An Analysis of Potential Standards for Post-Quantum Cryptosystems

by

Zachary Welch

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Science

in

Mathematics with Concentration in Mathematics

Carleton University
Ottawa, Ontario

Abstract

In the near future, general quantum computers could become a reality. Because of this we have a need for updating our public key infrastructure to resist quantum attacks. Among the many proposed cryptosystems that we could use to replace our modern cryptosystems are the four cryptosystems we will be analysing: McEliece, Classic McEliece, McNie, and NTRUEncrypt. We will cover the cryptosystems in full detail before analysing how secure and how efficient the cryptosystems are.

Acknowledgements

The author would like to thank his supervisors, professor Hugh Williams and professor Yuly Billig for all of their help providing corrections to the thesis and assistance with some of the mathematics behind it. The author would like to thank professor Jeffrey Hoffstein for assisting with the inconsistencies in the NTRUEncrypt cryptosystem. The author would like to thank professor Colin Ingalls for helping with the proof of the NTRUEncrypt public key generation failing. The author would like to thank his defence committee for their valuable feedback for this thesis.

Contents

List of Tables

1 Introduction

In recent years there have been many advances in quantum computing which suggest that full size, general quantum computers could become a reality in the near future [9]. One of the major concerns with quantum computers is that they are expected to be able to run Shor's Algorithm, compromising all of the public key cryptosystems that are widely used today [46]. Because of this it is important that we update our public key infrastructure to quantum-resistant cryptosystems.

Currently, quantum computers exist; however, they are not even close to breaking modern cryptosystems. There are two issues that need to be resolved before quantum computers will be effective against RSA and discrete logarithm problems. The first issue is that running Shor's Algorithm against modern cryptosystems requires several thousand qubits [46]. The second issue is that the qubits in the quantum computers we have developed are unreliable and require constant error correction to perform calculations, further increasing the number of required qubits. The largest quantum computer we know of is Google's 72-qubit quantum chip, named "Bristlecone", which is far short of being a threat to our cryptosystems [31].

While we do not know how long it will take for these issues to be resolved, some experts have predicted that quantum computers capable of breaking our current cryptosystems might be built within the next 20 years. It took almost 20 years to deploy our modern cryptosystem infrastructure [9]. Moreover, encoded information could easily contain

information someone would want to remain private for years into the future. Username and password combinations would be one such example, as most people do not like using different usernames and passwords if they can avoid it [12].

To upgrade our systems we first need to develop standards for quantum-resistant cryptography that can be easily introduced into our current public-key infrastructure. In 2017 NIST, the US National Institute for Standards and Technology, put out a call for proposals for post-quantum cryptosystems. At the end of 2017 they published proposals for 69 different cryptosystems and requested comments from the public as part of their evaluation process [39].

These 69 cryptosystems formed the first round of the Post-Quantum Cryptography Standardization competition; throughout 2018 attacks were developed and 5 submissions were withdrawn. There were other systems that were successfully attacked in this timeframe; however, parameter or algorithm changes were able to defend against these attacks.

In addition to the first round, the second round of the Post-Quantum Cryptography Standardization competition was announced. Currently, we only have details about which cryptosystems are being considered for the second round.

In this thesis we will cover the Goppa error correcting code, which is the basis of a number of so called "Hamming weight" cryptosystems. This will also give us insight into the main concept behind other error correcting code based cryptosystems. The next thing covered will be the McEliece Cryptosystem, the oldest proposed post-quantum cryptosystem and the

cryptosystem that served as the inspiration for a large number of the NIST submissions. The McEliece cryptosystem itself was not included in the NIST submission list; however, a lot of the systems and attacks currently under analysis are variants of those originally used with this cryptosystem.

Then we will cover two of the NIST submissions that were inspired by McEliece, those being the Classic McEliece cryptosystem and the McNie cryptosystem. The Classic McEliece cryptosystem, being a variant of the McEliece Cryptosystem, uses Goppa codes to identify an error vector in a ciphertext. The McNie cryptosystem applies the idea of the McEliece cryptosystem to a rank metric instead of a Hamming weight.

The last cryptosystem we will cover is the NTRUEncrypt cryptosystem, a variant of one of the NTRU series of cryptosystems that was submitted to NIST. NTRU cryptosystems have been undergoing improvement for over 20 years, and because of that, NTRUEncrypt is considered one of the top competitors for standardization.

In support of this analysis, this thesis will: prove the two Goppa code definitions are equivalent; describe the details behind a number of potential implementation attacks on McEliece; show how to modify an attack on Classic McEliece to attack the Initial McEliece cryptosystem; demonstrate that the documentation for NTRUEncrypt describes a key generation algorithm that will fail to generate a valid key pair; and will identify a potential miscalculation in the NTRUEncrypt decryption error rates.

2 Binary Goppa Codes

2.1 Error Correcting Codes

Error correcting codes provide a means of encoding information in such a way that should there be a small number of pieces of information read incorrectly, the original information can be reconstructed perfectly. Unfortunately, encoding information with error correcting codes expands the amount of information being transmitted.

A linear error correcting code C is a subspace in a vector space F^n over some finite base field F. The dimension k of this subspace is called the dimension of the code, and the dimension n of the vector space is called the length of the code.

These codes use 3 parameters, n, k, and t, all positive integers with $k < n$. The code will encode a vector of length k over R as a vector of length n over F by use of a k by n generator matrix G. To encode the vector $\vec{a}$, we calculate its associated codeword $\vec{c} = G\vec{a}$. The parity check matrix H of a code is an n by $n-k$ matrix with the property that $\vec{c}$ is a codeword if and only if $H\vec{c} = \vec{0}$. This tells us that the code C is the nullspace of the parity check matrix H.

As this is an error correcting code, there must also exist a error correction algorithm, $\Delta : F^n \to C \cup \{NA\}$, and some notion of weight, that is a method of quantifying how big or small a vector is. For almost any vector $\vec{r}$ such that the weight of $\vec{r} - \vec{c}$ is less than or equal to t, for some $\vec{c} \in C$, then $\Delta(\vec{r}) = \vec{c}$. In words, if $\vec{r}$ and $\vec{c} \in C$ differ by a vector small enough to be corrected, the algorithm is able to identify $\vec{c}$ from $\vec{r}$. If, for all $\vec{a} \in C$, the

weight of $\vec{r} - \vec{a}$ is greater than what the code is capable of correcting, the decoding algorithm can either output the nearest $\vec{c} \in C$, or $\{NA\}$. This $\{NA\}$ is typically used to denote a decoding failure, that is the algorithm could not find a close codeword to the given vector.

The most common weight that these algorithms use is the Hamming weight, which is defined as the number of non-zero entries in the vector. As an example, the binary vector $\vec{e} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}^T \in \mathbb{Z}_2^7$ has a Hamming weight of 3.

An important note is that because $G\vec{a}$ is a codeword for all $\vec{a}$, $H(G\vec{a}) = \vec{0}$ for all $\vec{a}$. By associativity we have $(HG)\vec{a} = \vec{0}$ for all $\vec{a}$, but this implies that HG is the zero matrix. Additionally, a code is defined by its set of codewords, and in general H and G are not unique for that code. In other words, a code can have many different generator and parity check matrices that define the same set of codewords. In fact, for applications in cryptography, this property is used extensively by generating different G and H for the same code.

A simple example of a linear error correcting code is the Hamming(7,4) code which encodes 4 bits of information into 7 bits capable of correcting an error with Hamming weight at most 1. The Hamming(7,4) code has generator matrix G and parity check matrix H defined as follows:

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

To encode 4 bits of information using this code we first write the data as a vector of length 4 and then simply multiply this vector on the left by the generator matrix G. For example to encode the binary information 0101 using this code, we write it as the column vector $\vec{a} = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}^T$ and then multiply it by G to give us the codeword $G\vec{a} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}^T$.

The Hamming code is very simple to decode; by multiplying the encoded information on the left by the parity check matrix H we can find if the word has an error as well as where it is. If we have an error in our encoded information we can represent our encoded information as $G\vec{a} + \vec{e}$ where the error vector $\vec{e}$ will have a single nonzero entry. If we multiply H on the right by $G\vec{a} + \vec{e}$ we get $HG\vec{a} + H\vec{e}$; however, we know $HG = 0$, so we simply end up with $H\vec{e}$. As $\vec{e}$ has only a single nonzero entry we need only find the column that matches the calculated $H\vec{e}$.

Continuing on with our previous example suppose we had an error in the third bit of $G\vec{a}$ and had instead read $\vec{b} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}^T$. When we multiply this vector by H we get $H\vec{b} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T$. Looking at the parity check matrix we can see that this matches the third column which tells us the error is in the third bit. Once we have corrected the error, finding the original 4 bits can be done by reading out the third, fifth, sixth and seventh

bits to obtain $\vec{a} = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}^T$.

This code can only correct 1 error. If instead there were 2 errors, the procedure would have given us the incorrect answer for $\vec{a}$. Thus, with our example, had there been errors in both the first and third bits, we would have concluded the error had instead been with the second bit and would obtain $\vec{a} = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}^T$.

2.2 Generating Binary Goppa Codes

Goppa codes are a type of linear error correcting code over finite fields and function similarly to the Hamming(7,4) code. Originally introduced by Valerii Denisovich Goppa in 1970 [26], Goppa codes have a generator matrix G, and parity check matrix H, but, unlike the Hamming(7,4) code, are capable of correcting errors with Hamming weight greater than 1. This does, however, make the decoding process more difficult as we cannot simply identify the column of H matching the vector $H(G\vec{a} + \vec{e})$.

Before we can generate our Goppa code we need to set some parameters which will define how much information will be encoded, the size of the encoded information and how many errors the code can correct. For a given $m \in \mathbb{N}$ we will define $q = 2^m$ which will give us an upper bound on the size of the code. The symbol $n \in \mathbb{N}$ will represent the number of bits in the encoded data, with the limitation that $n \leq q$. The last parameter we will choose is the number of errors t that the code can correct: $2 \leq t \in \mathbb{N}$, which has the additional restriction that $m \cdot t < n$. With all of these we can calculate the last parameter $k = n - mt$, which will be the number of bits

of information that will be encoded.

There are three more parameters which will define the code itself. The first is the field $F_q = \mathbb{Z}_2[z]/\langle f(z)\rangle$ where $f(z) \in \mathbb{Z}_2[z]$ is an irreducible monic polynomial of degree m. The second is a monic irreducible polynomial $g(x) \in F_q[x]$ of degree t. The last is an ordered list of n distinct elements $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ of F_q.

We will use these parameters to define the parity check matrix and then use the parity check matrix to calculate the generator matrix.

$$\bar{H} = \begin{bmatrix} \frac{\alpha_1^0}{g(\alpha_1)} & \frac{\alpha_2^0}{g(\alpha_2)} & \cdots & \frac{\alpha_n^0}{g(\alpha_n)} \\ \frac{\alpha_1^1}{g(\alpha_1)} & \frac{\alpha_2^1}{g(\alpha_2)} & \cdots & \frac{\alpha_n^1}{g(\alpha_n)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \frac{\alpha_2^{t-1}}{g(\alpha_2)} & \cdots & \frac{\alpha_n^{t-1}}{g(\alpha_n)} \end{bmatrix} \in Mat_{t,n}(F_q)$$

Every entry $\bar{H}_{i,j}$ of $\bar{H}$ is an element of $F_q = \mathbb{Z}_2[z]/\langle f(z)\rangle$ and can be represented by an element of $\mathbb{Z}_2[z]$ of degree less than the degree of $f(z)$. This representation will be a polynomial of degree less than m with binary coefficients and can be written as:

$$\bar{H}_{i,j} = \frac{\alpha_j^{i-1}}{g(\alpha_j)} = b_{i,j,m-1} \cdot z^{m-1} + \cdots + b_{i,j,1} \cdot z + b_{i,j,0}$$

We can use this to replace all elements of $\bar{H}$ with a column of binary elements, converting our t by n matrix of elements of F_q into a $t \cdot m$ by n matrix of binary elements. This new matrix will be our parity check matrix for our Goppa code.

$$H = \begin{bmatrix} b_{1,1,0} & b_{1,2,0} & \dots & b_{1,n,0} \\ b_{1,1,1} & b_{1,2,1} & \dots & b_{1,n,1} \\ \vdots & \vdots & \vdots & \vdots \\ b_{1,1,m-1} & b_{1,2,m-1} & \dots & b_{1,n,m-1} \\ b_{2,1,0} & b_{2,2,0} & \dots & b_{2,n,0} \\ \vdots & \vdots & \vdots & \vdots \\ b_{t,1,m-1} & b_{t,2,m-1} & \dots & b_{t,n,m-1} \end{bmatrix} \in Mat_{mt,n}(\mathbb{Z}_2) \tag{1}$$

It should be noted that H and $\bar{H}$ define the same code over $\mathbb{Z}_2$. That is, if we take the nullspace vectors over the fields that the matrices are over, $NULL(H) = NULL(H) \cap \mathbb{Z}_2^n$. However, for use in the McEliece cryptosystems the conversion to H is necessary.

We can use the parity check matrix to find a generator matrix for the code. From the previous section we know that $H\vec{a} = 0 \leftrightarrow \vec{a}$ is a codeword. In other words, the columns of G should generate the nullspace of H. This lets us conclude that G should be a binary matrix with n rows and $n - rank(H)$ linearly independent columns. This process does not define a unique G, in general a code will have many different generator matrices but identifying one is sufficient for most applications.

We can approximate the probability that $rank(H) \neq m \cdot t$ as the probability that a randomly generated binary matrix of similar dimensions is not of maximum rank. The probability that a matrix is of full rank can be calculated as the probability that each row is not a linear combination of previous rows, and because we are looking at a randomly generated binary matrix this can be simplified to the number of linear combinations of previous rows divided by the total number of possible rows. We obtain $\prod_{i=1}^{mt}(1 - 2^{n-i})$, which for the McEliece parameters that we will specify later,

is less than 2^{-188}; because this number is so low we will assume that $rank(H) = m \cdot t$.

2.3 Encoding and Decoding using Goppa Codes

Encoding information with a Goppa code is performed using the same method as the Hamming(7,4) code: the code will encode a length k binary vector $\vec{a}$ by multiplying it by the generator matrix to obtain the codeword $\vec{c} = G\vec{a}$. Decoding a codeword is a simple case of using linear algebra to solve the system of equations given by G and the codeword.

Correcting errors however, is not nearly as simple a task. An important note is that because we are dealing with a binary code, it is sufficient to locate the errors, because if an error is found we can do a simple bit flip on each of the error locations to get the nearest codeword. If, for a vector $\vec{r}$, $H\vec{r} \neq 0$, we know that $\vec{r}$ is not a codeword. Moreover, the output vector $H\vec{r}$ will be the sum of columns of H where the errors appear. We can determine $\vec{e}$ with an exhaustive search over vectors of low Hamming weight by checking if $H\vec{e} = H\vec{r}$. For any pair of vectors such that $H\vec{e} = H\vec{r}$ we know that $\vec{r} + \vec{e}$ is a codeword. However, there are $\binom{n}{|e|}$ possible $\vec{e}$ to try, so for large n and number of errors this method is impractical and we need a new approach. The approach we are going to describe is known as either Algebraic decoding or Syndrome decoding [3].

For this new approach we will need to use a different but equivalent definition of our codeword. A vector $\vec{c} = \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}^T \in \mathbb{Z}_2^n$ is a

codeword if and only if

$$\sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \quad mod\, g(x)$$

Note that because $g(x)$ is an irreducible polynomial over a field with degree $t \geq 1$, we have that there exists a unique polynomial $\beta_i(x)$ with degree less than t such that $\beta_i(x) \cdot (x - \alpha_i) \equiv 1 \quad mod\, g(x)$. This $\beta_i(x) \equiv \frac{1}{x-\alpha_i} \quad mod\, g(x)$, which allows us to restate the above equation as $\sum_{i=1}^{n} c_i \cdot \beta_i(x) \equiv 0 \quad mod\, g(x)$.

Theorem 1 *For a column vector* $\vec{c} = \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}^T \in \mathbb{Z}_2^n$ *and matrix* H *as defined in (1) using parameters* $\alpha_1, \dots, \alpha_n$ *and* $g(x)$*, we have that* $H\vec{c} = 0 \leftrightarrow \sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} \equiv 0 \quad mod\, g(x)$

The proof of this theorem can be found in Appendix A.

Suppose we want to decode the vector $\vec{r} = \begin{bmatrix} r_1 & r_2 & \dots & r_n \end{bmatrix}^T \in \mathbb{Z}_2^n$. We will assume there exists a codeword $\vec{c} = \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}^T \in \mathbb{Z}_2^n$ that differs in t bits or less from our vector. This codeword will be paired with the vector $\vec{e} = \begin{bmatrix} e_1 & e_2 & \dots & e_n \end{bmatrix}^T \in \mathbb{Z}_2^n$ where $\vec{r} = \vec{c} + \vec{e}$. This allows us to define the set $M = \{i | e_i = 1\}$. While we do not know what $\vec{c}$ and $\vec{e}$ are they will help us understand how the decoding algorithm works.

We are interested in two polynomials; the first one is the syndrome $S(x)$ which we can calculate as:

$$S(x) \equiv \sum_{i=1}^{n} \frac{r_i}{x - \alpha_i} \equiv \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} + \sum_{i=1}^{n} \frac{e_i}{x - \alpha_i} \equiv \sum_{i=1}^{n} \frac{e_i}{x - \alpha_i} \quad mod\, g(x)$$

The second polynomial we will be using is the error locator polynomial $\sigma(x) = \prod_{j \in M} (x - \alpha_j) \in F_q[x]$, and the derivative of this polynomial $\sigma'(x) = \sum_{i \in M} \prod_{j \in M, j \neq i} (x - \alpha_j) \in F_q[x]$. Important to note is that calculating these functions from the definitions would require knowledge of the set M, which we do not have. What we will instead do is identify $\sigma(x)$ using other means and calculate the error vector using $\sigma(x)$.

If we multiply $S(x)$ by $\sigma(x)$ we obtain

$$\begin{aligned} S(x) \cdot \sigma(x) &\equiv \sum_{i=1}^{n} \tfrac{e_i}{x - \alpha_i} \cdot \prod_{j \in M} (x - \alpha_j) \\ &\equiv \sum_{i \in M} \prod_{j \in M, j \neq i} (x - \alpha_j) \\ &\equiv \sigma'(x) \qquad\qquad mod\ g(x) \end{aligned} \tag{2}$$

From here there are two ways to proceed. The initial algorithm involves solving for $\sigma(x)$ by solving a system of t linear equations. This algorithm was very slow but a faster algorithm was discovered in 1975 by Patterson and has been named the Patterson algorithm [42]. We will first describe the initial algorithm, as the Patterson algorithm works in a very similar way.

To solve for $\sigma(x)$ we can first note that $\deg(\sigma(x))$ is equal to the cardinality of M we know that $\deg(\sigma(x)) \leq t$. Using the initial algorithm, we need to consider two possibilities, $\deg(\sigma(x)) < t$ or $\deg(\sigma(x)) = t$. While these cases will be dealt with very similarly, there exist key differences, and attempting one algorithm when dealing with the case of the other will not identify $\sigma(x)$.

If $\deg(\sigma(x)) < t$ then we can write out the polynomial $\sigma(x)$ as $\sigma_0 + \sigma_1 \cdot x + \cdots + \sigma_{t-1} \cdot x^{t-1}$ and observe that $\sigma'(x) = \sigma_1 + \sigma_3 \cdot x^2 + \cdots + \sigma_s \cdot x^{s-1}$ where s is the largest odd integer less than t. Because we know $S(x)$ and $g(x)$, we

can calculate $S(x) \cdot x^k mod g(x)$ for all $0 \leq k < t$. This will allow us to expand $S(x) \cdot \sigma(x) \equiv \sigma'(x)$ to $\sum_{i=0}^{t-1} S(x) \cdot x^i \cdot \sigma_i \equiv \sigma_1 + \sigma_3 \cdot x^2 + \cdots + \sigma_s \cdot x^{s-1} \ mod \ g(x)$. Considering each different exponent of x we obtain t linear equations with $\sigma_0, \ldots, \sigma_{t-1}$ as the unknowns. Solving for these $\sigma_0, \ldots, \sigma_{t-1}$ values will give us the set of soltions $\alpha \cdot \sigma(x) \ \forall \alpha \in F_q$.

We know that $\sigma(x)$ is the product of monic polynomials and is therefore monic itself. Because of this identifying $\sigma(x)$ from the set of solutions $\alpha \cdot \sigma(x)$ $\forall \alpha \in F_q$ should only require identifying the monic solution.

If instead $\deg(\sigma(x)) = t$, we can use much the same process, the only exception being that we need to consider the x^t term of the polynomial. By the construction of $\sigma(x)$ we know that it is monic and will be of the form $\sigma(x) = \sigma_0 + \sigma_1 \cdot x + \cdots + \sigma_{t-1} \cdot x^{t-1} + x^t$. This will give us that $\sigma'(x) = \sigma_1 + \sigma_3 \cdot x^2 + \cdots + \sigma_s \cdot x^{s-1}$ where s is the largest odd number less than or equal to t and $\sigma_t = 1$. Using the same process of calculating $S(x) \cdot x^k \ mod \ g(x)$ for all $0 \leq k \leq t$, the only difference being the addition of the $k = t$ term we again obtain a system of t linear equations with t unknowns. From this point we can solve for all σ_i values which will give us $\sigma(x)$.

To locate the errors in our code we calculate $\sigma(\alpha_j)$ for each α_j specified for our code. Because of how $\sigma(x)$ was defined we will have that $\sigma(\alpha_j) = 0$ if and only if $j \in M$. What we end up with is $M = \{j | \sigma(\alpha_j) = 0\}$; and now that we have M, we can solve for our error vector e. With this we can calculate the nearest codeword $c = r + e$, and we already know that decoding a codeword is a simple case of linear algebra. This concludes the initial decoding algorithm.

Before we describe the Patterson algorithm, we need to prove a theorem.

Theorem 2 *If F_q is of characteristic 2, then the Frobenius mapping Ψ:$F_q \rightarrow F_q$ by $\Psi(x) = x^2$ is an automorphism, and derivatives of polynomials in $F_q[x]$ are perfect squares.*

When considering the Frobenius mapping, we note that $1^2 = 1$, $(x+y)^2 = x^2 + 2xy + y^2 = x^2 + y^2$ and $(xy)^2 = x^2y^2$, so can conclude that $\Psi(x)$ is a homomorphism. The kernel of this homomorphism is the set of solutions to $x^2 = 1$ which we can rewrite as $x^2 + 1 = 0$ or $(x + 1)(x + 1) = 0$. We are in a field and so we have no zero divisors, implying that $x + 1 = 0$ or $x = 1$. Therefore $ker(\Psi) = \{1\}$, Ψ is an automorphism and importantly for the second half of the theorem, Ψ^{-1} exists.

Now we will consider the polynomial $p(x) = \sum_{i=0}^{n} p_i \cdot x^i \in F_q[x]$ and its derivative $p'(x) = \sum_{i=1}^{n} i \cdot p_i \cdot x^{i-1}$. For the terms of the derivative with an odd exponent we note that they are multiplied by an even integer i, which would give 0 over F_q. For even exponent terms i is an odd integer and multiplication by i is equivalent to multiplication by 1. This allows us to rewrite $p'(x) = \sum_{i=0}^{k} p_{2i+1} \cdot x^{2i}$ where $k = \lfloor \frac{n-1}{2} \rfloor$.

Given our $p(x)$ polynomial, we consider the polynomial $q(x) = \sum_{i=0}^{k} \Psi^{-1}(p_{2i+1}) \cdot x^i$. If we square $q(x)$ we will obtain

$$\sum_{j=0}^{k} \sum_{i=0}^{k} \Psi^{-1}(p_{2j+1}) \cdot x^j \cdot \Psi^{-1}(p_{2i+1}) \cdot x^i$$

$$= \sum_{j=0}^{k} \sum_{i=0}^{j-1} 2 \cdot \Psi^{-1}(p_{2j+1}) \cdot x^j \cdot \Psi^{-1}(p_{2i+1}) \cdot x^i + \sum_{i=0}^{k} (\Psi^{-1}(p_{2i+1}) \cdot x^i)^2$$

$$= 0 + \sum_{i=0}^{k} \Psi(\Psi^{-1}(p_{2i+1})) \cdot x^{2i} = p'(x)$$

As our choice of $p(x)$ was arbitrary, we conclude that the derivative of all polynomials in $F_q[x]$ are perfect squares. The other conclusion we can draw from this proof that will be useful is that a polynomial is a perfect square if and only if all terms with an odd exponent have coefficient 0. ■

An interesting note is that while $\Psi(y)$ is not a linear function, it can be treated as such over $F_q[x]/\langle g(x)\rangle$. Instead of thinking of y as an element of $F_q[x]/\langle g(x)\rangle$ we instead think of it as an element of $\mathbb{Z}_2[x,z]/\langle g(x), f(z)\rangle$ where $f(z)$ is the function that was used to define F_q. Squaring a function over this field is linear and therefore has an associated matrix τ such that if an element y of $\mathbb{Z}_2[x,z]/\langle g(x), f(z)\rangle$ was expressed as an $m \cdot t$ length vector then $\tau \cdot y = y^2$.

This might not seem that useful as squaring functions is already easy to compute, and τ is not trivial to calculate. However, τ only depends on $g(x)$ and $f(z)$, so it can be calculated beforehand and stored as part of the private key. More importantly, τ is a matrix associated with a bijective linear map, therefore τ^{-1} exists and can be used to calculate the square root of an element of $F_q[x]/\langle g(x)\rangle$. This gives us a quick method for calculating the square roots of functions over our field at the cost of storing an extra matrix as part of the private key.

With these results, we can now describe the Patterson algorithm. We will start by splitting our $\sigma(x)$ into terms with even and odd exponents. After factoring x from the odd exponent group identify the square roots of those two new polynomials, which necessarily exist by the previous theorem. This

will produce functions $\gamma(x)$ and $\delta(x)$ such that $\sigma(x) = (\gamma(x))^2 + x \cdot (\delta(x))^2$ with $\deg(\gamma(x)) \leq \frac{t}{2}$ and $\deg(\delta(x)) < \frac{t}{2}$. Taking the derivative of this gives us that $\sigma'(x) = (\delta(x))^2$.

As $S(x) \in F_q[x]/\langle g(x)\rangle$ is a non-trivial element of a field we know there exists $h(x) \in F_q[x]/\langle g(x)\rangle$ such that $S(x) \cdot h(x) \equiv 1 \ (mod\ g(x))$. If $h(x) = x$ then we just set $\sigma(x) = x$ and we are done identifying $\sigma(x)$. Replacing $\sigma(x)$ by $(\gamma(x))^2 + x \cdot (\delta(x))^2$ in (2) and multiplying both sides by $h(x)$ we get $(\gamma(x))^2 + x \cdot (\delta(x))^2 \equiv h(x) \cdot (\delta(x))^2 \ mod\ g(x)$. Because $F_q[x]/\langle g(x)\rangle$ is a field of characteristic 2 we can see that $\Psi(y) = y^2 \in F_q[x]/\langle g(x)\rangle$ is an automorphism. Therefore, there necessarily exists $\eta(x) = \Psi^{-1}(h(x) + x) \in F_q[x]/\langle g(x)\rangle$ with $(\eta(x))^2 = h(x) + x$. Manipulating the above equations we get:

$$(\gamma(x))^2 + x \cdot (\delta(x))^2 \equiv h(x) \cdot (\delta(x))^2 \ mod\ g(x)$$
$$(\gamma(x))^2 \equiv (h(x) + x) \cdot (\delta(x))^2 \ mod\ g(x)$$
$$\Psi^{-1}(\gamma(x))^2 \equiv \Psi^{-1}(\eta(x))^2 \cdot \Psi^{-1}(\delta(x))^2 \ mod\ g(x)$$
$$\gamma(x) \equiv \eta(x) \cdot \delta(x) \ mod\ g(x)$$

We are left with the equation $\gamma(x) \equiv \eta(x) \cdot \delta(x) \ mod\ g(x)$ where $\eta(x)$ is known and the degrees of $\gamma(x)$ and $\delta(x)$ are bounded. We will identify $\delta(x)$ such that $\eta(x) \cdot \delta(x) \ mod\ g(x)$ has degree at most $\lfloor \frac{t}{2} \rfloor$. Looking at coefficients of terms with degree greater than $\lfloor \frac{t}{2} \rfloor$ we obtain $\lfloor \frac{t-1}{2} \rfloor$ linear equations with the $\lfloor \frac{t-1}{2} \rfloor + 1$ coefficients of $\delta(x)$ as our unknowns. We are looking for a nontrivial set of unknowns which we can find by either setting one of coefficients of $\delta(x)$ to a nonzero value or by identifying the nullspace of these equations. With our $\delta(x)$ we can calculate $\gamma(x) \equiv \eta(x) \cdot \delta(x) \ mod\ g(x)$

and $\sigma(x) = (\gamma(x))^2 + x \cdot (\delta(x))^2$. Once we have our $\sigma(x)$ we can identify e using the same method as the previous decoding algorithm, concluding the explanation of the Patterson algorithm.

Solving a set of k linear equations with k unknowns involves about k^3 calculations [35]. The Patterson algorithm halves our k, effectively reducing the amount of calculations required for this step by over 87%.

Now we will show that the minimum distance between codewords is $2t+1$. To recall, a vector $d \in \mathbb{Z}_2^n$ is a codeword if and only if $\sum_{i=1}^{n} \frac{d_i}{x-\alpha_i} \equiv 0 \; mod \; g(x)$ and what we will demonstrate is that for any two codewords c_1 and c_2, $c_1 - c_2$ has weight at least $2t+1$. Suppose by contradiction that there exists two codewords c_1 and c_2 such that the Hamming distance between them is less than $2t+1$. Because this is a linear code, $c' = c_1 + c_2$ is a codeword with Hamming weight less than $2t+1$. By our definition we have that $\sum_{i=1}^{n} \frac{c'_i}{x-\alpha_i} \equiv 0 \; mod \; g(x)$. Because $F_q[x]/\langle g(x) \rangle$ is a field we can multiply it by the non-zero value $\prod_{j=1}^{n} (x - \alpha_j)$ to obtain $\sum_{i=1}^{n} c'_i \prod_{j=1, j\neq i}^{n} (x - \alpha_j) \equiv 0 \; mod \; g(x)$.

Let C' be a subset of $\{1, 2, \ldots, n\}$ such that $l \in C' \leftrightarrow c'_l = 1$. We can see that $\sum_{i=1}^{n} c'_i \prod_{j=1, j\neq i}^{n} (x - \alpha_j)$ is the derivative of $\prod_{i \in C'} (x - \alpha_i)$. Replacing that we obtain $\frac{d}{dx} \prod_{i \in C'} (x - \alpha_i) \equiv 0 \; mod \; g(x)$. As we have already seen $\frac{d}{dx} \prod_{i \in C'} (x - \alpha_i)$ is the square of an element $b(x) \in F_q[x]$.

The properties of $b(x)$ that we are interested in are that $\deg(b(x)) \leq \frac{2t-1}{2} < t$ and $b(x)^2 \equiv 0 \; mod \; g(x)$. By the field properties we have that $b(x) \equiv 0 \; mod \; g(x)$, which implies $g(x)$ divides $b(x)$. However, $b(x)$ is of smaller degree than $g(x)$ which is a contradiction and gives us the conclusion that the minimum distance between codewords is $2t + 1$.

2.4 Example

For our example we will use the following parameters and polynomials: $m = 4$, $q = 16$, $n = 10$, $t = 2$, $k = 2$, $f(z) = z^4 + z + 1$, $g(x) = x^2 + x + z^3$. The n elements α_i of $F_q = \mathbb{Z}_2[z]/\langle f(z) \rangle$ are listed below along with the associated $\frac{1}{g(\alpha_i)}$, $\frac{\alpha_i}{g(\alpha_i)}$ and $\frac{1}{x-\alpha_i}$.

α_i	$\frac{1}{g(\alpha_i)}$	$\frac{\alpha_i}{g(\alpha_i)}$	$\frac{1}{x-\alpha_i}$
1	z^3+z^2+z+1	z^3+z^2+z+1	$(z^3+z^2+z+1)x$
z	$z+1$	z^2+z	$(z+1)x+(z^2+1)$
z^2	z^3	z^2+z	$(z^3)x+(z^3+z^2+z)$
z^3	z^3+z	z^3+z^2+z+1	$(z^3+z)x+(z^2+1)$
$z+1$	$z+1$	z^2+1	$(z+1)x+(z^2+z)$
z^2+1	z^3	z^3+z^2+z	$(z^3)x+(z^2+z)$
z^2+z+1	z	z^2+1	$(z)x+(z^3+z^2)$
z^3+1	z^3+z	z^3+z^2+z+1	$(z^3+z)x+(z^3+z^2+z+1)$
z^3+z	z^3+z^2	1	$(z^3+z^2)x+(z^2)$
z^3+z+1	z^3+z^2	z^3+z^2+1	$(z^3+z^2)x+(1)$

Using the $\frac{1}{g(\alpha_i)}$ and $\frac{\alpha_i}{g(\alpha_i)}$ values we can calculate our parity check matrix H using (1). With H we can calculate our generator matrix G from the nullspace.

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This Goppa code encodes 2 bits of information into 10 bits and is

capable of correcting up to two errors. For this example we will encode the vector $\vec{v} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$. This vector will encode to $G\vec{v} = \vec{c} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}^T$. To demonstrate error correction we will attempt to recover $\vec{v}$ supposing that an error $\vec{e} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}^T$ occurred.

Our resulting vector is $\vec{r} = \vec{c} + \vec{e} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T$, we will now calculate the syndrome $S(x)$:

$$S(x) \equiv \sum_{i=1}^{n} \frac{r_i}{x - \alpha_i} \mod g(x)$$

$$S(x) \equiv (z^3)x + (z^3 + z^2 + z) + (z)x + (z^3 + z^2) + (z^3 + z^2)x + (1) \mod g(x)$$

$$S(x) \equiv (z^2 + z)x + (z) \mod g(x)$$

We can attempt to calculate $\sigma(x)$ under the assumption that the weight of our error is less than 2. What we will obtain is two linear equations with a single variable that has no solutions.

Now working under the assumption that the weight of our error is 2, we can write $\sigma(x) = \sigma_0 + \sigma_1 \cdot x + x^2$ and $\sigma'(x) = \sigma_1$. Applying these to the equation $S(x) \cdot \sigma(x) \equiv \sigma'(x)$ we obtain:

$$\begin{aligned} S(x) \cdot \sigma(x) &\equiv ((z^2 + z)x + (z)) \cdot (\sigma_0 + \sigma_1 \cdot x + x^2) \mod g(x) \\ \sigma'(x) &\equiv ((z^2 + z)x + (z)) \cdot ((\sigma_0 + z^3) + (\sigma_1 + 1)x) \\ \sigma_1 &\equiv (z^2 + z + \sigma_1(z^2 + z))x^2 + (\sigma_1(z) + \sigma_0(z^2 + z) + z^2 + z + 1)x \\ &\quad + (\sigma_0(z) + z + 1) \\ &\equiv (\sigma_1(z^2) + \sigma_0(z^2 + z) + 1)x + (\sigma_0(z) + z^2 + z + \sigma_1(z^2 + 1)) \end{aligned}$$

Because $g(x)$ is of greater degree than both sides of this equivalence we get that $\sigma_1 = (\sigma_1(z^2) + \sigma_0(z^2 + z) + 1)x + (\sigma_0(z) + z^2 + z + \sigma_1(z^2 + 1))$.

Differentiating between the different degrees of x, we obtain the two linear equations $\sigma_0(z) + z^2 + z + \sigma_1(z^2 + 1) = \sigma_1$ and $\sigma_1(z^2) + \sigma_0(z^2 + z) + 1 = 0$. Our two unknowns are σ_0 and σ_1, which we can solve for, to obtain:

$\sigma_0 = z^2 + 1$ and $\sigma_1 = z + 1$

Plugging these into our equation for $\sigma(x)$ we have that $\sigma(x) = x^2 + (z + 1)x + z^2 + 1 = (x - (z^3 + 1))(x - (z^3 + z))$. This allows us to conclude that the errors occur at positions i and j such that $\alpha_i = z^3 + 1$ and $\alpha_j = z^3 + z$.

Looking back at our table we have that $i = 8, j = 9$ and therefore $\vec{e} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}^T$. Adding this to our $\vec{r}$ will give us $\vec{c} = \vec{r} + \vec{e} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}^T$ and allow us to solve for our initial vector $\vec{v} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

3 McEliece Cryptosystem

The McEliece cryptosystem is a public key cryptosystem developed in 1978 by Robert McEliece [35]. The cyrptosystem was not widely implemented; this was almost entirely due to the large public key size of, at the time, about 524kb, compared to the single kilobit key sizes in RSA cryptosystems of comparable security [44]. However, unlike the RSA cryptosystem, the McEliece cryptosystem seems to be secure against attacks by a quantum computer [4]. This has brought a lot of recent attention to the system and it has become an important scheme to analyse for post-quantum cryptography.

The main concept behind the McEliece cryptosystem is that the public key is some form of obfuscated error correcting code. A message is encoded with this obfuscated error correcting code, and the ciphertext becomes that encoded message with some amount of errors that the code is capable of correcting. This concept is the inspiration for several candidates for post-quantum cryptography; however, the McEliece cryptosystem as it was originally designed is not one of the systems being analysed as part of the NIST Post-Quantum Cryptography Standardization competition. One possible reason this system not being included is the existence of the Classic McEliece cryptosystem which will be discussed in the next section.

3.1 Implementation Considerations

The first thing to generate for a McEliece cryptosystem is a binary Goppa code; for the public key we will need the generator matrix G and the parameters n, k and t. As we recall, these parameters are under the

constraint that $k + m \cdot t = n \leq 2^m$ for some m used to generate the binary Goppa code. We will also need to generate an invertible k by k binary matrix S and a n by n permutation matrix P, along with their respective inverses S^{-1} and P^{-1}. With these, we can calculate the matrix $G' = P \cdot G \cdot S$, which along with t will serve as our public key. The private key will be the matrices S^{-1} and P^{-1}, as well as the decryption algorithm for the code that generated G. The information needed for the decryption algorithm is the following: $f(z)$, $g(x)$ and $\begin{bmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_n \end{bmatrix}$ that were used to generate the Goppa code.

The McEliece cryptosystem encrypts k bits of information, in the form $\vec{u} = \begin{bmatrix} u_1 & u_2 & \dots & u_k \end{bmatrix}^T \in \mathbb{Z}_2^k$. To encrypt this message, we calculate $\vec{r} = G' \cdot \vec{u} \in \mathbb{Z}_2^n$ and generate a random vector $\vec{e} \in \mathbb{Z}_2^n$ of weight exactly t. Our ciphertext to be sent will be $\vec{c} = \vec{r} + \vec{e} \in \mathbb{Z}_2^n$.

For decryption, assume we received the ciphertext $\vec{c} = G'\vec{u} + \vec{e}$. The first step is to calculate $P^{-1}\vec{c} = P^{-1} \cdot G'\vec{u} + P^{-1}\vec{e}$. Expanding G', we get $P^{-1}\vec{c} = P^{-1} \cdot P \cdot G \cdot S\vec{u} + P^{-1}\vec{e}$, and by associativity of matrix multiplication $P^{-1}\vec{c} = G(S\vec{u}) + (P^{-1}\vec{e})$. Because S is a k by k matrix, $S\vec{u}$ is a binary vector of length k. Permutation matrices permute elements of a vector they are multiplied by, and therefore the Hamming weight of $P^{-1}\vec{e}$ is equal to the Hamming weight of $\vec{e}$.

Therefore $P^{-1}\vec{c}$ is a message of the form $G\vec{a} + \vec{e'}$, where $\vec{e'} = P^{-1}\vec{e} \in \mathbb{Z}_2^n$ is of weight t and $\vec{a} = S\vec{u} \in \mathbb{Z}_2^k$. Knowing $P^{-1}\vec{c}$ is in this form, we can apply the decryption algorithm for the Goppa code used to generate the code, using the information from the private key. If this decryption fails, or if $\vec{e}$ is not of Hamming weight t, we assume that $\vec{c}$ is not a possible

output for the encryption algorithm and return an error. This decryption algorithm will identify our $\vec{a}$, from here we can recover $\vec{u}$ by computing $S^{-1}\vec{a} = S^{-1} \cdot S\vec{u} = \vec{u}$.

3.2 Code Efficiency

The initial parameters for the McEliece cryptosystem when it was published in 1978 were $m = 10$, $n = 1024$, $t = 50$ and $k = 1024 - 10 \cdot 50 = 524$. This led to the public key matrix G' to have over half a million entries and the initial public key size around 524kb [35]; moreover, the ciphertext was about twice as long as the plaintext. By 1978 standards this was impractical, especially with 512 bit RSA being of similar security [44], with a public key size of only 1kb and the cipher and plaintexts being the same size.

For all of the cryptosystems we will analyse we need to understand different parameter sets and the bit security for each of those sets. Parameter sets are a list of recommended parameters for a cryptosystem which specify message lengths, key sizes and sometimes formatting.

The bit security of the system is a lower bound on the number of calculations on a base 2 logarithmic scale that, with our current best algorithms, would be required to attack the system in general. For example, a 256 bit secure cryptosystem would require, at least, 2^{256} calculations to attack it. NIST uses 128 bit for low security, 192 bit for medium security, and 256 bit for high security.

The McEliece cryptosystem was originally proposed to be 64 bit secure against the best known attack in 1978. With the introduction of both better algorithms and hardware the old parameters are no longer

considered secure and, as McEliece was never adopted as a public key cryptosystem, it doesn't have any standard sets of parameters to use. For the purposes of analysing the code we will use the same numbers as the Classic McEliece NIST submission, which as we will see, is of similar security level for the same parameters. There are two sets of parameters which we will list below; the names are not official, they simply allow us to reference them in the sequel.

name	m	n	t	k	bit security
initial6960	13	6960	119	5413	267
initial8192	13	8192	128	6528	>267

The security level of initial6960 is the same as that of the mceliece6960119 parameter set and cryptosystem we will cover later, the security of which is known to be 266.94 [4]. The security on initial8192, however, is not known, extrapolating the numbers from the improved Stern attack algorithm [5] it seems to be approximately 310. The exact number is not important as the security level is easily more than 256, and the system is secure short of a new attack algorithm.

When analysing code efficiency we are primarily interested in the data requirements for storing and sending keys and messages. The public key size will be the minimum amount of data required for me to communicate the information required for someone to encode a message with my public key. We assume that both parties have knowledge of the cryptosystem and parameters that will be used, as these are generally standardised and any variation would be a different system.

To determine how much information is required to send or store keys

we will be making use of the pigeonhole principle. We will assume that information is stored as binary strings, and there exists some mapping from those strings to the elements that make up the keys we want to send or store. This mapping must be both well defined and surjective, which implies that every element must have a pre-image unique to that element. By the pigeonhole principle we can conclude that the size of the set of binary strings is greater than or equal to the size of the set of possible elements. The set of binary strings of length $\bar{n}$ has size $2^{\bar{n}}$, so if we want to encode an element from a set of size N, we obtain that $2^{\bar{n}} \geq N$ or $\bar{n} \geq \lceil log_2(N) \rceil$ because $\bar{n}$ is a natural number. An important note is that this is a lower bound, in practice a much less efficient encoding is sometimes used to avoid having to perform excessive calculations to reconstruct the information.

The McEliece public key is just the matrix G', a n by k binary matrix, which can be encoded with $n \cdot k$ bits.

The private key consists of the invertible matrix S, the permutation matrix P, the Goppa code polynomials $f(z)$ and $g(x)$, the original generator matrix G and the ordered list of elements $(\alpha_1, \alpha_2, \ldots, \alpha_n)$. S is a k by k binary matrix and can thus be encoded with $k \cdot k$ bits. P is a permutation matrix and can theoretically be encoded using $\lceil log_2(n!) \rceil$ bits; however, to avoid having to perform excessive calculations P will likely be encoded less efficiently. A more practical implementation would store P as a list of n numbers between 1 and n, using $n \cdot \lceil log_2(n) \rceil$ bits instead.

For the Goppa code information $g(x)$ can be encoded by its coefficients, excluding the x^t term, as $g(x)$ is known to be monic. This will take $m \cdot t$ bits as each coefficient is an element of F_{2^m} and $g(x)$ is of degree t. G is

another n by k binary matrix, which we have seen can be encoded with $n \cdot k$ bits. Lastly, the ordered list of n elements can be encoded using $m \cdot n$ bits. This brings the total number of bits to encode the private key information up to $k \cdot k + n \cdot \lceil log_2(n) \rceil + n \cdot k + m \cdot t + m \cdot n$.

The plaintext and ciphertext are simply k and n bits respectively.

name	Public Key	Private Key	Plaintext	Ciphertext
initial6960	4.491MB	8.094MB	0.661KB	0.850KB
initial8192	6.375MB	11.481MB	0.797KB	1KB

Having a public key size of over 4 megabytes is impractical compared to modern RSA cryptosystems, which can function with as little as 2 kilobytes, this is the major drawback of the McEliece cryptosystem. The massive size of the private key information is also a huge drawback; however, the ratio of plaintext to ciphertext is around 0.78, which is pretty efficient compared to what we will see for other systems.

To give some context to these values, we will also give the values for the RSA2048 cryptosystem, currently the most widely used public key cryptosystem. For the RSA2048 cryptosystem, the public key consists of the modulus N, a 2048 bit number, and the public exponent e, a positive integer less than N. The private key only requires N and the private exponent d, another positive integer less than N. While more information is often given with the private key, such as the factors of N, this will be omitted as it is not necessary for decryption. Similarly, it is not uncommon for e to be a very small number compared to N, as low as 65537, but that is an implementation optimisation, so we will omit this from our

calculations.

Public Key	Private Key	Plaintext	Ciphertext	Bit Security
512B	512B	255B	256B	112

3.3 Implementation Reasoning

An interesting thing to note is that the decryption algorithm for the McEliece cryptosystem is a decoding algorithm for a binary error correcting code with the same parameters as the Goppa code used to generate it. As G' is our generator matrix for this code, we can identify a parity check matrix H' for the code as well. Moreover, using $G' = P \cdot G \cdot S$ we know that $H' = H \cdot P^{-1}$ because $H' \cdot G' = H \cdot P^{-1} \cdot P \cdot G \cdot S = H \cdot G \cdot S = 0$.

The McEliece cryptosystem specifically indicates the use of a binary Goppa code, but it seems to be that any Hamming weight, linear error correcting code can be substituted. In fact, the cryptosystem works perfectly well with any error correcting code capable of correcting t errors, even non-binary codes. Moreover, there exist error correcting codes that encode information more efficiently than Goppa codes, replacing Goppa codes with these error correcting codes would effectively reduce our n without changing our k or t.

As we saw in the previous section, the size of the public key is about $n \cdot k$ for binary codes and the ratio of length of ciphertext to plaintext is n to k. In short, decreasing n without affecting k will reduce the amount of ciphertext for the same amount of plaintext in addition to reducing the size

of the public key. Also, at the time McEliece was created, the best attack method against it would counter-intuitively require more time for smaller values n, if k and t are unchanged. With all of this, why does McEliece use Goppa codes?

In short, just about all of those codes are not secure, attacks have been developed against just about all of these proposals of McEliece variants. All variant codes proposed from before 2010 have been broken; and all but the most recently proposed variant have been broken to some extent.

Type of Code	Current Status
Binary Goppa Codes, 1978 [35]	Unbroken as of 2016
Generalized Reed Solomon Codes, 1986 [38]	Broken in 1992 [48]
Maximum Rank Distance Codes, 1991 [18] and 1993 [17]	Broken in 1994 [24] and 1996 [25]
Reed-Muller Codes, 1994 [47]	Broken in 2007 [36]
Quasi-cyclic subcodes of a primitive BCH code, 2005 [19]	Broken in 2008 [41]
Quasi-cyclic low density parity-check codes, 2007 [2]	Broken in 2008 [41]
Wild McEliece, 2010 [6]	Specific Instances broken in 2014 [11], [15]
Wild McEliece Incognito, 2011 [7]	Specific Instances broken in 2014 [15]
Quasi-cyclic moderate density parity-check codes, 2013 [37]	Broken in 2016 [43]
Random linear codes, 2016 [51]	Unbroken as of 2016

Table from [50], updated with an attack on MDPC codes.

One of the more interesting variants is the Maximum Rank Distance Codes, otherwise known as Gabidulin codes, which is a rank metric error correcting code instead of a Hamming weight error correcting code. This forces a number of changes to the cryptosystem such as the base field and error generation; more details about Gabidulin codes and the rank metric will be seen in Section 5.

From this table, we can see that several of the proposed variants went unbroken for years, and in one case, over a decade. Just because a system has not yet been broken, does not mean a break will not be developed. Excluding Binary Goppa codes, the only other code that has not been broken to some extent is the variant based on random linear codes. This variant is less than 3 years old at the time of writing; while the cryptosystem seems secure now, a new attack could be developed against it. While we can say the same thing about the initial McEliece system based on Binary Goppa codes, that system is over 40 years old and has had significantly more time spent analysing it.

3.4 Attacking McEliece

When describing attacks on cryptosystems, the attacks primarily fall into one of two types; plaintext recovery attacks and key recovery attacks. Plaintext recovery attacks are attacks that will only identify the plaintext associated with a given ciphertext. Key recovery attacks are attacks that will, given a public key, identify an associated private key that can be used to decode ciphertexts. When analysing the security level of a cryptosystem we do not care about this distinction, the security level is how secure the system is against all known attacks. It should be noted that key recovery attacks are more powerful as they allow an attacker to decode all ciphertexts for a given public key and allow the calculations to be done before a message is sent, if the attacker knows what public key they will be using.

In contrast to other error correcting codes, we do not have an efficient algorithm for breaking the McEliece cryptosystem for Goppa codes. We do

still have attacks against the system but for sufficiently large parameters these attacks are not feasible. The attack initially included with the paper proposing the cryptosystem involved guessing k bits of the error vector and attempting to solve for the plaintext using linear algebra. The fundamental idea behind the attack was that if k bits of the error vector were known, the ciphertext and public key could form a system of k linear equations with the k elements of the plaintext being the variables.

Here are the details of the attack, we first guess k bits of the error vector and assume they are all zeros. By using G' with $\vec{c} = G'\vec{u} + \vec{e}$, or after isolating the $\vec{u}$ term, $\vec{c} + \vec{e} = G'\vec{u}$, we can obtain k linear equations for the guessed entries of $\vec{c} + \vec{e}$, with the k entries of $\vec{a}$ being the unknowns. While you could solve the system by guessing what these entries are, this is less efficient than guessing new sets of k bits. The probability that the k selected bits are all zero is $\prod_{i=1}^{k} \frac{n-t-i}{n-i}$. On average, this will take 2^{263} attempts for the initial6960 parameter set and 2^{300} attempts for initial8192.

Moreover, for each of these attempts, an attacker would need to solve k linear equations with k variables, which as we have seen, requires roughly k^3 calculations. This also assumes there will exist a unique solution, which will occur exactly when the k rows of G associated with the known entries of $\vec{e}$ form an invertible k by k submatrix. A random k by k binary matrix will be invertible if and only if each successive row is not the sum of some number of previous rows. The probability of this is $\prod_{i=1}^{k}(1-2^{-i})$. An interesting note is that $\lim_{k\to\infty} \prod_{i=1}^{k}(1-2^{-i}) \approx 0.29$, so for large k we have that $\prod_{i=1}^{k}(1-2^{-i}) \approx 0.29$. This will only add an additional two bits of security at most, so in terms

of security level it is fairly insignificant. Multiplying everything through we can find that initial6960 is roughly 303 bit secure, and initial8192 is roughly 341 bit secure against this attack.

A more efficient attack is known as the Stern attack algorithm, developed by Jacques Stern in 1988 [49]. This algorithm uses the property of the minimum distance between any error correcing code codewords, that for any two codewords $\vec{c_1}$ and $\vec{c_2}$ such that $\vec{c_1} \neq \vec{c_2}$ we have that $\vec{c_1} - \vec{c_2}$ has weight of at least $2t + 1$. Suppose we were given the ciphertext $\vec{r} = \vec{c} + \vec{e}$ and we wanted to identify $\vec{u}$ such that $\vec{c} = G'\vec{u}$ and $\vec{e}$ has a weight less than or equal to t.

How the attack accomplishes this is to add the ciphertext, $\vec{r}$ as an additional column to our generator matrix G' to obtain $G^* = \begin{bmatrix} G' & \vec{r} \end{bmatrix}$. We will treat this matrix as a generator matrix for a code with parameters $n' = n$ and $k' = k + 1$. This code may or may not have error correcting capacity; however, the decoding algorithm requires knowledge of the decoding algorithm for G and is not useful to us. Regardless of if this code has error correction capacity or not, it is still a linear code, and so we can generate a full rank parity check matrix H^* of dimension $n' - k'$ by n', such that $H^* \cdot G^* = 0$.

The code generated by G^* is a linear code and so, if $\vec{c_1}$ and $\vec{c_2}$ are codewords, then so to is $\vec{c_1} + \vec{c_2}$. This vector will be $G^*\vec{u'}$, where $\vec{u'}$ is $\vec{u}$ with a 1 added onto the end. We have that $G^*\vec{u'} = G'\vec{u} + \vec{r} \cdot 1 = \vec{e}$, which by definition will be a vector of weight t.

By generating this code, we have reformulated the problem from an error correcting code to a problem called the shortest vector problem. As G' is

the generator matrix for an error correcting code capable of correcting an error of Hamming weight t, there does not exist an $\vec{u}$ such that the weight of $G'\vec{u}$ is less than $2t+1$. Consequently, if there existed a $\vec{y_0}$ such that the weight of $\vec{y} = G^*\vec{y_0}$ is less than $2t+1$, then $\vec{y}$ would not be in the code generated by G'.

By the structure of G^*, $G^*\vec{y_0}$ will be some element of the code generated by G plus an element of $\{0, \vec{r}\}$. However, if the element from $\{0, \vec{r}\}$ is 0, we have that $G^*\vec{y_0}$ is an element of the code generated by G, which we already know to be impossible because the weight of $G^*\vec{y_0}$ is less than $2t+1$. Therefore, $\vec{y} = G^*\vec{y_0}$ must be an element of the code generated by G plus $\vec{r}$, and by moving that addition by $\vec{r}$ to the other side of the equation, we can conclude that $\vec{y} + \vec{r}$ is in the code generated by G'. This tells us that $\vec{y} + \vec{e}$ is in the code generated by G'. This does not guarantee that $\vec{y} = \vec{e}$, it is possible that there is a non-zero codeword $\vec{c_1}$ such that $\vec{c_1} + \vec{e}$ has weight less than $2t+1$. If $\vec{y}$ has weight equal to t then we have $\vec{e} = \vec{y}$, $\vec{c} = \vec{y} + \vec{r}$ and identifying the plaintext is another system of equations that can be solved with linear algebra.

Now we just need to identify a vector of Hamming weight at most t in this code. If we assume $\vec{r}$ to be a ciphertext encoded by a McEliece cryptosystem, then there will exist exactly one vector of Hamming weight t and no other vectors of equal or lesser weight. To identify this vector we will specify two additional parameters l and $p \leq \frac{t}{2}$ for which we will determine values later. The code will work for reasonable values l and p; however, we can optimise these values to improve the speed of the attack.

We will select a uniformly random set Z of $n' - k'$ columns of H^*, and

a uniformly random subset Z_0 of l columns from Z. We then apply elementary row operations on H^* in an attempt to reduce the columns of Z to the identity matrix. This succeeds if and only if the columns of Z form an invertible binary matrix, which we have already seen happens with probability of roughly 0.29. If the matrix is not invertible and the row reduction fails, we choose a new Z and try again. Once a suitable Z is found, the remaining k' columns that are not in Z are randomly divided into two sets X and Y.

With the columns associated to Z forming the identity matrix, we can also associate each element of Z to a row of H^*, with the pairing being the row in which each column has the entry 1. We will look at the set of rows Z'_0 associated with the columns in the subset Z_0. For every set A of p elements from X, we calculate the sum of the columns over A, but only for the column elements in the rows specified by Z'_0. We will do the same for all sets B of p elements from Y, and again calculate the sum of the columns over the rows specified by Z'_0. We refer to these sums $\pi(A)$ and $\pi(B)$ respectively.

With these sums, we will look for any pair of sets, A and B, such that $\pi(A) = \pi(B)$. For each of those pairs, we calculate the sum of the $2p$ columns of $A \cup B$, this time not limiting ourselves to the rows specified by Z'_0. If the weight of this vector is equal to $t - 2p$, then we can add columns from Z to cancel out the remaining bits. We can then take the $2p$ columns of $A \cup B$ and the $t - 2p$ columns from Z used to cancel out the remaining bits to obtain a set of t columns of H^* that sum to zero. Setting elements of $\vec{y}$ to 1 if the position in $\vec{y}$ matches one of these columns will result in a $\vec{y}$ of weight t such that $H^*\vec{y} = 0$.

With this $\vec{y}$, which by definition is in the code defined by H^*, we have that one of $\vec{y}$ or $\vec{y}+\vec{r}$ is in the code defined by G'. As we have already seen, $\vec{y}$ has weight t and cannot be in this code, therefore $\vec{c} = \vec{y} + \vec{r}$ must be a codeword. This implies that $\vec{c}+\vec{y} = \vec{r}$, and $\vec{y}$ is within the error correcting capacity of the code. Therefore, the decryption of the code must result in this $\vec{y}$, and therefore $\vec{c} = G'\vec{m}$, where $\vec{m}$ is the plaintext. As we have already seen, identifying $\vec{m}$ from $\vec{c}$ is the process of solving a system of k linear equations.

Currently, the best algorithm for attacking a McEliece cryptosystem is a variant of the Stern attack algorithm, developed in 2008 [5]. The attack is not fundamentally different to the Stern attack algorithm, it is a series of five optimizations on the attack. The overview of those optimizations is as follows:

The first optimization is to reuse existing pivots; instead of attempting to reduce H^* for each new Z, the reduced matrices from previous Z are used, as a large number of the columns have only a single non-zero entry. The second optimization improves this further; by reusing a large number of columns from Z, the reduction to the identity matrix can be further optimized. This can, however, cause a problem; if too few columns are changed for each iteration of Z then the number of iterations required to identify $\vec{y}$ will increase.

The third optimization is to select multiple Z_i's at once instead of a single Z_0 for each X and Y, effectively skipping the row reduction for all but the first Z_i. The fourth optimization involves using $\pi(A)$ for all sets A that have already been calculated to compute $\pi(A)$ for new sets. Often

the calculations for a new $\pi(A)$ have already been done for a previous A. The final optimization is for pairs A and B such that $\pi(A) = \pi(B)$. When computing the sum of columns in $A \cup B$ the sum is terminated after the number of non-zero results exceeds $t - 2p$. Once the number of non-zero terms of the sum exceeds $t - 2p$, we know the sum will not produce our $\vec{y}$ and we know that any further calculations are a waste of effort.

The security of the system can be calculated by inverting the probability of a single iteration succeeding multiplied by the amount of computing time required to perform the iteration. For an iteration of the algorithm to find the error $\vec{e}$, we need a number of occurrences in the iteration. The probability that Z be invertible can largely be ignored as if that check fails, the later computations will not be performed. The first occurrence is that we need the columns of Z to be in the positions matching exactly $t-2p$ non-zero entries of e. This occurs with probability $\binom{w}{2p}\binom{n-w}{k-2p}/\binom{n}{k}$. The second occurrence is that the $2p$ remaining errors are evenly split between X and Y, such that each contains $2p$ errors. This occurs with probability $\binom{2p}{p}/2^{2p}$. The last occurrence is that the remaining $t-2p$ errors do not align with the columns of Z_0. This occurs with probability $\binom{n-k-t+2p}{l}/\binom{n-k}{l}$. The inverse of the product of these will tell us on average how many iterations will be required to break the cryptosystem [5].

This product will allow us to optimize the parameters p and l, identifying the pair such that the inverse product is minimized. This product, while accurate for Sterns algorithm as it was originally proposed, is not entirely accurate for the updated attack. Considering multiple Z_i's instead of a single Z_0 is one such example. Additionally, if the first occurrence fails, and

the number of changes between iterations of Z is small, it is likely that the first occurrence will continue to fail for several iterations. This is because the number of errors compared to n is small, and so, for the small number of columns switched between different Z, it is likely that the number of errors associated with those columns is 0.

The security of the McEliece cryptosystem against this attack was determined to be the following [5]:

n	t	security level
1744	35	84.88
2480	45	107.41
3408	67	147.94
4624	95	191.18
6960	119	266.94

The scaling between security level and n appears to be approximately linear; extrapolating the security level for $n = 8192$ will give approximately 310 bits of security.

What makes the McEliece cryptosystem quantum resistant is that it is considered secure against attacks using a quantum computer as well. While there are quantum attacks that will attack the cryptosystem, it has high enough security against those attacks to be considered safe. As both of the classical attacks described earlier on the McEliece cryptosystem, information set decoding and Stern's algorithm, are searches with very low success rates, Grover's algorithm is an obvious quantum attack to consider [27].

If an attack involving a classical computer required, on average, k iterations to successfully attack the system, a quantum computer running Grover's algorithm would only require $\sqrt{k}$ iterations. Because of this,

quantum attacks that make no further use of quantum computation beyond Grover's algorithm have a lower bound of halving the bit security of a cryptosystem. Both information set decoding and Stern's algorithm require significant calculations per iteration, and so this lower bound will not be reached.

Because the bit security of both parameter sets initial6960 and initial8192 is over 256, the quantum bit security level of the McEliece cryptosystem against these attacks is over 128, just by the lower bound of Grover's algorithm. As an example, information set decoding for initial 6960 required approximately 2^{265} attempts, including the two bits for invertibility, on a function that required approximately 2^{38} calculations. By use of Grover's algorithm, this can be reduced to $2^{132.5}$ iterations of 2^{38} calculations, or a security level of 170.

Other than Grover's algorithm, there do not appear to be any effective attacks against the McEliece cryptosystem [5]. Attempts have been made to develop one, notably the paper showing that the cryptosystem is resistant against Quantum Fourier Sampling attacks [14].

3.5 Implementation Attacks

Implementation attacks are an attack method that require information not normally obtainable from the cryptosystem. This extra information comes from the protocols or structure that is given to the cryptosystems when they are used for practical applications. As an example, an extra piece of information that could be used in an implementation attack would be the structure of the plaintext. If the plaintext is known to be of the form

$(username, padding, password, sessionkey)$ an attacker could use that information to form new attacks.

In addition to the attacks already described, the McEliece cryptosystem is vulnerable to a number of these implementation attacks. There is one such attack that works almost fundamentally on public key cryptosystems. This attack can be defended against using proper implementation; however, it gives us context for a number of other attacks. Suppose an attacker is eavesdropping on a conversation between a user and a web service. The user will likely at some point send his or her encrypted account information to the server, the eavesdropper will be able to see the encrypted plaintext. Obviously we do not want the attacker to just be able to regurgitate the same encrypted plaintext and gain access to the user's account. To counteract this, some other information will also be encoded into the message, such as a session key.

We have only considered attacks based on eavesdropping; there also exist attacks if an attacker can intercept and alter messages. One such attack is to intercept the public key from the server and instead send to the user a different public key that the attacker has generated. Any message will be intercepted by the attacker, decrypted and then encrypted with the other cryptosystem before being forwarded. This attack works regardless of the cryptosystem; however, this attack can be defended against by use of an SSL certificate. An SSL certificate is an additional message that can be used to verify that a given cryptosystem is associated with a given website. This functions by having a so called "Certificate Authority" generate a signature that any user can use to verify was generated with the given cryptosystem

and website.

SSL certificates are not trivial to obtain, while the price varies wildly, for major sites each certificate will be in the order of $50 per year. This tell us that websites generating a lot of public key cryptosystems are not practical, so cryptosystems should be expected to be reused. With this, we should assume that an attacker is able to message the server using the same public key cryptosystem as a user they have previously eavesdropped on.

Suppose, for example, an attacker eavesdropped on a user, and obtained the encrypted message the user sent to a server containing their login information. This attacker then contacted the server and the server prompts them with the same public key as the user they compromised, what is stopping the attacker from just sending the same message? In practice, the sent message would be decoded to the same plaintext sent by the compromised user, but the server will find fault with the content, typically a mismatched session key. If the server tells the eavesdropper that there was a session key mismatch, then this gives information to the eavesdropper that can be used to obtain the plaintext sent by the compromised user.

How this attack works is that instead of sending the eavesdropped message, send the message with two bit flips. This new ciphertext is not designed to let them into the account, the only output the attacker cares about is if the codeword is successfully decoded. If the codeword successfully decodes, then the new ciphertext is distance t from the nearest codeword. Excluding the case that the attacker is astronomically unlucky, this implies that exactly 1 of the two bits flipped had already been flipped

by the error vector $\vec{e}$. In practice, an attacker can identify $\vec{e}$ after testing at most $n - 2$ messages. As we have already seen, identifying the plaintext after determining $\vec{e}$ is trivial, and with this information we can conclude that an attacker can identify the plaintext associated with the eavesdropped ciphertext.

To defend against this attack, an attacker should not know if a codeword successfully decodes, which necessitates websites being vague on error messages. Depending on the nature of the website, it may be possible for an attacker to differentiate between error types without the server explicitly telling the attacker. One example is that different errors could take a different amount of time to get a response and could be differentiated that way. It should also be pointed out that these methods need not be 100% accurate; any method that is even somewhat accurate at differentiating these cases could identify an error vector given enough messages to the server.

Another implementation attack can solve a plaintext if it is reused with a different random error vector $\vec{e}$. In short, if the same plaintext is sent with two different error vectors $\vec{e_1}$ and $\vec{e_2}$, then an attacker can almost certainly obtain the plaintext with minimal effort. First, it is very easy to detect if a plaintext is repeated, the two ciphertexts should differ in at most $2t$ locations, the higher the better for the attacker. While it is possible for two close ciphertexts to be different plaintexts, we can approximate the odds of that occuring by calculating the odds a random vector is within $2t$ of our ciphertext. The odds of this can be calculated by the odds of a 50% binomial distribution with 6960 trials and less than 240 matches; this is

approximately 2^{-5461}.

We want $\vec{e_1}$ and $\vec{e_2}$ to differ in as many locations as possible, as this will account for as many of the error vector elements as possible. The odds that these vectors share no non-zero elements can be calculated as the number of possible $\vec{e_2}$ that do not share any non-zero elements with $\vec{e_1}$ divided by the total number of possible $\vec{e_2}$. This results in $\frac{\binom{n}{t}}{\binom{n-t}{t}}$, or 0.121 for initial6960 and 0.131 for initial8192 parameter sets.

The attack is a modification of the initial attack involving choosing k random elements in the vector and assuming $\vec{e}$ did not change them. With most of the errors accounted for we choose k random elements from the remaining elements of the vector. The more non-zero elements $\vec{e_1}$ and $\vec{e_2}$ share, the more errors will be unaccounted for and the less likely our random selection will have no errors. If, for example, only two errors are in the remaining vector elements, then the odds that this selection does not intersect either of the unidentified errors can be calculated as $\frac{\binom{n-2t}{k}}{\binom{n-2t+2}{k}}$. For our initial6960 and initial 8192 parameter sets this is approximately 0.04. This reduces the security level to the equivalent of solving $\frac{1}{0.04 \cdot 0.29}$ sets of k linear equations over k variables.

Moreover, the two plaintexts need not even match. If the difference between the plaintexts is known or can be reasonably inferred based on information an attacker can obtain, then an additional modification can be applied to identify both messages. With the two plaintexts $\vec{v_1}$ and $\vec{v_2}$, the two ciphertexts $\vec{c_1}$ and $\vec{c_2}$, the two errors $\vec{e_1}$ and $\vec{e_2}$, and known difference $\vec{v} = \vec{v_1} + \vec{v_2}$, we can apply the distributive property of matrix multiplication to calculate $\vec{c_1} + G\vec{v} = G\vec{v_1} + \vec{e_1} + G(\vec{v_1} + \vec{v_2}) = G\vec{v_2} + \vec{e_1}$. Pairing this

with $\vec{c_2} = G\vec{v_2} + \vec{e_2}$, we can apply the attack described above to identify the plaintext.

There is a method to defend against implementation attacks that rely on the plaintext. It is a variant of the method used by the NTRU cryptosystem, covered in Section 6. The decryption for McEliece identifies the error $\vec{e}$, and adds it to the ciphertext to then solve for the plaintext. To encrypt with this method we generate our error $\vec{e}$ as normal, but our ciphertext will not be $\vec{c} = G\vec{u} + \vec{e}$. Instead we take a known hash function $HASH$ that will map to an element in $\mathbb{Z}_2^k$ and calculate $\vec{u'} = \vec{u} + HASH(\vec{e})$. We then use this $\vec{u'}$ in place of our $\vec{u}$ for the encryption, so our ciphertext will be $c = G\vec{u'} + \vec{e}$. For the decryption we can recover $\vec{u'}$ as normal, but with the decryption we will also recover $\vec{e}$, which allows us to recover our $\vec{u}$ using $\vec{u} = \vec{u'} + HASH(\vec{e})$. This method only helps against implementation attacks and has no effect on the other types of attacks we previously mentioned.

There are variants of the above method that can be used to defend against these implementation attacks. Websites with good security will defend against these, such attacks should never work against something like an online bank. However, as these attacks recover the message plaintext; an attacker can learn a user's password. The issue is that users reuse usernames and passwords all the time [12]. If someone is able to use this attack on a website that has not incorporated something to defend against such attacks, an attacker could potentially learn username and password combinations for secure sites.

4 Classic McEliece NIST Submission

Despite its name, the Classic McEliece is not the same cryptosystem covered in the last section; it is a cryptosystem which, instead of using the generator matrix G to encrypt messages, uses instead the parity check matrix H. To differentiate between the systems the NIST submission will be referred to as the Classic McEliece cryptosystem and the cryptosystem covered in the last section will be referred to as the Initial McEliece Cryptosystem.

The Classic McEliece cryptosystem is an implementation of something called the Niederreiter variant of the Initial McEliece cryptosystem, developed by Harald Niederreiter in 1986 [38]. The Initial McEliece cryptosystem used an obfuscated generator matrix G' to encode a plaintext vector $\vec{a}$ with a random error $\vec{e}$. The Classic McEliece cryptosystem instead uses a manipulated parity check matrix H to encode the error $\vec{e}$, which is the plaintext for this system. This change drastically reduces the public and private key sizes while keeping the same level of security as the Initial system.

Because we are able to prove that the Classic McEliece cryptosystem is as secure as the Initial McEliece cryptosystem, it is a very notable contender in the Post-Quantum Cryptography Standardization competition [4]. The proof of this can be found in Section 4.4.

4.1 Implementation

As in the case of the initial McEliece Cryptosystem, we will start by generating a binary Goppa code with parity check matrix $\bar{H}$ as defined in

(1) and associated parameters n, m, k and t. We will row reduce our parity check matrix so that the first mt columns are the mt by mt identity matrix. More explicitly, row reduce to a matrix $\ddot{H}$ such that for $1 \leq i, j \leq mt$, $\ddot{H}_{ij} = 1$ for $i = j$ and $\ddot{H}_{ij} = 0$ for $i \neq j$. If H cannot be row reduced to this form then we will need to generate a new Goppa code and try again.

This could pose an issue if the probability of H not being able to be row reduced is very small. As we have seen previously, the probability that H can be row reduced this way is about 0.29. This is well within a reasonable success rate to generate codes until one works.

Our parity check matrix H will now look like $H = \begin{bmatrix} I_{mt} & T \end{bmatrix}$ with the identity in the first mt columns and a mt by k binary matrix in the last k columns. Our public key will be this matrix T.

The plaintext for this cryptosystem is a bit unusual; it will be a binary vector $\vec{e}$, of length n and Hamming weight t. We will use the public key T and concatenate this matrix with the mt by mt identity matrix to reconstruct H. We can then calculate our ciphertext as $\vec{c} = H\vec{e} \in \mathbb{Z}_2^{mt}$.

To decrypt the ciphertext $\vec{c} = \begin{bmatrix} c_1 & c_2 & \dots & c_{mt} \end{bmatrix}^T$ we will start by generating the vector $\vec{v} = \begin{bmatrix} c_1 & c_2 & \dots & c_{mt} & 0 & \dots & 0 \end{bmatrix}^T \in \mathbb{Z}_2^n$, by adding k zeros onto the end of the ciphertext. We then use the private key information for the Goppa code to perform syndrome decoding on $\vec{v}$, and recover $\vec{e}$, concluding the decryption. If the decoding algorithm for the Goppa code fails or if the weight of $\vec{e}$ is not t, we conclude that $\vec{c}$ was not a possible output for the encryption algorithm and return an error.

The reason this algorithm works is because the first mt columns of H are the identity matrix and all the non-zero entries of $\vec{v}$ are in the first mt entries. This tells us that $H\vec{v}$ will be the first mt entries of $\vec{v}$, or $\vec{c}$. We also know that $H\vec{e} = \vec{c}$, which when we add to $H\vec{v} = \vec{c}$ results in $H(\vec{v} + \vec{e}) = \vec{0}$. By definition of the parity check matrix, this tells us that $\vec{v}+\vec{e}$ is a codeword, which we will call $\vec{v_0}$. Isolating for $\vec{v}$, we obtain $\vec{v} = \vec{v_0} + \vec{e}$; where $\vec{v_0}$ is a codeword and $\vec{e}$ is a vector of small enough Hamming weight to be corrected by the code. Performing syndrome decoding on $\vec{v}$ should therefore return $\vec{e}$.

4.2 Code Efficiency

The parameters given in the Classic McEliece NIST submission are as follows:

NAME	m	n	t	k	security
mceliece6960119	13	6960	119	5413	267
mceliece8192128	13	8192	128	6528	>267

The security level for the system is the same as for the initial McEliece cyrptosystem. The documentation also specifies the function $f(x) = x^{13} + x^4 + x^3 + x + 1$ to be used in generating the field F_q. As of the time of writing this thesis, there do not appear to be any attacks that involve knowledge of $f(x)$, so this can largely be ignored.

Assuming that these parameters are standardised and will not need to be sent as part of the public key, then the public key will only depend on the content of our matrix T. Because T is a binary mt by k matrix, it can be encoded in $mt \cdot k$ bits.

Before we calculate the size of the private key, we have an interesting

property that will allow us to encode the private key more efficiently.

Lemma 1 *Over a field of characteristic 2, one of the generator matrices associated with the same code as* $H = \begin{bmatrix} I_{mt} & T \end{bmatrix}$ *is* $G = \begin{bmatrix} T \\ I_k \end{bmatrix}$.

First we note that the rank of both of these matrices is k and mt respectively, because of the identity submatrix in both G and H. Second, when we multiply the two matrices we obtain $HG = 2T = 0$, because we are in a field of characteristic 2. With this we can conclude that the rows of H must generate the nullspace of G, and that H is the parity check matrix associated with the generator matrix G. ■

This matrix is trivial to calculate, and so we can store T to encode both H and G without any loss in decryption speed.

For the private key, we need the public key T, which we already know can be encoded with $mt \cdot k$ bits, plus the Goppa code information. From the section on the initial McEliece cryptosystem, we know that this requires $m \cdot t + m \cdot n$ bits for our $g(x)$ and our ordered list of α_i elements, bringing our total for the private key up to $m \cdot t + m \cdot n + mt \cdot k$ bits.

The ciphertext for this code is a binary vector of length k and can be encoded using k bits. The plaintext on the other hand is not as easy to calculate. The plaintext is a binary vector of length n; however, only t of its entries are non-zero. This tells us the maximal amount of information we can encode is $log_2(\binom{n}{t})$ bits. This assumes a perfect encoding, which is not practical. The plaintext numbers listed below are maximal values, the actual values are likely to be less than these.

NAME	Public Key	Private Key	Plaintext	Ciphertext
mceliece6960119	0.998MB	1.009MB	0.105KB*	0.661KB
mceliece8192128	1.295MB	1.308MB	0.115KB*	0.797KB

A more realistic size for the plaintext might be $87B$ for mceliece6960119 $96B$ for mceliece8192128, as we can describe an encoding for those values. The encoding works by first encoding the data we want to send as a number in base $\lfloor \frac{n}{t} \rfloor$. This number will have to be limited to t digits and will therefore have a maximum of $\lfloor \frac{n}{t} \rfloor^t - 1$. For each digit of this number, we will have the plaintext be equivalent to that many zeroes modulo $\lfloor \frac{n}{t} \rfloor$, followed by a single 1. Once we run out of digits we pad the plaintext with zeroes until it is n bits. For example, the number 132 in base 4 with $n = 14$ could be encoded as 0100010010000, 0100000001001, or two other possibilities. The plaintext size of this encoding would be $t \cdot log_2(\lfloor \frac{n}{t} \rfloor)$.

It should be noted that the Private key information here includes the public key, so assuming that anyone who intends to use this system will also want to store the public key, the private key can be stored with only around 12KB extra storage.

Compared to the initial McEliece cryptosystem we see a very clear improvement in the size of the public and private keys. Key sizes of around 1MB are not ideal, but are reasonable for most modern applications. However, this system encodes information very inefficiently; thus, the user will not want to send large amounts of data using this code.

One method to use this code efficiently is to only use it to establish a shared private key, which can then be used for communications. One

possible candidate for private key communications is the AES256 private key cryptosystem. The protocol is simply to send an AES256 private key as a message after establishing the public key cryptosystem. This exchange requires a single message of 256 bits, or $32B$, which can be encoded in a single message. Thus, the data requirements for establishing an AES256 private key is the size of the public key plus the size of one ciphertext, assuming the plaintext is at least $32B$. Therefore, if the goal of this system is to share AES256 private keys, then the classic McEliece cryptosystem can establish a shared key in less than one quarter the data compared to the initial McEliece cryptosystem.

4.3 Security Comparison

We stated that the Classic McEliece cryptosystem is as secure as the Initial McEliece cryptosystem. To prove this we will show that any potential attack on one of the systems can be modified, with minimal overhead, to attack the other cryptosystem.

Theorem 3 *Suppose we have some algorithm capable of identifying the plaintext $\vec{e}$ of small Hamming weight associated with the Classic McEliece public key H and ciphertext $H\vec{e}$. There exists an algorithm requiring a similar amount of calculations capable of identifying the plaintext $\vec{u}$ associated with the Initial McEliece public key G' and ciphertext $G'\vec{u} + \vec{e}$.*

For proving this theorem, we are given the public key G' and ciphertext $G'\vec{u} + \vec{e}$, our goal is to calculate $\vec{u}$. With the public key, we are able to identify a parity check matrix H' associated with G'. We know that $G' =$

PGS, and if H was the parity check matrix for G then $H' = HP^{-1}$ as $HP^{-1}PGS = HGS = 0$. The matrix HP^{-1} is the permutation of columns of H, which, as H is a Goppa code, will just permute the α_i elements used to generate the code. But importantly is that H is still a parity check matrix for a Goppa code.

From here we attempt to row reduce H' to the identity matrix in the first mt columns, as we did when generating the Classic McEliece public key. This procedure might fail, but by our assumption, H' is of full rank, and therefore, there exists some set of mt columns that can be row reduced to the identity matrix. We can multiply H' by a permutation matrix P_0 to permute these columns to the first mt positions. We can then row reduce this matrix to identity in the first mt columns, and call this new matrix H''. An important note is that because row reduction does not change what code the parity check matrix defines, $H''P_0^{-1}$ is a parity check matrix associated with G'. With this, we have H'', a parity check matrix associated with $P_0^{-1}G'$, and we note that we can calculate $P_0^{-1}(G'\vec{u} + \vec{e}) = P_0^{-1}G'\vec{u} + P_0^{-1}\vec{e}$.

We now have a matrix H'', a parity check matrix for a Goppa code that has been row reduced to the identity matrix in the first mt columns. If we multiply this by $P_0^{-1}(G'\vec{u} + \vec{e})$ we obtain $(H''P_0^{-1})G'\vec{u} + H''(P_0^{-1}\vec{e})$, which by the property that $HG = 0$ we obtain $H''(P_0^{-1}\vec{e})$. We know that $\vec{e}$, and therefore $P_0^{-1}\vec{e}$, is an error of small enough weight to be corrected by the code. From here we can apply our supposed algorithm to identify $P_0^{-1}\vec{e}$ from these known values. Multiplying by P_0 will give us our $\vec{e}$ and, paired with our original ciphertext $G'\vec{u} + \vec{e}$, allow us to calculate $G'\vec{u}$. As we have seen before, identifying $\vec{u}$ from this is a simple case of linear algebra. ■

Theorem 4 *Suppose we have some algorithm capable of identifying the plaintext $\vec{u}$ associated with the Initial McEliece public key G' and ciphertext $G'\vec{u} + \vec{e}$, for $\vec{e}$ of small enough weight to be corrected by the code. There exists an algorithm requiring a similar amount of calculations capable of identifying the plaintext $\vec{e}$ associated with the Classic McEliece public key H and ciphertext $H\vec{e}$.*

We are given the public key H and ciphertext $H\vec{e}$, with $\vec{e}$ being of small enough Hamming weight to be corrected by the code. With H and $H\vec{e}$, identifying a solution $\vec{r}$ to the equation $H\vec{r} = H\vec{e}$ without the Hamming weight restriction is not difficult, it is another case of linear algebra. Additionally, we are able to identify a generator matrix G associated with H. Letting $P = I_n$ and $S = I_k$, we will apply our supposed algorithm to identify $\vec{u}$ from public key $G' = PGS = G$ and ciphertext $\vec{r}$.

With $\vec{r} = G\vec{u} + \vec{e'}$, we can multiply by H to obtain $H\vec{r} = HG\vec{u} + H\vec{e'}$, or $H\vec{e} = H\vec{e'}$. The weight of $\vec{e}$ and $\vec{e'}$ is small enough that if $H\vec{e} = H\vec{e'}$, we have that $\vec{e} = \vec{e'}$. Given $\vec{u}$ and G, we can calculate $G\vec{u}$; subtracting this from $G\vec{u} + \vec{e'}$ will give us our $\vec{e}$. ■

Therefore, being able to attack one cryptosystem would imply the ability to attack the other. Therefore the modified Stern attack algorithm will be the most effective attack against the Classic McEliece cryptosystem as well. By generating H^* from H as described, the attack will function without any changes. By the same logic as with the initial McEliece cryptosystem both mceliece6960119 and mceliece8192128 have over 256 bit security and are considered secure.

With this we have that the initial McEliece and the classic McEliece cryptosystems have the same security for the same parameters, but the classic McEliece cryptosystem has significantly reduced key sizes compared to the initial McEliece system. For the purposes of exchanging short messages the classic system is more efficient, but the extremely inefficient encoding of plaintext means the initial system is better for sending larger amount of data.

The implementation attack initial McEliece cryptosystem involving sending messages with two bit flips can be modified to work for the Classic McEliece cryptosystem. Instead of flipping two bits of a known ciphertext, an attacker will instead flip two bits of the plaintext. While the attacker does not know what the plaintext is, the attacker does know that the ciphertext associated with that bit flipped plaintext will be the ciphertext added to two columns of H. This works because $H\vec{e} + H\vec{e_0} = H(\vec{e} + \vec{e_0})$, which will allow an attacker to check pieces of the plaintext. As with the Initial McEliece attack, with knowledge whether $n - 2$ ciphertexts successfully decode or not, an attacker would be able to identify the plaintext.

The hash function method we described for the Initial McEliece cryptosystem cannot be used for the Classic McEliece cryptosystem as the encryption method does not have a random vector encoded into the ciphertext. Implementation attacks will need to be defended against using a different method or will need to use some part of the plaintext to implement the hash function method.

5 McNie NIST Submission

The McNie cryptosystem is another variant of the McEliece cryptosystem, but is different in that it is a McEliece variant that does not use binary Goppa codes. In fact, the code doesn't even use the same metric for determining what is a small vector. Other systems use the Hamming weight, but the McNie cryptosystem uses something we will call the rank metric.

This code as it was originally submitted was extremely efficient in terms of data usage, even being comparable to our currently implemented systems. However, the McNie cryptosystem as it was initially submitted was broken; however, changes were made to fix these breaks. The McNie cryptosystem was one of the cryptosystems being analysed for the Post-Quantum Cryptography Standardization competition [23]; however, it was not selected to move onto the second round of the competition [40].

5.1 Initial Implementation

The documentation for this cryptosystem [23] uses the convention of right multiplication of row rectors by matrices. Since we have have that $\vec{a}^T \cdot B^T = (B \cdot \vec{a})^T$, we will be maintaining consistency across this thesis by instead left multiplying column vectors by matrices.

The McNie cryptosystem also uses a different definition of weight compared to the other cryptosystems we are examining. To avoid confusion, we will instead use the term "rank" of a vector which we will define as follows: for an arbitrary base field F_q and vector

$\vec{v} = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} \in F_{q^m}^n$, $rank(\vec{v}) = dim(span_{F_q}\{v_i | 1 \leq i \leq n\}$. In words, the rank of a vector is the dimension of the span of its component elements over some base field F_q. More conceptually, the rank of a vector is the number of linearly independent entries it has.

The first things we need to produce are our parameters, starting with a prime q and positive integer m to generate the field F_{q^m} over which our calculations will be done and all of our matrix entries will be elements of. We can generate a parity check matrix H for an error correcting code over F_{q^m} capable of correcting errors of maximum rank r. The size of this parity check matrix will be $n \times (n-k)$. Our last parameter will be l such that $n-k < l < n$. With these parameters we will generate the last components of our code; an n by n permutation matrix P, an invertible $n-k$ by $n-k$ matrix S and a random n by l matrix G' such that G'^T has rank l.

With these matrices, we can generate our public key G', $F = SHP^{-1}G'$ and r. The private key for this cryptosystem will be the matrices P, S^{-1} and H along with the decoding algorithm for the error code associated with the matrix H. For any error vector $\vec{e}$ of rank at most r, This decoding algorithm will identify $\vec{e}$ given $H\vec{e}$.

For encryption we will encode our message into a column vector of length l over F_{q^m} to get $\vec{v} = \begin{bmatrix} v_1 & v_2 & \dots & v_l \end{bmatrix}^T \in F_{q^m}^l$. We will also generate an error vector $\vec{e} \in F_{q^m}^n$ of rank at most r. We then calculate vectors $\vec{c_1} = G'\vec{v} + \vec{e}$ of length n and $\vec{c_2} = F\vec{v}$ of length $n-k$, which give us our ciphertext as the pair $\vec{c} = (\vec{c_1}, \vec{c_2})$.

For decryption we will start by calculating

$$\begin{aligned}\vec{s'} &= HP^{-1}\vec{c_1} - S^{-1}\vec{c_2} \\ &= HP^{-1}(G'\vec{v} + \vec{e}) - S^{-1}SHP^{-1}G'\vec{v} \\ &= HP^{-1}\vec{e}\end{aligned}$$

Now $P^{-1}\vec{e} \in F_{q^m}^l$ is a vector of rank at most r, meaning that we can decode it from $HP^{-1}\vec{e}$. Applying the decoding algorithm we will have $P^{-1}\vec{e}$, which we can multiply by P to get $\vec{e}$. With this, we can subtract $\vec{e}$ from our $\vec{c_1}$ to get $\vec{c_1} - \vec{e} = G'\vec{v}$. Since we now have G' and $G'\vec{v}$, we can use linear algebra to recover $\vec{v}$ and complete the decryption.

5.2 Implementation using Low Rank Parity Check codes

By using Low Rank Parity Check codes, or LRPC codes for short, we are able to create an implementation of McNie that uses properties of a class of matrices known as circulant matrices to drastically reduce key sizes. Circulant matrices are square matrices with each successive row of a matrix being equal to the one before it cyclicly shifted one to the right. This implementation is a degree m field over the base field $F_q = \mathbb{Z}_2$, so all matrix components will be elements of F_{2^m}. Let H_1, H_2, H_3, G_1 and G_2 be $\frac{n}{3}$ by $\frac{n}{3}$ circulant matrices, we will generate our matrices as follows:

$$H = \begin{bmatrix} H_1 & H_2 & H_3 \end{bmatrix} \quad G' = \begin{bmatrix} I_{\frac{n}{3}} & 0_{\frac{n}{3}} \\ 0_{\frac{n}{3}} & I_{\frac{n}{3}} \\ G_1 & G_2 \end{bmatrix} \quad S = (H_1 + H_3G_1)^{-1} \quad P = I_n$$

For this implementation, P is not used as it would ruin one of the properties used for encoding keys. Evaluating F from this we have that:

$$F = \left[I_{\frac{n}{3}} \quad (H_1 + H_3 G_1)^{-1}(H_2 + H_3 G_2) \right]$$

Circulant matrices are used because they can be identified uniquely with only the first line, which is the primary method the size of the keys is kept low. Circulant matrices also have the property that addition, multiplication, transpose and inversion of circulant matrices will be a circulant matrix [13].

There is another requirement on our H_i's. If we write out the first row of all three of the matrices as a vector of length n, this vector has to have a rank of d, for some d that will be specified in the cryptosystem parameters.

It remains to show that H is the parity check matrix for some rank metric error correcting code and to define the decoding algorithm for that code. Due to the length of the decoding algorithm, details for this proof and explanation of the algorithm will be covered in appendix B [22]. The result we obtain is that H is a parity check matrix for a rank metric error correcting code capable of correcting almost all errors up to rank r. This code does have a small chance of decoding failure, which we will quantify later.

Once we have our H, S, G', F and decoding algorithm for the code associated with H, we can generate a McNie cryptosystem based on LRPC codes. The NIST submission does specifically include a protocol for how messages should be structured using LRPC codes. This protocol is to use the first four bytes of the plaintext to specify the length of the message and to pad the end bits to get $\vec{v} \in F_{q^m}^l$ where $\vec{v}$ is the concatenation of the first four bytes, the message and the pad bits.

The use of circulant matrices drastically reduced the size of the keys for

the McNie cryptosystem. When encoding the key, only 1 row of G_1, G_2 and the submatrix of F are required for the public key. Encoding the full matrix under similar parameters would require a significant amount more data, roughly n times more. The private key had a similar size reduction; however, the plaintext and ciphertext lengths are unchanged.

The NIST submission also had an implementation for 4-quasi-cyclic codes which at a similar security level to the 3-quasi-cyclic codes had a roughly 20% decrease in public key size in exchange for doubling the private key size and slightly reducing the efficiency by which messages are encoded. While there exist ways to implement s-quasi-cyclic codes for $s \geq 5$ these codes had larger key sizes for no noticeable benefits [23].

The matrices for 4-quasi-cyclic codes are as follows:

$$H = \begin{bmatrix} H_1 & H_2 & H_3 & H_4 \\ H_5 & H_6 & H_7 & H_8 \end{bmatrix} \quad G' = \begin{bmatrix} I_{\frac{n}{3}} & 0_{\frac{n}{3}} & 0_{\frac{n}{3}} \\ 0_{\frac{n}{3}} & I_{\frac{n}{3}} & 0_{\frac{n}{3}} \\ 0_{\frac{n}{3}} & 0_{\frac{n}{3}} & I_{\frac{n}{3}} \\ G_1 & G_2 & G_3 \end{bmatrix} \quad S = \begin{bmatrix} S_1 & S_2 \\ S_3 & S_4 \end{bmatrix} \quad P = I_n$$

Similarly to the 3-quasi-cyclic codes, the H_i's, G_i's and S_i's are circulant matrices and H is of low rank, forming a LRPC code.

5.3 Initial Code Efficiency

When we were introducing this cryptosystem we mentioned that the system as it was initially proposed was broken; the parameters specified below are as the cryptosystem was originally proposed. We are interested

in these numbers because they give context to how large the parameters were, especially relative to each other, when looking at the various attacks developed against McNie. Parameters given in the NIST submission for 3-quasi-cyclic LRPC, their associated data usages and initially claimed security levels are as follows:

n	k	l	r	m	Public	Private	plaintext	ciphertext	claimed security
93	62	62	5	37	431B	194B	314B	579B	128
105	70	70	5	37	486B	218B	358B	653B	128
111	74	74	7	41	569B	247B	454B	764B	192
123	82	82	7	41	631B	274B	505B	846B	192
111	74	74	7	59	819B	337B	636B	1097B	256
141	94	94	9	47	829B	348B	699B	1110B	256

For all posted parameters sets, the values of $d = 3$ and $q = 2$ were used.

As we can clearly see these are extremely good numbers compared to the two McEliece systems we have examined. If this system had been secure, it would have been a very good choice of cryptosystem due to its efficiency. The public key was less than a kilobyte for claimed 2^{256} security and the ratio of plaintext to ciphertext was reasonable as well. In fact, these were arguably not even the most efficient implementation of LRPC codes; the 4-quasi-cyclic LPRC public keys were around 75% the size of equivalent security 3-quasi-cyclic LPRC keys. This was however, in exchange of most of the other listed parameters [23].

There are two problems with this system compared to the other McEliece systems we have examined. First, there is a possibility that the decoding algorithm will fail; the odds of this happening for our parameters is somewhere between 2^{-21} and 2^{-16} [23]. While not likely for any

individual message, this is high enough that it can be expected to happen on a regular basis if the system sees heavy usage. Second, the security numbers are no longer accurate and in fact the 2^{256} security systems are not even 2^{128} secure [32].

5.4 Rank Metric and Hamming Weight Comparison

McNie uses a different metric for determining the weight of a vector, and there is a very good reason for this. Error correcting codes have a limit to the amount of errors they can correct for a given pair $k < n$, this limit is that a single element of $\mathbb{Z}_{q^m}^n$ cannot decode to two different plaintexts. To avoid this, we need that the total number of vectors divided by the the total number of codewords multiplied should exceed the number of error vectors the code is capable of correcting. For a Hamming weight error correcting code, this can be written as the inequality $\sum_{j=0}^{E} \binom{n}{j}(q^m-1)^j \leq \frac{(q^m)^n}{(q^m)^k} = q^{m\cdot(n-k)}$, where E is the maximum possible number of errors in the code. For a rank metric error correcting code the inequality will instead be $q^{R\cdot n} \leq q^{m\cdot(n-k)}$, where R is the maximum rank of error the code is capable of correcting. What we note is that E does not scale with m, but D does.

One of the major advantages to using the Rank Metric is that we can increase the parameter m to increase security, instead of having to increase n and k. This is not the case for Hamming weight error correcting code based cryptosystems, the probability of an iteration of information set decoding succeeding is $\prod_{i=1}^{k} \frac{n-t-i}{n-i}$. Solving the system of linear equations requires at most $(k \cdot m)^3$ calculations for $q = 2$, while not insignificant, is

small compared to the previous product. Moreover, E does not scale with m, and so increasing m will have a relatively insignificant effect on the security of the cryptosystem. What we will see in Section 5.5 is that the security level of rank metric codes scales linearly with m, in addition to the scaling by R, which we have already seen is bound above by a function of m.

The advantage of increasing m instead of k and n is that, over F_{q^m}, the generator and parity check matrices can be encoded with $m \cdot k \cdot n$ and $m \cdot n \cdot (n-k)$ bits respectively. If the ratio between k and n is kept constant, a public key would scale linearly with m and quadratically with n and k. This, in addition to the existence of LRPC codes that make use of circulant matrices, would in theory allow for a cryptosystem with exceptionally low key sizes while maintaining a good ratio between plaintext and ciphertext. In practice, there exist other attacks for these cryptosystems, and developing such a cryptosystem that is not vulnerable to those attacks is what the creators of McNie set out to accomplish.

The attacks on McEliece relied heavily on the Hamming weight of the error in the code. If we try some of the McEliece attacks on McNie, we conclude that none of them will work because it is entirely possible that all of the entries of $G\vec{a}$ have been changed by $\vec{e}$, and that inverting sub-matrices of G does not appear to give a useful answer. With this, we will need to describe new attacks, specifically for use against the McNie cryptosystem and the rank metric.

5.5 Attacking McNie

Fundamentally, McNie is based on a different problem compared to McEliece, so it seems reasonable that we need an entirely different attack. This problem is referred to as the Rank Syndrome Decoding problem, abbreviated RSD [20]. Suppose that H is a full rank $n-k$ by n matrix over F_{q^m} and we are given $H\vec{v} = \vec{c}$ then the RSD is to identify a $\vec{v}$ such that $rank(\vec{v}) \leq r$. It should be noted that if we were instead given $G\vec{a} + \vec{v} = \vec{c'}$, we could identify an H such that $HG = 0$ and reduce the problem to $H\vec{v} = \vec{c} = H\vec{c'}$.

For our first attack [21], we will be considering subspaces in F_{q^m} of various dimensions over F_q. The subspace E will be the subspace of rank r that contains all elements of the error vector for a given message $\vec{c} = H\vec{e}$. The general overview of how the attack works is that it generates subspaces E' of rank $r' \geq r$, then solves for $\vec{e}$ assuming $E \subseteq E'$.

First, we need to determine how big our r' should be: too large and we won't be able to identify $\vec{e}$ uniquely, too small and the probability that $E \subseteq E'$ drops significantly. If we rewrite elements of $\vec{e}$ as the sum of the r' basis vectors of E', then we will have $n \cdot r'$ unknowns in F_q. From the equation $\vec{c} = H\vec{e}$ we have $n-k$ linear equations over F_{q^m}, which is equivalent to $(n-k) \cdot m$ linear equations over F_q. With $n \cdot r'$ unknowns and $(n-k) \cdot m$ linear equations this should be solvable if $n \cdot r' \leq (n-k) \cdot m$. Isolating for r' we get that $r' \leq \lfloor \frac{(n-k) \cdot m}{n} \rfloor$, or $r' = \lfloor \frac{(n-k) \cdot m}{n} \rfloor$ if we try to maximise r'.

For a randomly chosen subspace E' of rank r', the odds that any element of F_{q^m} is also in E' is $\frac{q^{r'}}{q^m} = q^{r'-m}$. Therefore, the odds that r

linearly independent elements of F_{q^m} are in E' is roughly $q^{(r'-m)r}$. With our maximised r', we have that this attack will require $q^{(r'-m)r} = q^{r \cdot \lfloor \frac{km}{n} \rfloor}$ iterations, with each iteration involves solving a system of linear equations.

With a baseline on attacking the RSD problem, we can consider attacks on McNie. Looking at $\vec{c_2} = F\vec{v}$, we can use the properties of matrix multiplication to split up $F\vec{v}$. We split F into its first $n-k$ columns which we will call F_1' and its remaining $l-(n-k)$ columns which we will call F_2'. We will also split $\vec{v}$ into its first $n-k$ entries $\vec{v_1'}$ and its remaining $l-(n-k)$ entries $\vec{v_2'}$. What we have is that $\vec{c_2} = F\vec{v} = F_1'\vec{v_1'} + F_2'\vec{v_2'}$, which after some rearranging we get $\vec{v_1'} = F_1'^{-1}\vec{c_2} - F_1'^{-1} \cdot F_2'\vec{v_2'}$.

Doing a similar manipulation for the first equation $\vec{c_1} = G'\vec{v} + \vec{e}$ and a similar split as above for G' we get that $\vec{c_1} = G_1'\vec{v_1'} + G_2'\vec{v_2'} + \vec{e}$. Substituting our equation for $\vec{v_1'}$ from above we obtain $\vec{c_1} = \vec{e} + G_1' \cdot (F_1^{-1} \cdot F_2'\vec{v_2'} - F_1'^{-1} \cdot \vec{c_2}) - G_2'\vec{v_2'}$. Because of the structure of F and G' from the LRPC code, we have that F_1' is the identity matrix which allows us to simplify the formula to $\vec{c_1} = \vec{e} + G_1' \cdot F_2\vec{v_2'} - G_1'\vec{c_2} - G_2'\vec{v_2'} = \vec{e} + (G_1' \cdot F_2 - G_2')\vec{v_2'} - G_1'\vec{c_2}$. Finally, this can be rewritten to $\vec{c_1} + G_1'\vec{c_2} = \vec{e} + (G_1' \cdot F_2 - G_2')\vec{v_2'}$. If there existed more than one pair $\vec{e}$ of rank at most r and $\vec{v_2'}$ such that this equation held, that would imply that this ciphertext pair could have been generated by more than one plaintext, which we know to be impossible because of the existence of a decoding algorithm for the cryptosystem. Moreover, for every $\vec{v_2'}$ and $\vec{e}$, there will exist a $\vec{c_1} + G_1'\vec{c_2}$ such that this equation holds.

This implies that $(G_1' \cdot F_2 - G_2')$ is a generator matrix for a rank metric code, which implies $\vec{c_1} + G_1'\vec{c_2} = \vec{e} + (G_1' \cdot F_2 - G_2')\vec{v_2'}$ is of the form $\vec{a} = \bar{G}\vec{m} + \vec{e'}$ where $\vec{a}$ and $\bar{G}$ are known. This is exactly the rank syndrome decoding

problem, and applying an algorithm for that would solve this system, and allow us to identify $\vec{v_2'}$. With this $\vec{v_2'}$, we can recover the other half of the plaintext using $\vec{v_1'} = F_1'^{-1}\vec{c_2} - F_1'^{-1} \cdot F_2'\vec{v_2'}$.

More recently, there was an attack posted [33] that, even after the parameter update following the previous attack, makes the cryptosystem insecure. The attack will recover a private key for a given public key by restating the equation $F = SHG'$ as $S^{-1}F = HG'$ and attempting to solve for S and H. This attack works for all s-Quasi-Cyclic LRPC codes but the details will only be given for the case of 3-Quasi-Cyclic LRPC Codes as the biggest difference is how messy the notation becomes.

The reason we restated the equation as $S^{-1}F = HG'$ is that now we have a system of $l \cdot (n-k)$ linear equations, one for each entry in the resulting matrix. We also have $(n-k) \cdot (n-k)$ unknowns from S and $(n-k) \cdot n$ unknowns from H which results in $(n-k) \cdot (2n-k)$ unknowns for our equation. Because $l \cdot (n-k)$ is much less than $(n-k) \cdot (2n-k)$ we can not directly solve this. However, our H and S have a structure that was used to generate the code. From earlier we had:

$$H = \begin{bmatrix} H_1 & H_2 & H_3 \end{bmatrix} \quad G' = \begin{bmatrix} I_{\frac{n}{3}} & 0_{\frac{n}{3}} \\ 0_{\frac{n}{3}} & I_{\frac{n}{3}} \\ G_1 & G_2 \end{bmatrix} \quad S = (H_1 + H_3G_1)^{-1}$$

$$F = \begin{bmatrix} I_{\frac{n}{3}} & (H_1 + H_3G_1)^{-1}(H_2 + H_3G_2) \end{bmatrix} = \begin{bmatrix} I_{\frac{n}{3}} & F' \end{bmatrix}$$

where F', G_i and H_i were all circulant $\frac{n}{3}$ by $\frac{n}{3}$ matrices. Using $S^{-1} = H_1 + H_3G_1$ and multiplying out $S^{-1}F$ and HG' we obtain:

$$HG' = \begin{bmatrix} H_1 + H_3G_1 & H_2 + H_3G_2 \end{bmatrix}$$
$$S^{-1}F = \begin{bmatrix} H_1 + H_3G_1 & (H_1 + H_3G_1)F' \end{bmatrix}$$

Now we have that $(H_1 + H_3G_1)F' = H_2 + H_3G_2$ with G_1, G_2 and F' known. Manipulating things to one side of the equation and factoring out the H_i terms we can obtain:

$$0_{\frac{n}{3} \cdot \frac{n}{3}} = H_1F' - H_2 + H_3(G_1F' - G_2) = H \begin{bmatrix} F' \\ -I_{\frac{n}{3}} \\ G_1F' - G_2 \end{bmatrix} \tag{3}$$

There was an additional requirement on our H_i's, that the first row of all three concatenated would form a vector of rank d. Let $\vec{h_i}$ be the first row of H_i, take the transpose of (3) and consider the first column of this transposed equation:

$$0_{\frac{n}{3}} = \begin{bmatrix} F' & -I_{\frac{n}{3}} & G_1F' - G_2 \end{bmatrix} \begin{bmatrix} \vec{h_1} & \vec{h_2} & \vec{h_3} \end{bmatrix}^T$$

This reduces the problem to the Rank Syndrome Decoding problem, the exact problem we had already reduced the cryptosystem to. The difference here is that the rank of our vector is d instead of r. For the proposed parameter sets, r was always greater than d and in fact, the parameter changes caused by the last attack were defended against largely by increasing r, which doesn't help against this attack. This dropped the security levels of even the 256 bit secure parameters to worse than 128 bit security. (Details can be seen in Section 5.6).

In addition to this attack, one of the major issues with the McNie cryptosystem using LRPC codes is that the chance of decoding error is

large enough that decoding errors are likely to be seen over the cryptosystems lifetime. A set of changes to McNie were published with the recent attack which removes this chance of decoding error and makes the cryptosystem secure against all currently known attacks [32]. The first change was a switch of error correcting code, from LRPC to Gabidulin codes [16]. Gabidulin codes are rank metric error correcting codes generated by a set of n linearly independent elements $\{g_1, g_2, \ldots, g_n\}$ of F_{q^m}.

$$G = \begin{bmatrix} g_1 & g_2 & \cdots & g_n \\ g_1^{q^1} & g_2^{q^1} & \cdots & g_n^{q^1} \\ \vdots & \vdots & \ddots & \vdots \\ g_1^{q^{k-1}} & g_2^{q^{k-1}} & \cdots & g_n^{q^{k-1}} \end{bmatrix}$$

Notably, this is the generator matrix and not the parity check matrix like what we had for Goppa codes. The important factor of Gabidulin codes is that there is no chance of a decoding error, unlike the LRPC codes. The second change came from splitting the parity check matrix H and applying a second error to $\vec{c_2}$ to prevent an attacker using it to get additional information about the plaintext.

For parameters l, n, q, m and k a parity check matrix H for a Gabidulin code over F_{q^m} is generated using parameters $2n - k$ and n along with a decryption algorithm Φ_H. We will split H into the first k columns H_1 and the last $n - k$ columns H_2 so $H = \begin{bmatrix} H_1 & H_2 \end{bmatrix}$. We will also require that H_2 be invertible, if it is not we need to generate a new H. An optimization on this code is to row reduce H to the form $\begin{bmatrix} H_1 & I_{n-k} \end{bmatrix}$ to reduce the private key size.

We will also generate a partially circulant l by n matrix G'^T and take its transpose to get G'. A partially circulant matrix has the same definition as a circulant matrix with the exception that the matrix need not be square. The last matrix we need to generate is an n by n permutation matrix P, with these we can calculate $F = H_2^{-1}H_1P^{-1}G'$. The public key will be the matrix F and the first row of G', the private key will be the matrices P and H as well as the decryption algorithm Φ_H.

To encrypt a message $\vec{u} \in F_{q^m}^l$, we generate a random error vector $\vec{e} \in F_{q^m}^{2n-k}$ with rank at most $r = \lfloor \frac{n-k}{2} \rfloor$. We then generate two error vectors by splitting this $\vec{e}$ into the first n elements $\vec{e_1}$, and the last $n-k$ elements $\vec{e_2}$. We compute $\vec{c_1} = G'\vec{u} + \vec{e_1}$ and $\vec{c_2} = F\vec{u} + \vec{e_2}$ and our ciphertext becomes $(\vec{c_1}, \vec{c_2}) \in F_{q^m}^{2n-k}$.

To decrypt a message we compute $H_1P^{-1}\vec{c_1} - H_2\vec{c_2} = H_1P^{-1}G'\vec{u} + H_1P^{-1}\vec{e_1} - H_2H_2^{-1}H_1P^{-1}G'\vec{u} - H_2\vec{e_2} = H_1P^{-1}\vec{e_1} - H_2\vec{e_2} = H[P^{-1}\vec{e_1} \quad \vec{e_2}]^T$. Because permuting elements does not change the rank of a vector, this $\left[P^{-1}\vec{e_1} \quad \vec{e_2}\right]$ has rank r which can be corrected using Φ_H. Now with $P^{-1}\vec{e_1}$ we can multiply by P to get $\vec{e_1}$ and get $G'\vec{u} = \vec{c_1} - \vec{e_1}$. As we have done previously, we use linear algebra to recover our message $\vec{u}$.

5.6 Updated Code Efficiency

After the submitters of the McNie cryptosystem were informed of the break that split the plaintext into $\vec{v_1'}$ and $\vec{v_2'}$, the parameters for the cyrptosystem were updated to the following. Prop Sec is short for Proposed Security and is what the security level was calculated as before the Key Recovery

Attack, that is the attack that identified H using rank syndrome decoding, was published. KRA Sec is short for Key Recovery Attack Security and is the security level of the parameters against the key recovery attack.

NAME	n	k	l	d	r	q	m	Prop Sec	KRA Sec
McNie3QC120	120	80	80	3	5	2	53	128	87
McNie3QC138	138	92	92	3	10	2	67	192	103
McNie3QC156	156	104	104	3	12	2	71	256	107

NAME	n	k	l	d	r	q	m	Prop Sec	KRA Sec
McNie4QC92	92	46	69	3	10	2	59	128	107
McNie4QC112	112	56	84	3	13	2	67	192	119
McNie4QC128	128	64	96	3	16	2	73	256	127

With the Key Recovery Attack the change of the system to the use of Gabidulin codes was proposed with the following parameters [33]:

NAME	n	k	l	r	q	m
McNie2-Gabidulin128	24	12	22	6	2	41
McNie2-Gabidulin192	32	16	24	8	2	53
McNie2-Gabidulin256	36	18	29	9	2	59

To calculate the public key size, we remember that the public key is the first row of the circulant l by n matrix G' and the full l by $n-k$ matrix F. All entries are in the field F_{2^m} and require m bits to represent. There are 8 bits in a byte, so our formula of how many bytes are required to send the public key becomes $\frac{m(n+l(n-k))}{8}$.

The private key consists of the n by n permutation matrix P, the parity check matrix H and the decryption algorithm Φ_H. As we have seen in Section 3.2, permutation matrix requires at least $\lceil log_2(n!) \rceil$ bits to encode. If we used the optimization, we can encode H using only H_1 which is a $n-k$

by n matrix using $\frac{n\cdot(n-k)\cdot m}{8}$ bytes. It is also possible to solve for H_1 using $F = H_2^{-1}H_1P^{-1}G' = H_1P^{-1}G'$, but this would require a large amount of computation to the point of being less practical than storing H_1.

The plaintext is a vector in $F_{q^m}^{l}$ and therefore takes $\frac{l\cdot m}{8}$ bytes to encode. The ciphertext is a vector in $F_{q^m}^{2n-k}$ and therefore takes $\frac{(2n-k)\cdot m}{8}$ bytes to encode.

The following is a table from the paper proposing the update [32] which describes the code efficiency for different security levels.

NAME	Public Key	Private Key	plaintext	ciphertext	security
Gabidulin128	1.476KB	0.308KB	0.113KB	0.185KB	128
Gabidulin192	2.756KB	0.530KB	0.159KB	0.318KB	192
Gabidulin256	4.116KB	0.664KB	0.214KB	0.399KB	256

6 NTRUEncrypt NIST Submission

NTRUEncrypt is the only cryptosystem we will be examining that is not a variant of McEliece. Instead of being an obfuscated error correcting code, NTRU is a lattice-based cryptosystem. Calculations are done over a polynomial ring, where functions with small coefficients are used to make ring reductions predictable.

NTRU is a series of cryptosystems originally developed more than 20 years ago [30]. Since then, NTRU has been through many iterations, each iteration improving the security or the efficiency of the system. NTRUEncrypt is one of the most recent iterations that was submitted for post-quantum standardization [8]. NTRUEncrypt has both a relatively long history and impressively small key sizes; because of these features it is one of the top contenders in the NIST Post-Quantum Cryptography Standardization competition.

6.1 Implementation

Before we look at the definition of this scheme, it needs to be made clear that the submission appears to have a typo in it. With this typo the system does not function for the given parameters, and it is not clear what the correct implementation is. After the author contacted Jeffrey Hoffstein, one of the submitters of this cryptosystem, he was given gave two possible corrections to the implementation that would fix this issue. Because these corrections are fairly minor, and the parameters are important in seeing why this system fails, we will first cover the system and its parameters as they were originally

submitted. We will then cover the changes suggested by Hoffstein as well as one other possible change that was taken from a previous iteration of NTRUEncrypt.

The NTRUEncrypt cryptosystem is done largely over the polynomial ring $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ for some given positive integers q and n. An important note is that this is a ring, not a field and the inverse of elements need not exist. An alternative polynomial ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ is sometimes used instead, this could be a field depending on values of q and n but rarely is.

When dealing with elements of $\mathbb{Z}_q$, instead of using the standard representation of elements $(0, 1, \ldots, q-1)$, we will have the elements represented as the integers between $\lceil \frac{-q+1}{2} \rceil$ and $\lfloor \frac{q}{2} \rfloor$, including both endpoints. We will assume this notation for all calculations in this section unless stated otherwise.

As with the other cryptosystems, we need to start with our parameters. We have q and p positive integers strictly greater than 2, such that $gcd(p, q) = 1$ and q much larger than p. We also need n which will finish our definitions of $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ and $\mathbb{Z}_p[x]/\langle x^n - 1\rangle$ over which we will be doing our calculations. Furthermore, we will have a positive integer d with $d < \frac{q}{2}$ that will determine the number of non-zero entries of our private key functions.

We need to define T_N and $T_N(a, b)$. T_N is the set of polynomials of degree at most $N - 1$ with all coefficients in the set $\{-1, 0, 1\}$. $T_N(a, b)$ is the set of all elements of T_N with exactly a ones and b minus ones as the coefficients.

We will require a hash function, which we will denote $HASH(y) = z$,

where y is a binary string of any length and z is a binary string of length $|HASH|$. We will need a mapping $\Upsilon(z)$: $\mathbb{Z}_2^{|HASH|} \rightarrow T_n$, the outputs of which we will treat as elements of $\mathbb{Z}_q[x]/\langle x^n-1\rangle$. Last, we will need a method to encode our message, with padding information msg, into $\mathbb{Z}_p[x]/\langle x^n-1\rangle$, which we will call $\kappa(msg)$. All of these will need to be shared between the encoder and decoder, but these functions can be standardised and omitted from the public and private keys.

We will generate our private key functions $f(x)$ and $g(x)$ from the set $T_n(d+1,d)$. Generating these polynomials can be achieved using a modification of the mapping $\Upsilon(z)$; however, as this function need not be used by anyone else, this can be any convenient method to generate two random functions from $T_n(d+1,d)$. Our $f(x)$ function will also have the additional requirement that $p \cdot f(x)+1$ is invertible over $\mathbb{Z}_q[x]/\langle x^n-1\rangle$. If this check fails we will need to generate a new $f(x)$ and try again. With this we can then calculate our public key $h(x) = \frac{g(x)}{p \cdot f(x)+1} \in \mathbb{Z}_q[x]/\langle x^n-1\rangle$ and our associated private key $p \cdot f(x)+1$.

To encrypt the message msg into this system we need to encode it using the mapping $m = \kappa(msg)$ which we concatenate with $h(x)$ and calculate $r(x) = \Upsilon(HASH(m||h(x)))$. Then we calculate $t(x) = p \cdot r(x) \cdot h(x) \in \mathbb{Z}_q[x]/\langle x^n-1\rangle$ and use that to calculate $m_{mask} = \Upsilon(HASH(t(x)))$. With this we calculate $m' = m - m_{mask} \in \mathbb{Z}_p[x]/\langle x^n-1\rangle$ and our ciphertext $c = t(x) + m' \in \mathbb{Z}_q[x]/\langle x^n-1\rangle$.

To decode the message with the ciphertext $c = p \cdot r(x) \cdot \frac{g(x)}{p \cdot f(x)+1} + m'$ we

calculate

$$\begin{aligned}(p \cdot f(x) + 1) \cdot c &= p \cdot r(x) \cdot g(x) + (p \cdot f(x) + 1) \cdot m' \\ (p \cdot f(x) + 1) \cdot c &\equiv m' \mod p\end{aligned} \tag{4}$$

In the second line we took the values modulo p, however the numbers are elements of $\mathbb{Z}_q$ with $gcd(p, q) = 1$. This calculation is instead done by first mapping the element of $\mathbb{Z}_q$ to its representative in $\mathbb{Z}$ between $\lceil \frac{-q+1}{2} \rceil$ and $\lfloor \frac{q}{2} \rfloor$. We then take this value modulo p we will obtain an element of $\mathbb{Z}_p$ which will be our m'. This does however have the problem that $p \cdot d$ is not necessarily 0 modulo p, and addition is not well defined. This calculation does work with a very small decryption error rate, the reasoning will be discussed in Section 6.4.

Now we can calculate $t(x) = c - m'$ which gives us $m_{mask} = \Upsilon(HASH(t(x)))$ and $m \equiv m' + m_{mask} \mod p$. From here we can reverse the encoding to get our original message msg. We are not done yet, as we will use the same process as the encryption algorithm on m to calculate our $t'(x)$. If this $t'(x)$ matches the $t(x)$ we calculated earlier, we conclude that we have the intended message m, if they do not match we assume that some kind of error has occurred.

6.2 Code Efficiency

The parameters given in the NTRUEncrypt NIST submission are as follows:

NAME	n	q	p	d
NTRU-443	443	2048	3	143
NTRU-743	743	2048	3	247

The public key of the NTRUEncrypt cryptosystem is our $h(x) \in \mathbb{Z}_q[x]/\langle x^n - 1\rangle$. An element of $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ can be represented as an ordered list of n elements of $\mathbb{Z}_q$ in $n \cdot log_2(q)$ bits.

The decryption algorithm requires $h(x)$, so we will include the public key information in the private key. In addition to $h(x)$ we have in our private key information $f(x)$ and $g(x)$. In general, the amount of storage for these polynomials is not easy to calculate, the simplest way to store them would be to assign 2 bits for each coefficient and map each to the set $\{-1, 0, 1\}$. This would give our private key the size $n \cdot log_2(q) + 4 \cdot n$ bits

The theoretically most efficient encoding would require $log_2(p)$ bits, where p is the total number of possible $f(x)$. As each polynomial has $2d + 1$ nonzero entries and that set is split into two categories of sizes d and $d + 1$ we conclude that $p = \binom{n}{2d+1} \cdot \binom{2d+1}{d}$. And thus the private key size will be somewhere between $n \cdot log_2(q) + 2 \cdot log_2(\binom{n}{2d+1} \cdot \binom{2d+1}{d})$ and $n \cdot log_2(q) + 4 \cdot n$ bits.

The value listed in the table will be a balance between the two, converting the values of the coefficients into digits in a base 3 number and converting it to binary. This leaves us with the private key size of $n \cdot log_2(q) + 2 \cdot \lceil n \cdot log_2(3) \rceil$.

The plaintext for this cryptosystem is an element of T_n which we can encode in exactly the same way we encoded our $f(x)$ and $g(x)$ in the previous paragraph, and therefore can store up to $\lceil n \cdot log_2(3) \rceil$ bits worth of information.

The ciphertext for this cryptosystem is an element of $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ which gives it the same size as our public key: $n \cdot log_2(q)$ bits.

NAME	Public Key	Private Key	Plaintext	Ciphertext	Security
NTRU-443	0.595KB	0.776KB	88B	0.595KB	128
NTRU-743	0.998KB	1.285KB	147B	0.998KB	268

It should be noted that the private key information here also includes the public key information, assuming that anyone who intends to use this system will want to store both the public and private keys, the private key information can be stored with only around 1.4-2.4KB extra storage.

6.3 Corrected Implementation

As mentioned previously, the cryptosystem as it is defined above is not adequate. We need first to establish what the problem is that causes this cryptosystem to fail. The difficulty arises from "$p \cdot f(x)+1$ is invertible over $\mathbb{Z}_q[x]/\langle x^n-1\rangle$". Specifically, there does not exist an $f(x)$ in $T_n(d+1,d)$ that will have this property for either parameter set NTRU-443 or NTRU-743.

Because x^n-1 has a root of 1, the evaluation map at 1, $ev_1 : \mathbb{Z}_q[x]/\langle x^n-1\rangle \to \mathbb{Z}_q$ defined by $ev_1(f(x)) = f(1)$ is a homomorphism. For all $f(x) \in T_n(d+1,d)$, $ev_1(3f(x)+1) = 3f(1)+1 = 4$. Ring homomorphisms map units to units, so if there exists an $f(x) \in T_n(d+1,d)$ such that $3f(x)+1$ is invertible over $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ then $ev_1(3f(x)+1) = 4$ must also be invertible over $\mathbb{Z}_q$. However, 4 is even and therefore not invertible over $\mathbb{Z}_{2048}$. This tells us that there does not exist an $f(x) \in T_n(d+1,d)$ such that $3f(x)+1$ is invertible over $\mathbb{Z}_q[x]/\langle x^n-1\rangle$.

Because of this, the key generation function will always fail by never producing a public/private key pair. The solution to this problem is not

clear, not because possible solutions are not known, but rather what solution should be used. The three most likely options for correcting this issue would be either inverting by $f(x)$ instead of $3f(x)+1$, taking $f(x) \in T_n(d,d)$, or a different method for generating $f(x)$ from the previous iteration of NTRUEncrypt.

The first option of inverting $f(x)$ instead of $3f(x)+1$ is the option Hoffstein said was the most likely correction. This correction would require the most changes to the cryptosystem. The first such change would be that $f(x)$ should be invertible over both $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ and $\mathbb{Z}_3[x]/\langle x^n-1\rangle$. The inverse over $\mathbb{Z}_3[x]/\langle x^n-1\rangle$ we will call $f_3^{-1}(x)$. When we do the decryption calculation specified in (4), we will instead have:

$$f(x)\cdot c = p\cdot r(x)\cdot g(x) + f(x)\cdot m' = f(x)\cdot m'$$

Now after we do the reduction modulo 3, we can multiply by $f_3^{-1}(x)$ to obtain our m'. The decryption process continues as normal from here.

The second possible correction is the simplest change, and the other correction Hoffstein suggested. By taking $f(x) \in T_n(d,d)$ instead of $T_n(d+1,d)$ this issue is resolved with no other changes to the cryptosystem.

The last possible correction is the generation method used in the previous iteration of NTRUEncrypt [29]. Instead of one parameter d there were 3: d_1, d_2, and d_3. Three functions were generated, $f_i(x) \in T_n(d_i,d_i)$ for $i=1,2,3$ and $f(x)$ was taken to be $f_1(x)\cdot f_2(x)+f_3(x)$. No other changes to the cyrptosystem are required.

This generation method, however, did give a new attack method

involving a meet in the middle search of $f_i(x)$. Because the values for these d_i were small, this method was faster than the hybrid attack that will be described later for NTRU-743 [29]. In fact, the parameters other than d for NTRU-743 were selected so the system would be 256 bit secure against this attack.

An interesting note is that all three of these corrections only affect the key generation and possibly the decryption algorithms. Because of this, all three could be used for different implementations of the same cryptosystem, as the encryption process is the same across all three.

6.4 Cryptosystem Analysis

Several questions arise about this cryptosystem, and for many of these questions the answers are tied very closely together. Why do we use $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ instead of $\mathbb{Z}_p[x]/\langle g(x)\rangle$ for some irreducible $g(x)$ and prime p? In this case we don't need to worry about non-zero element inverses existing. Why did we restrict our private key polynomials to $T_n(d+1, d)$? If the private key consists of the functions that we used to form $h(x)$, why can't we use different functions $\bar{f}(x)$ and $\bar{g}(x)$ such that $h(x) = \bar{g}(x)/(p \cdot \bar{f}(x) + 1)$? Why did we take the representatives of $\mathbb{Z}_q$ between $\lceil \frac{-q+1}{2} \rceil$ and $\lfloor \frac{q}{2} \rfloor$ instead of 0 and $q-1$?

All of these questions are tied to the calculation we did that is not mathematically correct. As we mentioned in the description of the decryption algorithm, addition and multiplication of elements is not well defined for the modulo p mapping in (4). Looking at the equation we have that $p \cdot r(x) \cdot g(x) + (p \cdot f(x) + 1) \cdot m'$ consists of multiplying and adding

functions of small coefficients.

When we multiply two polynomials with small coefficients together in $\mathbb{Z}[x]/\langle x^n - 1\rangle$, we can first do the calculation over $\mathbb{Z}[x]$, then taking the remainder when dividing by $x^n - 1$. This reduction will add terms together if their exponents differ by n. Thus, each term in the resulting polynomial will be the sum of up to n products of coefficients from our original polynomials. Moreover, if only e of the coefficients of one of the original polynomials are nonzero, then each term will be the sum of up to e products of coefficients instead. If all the coefficients of the two original polynomials are in the set $\{-1, 0, 1\}$, then the maximum value for the coefficient of the resulting polynomial is e. For NTRUEncrypt, because $g(x)$, $f(x) \in T_n(d+1, d)$, if $f(x)$ or $g(x)$ are multiplied by a polynomial in T_n, then

For our $g(x)$ and $f(x)$ we have that $e = 2d + 1$ so the maximum possible coefficients before taking the modulus of $p \cdot r(x) \cdot g(x) + (p \cdot f(x) + 1) \cdot m'$ would be $p \cdot (2d + 1) + p \cdot (2d + 1) + 1$. From what we have seen earlier, if the absolute value of this number is less than $\frac{q}{2}$ then taking the modulo q will not change the value. With the parameters given in the submission we have this maximum value is 1723 for NTRU-443 and 2971 for NTRU-743. Unfortunately, both of these exceed our $\frac{q}{2}$, and therefore the cryptosystem can have decryption failures.

The probability of a decryption failure happening, however, is extremely low. For NTRU-443 the decryption failure probability is less than 2^{-196}, and for NTRU-743 the probability is less than 2^{-112}, according to the NTRUEncrypt NIST submission [8]. However, the submission references and uses the decryption error rate of a previous iteration of

NTRUEncrypt [29]. The decryption error rate is dependent on how $f(x)$ and $r(x)$ are generated, both of which are different between the two iterations. The two iterations seem to have similar chances of decryption failure; however, further investigation would be necessary to conclude this.

If a decryption failure does not occur; that is, none of the coefficients exceed $\frac{q}{2}$, we have that $(p \cdot f(x)+1) \cdot c = p \cdot r(x) \cdot g(x) + (p \cdot f(x)+1) \cdot m' \equiv m'$ *mod* p. This is the result we need for (4) to hold and for the decryption to function.

We will now analyse what would occur if the decryption was attempted with polynomials $\bar{f}(x)$ and $\bar{g}(x)$ such that $h(x) = \bar{g}(x)/(p \cdot \bar{f}(x) + 1)$. By replacing $f(x)$ and $g(x)$ by $\bar{f}(x)$ and $\bar{g}(x)$, we will discover that the decryption will fail if any of the coefficients of $c \cdot (p \cdot \bar{f}(x) + 1) = p \cdot r(x) \cdot \bar{g}(x) + (p \cdot \bar{f}(x) + 1) \cdot m'$ exceed $\frac{q}{2}$. Each coefficient of $c \cdot (p \cdot \bar{f}(x) + 1)$ will be the sum of each of the coefficient of $\bar{f}(x)$ and $\bar{g}(x)$, each multiplied by a random element of $\{-1, 0, 1\}$.

For arbitrary $\bar{f}(x)$ and $\bar{g}(x)$, it is very difficult to calculate the probability of the magnitude of any coefficient of $c \cdot (p \cdot \bar{f}(x) + 1)$ exceeding $\frac{q}{2}$. If we instead suppose that the magnitudes of coefficients of $\bar{f}(x)$ and $\bar{g}(x)$ were all the same, in other words every coefficient of $\bar{f}(x)$ and $\bar{g}(x)$ was k or $-k$ for some $k \in \mathbb{Z}$, we can approximate the decryption error rate. By taking the further approximation that $\frac{2}{3}$ of the coefficients of $r(x)$ and $m(x)$ will be non-zero, rounded to the nearest whole number, every coefficient of $c \cdot (p \cdot \bar{f}(x) + 1)$ will be the sum of $\lfloor \frac{2n}{3} \rceil$ integers of magnitude $p \cdot k$ plus some small coefficients. This distribution is the same as $p \cdot k$ multiplied by a binomial distribution plus some small coefficients, so we can calculate the

odds that the magnitude of this binomial distribution exceeds $\frac{q}{2 \cdot p \cdot k}$.

Using the values for NTRU-443 with $k = 10$ we will find that this occurs with probability 0.0016. If we assume this probability is independent for different coefficients of $c \cdot (p \cdot \bar{f}(x) + 1)$ we can calculate the approximated decryption error rate using $1 - (1 - 0.001)^{443} \approx 0.358$. In other words these functions will successfully decrypt messages with probability approximately 0.642.

6.5 Attacking NTRUEncrypt

The most efficient attacks on the NTRUEncrypt cryptosystem are attacks that use lattices to identify small $\bar{f}(x)$ and $\bar{g}(x)$. For our purposes we are only concerned with $\mathbb{Z}$ lattices in $\mathbb{R}^k$, so we will give the definition for those instead. A $\mathbb{Z}$ lattice in $\mathbb{R}^k$ is a subgroup of $(\mathbb{R}^k, +)$ generated by a basis of $\mathbb{R}^k$. More explicitly, let $\{\vec{b_1}, \vec{b_2}, \ldots, \vec{b_k}\}$ be a basis for $\mathbb{R}^k$; the $\mathbb{Z}$ lattice L generated by these vectors will be $L = \{\sum_{i=1}^{n} t_i \cdot \vec{b_i} | t_i \in \mathbb{Z}\}$.

Two different sets of basis vectors can generate the same lattice, $\{\vec{b_1}, \vec{b_2}, \ldots, \vec{b_k}\}$ and $\{\vec{b_1}, \vec{b_1} + \vec{b_2}, \vec{b_3}, \ldots, \vec{b_k}\}$ is one example. The important conclusion we draw from this is that a lattice can have different bases, and a basis can be manipulated in ways that will not change the lattice it generates. Conceptually, the basis of a $\mathbb{Z}$ lattice over $\mathbb{R}^k$ is called reduced if the basis is comprised of short, nearly orthogonal elements. For this thesis we will be using the definition given for the LLL algorithm [34]. This definition requires other definitions from the LLL algorithm and so the formal definition will be stated in Appendix C with other details about the LLL algorithm. Identifying short vectors from a reduced basis is trivial,

taking the shortest basis vector is the easiest method.

Suppose we have a public key $h(x) = h_{n-1} \cdot x^{n-1} + \cdots + h_1 \cdot x + h_0$ and we are looking for a pair $\bar{f}(x)$ and $\bar{g}(x)$ such that $h(x) = \frac{\bar{g}(x)}{\bar{f}(x)}$ to break this system. For attacking NTRUEncrypt with this method, we do not require the denominator to be of the form $3f(x)+1$, and in fact limiting our denominator like this will only make lattice attacks more difficult. Finding such a pair would be equivalent to identifying a short vector in the lattice generated by the columns of:

$$L = \begin{bmatrix} q \cdot I_n & H \\ 0_n & I_n \end{bmatrix} \in Mat_{2n,2n}(\mathbb{Z}) \tag{5}$$

Where H is the circulant matrix defined by the coefficients of $h(x)$. An important note is that this matrix is only circulant because we are over the ring $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$. In general the entries of the i^{th} column of the matrix H are the coefficients of $h(x) \cdot x^{i-1}$ over the ring $\mathbb{Z}[x]/\langle x^n - 1\rangle$. Here

$$H = \begin{bmatrix} h_0 & h_{n-1} & \dots & h_1 \\ h_1 & h_0 & \dots & h_2 \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & \dots & h_0 \end{bmatrix} \in Mat_{n,n}(\mathbb{Z})$$

Lemma 2 *If the coefficients of a polynomial $f(x)$ of degree less than n are written out as a length n vector $\vec{f}$, then $H\vec{f}$ will be the vector of the coefficients of $h(x) \cdot f(x)$ over the ring $\mathbb{Z}[x]/\langle x^n - 1\rangle$.*

Suppose $f(x)$ only had a single term, $f(x) = f_j \cdot x^j$, then $\vec{f}$ would have a single non-zero entry of f_j in the $j + 1^{th}$ position. Therefore $H\vec{f}$ would be f_j multiplied by the $j + 1^{th}$ column. Now we consider $h(x) \cdot f(x)$ over the ring $\mathbb{Z}[x]/\langle x^n - 1\rangle$. We have that $f(x) = f_j \cdot x^j$ and that the entries of

the i^{th} column of the matrix H are the coefficients of $h(x) \cdot x^{i-1}$. By setting $i = j+1$ we have that the $j+1^{th}$ column of the matrix H are the coefficients of $h(x) \cdot x^j$. Therefore, the coefficients of $h(x) \cdot f_j \cdot x^j$ should be f_j multiplied by the $j+1^{th}$ column of the matrix H. This matches what we obtained by calculating this as a vector.

To obtain this result for arbitrary $f(x)$ we note that both multiplication by H and multiplication by $h(x)$ have the distributive property. Notably, $H(\vec{f}+\vec{f^*}) = H\vec{f}+H\vec{f^*})$ and $h(x) \cdot (f(x)+f^*(x)) = h(x) \cdot f(x)+h(x) \cdot f^*(x)$ will allow us to decompose both $\vec{f}$ and $f(x)$ into individual components and then obtain the same result. ■

Lemma 3 *Vectors in the lattice generated by the columns of L correspond to polynomial pairs $\bar{g}(x)$ and $\bar{f}(x)$ such that $h(x) = \frac{\bar{g}(x)}{\bar{f}(x)}$ and short vectors in this lattice correspond to these polynomial pairs $\bar{f}(x)$ and $\bar{g}(x)$ with small coefficients.*

Consider an arbitrary vector $\vec{S} = \begin{bmatrix} S_1 & S_2 & \dots & S_{2n} \end{bmatrix}^T$ in this lattice, this vector can be written as $\vec{S} = L \cdot \vec{s^*}$ for some vector $\vec{s^*} = \begin{bmatrix} s_1^* & s_2^* & \dots & s_{2n}^* \end{bmatrix}^T$. If we split $\vec{s^*}$ into two halves, $\vec{t} = \begin{bmatrix} s_1^* & s_2^* & \dots & s_n^* \end{bmatrix}^T$ and $\vec{t^*} = \begin{bmatrix} s_{n+1}^* & s_{n+2}^* & \dots & s_{2n}^* \end{bmatrix}^T$, we can see how each of these individually affects our $\vec{S}$.

The first half, $\vec{t}$, will increase or decrease the first half of $\vec{S}$ by some multiple of q, specifically $q \cdot \vec{t}$; the second half of $\vec{S}$ is unaffected by $\vec{t}$. The second half of $\vec{s^*}$, $\vec{t^*}$, is multiplied by the identity matrix in the bottom half of L. As the second half of $\vec{S}$ was unaffected by $\vec{t}$, we obtain that the second half of $\vec{S}$ is $\vec{t^*}$. The first half of $\vec{S}$ is the sum of $q \cdot \vec{t}$ and $H\vec{t^*}$, but as we

know from our previous lemma, this vector is the coefficients of $h(x) \cdot t^*(x)$. If we allow $\vec{t}$ to be arbitrary, we obtain that $\vec{S}$ is the concatenation of the coefficients of $h(x) \cdot t^*(x)$ modulo q, and the coefficients of $t^*(x)$. This corresponds exactly to the pairs $\bar{g}(x)$ and $\bar{f}(x)$ such that $h(x) = \frac{\bar{g}(x)}{\bar{f}(x)}$.

As $\vec{t}$ allowed us to take arbitrary reductions modulo q for the first half of $\vec{S}$, for all $\vec{t^*}$ there exists $\vec{t}$ such that all coefficients of the first half of $\vec{S}$ are between $\lceil \frac{-q+1}{2} \rceil$ and $\lfloor \frac{q}{2} \rfloor$. As the vectors in this lattice are the polynomial pairs $\bar{g}(x)$ and $\bar{f}(x)$, the result that short vectors in this lattice correspond to $\bar{f}(x)$ and $\bar{g}(x)$ with small coefficients is immediate. ■

We know by construction of the cryptosystem that there exist $\bar{f}(x)$ and $\bar{g}(x)$ such that these coefficients are extremely small. Specifically there is a lattice point at the concatenation of the coefficients of $g(x)$ and $3f(x)+1$. If this lattice point, or any other of similar or less magnitude, can be identified, then an attacker can use it to decrypt messages encrypted with $h(x)$.

Finding short vectors in a lattice is not an easy problem; however, we do have methods to find somewhat short vectors. The most notable of these algorithms is the LLL algorithm [34]. The general overview of the algorithm is that it will use a variation on the Gram-Schmidt orthonormalization process to manipulate the basis vectors $(\vec{b_1}, \vec{b_2}, \ldots, \vec{b_n})$ into a new reduced basis $(\vec{b_1^*}, \vec{b_2^*}, \ldots, \vec{b_n^*})$. More details concerning the LLL algorithm can be found in Appendix C.

The LLL algorithm is only guaranteed to find a vector at most 2^n times the length of the actual shortest vector in the lattice. In practice the LLL algorithm tends to find vectors significantly shorter than this [45]. For our parameters we have $q < 2^n$; this tells us that we have no way of predicting

how short a vector LLL will find.

The LLL algorithm is not the only algorithm for finding a reduced basis of a lattice. The current best algorithm for finding a reduced basis of a lattice is a hybrid attack [29], which is a combination of the BKZ2.0 lattice reduction algorithm and combinatorial search. Combinatorial search can be thought of as a partial brute force technique, using some logic to reduce the size of the set that needs to be brute-forced.

The Block Korkin Zolotarev reduction algorithm was originally developed in 1987 but it was not until 1994 that a practical version of the algorithm was published [45]. The BKZ algorithm uses the LLL algorithm as a subroutine on 'blocks' of the lattice, these blocks being the lattice generated by a subset of the basis along with an element generated from the remaining elements of the basis. The BKZ2.0 algorithm [10] consists of a few minor additions and optimizations to the original BKZ algorithm designed to improve runtime. There are no major changes to the structure of the algorithm.

Defending against this hybrid attack is what the current NTRUEncrypt parameters are based on. For example, the parameter set for NTRU-443 is 133 secure against the hybrid attack, and therefore meets the requirement for 128 security.

There is a further addition to the BKZ2.0 algorithm known as sieving. This actually does reduce the number of required calculations that would need to be done to successfully attack the system. The issue with this attack is the space requirement, in addition to the still very large amount of calculations, 2^{93} for NTRU-443 and 2^{176} for NTRU-743, that would need to

be done, an attacker would also need a very large amount of data storage. The attack requires that an attacker has over 2^{66} bits of storage, roughly 25% of the worlds data storage capacity, for an attack on NTRU-443, or over 2^{125} bits of storage for an attack on NTRU-743 [8].

This is the most efficient attack in terms of number of calculations, however due to the excessive storage requirement, this attack was not used when analysing the security level of the cryptosystem. This attack also can be augmented by use of quantum sieving, an algorithm that can be run on a quantum computer. This will not change the data storage requirements, but it will reduce the number of calculations required to 2^{85} for NTRU-443 and 2^{159} for NTRU-743 [8].

No other quantum attacks were mentioned in the submission, so as with the McEliece cryptosystem, NTRUEncrypt appears secure against attacks with a quantum computer.

7 Summary

The question of which of these systems might be used as standard cryptosystems for post-quantum communications does not have a simple answer. Currently, we use multiple cryptosystems for communications so it is very much possible that more than one of these systems could become standard post-quantum cryptosystems. However, it is also possible that the standard cryptosystems will be ones not covered in this thesis.

Before we compare the cryptosystems, it should be noted that the second round of the NIST submissions does something very similar to what this chapter will. On January 30th 2019 the second round candidate list was announced [40]; the selected candidates were given the deadline of March 15th 2019 to submit an updated version of their cryptosystem so details about the submissions are not expected until late March or early April. At this point we only know which cryptosystems can be submitted; however, almost all of these were in the first round of submissions and we can look at the details each had posted there.

With regard to this thesis, two of the three submissions covered were accepted into the second round, the two systems being Classic McEliece and NTRUEncrypt. NTRUEncrypt, however, was merged with NTRU-HRSS-KEM to form the NTRU cryptosystem. NTRU-HRSS-KEM describes itself as "a direct parameterization of NTRUEncrypt" [28] so it appears that NTRU will be almost identical to NTRUEncrypt.

Given that there were multiple successful attacks on the McNie cryptosystem, is is not surprising it was not chosen as a round two

candidate. NIST was very clear that security is the most important evaluation criterion [1]. Because there were other cryptosystems submitted based on rank metric error correction, McNie was not a priority for inclusion.

For a cryptosystem, the most important factor is how secure the system is against any and all possible attacks. For all of the covered systems there were parameters for which these systems are secure according to our current knowledge. The issue however, is that it is possible new attacks could be developed against the systems that reduces this security below what we consider secure.

Just looking at the security criteria, the initial and Classic McEliece cryptosystems are currently the most secure. As we saw with the McEliece variants, it can often take years for an attack to be developed against a cryptosystem, for newer systems that amount of time has not passed. While it is certainly possible for older systems to be attacked, it is significantly less common. The varying security levels are secondary, if the currently analysed security levels are accurate, then the systems will be secure. The same holds for quantum security levels, the only difference being that we do not currently have a general quantum computer capable of testing our theorized algorithms.

The initial McEliece cryptosystem has only had parameter updates since it was first developed over 40 years ago, and the classic system we know is as secure as the initial system. NTRUEncrypt remained unbroken, as it was included in the first round of NIST submissions over a year ago. In contrast McNie was successfully attacked on multiple occasions, and the

recent change from LRPC to Gabidulin codes means that we haven't had a lot of time to identify an attack vector using this new variant of the McNie cryptosystem. Moreover, because McNie was not chosen for round two it likely will not be subject to much future analysis. This means that even if no new attack is posted, it still will not be considered as secure as any competition that has seen more analysis.

The second criterion we will look at is the overall efficiency of the code. This largely looks at key and text sizes for implementations at different security levels. For an efficient code we are looking to have very small public and private keys, and a plaintext size exactly equal to the ciphertext size.

For example the NTRUEncrypt systems have the smallest public keys excluding the now broken LRPC iterations of the McNie system. However, NTRUEncrypt also has a ciphertext size roughly seven times the size of its plaintext. As an example, if someone wanted to send a gigabyte of information encrypted using NTRUEncrypt, it would require almost 7 gigabytes of information to be sent. By contrast McNie2-Gabidulin codes would require less than 2 gigabytes and the initial McEleice would only require about 1.26 gigabytes.

For short messages, the total amount of communication will only be the size of the public key plus the size of one encrypted message. This give us that NTRU743 would require about 2KB, McNie2-Gabidulin256 would require about 4.5KB and mceliece8192128 would require 1.3MB. Interestingly, the order for this is reversed to what it was for longer messages.

To decide which cryptosystem has the best code efficiency we need to remember that exchanging AES private keys for future communications is an option. Given this, NTRUEncrypt has the most efficient encoding, but the difference between NTRUEncrypt and McNie2-Gabidulin is relatively small. Both variants of McEliece have terrible code efficiency in this regard. Private keys, as well as all factors at the other security levels show almost identical relative results, which allows us to conclude that NTRUEncrypt has the best overall code efficiency with McNie2-Gabidulin a close second.

The short answer to which system is best is that there isn't one best system. For some applications a key size of a few megabytes is not a major issue, but data security is a major factor. In that case the mceliece8192128 system would be the most likely choice of the analysed cryptosystems. For another application, a message need only be secure for a few minutes but sending large amounts of data is impractical or expensive, then NTRU-443 would likely be a better selection. Each application would need to be dealt with on a case-by-case basis.

For the purposes of exchanging an AES private key the NTRUEncrypt systems would likely be the most widely used with their massively reduced key sizes compared to Classic McEliece. For more secure applications Classic McEliece is likely to be a better choice, but this could easily change in the future.

8 Appendices

Appendix A - Goppa Code Equivalent Definitions

For H, α_i and $g(x)$ as defined in Section 4 along with a vector $\vec{c} = \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}^T \in \mathbb{Z}_2^n$ we will show that

$$H \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = 0 \Leftrightarrow \sum_{j=1}^{n} \frac{c_j}{x - \alpha_j} \equiv 0 \mod g(x)$$

First we show that $Hc = 0 \Leftrightarrow \bar{H}c = 0$ with $\bar{H}$ as defined in Section 4. As in Section 4 we can rewrite elements of $\bar{H}$ as polynomials of degree $m - 1$.

$$\bar{H}_{i,j} = \frac{\alpha_j^{i-1}}{g(\alpha_j)} = b_{i,j,m-1} \cdot z^{m-1} + \cdots + b_{i,j,1} \cdot z + b_{i,j,0}$$

When we calculate Hc the resulting vector will be the zero vector if and only if

$$\sum_{j=1}^{n} c_j \cdot b_{i,j,k} \equiv 0 \mod 2 \quad \forall\, 1 \leq i \leq t, 0 \leq k \leq m - 1$$

By contrast $\bar{H}c$ will result in the zero vector if and only if

$$\sum_{j=1}^{n} c_j \cdot (b_{i,j,m-1} \cdot z^{m-1} + \cdots + b_{i,j,1} \cdot z + b_{i,j,0}) = 0 \quad \forall\, 1 \leq i \leq t$$

We will distribute the c_j term and then rewrite the equation as a double

summation to get

$$\sum_{k=0}^{m-1}\sum_{j=1}^{n} c_j \cdot b_{i,j,k} \cdot z^k \equiv 0 \mod f(z) \quad \forall\, 1 \leq i \leq t$$

As we are only dealing with exponents less than m this polynomial will only be equivalent to $0 \mod f(x)$ if and only if each coefficient is equal to zero.

$$\sum_{j=1}^{n} c_j \cdot b_{i,j,k} \equiv 0 \mod 2 \quad \forall\, 1 \leq i \leq t, 0 \leq k \leq m-1$$

This allows us to conclude that $Hc = 0 \Leftrightarrow \bar{H}c = 0$. It remains to show that $\bar{H}c = 0 \Leftrightarrow \sum_{j=1}^{n} \frac{c_j}{x-\alpha_j} \equiv 0 \mod g(x)$.

We can replace $\bar{H}c = 0$ with $\sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{i-1}}{g(\alpha_j)} = 0 \quad \forall\, 1 \leq i \leq t$.

Because $g(x)$ is a monic irreducible polynomial of degree t we know that $x - \alpha_j$ will not divide $g(x)$ when $t > 1$ and thus $x - \alpha_j$ has an inverse in $F_q[x]/\langle g(x)\rangle$. When we divide $g(x)$ by $x - \alpha_j$ we obtain $g(x) = h_j(x) \cdot (x - \alpha_j) + d_j$ where $h_j(x)$ is a degree $t-1$ polynomial in $F_q[x]$ and d_j is an element of F_q. If we set $x = \alpha_j$ we get the equation $g(\alpha_j) = h_j(\alpha_j) \cdot (\alpha_j - \alpha_j) + d_j = d_j$. This tells us that $g(x) = h_j(x) \cdot (x - \alpha_j) + g(\alpha_j)$.

Additionally $g(x)$ is irreducible so for $t > 1$ we have that $x - \alpha_j$ does not divide $g(x)$ and therefore $g(\alpha_j) \neq 0$.

We can rearrange this equation and note that addition is equal to subtraction in $F_{2^m}[x]$ to obtain $g(x) + h_j(x) \cdot (x - \alpha_j) = g(\alpha_j)$. As F_q is a field, the inverse of non-zero elements exist, specifically $\frac{1}{g(\alpha_j)}$. We can now conclude that $\frac{g(x)}{g(\alpha_j)} + \frac{h_j(x)}{g(\alpha_j)} \cdot (x - \alpha_j) = 1$ and more importantly that

$\frac{1}{x-\alpha_j} \equiv \frac{h_j(x)}{g(\alpha_j)} \ mod\, g(x)$. This allows us to conclude that

$$\sum_{j=1}^{n} \frac{c_j}{x-\alpha_j} \equiv 0 \ mod\, g(x) \Leftrightarrow \sum_{j=1}^{n} \frac{c_j \cdot h_j(x)}{g(\alpha_j)} \equiv 0 \ mod\, g(x)$$

We can observe that $c_j \cdot h_j(x)$ will either be $h_j(x)$ or zero and that because summing polynomials cannot increase the order $\sum_{j=1}^{n} \frac{c_j \cdot h_j(x)}{g(\alpha_j)} \equiv 0 \ mod\, g(x) \Leftrightarrow \sum_{j=1}^{n} \frac{c_j \cdot h_j(x)}{g(\alpha_j)} = 0$. Additionally we will express $h_j(x) = h_{j,1} \cdot x^{t-1} + h_{j,2} \cdot x^{t-2} + \cdots + h_{j,t}$ which we can use to rewrite $g(x) = h_j(x) \cdot (x - \alpha_j) + g(\alpha_j)$ as:

$$g(x) = \sum_{i=1}^{t} (h_{j,i} \cdot x^{t-i+1}) + \sum_{i=1}^{t} (\alpha_j \cdot h_{j,i} \cdot x^{t-i}) + g(\alpha_j) \tag{6}$$

We are going to expand $g(x)$ similarly to how we expanded $h(x)$ to get $g(x) = g_0 \cdot x^t + g_1 \cdot x^{t-1} + g_2 \cdot x^{t-2} + \cdots + g_t$, as $g(x)$ is monic we have that $g_0 = 1$. Splitting (6) by exponents of x we can isolate for the terms of $h_j(x)$ to find that $h_{j,1} = 1$ and $h_{j,i} = g_{i-1} + \alpha_j \cdot h_{j,i-1} \ \forall i$ such that $2 \leq i \leq t$. We we can apply this equation to itself repeatedly to obtain $h_{j,i} = \sum_{t=1}^{i} g_{t-1} \cdot \alpha_j^{i-t}$.

The expansion of $h_j(x)$ can also be used to rewrite $\bar{H}c = 0 \Leftrightarrow \sum_{j=1}^{n} \frac{c_j}{x-\alpha_j} \equiv 0 \ mod\, g(x)$ as the following theorem:

Theorem 5

$$\sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{i-1}}{g(\alpha_j)} = 0 \quad \forall\, 1 \leq i \leq t \Leftrightarrow \sum_{j=1}^{n} \frac{c_j \cdot h_{j,i}}{g(\alpha_j)} = 0 \quad \forall\, 1 \leq i \leq t$$

We will prove this theorem by demonstrating that the following equations hold for all s such that $1 \leq s \leq t$.

$$\begin{aligned} &\sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{i-1}}{g(\alpha_j)} = 0 \quad \forall\, 1 \le i \le s \\ \Leftrightarrow \quad &\sum_{j=1}^{n} \frac{c_j \cdot h_{j,i}}{g(\alpha_j)} = 0 \quad \forall\, 1 \le i \le s \end{aligned} \tag{7}$$

With this we can prove the base case of $s = 1$ in (7). Considering $\sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{i-1}}{g(\alpha_j)}$ for $i = 1$ we can simplify this sum to $\sum_{j=1}^{n} \frac{c_j}{g(\alpha_j)}$. Additionally by taking $i = 1$, $\sum_{j=1}^{n} \frac{c_j \cdot h_{j,i}}{g(\alpha_j)} = \sum_{j=1}^{n} \frac{c_j}{g(\alpha_j)}$ using the result that $h_{j,1} = 1$. Obviously $\sum_{j=1}^{n} \frac{c_j}{g(\alpha_j)} = 0$ if and only if $\sum_{j=1}^{n} \frac{c_j}{g(\alpha_j)} = 0$ and thus (7) holds for our base case $s = 1$.

With our base case of $s = 1$ we need one equation before we proceed by induction on s. We first recall that $h_{j,k} = \sum_{t=1}^{k} g_{t-1} \cdot \alpha_j^{k-t}$, we can use this in the second half of (7) to obtain:

$$\begin{aligned} \sum_{j=1}^{n} \tfrac{c_j \cdot h_{j,i}}{g(\alpha_j)} &= \sum_{j=1}^{n} \sum_{t=1}^{k} g_{t-1} \cdot \tfrac{c_j \cdot \alpha_j^{k-t}}{g(\alpha_j)} \\ &= \sum_{t=1}^{k} g_{t-1} \cdot \sum_{j=1}^{n} \tfrac{c_j \cdot \alpha_j^{k-t}}{g(\alpha_j)} \\ &= \sum_{t=2}^{k} g_{t-1} \cdot \sum_{j=1}^{n} \tfrac{c_j \cdot \alpha_j^{k-t}}{g(\alpha_j)} \quad + g_0 \cdot \sum_{j=1}^{n} \tfrac{c_j \cdot \alpha_j^{k-1}}{g(\alpha_j)} \end{aligned} \tag{8}$$

We can apply this equation to replace the equation in the second half of (7). Proceeding by induction, we will assume that (7) holds for $1 \le s < k$ and will prove the statement for $s = k$. To prove the $\Rightarrow$ direction of (7) for $s = k$, we can take the second line of (8) and use the left hand side of (7) to obtain $\sum_{j=1}^{n} \frac{c_j \cdot h_{j,k}}{g(\alpha_j)} = \sum_{t=1}^{k} g_{t-1} \cdot \sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{k-t}}{g(\alpha_j)} = \sum_{t=1}^{k} g_{t-1} \cdot 0 = 0$.

To prove the $\Leftarrow$ direction of (7) we will take the third line of (8). By our induction assumption we have that

$\sum_{t=2}^{k} g_{t-1} \cdot \sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{k-t}}{g(\alpha_j)} = \sum_{t=1}^{k-1} g_t \cdot \sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{(k-1)-t}}{g(\alpha_j)} = \sum_{t=1}^{k-1} g_t \cdot 0 = 0$ which, when used with the right hand side equation from (7), implies $0 = \sum_{j=1}^{n} \frac{c_j \cdot h_{j,k}}{g(\alpha_j)} = g_0 \cdot \sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{k-1}}{g(\alpha_j)}$. As $g_0 = 1$ we have that $\sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{k-1}}{g(\alpha_j)} = 0$ which concludes the proof by induction of (7) which gives us that $\sum_{j=1}^{n} \frac{c_j \cdot \alpha_j^{i-1}}{g(\alpha_j)} = 0 \quad \forall\, 1 \leq i \leq t$ if and only if $\sum_{j=1}^{n} \frac{c_j \cdot h_{j,i}}{g(\alpha_j)} = 0 \quad \forall\, 1 \leq i \leq t$. ■

We have already shown this theorem to be equivalent to our initial problem proposed in this Appendix and allows us to conclude that:

$$H \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = 0 \Leftrightarrow \sum_{j=1}^{n} \frac{c_j}{x - \alpha_j} \equiv 0 \mod g(x)$$

Appendix B - LRPC Error Correcting Code

We are looking for a decoding algorithm for a Low Rank Parity Check code (LRPC) where elements of the parity check matrix H can be expressed as elements of the subspace of dimension d generated by $\bar{F}_i$ for $0 \leq i \leq d$. We have $H\vec{e}$ where $\vec{e}$ is of rank r, a set of $\bar{F}_i$'s which we will define later and parameters n, k, and m. This algorithm will solve for $\vec{e}$ which we can use to solve the rest of the system.

Because $\vec{e} = \begin{bmatrix} e_1 & e_2 & \dots & e_{n-k} \end{bmatrix}^T \in F_{q^m}^{n-k}$ is of rank r we can form a basis $\vec{E_1}, \vec{E_2}, \dots, \vec{E_r}$ of the elements so that $\forall 1 \leq i \leq n-k$ we can write $\vec{e_i} = \sum_{j=1}^{r} e_{i,j} \cdot \vec{E_j}$. As this is a LRPC code we similarly can write out our elements of the parity check matrix H as $h_{i,j} = \sum_{l=1}^{d} h_{i,j,l} \bar{F}_l$ for a set of d $\bar{F}_i \in F_{q^m}$. With both of these bases and all elements are written in them we

know that $H\vec{e}$ can be written in the basis $\{\bar{F}_j \cdot \vec{E}_i | 1 \leq i \leq r, 1 \leq j \leq d\}$.

As $\bar{F}_i \in F_{q^m}$ we have that $\bar{F}_i^{-1}$ exists for all i. We will take the vector $H\vec{e}$ and find a set of generators $\{\vec{s_1}, \vec{s_2}, \ldots, \vec{s_z}\}$ for the elements. As we know each of these will be some linear combination of elements from the set $\bar{F}_j \cdot \vec{E}_i$. For $1 \leq l \leq d$ we will define S_l as the subspace generated by $\{\bar{F}_l^{-1} \cdot \vec{s_i} | 1 \leq i \leq z\}$. Notably there should almost always exist $\bar{F}_j^{-1} \cdot \bar{F}_j \cdot \vec{E}_i = \vec{E}_i$ in this subspace for all S_l.

Taking the intersection of all of these we are very likely to have all of our $\vec{E}_i$ elements and nothing more. For any of the $\vec{E}_i$ to not show up would require there to exist an l such that S_l written in our $\{\bar{F}_j \cdot \vec{E}_i\}$ basis will have a coefficient of $0 \in F_{q^m}$ for one of the $\bar{F}_j \cdot \vec{E}_i$ terms. While there are a lot of these coefficients the number should in general be very small compared to the size of F_{q^m} and therefore is unlikely to occur. This is however, where the chance of the McNie cryptosystem failing to decode a message comes from. On the other hand if there was an extra element in the set of generators then it would have to effectively randomly appear in each and every S_l. The odds of this for our parameters are always lower than the probability of not fully generating $\vec{E}_i$ so we can practically ignore it when determining the probability of decoding failure.

If the intersection fails to generate an r dimensional subspace we say the decoding algorithm has failed and had this been part of the decoding algorithm for the McNie cryptosystem we would require a new message to be sent.

Assuming we obtain r elements from the S_l intersection we have a basis for our error vector $\vec{E}_1, \vec{E}_2, \ldots, \vec{E}_r$. With this plus the $\bar{F}_j$ basis for our H and

our $\bar{F}_j \cdot \vec{E}_i$ basis for our $H\vec{e}$ we can form a system of $n-k$ linear equations over F_q^m for $r \cdot n$ unknowns (the $e_{i,j}$ values) over F_q. We can rewrite the $n-k$ linear equations as $(n-k) \cdot m$ linear equations over F_q with the same number of unknowns. Assuming $r \cdot n \leq (n-k) \cdot m$ we already know how to solve this using methods from previous sections. Once we have our $e_{i,j}$ elements we can use them to calculate our $\vec{e}$ by using $e_i = \sum_{j=1}^{r} e_{i,j} \cdot \vec{E}_j$.

Appendix C - The LLL Algorithm

The LLL algorithm uses the values of $B_j = |b_j^*|^2 = (b_j^*, b_j^*)$ and $u_{i,j} = \frac{(b_i^*, b_j^*)}{B_j}$ throughout, these values will be assumed to use the most up to date b_i^* values. The (b_i^*, b_j^*) denotes the dot product between b_i^* and b_j^*. Additionally we will have $r_{i,j} = \lfloor u_{i,j} \rceil$ which is the closest integer to our $u_{i,j}$ value and the calculation $\Theta(b_i^*, b_j^*) = b_i^* - r_{i,j} \cdot b_j^*$. The algorithm is as follows:

Set b_i^* to b_i

Set a variable $k = 2$

(1) Set $b_k^* = \Theta(b_k^*, b_{k-1}^*)$

If $B_k < (\frac{3}{4} - u_{k,k-1}^2) \cdot B_{k-1}$ then go to (2)

For $j = k-2, k-3, \ldots, 1$ in that order set $b_k^* = \Theta(b_k^*, b_j^*)$

If $k = n$ algorithm is complete

Else set $k = k+1$ and go to (1)

(2) Switch b_k^* and b_{k-1}^*

If $k > 2$ set $k = k-1$

Go to (1)

In the algorithm the value $\frac{3}{4}$ appeared in line 4. This value was set for

the initial paper for the LLL algorithm but later papers replaced this with a variable δ which was modified for different applications.

With our definitions of B_j and $u_{i,j}$, we can give a formal definition to a reduced basis. A basis $\{b_1^*, b_2^*, \ldots, b_n^*\}$ is reduced if both $u_{i,j} \leq \frac{1}{2}$ for all $1 \leq j < i \leq n$ and $|b_i^* + u_{i,i-1} \cdot b_{i-1}^*|^2 \geq \frac{3}{4} B_{i-1}$ for all $1 < i \leq n$.

References

[1] Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Liu, Y.-K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., and Smith-Tone, D. (2017). *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. US Department of Commerce, National Institute of Standards and Technology.

[2] Baldi, M. and Chiaraluce, F. (2007). Cryptanalysis of a new instance of McEliece cryptosystem based on QC-LDPC codes. In *2007 IEEE International Symposium on Information Theory*, pages 2591–2595. IEEE.

[3] Berlekamp, E. (1973). Goppa codes. *IEEE Transactions on Information Theory*, 19(5):590–592.

[4] Bernstein, D., Chou, T., Lange, T., Maurich, I. v., Misoczki, R., Niederhagen, R., Persechetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., and Wang, W. (2017). *Classic McEliece: conservative code-based cryptography*. US Department of Commerce, National Institute of Standards and Technology.

[5] Bernstein, D. J., Lange, T., and Peters, C. (2008). Attacking and defending the McEliece cryptosystem. In *International Workshop on Post-Quantum Cryptography*, pages 31–46. Springer.

[6] Bernstein, D. J., Lange, T., and Peters, C. (2010). Wild mceliece. In *International Workshop on Selected Areas in Cryptography*, pages 143–158. Springer.

[7] Bernstein, D. J., Lange, T., and Peters, C. (2011). Wild mceliece incognito. In *International Workshop on Post-Quantum Cryptography*, pages 244–254. Springer.

[8] Chen, C., Hoffstein, J., Whyte, W., and Zhenfei, Z. (2017). *NIST PQ Submission: NTRUEncrypt A lattice based encryption algorithm.* US Department of Commerce, National Institute of Standards and Technology.

[9] Chen, L., Jordan, S., Liu, Y.-K., Moody, D., Peralta, R., Perlner, R., and Smith-Tone, D. (2016). *Report on post-quantum cryptography.* US Department of Commerce, National Institute of Standards and Technology.

[10] Chen, Y. and Nguyen, P. Q. (2011). BKZ 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer.

[11] Couvreur, A., Otmani, A., and Tillich, J.-P. (2017). Polynomial time attack on wild McEliece over quadratic extensions. *IEEE Transactions on Information Theory*, 63(1):404–427.

[12] Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. (2014). The Tangled Web of Password Reuse. In *NDSS*, volume 14, pages 23–26.

[13] Diaconis, P. (1990). Patterned matrices. In *Proc. of Symposia in Applied Mathematics*, volume 40, pages 37–58.

[14] Dinh, H., Moore, C., and Russell, A. (2010). The mceliece cryptosystem resists quantum fourier sampling attacks. *arXiv preprint arXiv:1008.2390*.

[15] Faugère, J.-C., Perret, L., and De Portzamparc, F. (2014). Algebraic attack against variants of McEliece with Goppa polynomial of a special form. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 21–41. Springer.

[16] Gabidulin, E. M. Rank-metric codes and applications. `https://www.researchgate.net/profile/Ernst_Gabidulin/publication/267791524_Rank-metric_codes_and_applications/links/557bd7cc08aeb61eae21cf6e/Rank-metric-codes-and-applications.pdf`. [last accessed 08-May-2019].

[17] Gabidulin, E. M. (1995). *Public-key cryptosystems based on linear codes*. Citeseer.

[18] Gabidulin, E. M., Paramonov, A., and Tretjakov, O. (1991). Ideals over a non-commutative ring and their application in cryptology. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 482–489. Springer.

[19] Gaborit, P. (2013). Shorter keys for code based cryptography. In *Proceedings of the Workshop on Coding and Cryptography WCC*.

[20] Gaborit, P., Murat, G., Ruatta, O., and Zémor, G. (2013). Low rank parity check codes and their application to cryptography. In *Proceedings of the Workshop on Coding and Cryptography WCC*, volume 2013.

[21] Gaborit, P., Ruatta, O., and Schrek, J. (2016). On the complexity of the rank syndrome decoding problem. *IEEE Transactions on Information Theory*, 62(2):1006–1019.

[22] Gaborit, P., Ruatta, O., Schrek, J., Tillich, J.-P., and Zémor, G. Rank based cryptography: a credible post-quantum alternative to classical cryptography.

[23] Galvez, L., Kim, J.-L., Kim, M. J., Kim, Y.-S., and Lee, N. (2017). *McNie: Compact McEliece-Niederreiter Cryptosystem*. US Department of Commerce, National Institute of Standards and Technology.

[24] Gibson, J. (1995). Severely denting the Gabidulin version of the McEliece public key cryptosystem. *Designs, Codes and Cryptography*, 6(1):37–45.

[25] Gibson, K. (1996). The security of the Gabidulin public key cryptosystem. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 212–223. Springer.

[26] Goppa, V. D. (1970). A new class of linear correcting codes. *Problemy Peredachi Informatsii*, 6(3):24–30.

[27] Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*.

[28] Hülsing, A., Rijneveld, J., Schanck, J. M., and Schwabe, P. (2017). *NTRU-HRSS-KEM Algorithm Speci-cations And Supporting*

Documentation. US Department of Commerce, National Institute of Standards and Technology.

[29] Hoffstein, J., Pipher, J., Schanck, J. M., Silverman, J. H., Whyte, W., and Zhang, Z. (2017). Choosing parameters for NTRUEncrypt. In *Cryptographers' Track at the RSA Conference*, pages 3–18. Springer.

[30] Hoffstein, J., Pipher, J., and Silverman, J. H. (1998). NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer.

[31] Kelly, J. (2018). A Preview of Bristlecone, Google's New Quantum Processor. `https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html`. [last accessed 08-April-2019].

[32] Kim, J.-L., Kim, Y.-S., Galvez, L., and Kim, M. J. (2018). McNie2-Gabidulin: An improvement of McNie public key encryption using Gabidulin code. *arXiv preprint arXiv:1812.05015*.

[33] Lau, T. S. C. and Tan, C. H. (2018). Key Recovery Attack on McNie Based on Low Rank Parity Check Codes and Its Reparation. In *International Workshop on Security*, pages 19–34. Springer.

[34] Lenstra, A. K., Lenstra, H. W., and Lovász, L. (1982). Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534.

[35] Mceliece, R. J. (1978). A public-key cryptosystem based on algebraic. *Coding Thv*, 4244:114–116.

[36] Minder, L. and Shokrollahi, A. (2007). Cryptanalysis of the Sidelnikov cryptosystem. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 347–360. Springer.

[37] Misoczki, R., Tillich, J.-P., Sendrier, N., and Barreto, P. S. (2013). MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *2013 IEEE international symposium on information theory*, pages 2069–2073. IEEE.

[38] Niederreiter, H. (1986). Knapsack-type cryptosystems and algebraic coding theory. *Prob. Control and Inf. Theory*, 15(2):159–166.

[39] NIST (2018). Post-Quantum Cryptography - Round 1 Submissions. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`. [last accessed 08-May-2019].

[40] NIST (2019). PQC Standardization Process: Second Round Candidate Announcement. `https://csrc.nist.gov/news/2019/pqc-standardization-process-2nd-round-candidates`. [last accessed 08-May-2019].

[41] Otmani, A., Tillich, J.-P., and Dallot, L. (2010). Cryptanalysis of two McEliece cryptosystems based on quasi-cyclic codes. *Mathematics in Computer Science*, 3(2):129–140.

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

[43] Qian Guo and Thomas Johansson and Paul Stankovski (2016). A key recovery attack on mdpc with cca security using decoding errors. Cryptology ePrint Archive, Report 2016/858. `https://eprint.iacr.org/2016/858`.

[44] Robshaw, M. (1995). Security Estimates for 512-bit RSA.

[45] Schnorr, C.-P. and Euchner, M. (1994). Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199.

[46] Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332.

[47] Sidelnikov, V. M. (1994). A public-key cryptosystem based on binary Reed-Muller codes. *Discrete Mathematics and Applications*, 4(3):191–208.

[48] Sidelnikov, V. M. and Shestakov, S. O. (1992). On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Mathematics and Applications*, 2(4):439–444.

[49] Stern, J. (1988). A method for finding codewords of small weight. In *International Colloquium on Coding Theory and Applications*, pages 106–113. Springer.

[50] Vries, S. (2016). Achieving 128-bit security against quantum attacks in OpenVPN. Master's thesis, University of Twente.

[51] Wang, Y. (2016). Quantum Resistant Random Linear Code Based Public Key Encryption Scheme RLCE. https://eprint.iacr.org/2015/298.pdf. [last accessed 08-May-2019].