Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science,
Coding Theory and Cryptology

White Paper on McEliece with Binary Goppa Codes

Michiel Marcus

Supervisors:
prof.dr. T. (Tanja) Lange
dr. P. (Peter) Schwabe

Eindhoven, February 2019

Contents

Contents **iii**

1 Introduction **1**

2 Coding Theory **3**
2.1 Encoding . . . 3
2.2 Decoding . . . 4

3 Binary Goppa Codes **9**
3.1 Field Exponent . . . 9
3.2 Support . . . 10
3.3 Error Correction Capacity . . . 10
3.4 The Code . . . 10
3.5 Parity Check Matrix . . . 12
3.6 Nicer Parity Check Matrix . . . 14
3.7 Decoding Binary Goppa Codes . . . 20

4 McEliece Cryptosystem **25**
4.1 Encryption . . . 25
4.2 Decryption . . . 26
4.3 Keys . . . 26
4.3.1 General Codes . . . 26
4.3.2 Binary Goppa Codes . . . 27

5 Niederreiter Cryptosystem **29**
5.1 Encryption . . . 29
5.2 Decryption . . . 29
5.3 Keys . . . 30
5.3.1 General codes . . . 30
5.3.2 Binary Goppa Codes . . . 30

6 Security **31**
6.1 Attacks . . . 31
6.1.1 Intuitive Brute Force Attack . . . 31
6.1.2 Information Set Decoding . . . 32
6.2 Security Notes . . . 33
6.2.1 Key sizes . . . 33
6.2.2 How to use the Cryptographic Systems . . . 34

Bibliography **35**

Chapter 1

Introduction

When you open your browser and visit a website, you usually see a little lock next to the web page URL, indicating that all information exchanged between you and the website is secured. This means that your computer and the server have initiated a connection for everyone on the network to see, yet only your computer and the server know what was said. Cryptography is the field that makes this all possible. However, cryptography is not limited to your browser. It is applied when you make a call or send a digitally signed e-mail. It is present in modern cars, credit-card readers and even in passports. But apart from personal use, cryptography is also widely applied within companies to for example encrypt files that should only be accessible to certain people. A lot of our current society revolves around this ubiquitous secrecy, but what if in 10 or 15 years, some device is created that can break all this mathematical magic that keeps our information secret? This is not some fictitious Armageddon story, this threat is real and is known as the *quantum computer.*

To get a better understanding of the problem and the solution, let's take a step back. The way cryptography works, is that there is some mathematical problem that is very hard to solve unless you have some specific secret, so that only the person or computer that has this secret can solve the particular problem. When somebody wants to send a secret message, they embed it into this problem and then send the result. As nobody can solve this problem except the recipient, the message is only known to the recipient and the sender. All widely-used cryptographic systems, such as RSA and ECC [1], use some mathematical problem that is supposed to be hard on a 'classical' computer [2]. In this case, hard means that it takes trillions of years to break a 128-bit security level [3] if all the computing power in the world is used, which is many times the lifetime of the universe [4]. However, quantum computers can abuse a certain parallelism that can easily solve the mathematical problems underlying RSA and ECC. Fortunately, there are also a handful of mathematical problems that are still believed to be hard for quantum computers and can be processed on classical computers, which gives rise to the notion of *post-quantum cryptography.*

The real issue that we face now is that firstly, the new propositions for cryptographic systems have to be analysed to guarantee that they give us a certain level of security and secondly, that the world has to switch to these new systems. Some very hard-to-update systems need to last longer than the period between now and the expected time of arrival of quantum computers. This means that such systems that are created today should already use post-quantum cryptography. Additionally, critical encrypted information that is exchanged at this very moment could by stored by malicious parties and decrypted when they have access to quantum computers. This makes it all the more imperative that the shift of cryptographic systems happens as soon as possible.

One very confidence-inspiring candidate for post-quantum cryptography is code-based cryptography, because the mathematical problem that is fundamental to this system has been analysed

[1]It is not necessary to know what these are, only that these are the most used crytographic systems.
[2]A classical computer is a non-quantum computer.
[3]This is a security level that is usually taken as a starting point.
[4]To estimate the total computing power in the world, the Bitcoin hash rate per second was used.

for almost 60 years, contrary to many other candidates, which have only been analysed over the recent years. This gives a lot of insight into what a classical computer and a quantum computer could do to break it. The consensus has been for a long time and still is that the problem is hard for both classical computers and quantum computers. The purpose of this document is to get the reader some familiarity with code-based cryptography, using a step-by-step guide on what coding theory is about and how it is used in the McEliece cryptographic system. An implementation of this cryptographic system that adheres to current security standards has been realised in [2].

This document comes with an interactive Jupyter notebook that goes through all the steps described in this document with user-specified parameters and outputs the results. This notebook can be found at https://www.hyperelliptic.org/tanja/students/m_marcus/notebook.ipynb. The notebook can be used in combination with this document to get the best understanding of binary Goppa codes and how they are used in the McEliece cryptographic system. In order to use the notebook, download SageMath for the correct operating system. There is a detailed description on this at https://wiki.sagemath.org/DownloadAndInstallationGuide. To run the notebook using SageMath on Linux, use the command line arguments

sage -n jupyter [*title*],

where [*title*] denotes the name of the notebook file. On a Windows machine, the user gets 3 shortcuts on their desktop by default, namely *SageMath* 8.5, *SageMath* 8.5 *shell* and *SageMath* 8.5 *notebook*. The version number 8.5 can vary, but any SageMath version above 8.5 (including 8.5) should be able to run the notebook. Now the user can double-click the shortcut for *SageMath* 8.5 *notebook*, which opens an interface in the default browser. In the top right corner, there is a button that says *upload*. Using this button, the user can select the downloaded notebook to add it to their locally running server and run it by clicking on it.

Chapter 2

Coding Theory

First of all, this document assumes a basic understanding of linear algebra. This means that the reader should be familiar with matrix multiplication, inversion and transposition [1]. Additionally, knowledge of finite fields is recommended[2].

This chapter provides some background information on the mathematics involved in the McEliece system, which is based on coding theory. Coding theory was originally developed to send information from one place to another under the assumption that errors may occur [9]. This is a very well-studied phenomenon, because this problem arose in the early days of the computer. Imagine sending a bit string over a copper cable or WiFi. During the transmission, one or more bits could be corrupted for a variety of reasons. To make up for these errors, coding theory describes a mathematical way of adding smart redundancy to recover from these possible errors. This and more can be found in [9]. As we work with bits only, we will assume that all values are in $\mathbb{F}_2$, the binary field, so addition and subtraction default to the xor operation.

2.1 Encoding

In general, when we want to send a message of length k, we extend the message up to a length n, which leads to $n-k$ redundant bits. This process is called *encoding*. Let's go over an example. We take $k = 4$. For clarity, the message, indicated by the letter m, is visualised as the vector

$$m = \begin{pmatrix} \alpha & \beta & \gamma & \delta \end{pmatrix}.$$

A very intuitive step would be to simply send the message twice. Then we get

$$\begin{pmatrix} \alpha & \beta & \gamma & \delta & \alpha & \beta & \gamma & \delta \end{pmatrix}.$$

If we receive a message where the first k bits are not equal to the last k bits, we know that an error occurred. However, we do not know what the actual message was. It could either be the first k bits or the last k bits. This means that we can *detect* errors, but we cannot *resolve* the errors.

In order to resolve errors, we want to introduce some relations to pinpoint where an error occurred after another person receives the message. One way is to include the bits $\alpha+\beta$, $\gamma+\delta$, $\alpha+\gamma$ and $\beta+\delta$ in the message. This final vector is called the code word. The length n of the code word c now has the value 8. We append the relations in the following way.

$$c = \begin{pmatrix} \alpha & \beta & \gamma & \delta & \alpha+\beta & \gamma+\delta & \alpha+\gamma & \beta+\delta \end{pmatrix}$$

As we work in a binary field, all the relations, which will be referred to as *parity bits*, are one bit. This extension of the message into the code word can also be done using a matrix, which is

[1]Detailed explanations and examples can be found at https://www.khanacademy.org/math/linear-algebra/matrix-transformations

[2]Literature on this can be found at http://www.hyperelliptic.org/tanja/teaching/cryptoI13/nt.pdf and http://www.hyperelliptic.org/tanja/teaching/CCI11/online-ff.pdf

called the *generator matrix*. For this specific transformation, the generator matrix G would be

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

This matrix transforms the message m of length 4 into the code word c of length 8. To see this is correct, we compute $m \cdot G$.

$$\begin{pmatrix} \alpha & \beta & \gamma & \delta \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \alpha & \beta & \gamma & \delta & \alpha+\beta & \gamma+\delta & \alpha+\gamma & \beta+\delta \end{pmatrix} = c$$

In this matrix, the first k, here 4, columns of G form an *identity matrix*, which guarantees that the first k entries of c are just the message m. The other $n-k$, here also 4, columns of G are the so-called *parity relations* that help detecting and correcting errors. When the matrix is of the form $\big(I_k | P_{n-k}\big)$, I_k being the Identity matrix of size $k \times k$ and P_{n-k} being the parity relations matrix of size $k \times (n-k)$, we say that the matrix is in *systematic form*. As we will see further along this chapter, it is very useful to have the generator matrix in the systematic form.

2.2 Decoding

To show how errors are corrected, which is also known as *decoding*, let's assume we receive a vector over an unstable medium, so it could contain errors. Here we say vector, because we only know that it is a bit string of n elements. If the vector contains 0 errors, it is a code word, but if it does contain errors, it is simply a vector of length n. We visualise the received vector w as follows.

$$w = \begin{pmatrix} \alpha' & \beta' & \gamma' & \delta' & \epsilon' & \zeta' & \eta' & \theta' \end{pmatrix}$$

where $\alpha', \beta', \gamma', \delta', \epsilon', \zeta', \eta', \theta'$ are separately received bits. If no errors occurred, the original message is simply the first k bits of w, but to check for errors, we will go over all the *parity relations* and perform the following *parity checks*. If everything is correct, then

$$\begin{aligned} \alpha' + \beta' + \epsilon' &= 0 \\ \gamma' + \delta' + \zeta' &= 0 \\ \alpha' + \gamma' + \eta' &= 0 \\ \beta' + \delta' + \theta' &= 0 \end{aligned}$$

Using these checks, we can pinpoint where an error occurred. For Example, if

$$\alpha' + \beta' + \epsilon' = 1$$

Assuming there was only 1 error, we know that either α',β' or ϵ' was flipped during transmission. If the other parity checks yield 0, then we know that the flipped bit was ϵ', because α' and β' occur separately in other relations, and these were correct. If α' or β' had been the flipped bit, we should have seen another incorrect parity check. This checking for errors can also be done using a so-called *parity check matrix*, usually called H. The parity check matrix for our example is as follows.

$$H = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

To do the parity checks, we multiply the parity check matrix by the transposed vector w. The first row checks if the addition of the bits α', β' and ϵ', which should be α, β and $\alpha+\beta$ respectively, yields 0. All the other rows consist of the other parity checks. We can confirm this by multiplying H by the transposed version of a code word c.

$$H \cdot c^T = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \\ \alpha+\beta \\ \gamma+\delta \\ \alpha+\gamma \\ \beta+\delta \end{pmatrix} = \begin{pmatrix} \alpha+\beta+(\alpha+\beta) \\ \gamma+\delta+(\gamma+\delta) \\ \alpha+\gamma+(\alpha+\gamma) \\ \beta+\delta+(\beta+\delta) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The attentive reader might have spotted that the first half of parity check matrix H is actually the transposed second half of Generator Matrix G. This is because in the Generator part, we add these bits together into *parity bits* and in the Parity Check part, we check if these *parity bits* are still the addition of the respective bits. The second half of Parity Check Matrix H is once again an *identity matrix*, because we check one parity bit per row. With this knowledge, it is trivial to construct the parity check matrix from the generator matrix and vice versa if they are in systematic form. In both cases, we take away the Identity matrix part and transpose what is left. Then we append an Identity matrix again.

$$\underset{\text{Generator Matrix}}{\left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ 0 & 1 & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 1 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & 0 & 0 & 1 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array}\right)} \quad \underset{\text{Parity Check Matrix}}{\left(\begin{array}{cccc|cccc} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & 1 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & 0 & 1 & 0 & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & 0 & 0 & 1 & 0 \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & 0 & 0 & 0 & 1 \end{array}\right)}$$

In our case, k is the same as $n-k$, which is why the Identity matrices for the generator matrix and the parity check matrix are of the same size. For other cases these values might be different. Additionally, note that our generator matrix was already in systematic form. This is not necessarily always the case, which is why our generator matrix is an *example* of a generator matrix for this code and not the unique generator matrix for this code. However, it is often possible to write a generator matrix in systematic form using *Gaussian Elimination.* For simplicity, we assume in the following that G and H are in systematic form.

Now we show how the parity check matrix can be used to correct errors. First of all, the result of multiplying a vector of length n (which is not necessarily a code word) with the Parity Check Matrix H is called the *syndrome* of that vector. As we saw before, if the syndrome is the all-zero vector, then that vector is a code word. Therefore, the formal definition of a code word is a vector for which multiplication with the Parity Check Matrix yields the all-zero vector. This is mathematically denoted as

$$Hc^T = 0^T.$$

where 0^T is a transposed vector with $n-k$ zeroes (since there are $n-k$ parity bits), which in our example is 4 zeroes. All these code words together form the *code* C. One definition of the code is all vectors c for which multiplication with the Parity Check Matrix yields the all-zero vector, denoted as

$$C = \{c | Hc^T = 0^T, c \in \{0,1\}^n\}.$$

As all code words can be generated by some message of length k multiplied by the Generator

Matrix, the code can alternatively be defined as

$$C = \{mG | m \in \{0,1\}^k\}.$$

One property of our code is that adding two code words results in another code word. We show this by using the fact that every code word can be generated by a message of length k multiplied by the Generator Matrix. In a binary field, adding two bit strings of length k results once again in a bit string of length k, so adding two distinct code words of length n yields

$$c_1 + c_2 = m_1 \cdot G + m_2 \cdot G = (m_1 + m_2) \cdot G = m' \cdot G.$$

which is another code word. This property makes our code a *linear code.*

To decode using our matrices, we simply take a look at the syndrome of the received vector and scan the columns of the Parity Check Matrix. If the first column of the Parity Check Matrix is our syndrome, then the first bit of the initial code word was flipped. In general, if column i is the syndrome, then the ith bit was flipped. flipping that bit again will correct the error. To demonstrate this, we will assume we received a vector where the second bit was flipped. We will call this vector w'.

$$w' = \begin{pmatrix} \alpha & \beta+1 & \gamma & \delta & \alpha+\beta & \alpha+\delta & \alpha+\gamma & \beta+\delta \end{pmatrix}$$

Then multiplying Parity Check Matrix H with the transposed received vector w' yields the following.

$$H \cdot w'^T = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta+1 \\ \gamma \\ \delta \\ \alpha+\beta \\ \gamma+\delta \\ \alpha+\gamma \\ \beta+\delta \end{pmatrix} = \begin{pmatrix} \alpha+\beta+1+(\alpha+\beta) \\ \gamma+\delta+(\gamma+\delta) \\ \alpha+\gamma+(\alpha+\gamma) \\ \beta+1+\delta+(\beta+\delta) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

As we can see, the computed syndrome is actually the second column of H, which means the second bit, β, was flipped. The same result could have resulted from 2 bit flips, namely the fifth bit, $\alpha + \beta$, and the eight bit, $\beta + \delta$, or even from 3 bit flips, namely the fourth entry, δ, the fifth entry, $\alpha + \beta$ and the sixth entry, $\gamma + \delta$. However, two or three errors are much less likely than one error, which is why we assume that the minimum number of errors occurred.

The reason we can use the columns of the parity check matrix to identify which bit was flipped is the following: We know that a code word multiplied by the Parity Check Matrix yields the all-zero vector and we know that a received erroneous vector can be written as

$$w' = c' + e.$$

where w' is the received erroneous vector, c' is some code word and e is the error vector. For example, the transposed received vector from the previous example, where β was flipped, can be written as follows.

$$\begin{pmatrix} \alpha \\ \beta+1 \\ \gamma \\ \delta \\ \alpha+\beta \\ \gamma+\delta \\ \alpha+\gamma \\ \beta+\delta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \\ \alpha+\beta \\ \gamma+\delta \\ \alpha+\gamma \\ \beta+\delta \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This means that whenever we multiply a transposed erroneous received vector w' with the parity check matrix, we get

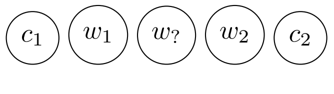
$$H \cdot w'^T = H \cdot (c'^T + e^T) = 0^T + H \cdot e^T.$$

where 0^T is once again the transposed all-zero vector. As e^T is a vector with only the erroneous bit set to 1, $H \cdot e^T$ gives exactly the column of the Parity Check Matrix where the flipped bit was. This method is called *syndrome decoding.*

In general, there are two types of decoding problems. *Regular decoding* is the problem where we have a received vector w' with $w' = c + e$ and we want to find the code word c. The other problem is *syndrome decoding*, which tries to retrieve an error vector e from the syndrome of a received vector w', where the syndrome s is Hw'. These are equivalent problems as the following reduction shows. If we want to regularly decode some received vector w' with a syndrome decoder, we can compute the syndrome Hw', apply the syndrome decoder to retrieve the error vector and calculate the code word as $c = w' + e$. Similarly, we can do syndrome decoding using a regular decoder. We have access to the syndrome $s = Hw'$, but not to w'. Assume that the parity check matrix was in systematic form. Recall from the first chapter that the matrix then first has the transposed parity relations and then an identity matrix. This means that we can reconstruct w' by prepending k zeroes to s. As the parity relations part is cancelled out by the zeroes, we are left with the multiplication of an identity matrix and s, which is simply s. So, $w' = (00\ldots00)||s$ and $Hw' = s$. Now we can use the regular decoder to retrieve the code word c and therefore the error vector e.

As shown, we are able to identify and correct up to 1 bit, so we say that this code can correct up to 1 error.

Abstractly, we could see a code as points in space that are a certain distance away from each other. Whenever we want to know which code word a received vector with errors belongs to, we take the code word for which the distance to the received vector is minimal. How many errors a code can correct is therefore directly related to the minimum distance between any pair of code words. Let's say we have two code words, c_1 and c_2 and they have a distance of 4. In other words, there are 4 hops between the code words, so 3 vectors between them.



If we received a vector that corresponds to w_1 or w_2, we know that it should have been c_1 or c_2 respectively, unless more than 1 error occurred. If we receive $w_?$, we simply have no idea whether it was c_1 or c_2, so we cannot correct a vector with two hops from a code word in this setting. As we can see, we can correct only 1 error if there are 2 or 3 vectors between the code words or equivalently, if the distance between the code words is 3 or 4. In general, if we want to correct t errors we need a distance of at least $2 \cdot t + 1$ hops between any pair of code words.

In our linear code, these hops are actually bit flips. The distance between two code words is the minimum number of bit flips required to get from one code word to the other. For example, let's take the code words c_1 and c_2 as described below. These code words were generated by applying the generator matrix to $(1, 1, 1, 0)$ and $(1, 1, 1, 1)$ respectively.

$$c_1 = \begin{pmatrix}1 & 1 & 1 & 0 & 0 & 1 & 0 & 1\end{pmatrix}$$
$$c_2 = \begin{pmatrix}1 & 1 & 1 & 1 & 0 & 0 & 0 & 0\end{pmatrix}$$

To get from c_1 to c_2, we have to flip the fourth, the sixth and the eight bit. This is a distance of 3. This particular way of calculating the distance is called the *Hamming distance*. This is based on the *Hamming weight*, which is the sum of all non-zero entries of a vector. Therefore, we can write the Hamming distance of two vectors as the Hamming weight of the sum of the two vectors, denoted as $||c_1 + c_2||$. This works, because adding two binary strings results in a string with 1s in the positions where they were different. However, we only calculated the distance between one pair of code words. To guarantee that an entire code can correct up to t errors, the distance between *all* pairs of code words needs to be $2t+1$ or higher. This is where the advantage of linear codes comes in. We write the minimum distance d of a code C as

$$d = \min_{c_1 \neq c_2} \{||c_1 + c_2|| \ \ |c_1, c_2 \in C\}.$$

but we know that the sum of two code words is yet another code word in a linear code. As a result, the minimum distance for a linear code C can be written as

$$d = \min_{c \neq 0} \{||c|| \ \ |c \in C\}.$$

We need to include the requirement that c should not be the all-zero vector, because that is not possible from the general definition as c_1 and c_2 should be distinct, and the all-zero vector can only be acquired when we add something to itself.

This concludes the basic notions related to coding theory and can function as the fundamentals to any code. Nowadays, there are many different kinds of code that for example use non-binary values or can correct more than 1 error. The codes that are interesting for cryptography, are those that have a fast decoding algorithm and can correct multiple errors. An example of such a code is the *Binary Goppa Code*. The next section aims to show why binary Goppa codes can correct up to a predefined number of errors and how this is achieved.

Chapter 3

Binary Goppa Codes

Goppa codes were introduced in 1970 by V. Goppa [3]. Just like the linear code from the previous chapter, binary Goppa codes have a generator matrix that extends a message of length k to a code word of length n and a respective parity check matrix that transforms a vector of length n into a syndrome of length $n - k$. However, they are defined a little differently. A binary Goppa code is defined by a few characteristics. From the following introductions it might not be clear what the characteristics will be used for, but a detailed explanation of their usages is provided after the introduction of these characteristics.

3.1 Field Exponent

The first characteristic is the field exponent m, which defines in which binary extension field the code will be. For a certain m, the extension field is $\mathbb{F}_{2^m}$, which can be thought of as polynomials with binary coefficients and a degree of at most $m-1$ [1]. Each of these polynomials can alternatively be written as

$$a(z) = \sum_{j=0}^{m-1} a_j z^j \tag{3.1}$$

where a_j is the jth coefficient of the polynomial. Usually, x is used for the terms of polynomials, but we will use x later on in a different context, so to clarify this distinction, we will use z for elements of $\mathbb{F}_{2^m}$. An example would be $m = 5$, $a(z) = z^4 + z + 1$, so the coefficients are $a_0 = 1$, $a_1 = 1$, $a_2 = 0$, $a_3 = 0$, $a_4 = 1$. For clarity, all elements of $\mathbb{F}_{2^m}$ are polynomials like $a(z)$. To enforce that there are 2^m elements in this field, we use a polynomial $f(z)$ of degree m as the modulus for the field. For example, let's take $\mathbb{F}_{2^3}$ with modulus $z^3 + z + 1$, so the elements are $0, 1, z, z + 1, z^2, z^2 + 1, z^2 + z, z^2 + z + 1$. Note that the coefficients are binary, so if the addition of two elements produces a polynomial where a coefficient is 2, this term will be cancelled out. For example, adding elements $z^2 + z$ and $z^2 + 1$ yields $z + 1 \bmod f(z)$. As with all fields, there is a multiplication operator, so we can multiply two polynomials. However, multiplying two polynomials of degree $m - 1$ or lower results in a polynomial of degree $2m - 2$ or lower, which is more than $m - 1$. For example, if we multiply the elements z^2 and $z^2 + 1$, we get $z^4 + z^2$, which is not an element of $\mathbb{F}_{2^3}$. To *reduce* it to an element of the field again, we apply the modulus. We multiply the modulus by a factor that results in a polynomial of the same degree as the element that we have to reduce. For the example, the degree of $z^4 + z^2$ is 4, so we multiply the modulus by z to get $z^4 + z^2 + z$. We then add $z^4 + z^2 + z$ to $z^4 + z^2$ and get z. This is an element our field, so $z^2 \cdot (z^2 + 1) = z \bmod (z^3 + z + 1)$. It is possible that after one application of the modulus, the degree of the element is still too high, so this process has to be done recursively until the result is an element of the field.

[1]This means that the largest term of the polynomial is z^{m-1}

However, the modulus should not just be of degree m, but also *irreducible.* An irreducible polynomial is a polynomial that cannot be divided by any polynomial of a lower degree except 1, which is officially a polynomial of degree 0. They are also known as prime polynomials, because they work like prime numbers. If the modulus would not be irreducible, there could be polynomials of a degree smaller than m that have no inverse. Assume the same field as in the previous example, $\mathbb{F}_{2^3}$, but we use the modulus z^3. Now z^2 and z do not have an inverse, because the greatest common divisors of z^2 and z with z^3 are z and not 1, which is necessary to find an inverse.

The field $\mathbb{F}_{2^m}$ can also be written as $\mathbb{F}_2[z]/f(z)$, because it extends the binary field using the modulus $f(z)$. Note that any polynomial modulus $f(z)$ will always result in a polynomial with a maximum degree of $m-1$ and is therefore an element of $\mathbb{F}_{2^m}$. It can easily be seen that this field contains exactly 2^m elements, because we have all combinations of m bits as the coefficients.

3.2 Support

The second characteristic is a *sequence* σ of n distinct elements of $\mathbb{F}_{2^m}$, which is also known as the *support.* It is trivial that n should therefore be smaller or equal to 2^m, as σ is a subset of $\mathbb{F}_{2^m}$. According to equation 4.1, this means for our support that every element σ_i can be written as

$$\sigma_i(z) = \sum_{j=0}^{m-1} \sigma_{i,j} z^j \tag{3.2}$$

Where $\sigma_{i,j}$ is the jth coefficient of the support element $\sigma_i(z)$. This support will be directly used in the definition of the code and in the creation of the parity check matrix.

3.3 Error Correction Capacity

The third characteristic is a positive integer t, which represents the number of errors that the code will be able to correct. We will use a polynomial $g(x)$ of degree t to achieve this. Note that we use x here instead of z, which will be very important later on. One property that is required for this polynomial, is that for all elements of the support σ, $g(\sigma_i(z))$ should not yield 0, denoted as

$$g(\sigma_i(z)) \neq 0, \sigma_i(z) \in \sigma \tag{3.3}$$

The reason for this will become clear in the next section. Note that all $\sigma_i(z)$ are polynomials in $\mathbb{F}_{2^m}$, so this means that if we substitute each x in $g(x)$ for any polynomial $\sigma_i(z)$, which is made up of z's, not all coefficients will cancel out. To achieve this property, we need $g(x)$ to be square-free, which means there should not be any polynomial of degree lower than t that results in $g(x)$ when it is squared. For simplicity, we take an irreducible polynomial for $g(x)$, because they are inherently square-free.

As before, the code uses dimension values n and k, where k is the length of the message. The length k for binary Goppa codes is $n - m \cdot t$, which is why $m \cdot t$ should be smaller than n when choosing parameters.

3.4 The Code

We now have a support σ, with elements in $\mathbb{F}_{2^m}$ and an irreducible polynomial $g(x)$ of degree t. The official definition of a binary Goppa code is then

$$C = \{c \in \{0,1\}^n \mid \sum_{i=0}^{n-1} \frac{c_i}{x - \sigma_i(z)} \equiv 0 \bmod g(x)\} \tag{3.4}$$

This might look a bit puzzling, but we will go over all the details one by one. First of all, let's take a look at the fraction. Since we work in a finite field, recall that the inverse of an element can be found using the extended Euclidean algorithm. if we want to find the inverse of an element $\epsilon(x) \bmod g(x)$, in other words $\frac{1}{\epsilon(x)} \bmod g(x)$, we apply the algorithm and get the following equation

$$1 \;=\; \epsilon(x) \cdot I_\epsilon(x) + g(x) \cdot I_g(x) \;=\; \epsilon(x) \cdot I_\epsilon(x) \bmod g(x) \tag{3.5}$$

If the reader does not know how this is done, they are encouraged to consult (online) sources on the extended Euclidean algorithm [2]. After acquiring equation 3.5, we know that $I_\epsilon(x)$ is the inverse of $\epsilon(x) \bmod g(x)$. For binary Goppa codes, this is a little more complicated, but the principle remains the same. The difference is that instead of $\frac{1}{\epsilon(x)}$, we have $\frac{1}{x-\sigma_i(z)}$, and $(x - \sigma_i(z))$ is a not just a function of x, but a function of both x and z. We still need to find an inverse mod $g(x)$, so x is seen as the main term and all polynomials in z are seen as coefficients. As an example we take $\sigma_0 = z^3 + z$. Now $x - \sigma_0(z) = x - z^3 - z = x + z^3 + z$, because we are still in a binary field, so subtraction and addition are the same. Consequently, $x+z+z^3 = x^0 \cdot (z^1+z^3)+x^1 \cdot 1$, so

$$x - \sigma_i(z) = x + z^3 + z = \begin{array}{ccc} x^0 & \cdot & (0 \cdot z^0 + 1 \cdot z^1 + 0 \cdot z^2 + 1 \cdot z^3 + 0 \cdot z^4 + \cdots + 0 \cdot z^{m-1}) & + \\ x^1 & \cdot & (1 \cdot z^0 + 0 \cdot z^1 + \cdots + 0 \cdot z^{m-1}) & + \\ x^2 & \cdot & 0 & + \\ \vdots & & \vdots & \\ x^{t-1} & \cdot & 0 & \end{array} \tag{3.6}$$

Compare this to a regular polynomial such as $5x^4 + 3x + 7$. Instead of numbers like 5, 3 and 7, we now use polynomials with terms in z as the coefficients. The reason that the highest term of x is x^{t-1} is the fact that we will apply $g(x)$ as the modulus, which has degree t. Similarly, all coefficients are in $\mathbb{F}_{2^m}$, so they go up to z^{m-1}. Formally, we say that $(x - \sigma_i(z)) \in \mathbb{F}_{2^m}[x]/g(x)$, because all *coefficients* of the terms x are in $\mathbb{F}_{2^m}$. This field has a total of $(2^m)^t = 2^{m \cdot t}$ elements. The fact that the polynomial g is referred to as $g(x)$ in most literature might be confusing, because $g(x)$ is actually a polynomial in z and x, where x is the main term and z is the term of the coefficients, so we will refer to it as $g(x, z)$ in future references to avoid this confusion. Let's take the default example of the notebook to clarify this.

Take $m = 5$, $t = 4$, $n = 32$ and $k = 12$. An example of an irreducible polynomial for $f(z)$ would be $z^5 + z^2 + 1$. Now there are quite a few possible values for $g(x, z)$ that only use the variable x, such as $x^4 + x + 1$ and $x^4 + x^3 + 1$. In other words, the coefficients of all x^i are 1, which is a value of $\mathbb{F}_{2^m}$. However, there are also possible values for $g(x, z)$ that have both x and z, such as $(z^4 + z^2)x^4 + (z^3 + z^2 + z)x^2 + (z^2 + 1)x + z^4 + z^2 + 1$ and $(z^4 + z)x^4 + (z^4 + z^3)x^3 + (z^2 + 1)x^2 + (z^4 + z)x + z^3 + 1$ .All of the above are irreducible polynomials in $\mathbb{F}_{2^m}[x]$ and are therefore equally viable candidates for $g(x, z)$. Now instead of 3.5, the extended Euclidean algorithm gives

$$1 \;=\; (x - \sigma_i(z)) \cdot I_\epsilon(x, z) + g(x, z) \cdot I_g(x, z) \;=\; (x - \sigma_i(z)) \cdot I_\epsilon(x, z) \bmod g(x, z). \tag{3.7}$$

[2] For example https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.

which means that the inverse of every $(x - \sigma_i(z))$ is also in $\mathbb{F}_{2^m}[x]/g(x,z)$. We will use this to write the definition of the binary Goppa code in a less complex way. Using equation 3.7, we can rewrite the fractions as

$$I_i(x,z) = \frac{1}{x - \sigma_i(z)} \bmod g(x,z), \sigma_i(z) \in \sigma. \tag{3.8}$$

We can now rewrite definition 3.4 of the Goppa code as

$$C = \{c \in \{0,1\}^n \ \mid \ \sum_{i=0}^{n-1} c_i I_i(x,z) \equiv \ 0 \cdot \bmod g(x,z)\}. \tag{3.9}$$

In essence, this description of the code defines a parity relation, because it describes that a certain relation between the bits of the code word has to yield 0. The following section shows how this definition can be turned into a Parity Check Matrix.

3.5 Parity Check Matrix

First of all, using the same reasoning as equation 3.2, we can rewrite the sum in definition 3.9 as

$$\sum_{i=0}^{n-1} c_i I_i(x,z) = \sum_{i=0}^{n-1} c_i \sum_{j=0}^{t-1} I_{i,j}(z)x^j \equiv 0 \bmod g(x,z) \implies \sum_{i=0}^{n-1} c_i \sum_{j=0}^{t-1} I_{i,j}(z)x^j = 0. \tag{3.10}$$

because we have n bits in the code word c, which determine whether each $\sum_{j=0}^{t-1} I_{i,j}(z)x^j$ becomes 0 or remains a non-zero polynomial. As each $I_i(x,z)$ is already reduced modulo $g(x,z)$ and addition cannot increase the degree of a polynomial, we do not have to reduce anymore, so we can use "=" instead of "≡" and take away the "mod $g(x,z)$" part. Additionally, recall that all $I_i(x,z)$ are in $\mathbb{F}_{2^m}[x]/g(x,z)$, so if we iterate over all the terms x, then every $I_{i,j}(z)$ is an element of $\mathbb{F}_{2^m}$, which is a polynomial in only z. We can visualise this as the sum of the following rows.

$$\begin{array}{lcl} c_0 & \cdot & (I_{0,0}(z) \cdot x^0 + I_{0,1}(z) \cdot x^1 + \cdots + I_{0,t-1}(z) \cdot x^{t-1}) \\ c_1 & \cdot & (I_{1,0}(z) \cdot x^0 + I_{1,1}(z) \cdot x^1 + \cdots + I_{1,t-1}(z) \cdot x^{t-1}) \\ \vdots & & \qquad\qquad\qquad\vdots \\ c_n & \cdot & (I_{n,0}(z) \cdot x^0 + I_{n,1}(z) \cdot x^1 + \cdots + I_{n,t-1}(z) \cdot x^{t-1}) \\ \hline 0 & & \end{array} + \tag{3.11}$$

We want to use this nice format to build a parity check matrix. To achieve this, we will show that this can be written in a different form that directly translates into a parity check matrix. From 3.11 it trivially follows that

$$\sum_{i=0}^{n-1} c_i \sum_{j=0}^{t-1} I_{i,j}(z)x^j = \sum_{i=0}^{n-1}\sum_{j=0}^{t-1} c_i I_{i,j}(z)x^j = 0, \tag{3.12}$$

which is simply

$$\begin{array}{rrr} c_0 \cdot I_{0,0}(z) \cdot x^0 + & c_0 \cdot I_{0,1}(z) \cdot x^1 + \cdots + & c_0 \cdot I_{0,t-1}(z) \cdot x^{t-1} \\ c_1 \cdot I_{1,0}(z) \cdot x^0 + & c_1 \cdot I_{1,1}(z) \cdot x^1 + \cdots + & c_1 \cdot I_{1,t-1}(z) \cdot x^{t-1} \\ & \vdots & \\ c_n \cdot I_{n,0}(z) \cdot x^0 + & c_n \cdot I_{0,1}(z) \cdot x^1 + \cdots + & c_n \cdot I_{0,t-1}(z) \cdot x^{t-1} \\ \hline 0 & & \end{array} + . \tag{3.13}$$

Adding these rows together should result in 0. As we are adding polynomials, this sum can be done independently for each term x^j, because adding all coefficients for a term x^j for a number of polynomials does not affect any other term. For example, if adding polynomials $5x^3 + 2x^2 + 3x + 4$ and $ax^3 + bx^2 + cx + d$ needs to be 0, then $(5 + a)x^3 = 0, (2 + b)x^2 = 0, (3 + c)x = 0, 4 + d = 0$. This creates 4 independent equations. This means for 3.13 that we get t independent sums that should all be 0. Mathematically, we write this as

$$\text{for } 0 \le j < t,\ \sum_{i=0}^{n-1} c_i I_{i,j}(z) x^j = x^j \cdot \sum_{i=0}^{n-1} c_i I_{i,j}(z) = 0. \tag{3.14}$$

which says that the sum of each column in 3.13 should be 0, independently of the other columns. This is visualised as

$$\begin{array}{ccccc} x^0 & \cdot & (c_0 \cdot I_{0,0}(z) + c_1 \cdot I_{1,0}(z) + \cdots + c_n \cdot I_{n,0}(z)) & = & 0 \\ x^1 & \cdot & (c_0 \cdot I_{0,1}(z) + c_1 \cdot I_{1,1}(z) + \cdots + c_n \cdot I_{n,1}(z)) & = & 0 \\ \vdots & & \vdots & & \vdots \\ x^{t-1} & \cdot & (c_0 \cdot I_{0,t-1}(z) + c_1 \cdot I_{1,t-1}(z) + \cdots + c_n \cdot I_{n,t-1}(z)) & = & 0 \end{array} . \tag{3.15}$$

If $x^j \cdot \sum_{i=0}^{n-1} c_i I_{i,j}(z) = 0$, then $\sum_{i=0}^{n-1} c_i I_{i,j}(z)$ has to be 0, which shows that the checks only depend on the bits c_i and the coefficients $I_{i,j}(z)$. We could store all $I_{i,j}(z)$ values in a matrix to create a parity check matrix, because then each column would be multiplied by a bit of the code word, which creates exactly these checks. However, we can do better. Recall that our $I_{i,j}(z)$ are elements of $\mathbb{F}_{2^m}$. We can apply the same trick we used in 3.15 to create binary parity checks. Recall that elements of $\mathbb{F}_{2^m}$ are polynomials of degree $m-1$ in the term z. Just like we did in 3.14, we can create m independent checks out of this for each j. Mathematically, we write this as

$$\text{for } 0 \le j < t,\ x^j \cdot \sum_{i=0}^{n-1} c_i I_{i,j}(z) = x^j \cdot \sum_{i=0}^{n-1} \sum_{k=0}^{m-1} c_i I_{i,j,k} z^k = x^j \cdot \sum_{k=0}^{m-1} (z^k \cdot \sum_{i=0}^{n-1} c_i I_{i,j,k}) = 0. \tag{3.16}$$

where all $I_{i,j,k}$ are binary values. A binary matrix is much easier to work with for a computer in terms of storage, calculation and transmission, which is why this is the preferred representation. The matrix is visualised as the following.

$$\begin{array}{ccccccl} x^0 & \cdot & z^0 & \cdot & (c_0 \cdot I_{0,0,0} + c_1 \cdot I_{1,0,0} + \cdots + c_n \cdot I_{n-1,0,0}) & = & 0 \\ x^0 & \cdot & z^1 & \cdot & (c_0 \cdot I_{0,0,1} + c_1 \cdot I_{1,0,1} + \cdots + c_n \cdot I_{n-1,0,1}) & = & 0 \\ x^0 & \cdot & \vdots & & \vdots & & \vdots \\ x^0 & \cdot & z^{m-1} & \cdot & (c_0 \cdot I_{0,0,m-1} + c_1 \cdot I_{1,0,m-1} + \cdots + c_n \cdot I_{n-1,0,m-1}) & = & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ x^{t-1} & \cdot & z^0 & \cdot & (c_0 \cdot I_{0,t-1,0} + c_1 \cdot I_{1,t-1,0} + \cdots + c_n \cdot I_{n-1,t-1,0}) & = & 0 \\ x^{t-1} & \cdot & z^1 & \cdot & (c_0 \cdot I_{0,t-1,1} + c_1 \cdot I_{1,t-1,1} + \cdots + c_n \cdot I_{n-1,t-1,1}) & = & 0 \\ x^{t-1} & \cdot & \vdots & & \vdots & & \vdots \\ x^{t-1} & \cdot & z^{m-1} & \cdot & (c_0 \cdot I_{0,t-1,m-1} + c_1 \cdot I_{1,t-1,m-1} + \cdots + c_n \cdot I_{n-1,t-1,m-1}) & = & 0 \end{array} \tag{3.17}$$

This makes a total of $t \cdot m$ parity checks, because for all t checks on x, we have m checks on z. As we can see yet again, for all $x^j \cdot z^k \cdot \sum_{k=0}^{m-1} c_i I_{i,j,k} = 0$, the equation only depends on $\sum_{k=0}^{m-1} c_i I_{i,j,k}$. Since the c_i have to come from the code word, we can create a binary parity check matrix by storing all $I_{i,j,k}$ in a matrix in the same fashion as done in 3.17. The $(t \cdot m) \times n$ Parity Check Matrix H looks like the following.

$$H = \begin{bmatrix} I_{0,0,0} & I_{1,0,0} & \cdots & I_{n-1,0,0} \\ I_{0,0,1} & I_{1,0,1} & \cdots & I_{n-1,0,1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,0,m-1} & I_{1,0,m-1} & \cdots & I_{n-1,0,m-1} \\ I_{0,1,0} & I_{1,1,0} & \cdots & I_{n-1,1,0} \\ I_{0,1,1} & I_{1,1,1} & \cdots & I_{n-1,1,1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,1,m-1} & I_{1,1,m-1} & \cdots & I_{n-1,1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,t-1,0} & I_{1,t-1,0} & \cdots & I_{n-1,t-1,0} \\ I_{0,t-1,1} & I_{1,t-1,1} & \cdots & I_{n-1,t-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,t-1,m-1} & I_{1,t-1,m-1} & \cdots & I_{n-1,t-1,m-1} \end{bmatrix} \tag{3.18}$$

However, this means that we need to calculate the inverse of $(x - \sigma_i(z))$ for all i first, which requires n times an entire extended Euclidean algorithm execution. There is a more efficient way to acquire this parity check matrix that instead of n extended Euclidean algorithm executions, uses the support elements directly. The next section shows this method.

3.6 Nicer Parity Check Matrix

The alternative way of creating the parity check matrix can be shown by symbolically applying the extended Euclidean algorithm to any $(x - \sigma_i(z))$ and $g(x, z)$, which defines the inverse of $(x - \sigma_i(z)) \bmod g(x, z)$. This yields a generalised solution for any element of any support σ. This means that no extended Euclidean algorithm has to be run.

To start off, note that we cannot stay in the field $\mathbb{F}_{2^m}[x]/g(x, z)$ for this, because then we would take the gcd of $(x - \sigma_i(z))$ and 0. We will simply refrain from applying $g(x, z)$, which means we work in the so-called *ring* $\mathbb{F}_{2^m}[x]$ [3]. Bear in mind that our coefficients are still in $\mathbb{F}_{2^m}$, so $f(z)$ does have to be applied to the coefficients. For this example we will take $g(x, z) = x^3 + x + 1$.

[3]Because there is no modulus, the elements do not have inverses, which is a requirement for a field.

The algorithm starts as follows

Table 3.1: extended Euclidean algorithm

index	quotient	remainder	$g(x,z)$	$x-\sigma_i(z)$
0		$g(x,z)$	1	0
1		$x-\sigma_i(z)$	0	1
2				

The quotient of index 2 will be $\frac{g(x,z)}{x-\sigma_i(z)}$. We can use long division to calculate the quotient and the remainder. Recall that we are still in $\mathbb{F}_{2^m}[x]$, so addition and subtraction are still the same.

Step (1): The first term of the quotient

$$\begin{array}{rl}
 & \boldsymbol{x^2} \\ \cline{2-2}
x-\sigma_i(z) & \big) x^3+x+1 \\
+ & \boldsymbol{x^3+x^2(\sigma_i(z))} \\ \hline
 & x^2(\sigma_i(z))+x+1
\end{array}$$

Step (2): The second term of the quotient

$$\begin{array}{rl}
 & x^2+\boldsymbol{x(\sigma_i(z))} \\ \cline{2-2}
x-\sigma_i(z) & \big) x^3+x+1 \\
+ & x^3+x^2(\sigma_i(z)) \\ \hline
 & x^2(\sigma_i(z))+x+1 \\
+ & \boldsymbol{x^2(\sigma_i(z))+x(\sigma_i(z)^2)} \\ \hline
 & x(1+\sigma_i(z)^2)+1
\end{array}$$

Step (3): The final term of the quotient

$$\begin{array}{rl}
 & x^2 + x(\sigma_i(z)) + \boldsymbol{1 + \sigma_i(z)^2} \\
\cline{2-2}
x - \sigma_i(z) & \big) x^3 + x + 1 \\
+ & x^3 + x^2(\sigma_i(z)) \\
\cline{2-2}
 & x^2(\sigma_i(z)) + x + 1 \\
+ & \boldsymbol{x^2(\sigma_i(z)) + x(\sigma_i(z)^2)} \\
\cline{2-2}
 & x(1 + \sigma_i(z)^2) + 1 \\
+ & \boldsymbol{x(1 + \sigma_i(z)^2) + \sigma_i(z) + \sigma_i(z)^3} \\
\cline{2-2}
 & \sigma_i(z)^3 + \sigma_i(z) + 1
\end{array}$$

We see that for $g(x, z) = x^3 + x + 1$, the remainder of $\frac{g(x,z)}{x-\sigma_i(z)}$ is actually $g(\sigma_i(z))$. It can be shown that this is true for all $g(x, z)$. How that is done is out of scope, but the curious reader could try to prove that themselves after reading the whole chapter. Now we can plug the remainder and the quotient into the extended Euclidean algorithm table.

Table 3.2: extended Euclidean algorithm

index	quotient	remainder	$g(x,z)$	$x - \sigma_i(z)$
0		$g(x,z)$	1	0
1		$x - \sigma_i(z)$	0	1
2	$\frac{g(x,z)}{x-\sigma_i(z)} = x^2 + x(\sigma_i(z)) + 1 + \sigma_i(z)^2$	$g(\sigma_i(z))$	1	$x^2 + x(\sigma_i(z)) + 1 + \sigma_i(z)^2$

This remainder of $g(\sigma_i(z))$ explains why we have the restriction that $g(\sigma_i(z)) \neq 0$ for all i, because if $g(\sigma_i(z))$ is 0, that means that the algorithm terminates at index 2 and that the greatest common divisor is not 1, while it needs to be 1 for $x - \sigma_i(z)$ to have an inverse modulo $g(x, z)$. To show how we now get to the inverse, we compose the following equation from index 2 of the table above.

$$g(\sigma_i(z)) = g(x,z) + (x - \sigma_i(z))(x^2 + x \cdot \sigma_i(z) + 1 + \sigma_i(z)^2) \equiv (x - \sigma_i(z))(x^2 + x \cdot \sigma_i(z)) + 1 + \sigma_i(z)^2) \bmod g(x,z) \tag{3.19}$$

If we now divide everything by $g(\sigma_i(z))$, we get

$$1 \equiv (x - \sigma_i(z)) \cdot \frac{x^2 + x(\sigma_i(z)) + 1 + \sigma_i(z)^2}{g(\sigma_i(z))} \bmod g(x,z). \tag{3.20}$$

which means that

$$\frac{1}{x - \sigma_i(z)} \equiv \frac{x^2 + x(\sigma_i(z)) + 1 + \sigma_i(z)^2}{g(\sigma_i(z))} \bmod g(x,z). \tag{3.21}$$

for $g(x, z) = x^3 + x + 1$. This is another reason why $g(\sigma_i(z))$ should not be 0, because the coefficient 0 does not have an inverse in $\mathbb{F}_{2^m}$. Note that $\frac{1}{g(\sigma_i(z))}$ is an element of $\mathbb{F}_{2^m}$, so we can consider this a scalar and distribute it over the different terms of the polynomial. Compare this to dividing x^2+x+1 by a scalar, for example 7. then $\frac{x^2+x+1}{7} = x^2(\frac{1}{7}) + x(\frac{1}{7}) + 1(\frac{1}{7})$. Following the same logic, we can rewrite equation 3.21 as

$$\frac{1}{x-\sigma_i(z)} \equiv x^2(\frac{1}{g(\sigma_i(z))}) + x(\frac{\sigma_i(z))}{g(\sigma_i(z))}) + 1(\frac{1+\sigma_i(z)^2)}{g(\sigma_i(z))}) \bmod g(x,z). \tag{3.22}$$

Now we can easily compute the inverse for any $x-\sigma_i(z)$ mod x^3+x+1. We have the coefficients for each term, so we can easily create a parity check matrix like 3.18. For just $\sigma_0(z)$ and $g(x,z) = x^3+x+1$, the matrix that computes the inverse would be

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ \sigma_0(z) \\ \sigma_0(z)^2 \end{bmatrix} \cdot \frac{1}{g(\sigma_0(z))}. \tag{3.23}$$

The first and second matrices make up the numerators of the coefficients in 3.22. The scalar, which makes up the denominator, is then applied to all the coefficients. To confirm that this multiplication yields the coefficients, we calculate this multiplication and get

$$H = \begin{bmatrix} \frac{1}{g(\sigma_0(z))} \\ \frac{\sigma_0(z)}{g(\sigma_0(z))} \\ \frac{1+\sigma_0(z)^2}{g(\sigma_0(z))} \end{bmatrix}. \tag{3.24}$$

These are the coefficients of the inverse as we saw in 3.22. After computation with actual values, we can extend this matrix from 3 entries to to $3m$ entries by sorting by the m different powers of z and create the first column of 3.18. If we want to extend this to all n elements of the support, we need to extend the second matrix so that all $\sigma_i(z)$ are involved instead of just $\sigma_0(z)$. Additionally, we have to change the scalar into a matrix, because the scalar $\frac{1}{g(\sigma_0(z))}$ should only be applied to the first column, $\frac{1}{g(\sigma_1(z))}$ should only be applied to the second column, etc. The matrix that computes all the inverses mod $g(x,z) = x^3+x+1$, is as follows.

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & \dots & 1 \\ \sigma_0(z) & \sigma_1(z) & \dots & \sigma_{n-1}(z) \\ \sigma_0(z)^2 & \sigma_1(z)^2 & \dots & \sigma_{n-1}(z)^2 \end{bmatrix} \begin{bmatrix} \frac{1}{g(\sigma_0(z))} & 0 & 0 & 0 \\ 0 & \frac{1}{g(\sigma_1(z))} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \frac{1}{g(\sigma_{n-1}(z))} \end{bmatrix} \tag{3.25}$$

Compare this to 3.23. The first matrix is still the same, because the numerators for the coefficients in 3.22 are picked in the same way for each $\sigma_i(z)$. The second and third matrix have been extended to calculate the inverses for all $(x-\sigma_i(z))$. The result of the multiplication of these matrices yields

$$H = \begin{bmatrix} \frac{1}{g(\sigma_0(z))} & \frac{1}{g(\sigma_1(z))} & \dots & \frac{1}{g(\sigma_{n-1}(z))} \\ \frac{\sigma_0(z)}{g(\sigma_0(z))} & \frac{\sigma_1(z)}{g(\sigma_1(z))} & \dots & \frac{\sigma_{n-1}(z)}{g(\sigma_{n-1}(z))} \\ \frac{1+\sigma_0(z)^2}{g(\sigma_0(z))} & \frac{1+\sigma_1(z)^2}{g(\sigma_1(z))} & \dots & \frac{1+\sigma_{n-1}(z)^2}{g(\sigma_{n-1}(z))} \end{bmatrix}. \tag{3.26}$$

Now each ith column contains the coefficients of the inverse of $(x-\sigma_i(z))$ mod $g(x,z)$, where the top value belongs to the term x^2, the middle value belongs to x and the bottom value belongs to 1. As shown in the previous section, this is the parity check matrix. After computing this matrix, we have a matrix where all elements are in $\mathbb{F}_{2^m}$ and we can rewrite it into a binary matrix as shown in 3.18.

The last step is to generalise the parity check matrix for any $g(x, z)$. The matrix that needs to be generalised is the first matrix in 3.25. This matrix was created by checking which elements of the sequence $(1, \sigma_i(z), \sigma_i(z)^2)$ were required in the numerators of equation 3.22. Recall from table 3.2 that these numerators are the quotient of the long division of $g(x, z)$ divided by $(x - \sigma_i(z))$, so generalising the parity check matrix for any $g(x, z)$, requires us to inspect the long division in steps $1, 2$ and 3 of the start of this section. Assume any $g(x, z)$, written as

$$g(x, z) = \sum_{i=0}^{t} g_i \cdot x^i. \tag{3.27}$$

If we now start the long division, we see that the first term is always x^{t-1}, because g_t is always 1.

$$\begin{array}{rl}
 & \boldsymbol{x^{t-1}} \\ \cline{2-2}
x - \sigma_i(z) & \big)x^t + g_{t-1}x^{t-1} + \cdots + g_1 x + g_0 \\
+ & \boldsymbol{x^t + x^{t-1}(\sigma_i(z))} \\ \hline
 & x^{t-1}(g_{t-1} + \sigma_i(z)) + \sum_{j=0}^{t-2} g_j \cdot x^j
\end{array}$$

Now the next value of the quotient depends on the value of g_{t-1}. As g_{t-1} is a value in $\mathbb{F}_{2^m}$, the next term of our quotient will be $x^{t-2}(g_{t-1} + \sigma_i(z))$.

$$\begin{array}{rl}
 & x^{t-1} + \boldsymbol{x^{t-2}(g_{t-1} + \sigma_i(z))} \\ \cline{2-2}
x - \sigma_i(z) & \big)x^t + g_{t-1}x^{t-1} + \cdots + g_1 x + g_0 \\
+ & x^t + x^{t-1}(\sigma_i(z)) \\ \hline
 & x^{t-1}(g_{t-1} + \sigma_i(z)) + g_{t-2}x^{t-2} + \cdots + g_1 x + g_0 \\
+ & \boldsymbol{x^{t-1}(g_{t-1} + \sigma_i(z)) + x^{t-2}(g_{t-1} + \sigma_i(z)) \cdot \sigma_i(z)} \\ \hline
 & x^{t-2}((g_{t-1} + \sigma_i(z)) \cdot \sigma_i(z) + g_{t-2}) + \sum_{j=0}^{t-3} g_j \cdot x^j
\end{array}$$

Note that the coefficient of the third term of our quotient will have to eliminate the first coefficient of our remainder now. So the next coefficient will be $x^{t-3}((g_{t-1} + \sigma_i(z)) \cdot \sigma_i(z) + g_{t-2})$. The pattern that arises here, is that the third coefficient is the second coefficient times $\sigma_i(z)$ plus g_{t-2}. In general, we see that for the jth coefficient of the quotient, starting from $j = 1$,

$$q_j = q_{j-1} \cdot \sigma_i(z) + g_{t-j}. \tag{3.28}$$

where $q_0 = 1$. This means that each coefficient depends on all the coefficient that came before it. This can actually be beautifully incorporated in a matrix, as the following paragraph will show. Let's first take a look at how this would work out for the first element of any support, $\sigma_0(z)$. We need to create a matrix M such that

$$M \cdot \begin{bmatrix} 1 \\ \sigma_0(z) \\ \sigma_0(z)^2 \\ \vdots \\ \sigma_0(z)^{t-1} \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{t-1} \end{bmatrix}. \tag{3.29}$$

where q_0 through q_{t-1} are the coefficients of the quotient. We know that q_0 has to be 1, as stated in 3.28. This means that the first row of M can be written as follows.

$$\begin{bmatrix} \mathbf{1} & 0 & 0 & \dots & 0 \\ ? & ? & ? & ? & ? \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ ? & ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \sigma_0(z) \\ \sigma_0(z)^2 \\ \vdots \\ \sigma_0(z)^{t-1} \end{bmatrix} = \begin{bmatrix} \mathbf{1} \\ q_1 \\ q_2 \\ \vdots \\ q_{t-1} \end{bmatrix} \tag{3.30}$$

Then, following the rule from 3.28, we multiply q_0 by $\sigma_i(z)$ and add g_{t-1} to calculate q_1. For M, to achieve multiplication by $\sigma_i(z)$, we can simply shift all the elements 1 cell to the right, because the matrix we are multiplying by contains elements that increasingly have a higher power of $\sigma_i(z)$. Then we can put g_{t-1} in the cell that multiplies with 1, which is the first cell. This means that we can write the second row of M as follows.

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \boldsymbol{g_{t-1}} & \mathbf{1} & 0 & \dots & 0 \\ ? & ? & ? & ? & ? \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ ? & ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \boldsymbol{\sigma_0(z)} \\ \sigma_0(z)^2 \\ \vdots \\ \sigma_0(z)^{t-1} \end{bmatrix} = \begin{bmatrix} 1 \\ \boldsymbol{g_{t-1} + \sigma_i(z)} \\ q_2 \\ \vdots \\ q_{t-1} \end{bmatrix} \tag{3.31}$$

In the same fashion, we can create q_2 by multiplying q_1 by $\sigma_i(z)$ and adding g_{t-2}. Note that when we multiply q_1 by $\sigma_i(z)$, we multiply the addition of two elements by $\sigma_i(z)$, which is the same as multiplying both elements with $\sigma_i(z)$. This is clarified by computing

$$q_2 = q_1 \cdot \sigma_i(z) + g_{t-2} = (g_{t-1} + \sigma_i(z))\sigma_i(z) + g_{t-2} = g_{t-1}\sigma_i(z) + \sigma_i(z)^2 + g_{t-2}. \tag{3.32}$$

This means that for the third row of M, we can simply shift all the elements of the second row 1 cell to the right again and put g_{t-2} in the first cell. Then our matrix M looks like the following.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ g_{t-1} & 1 & 0 & 0 & \dots & 0 \\ \boldsymbol{g_{t-2}} & \boldsymbol{g_{t-1}} & \mathbf{1} & 0 & \dots & 0 \\ ? & ? & ? & ? & ? & ? \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ ? & ? & ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \boldsymbol{\sigma_0(z)} \\ \boldsymbol{\sigma_0(z)^2} \\ \vdots \\ \sigma_0(z)^{t-1} \end{bmatrix} = \begin{bmatrix} 1 \\ g_{t-1} + \sigma_i(z) \\ \boldsymbol{g_{t-2} + g_{t-1}\sigma_i(z) + \sigma_i(z)^2} \\ q_3 \\ \vdots \\ q_{t-1} \end{bmatrix} \tag{3.33}$$

This pattern continues until the tth coefficient q_{t-1}, because we calculate the coefficients of x^{t-1} down to x^0. According to 3.28, $q_{t-1} = q_{t-2} \cdot \sigma_i(z) + g_{t-(t-1)} = q_{t-2} \cdot \sigma_i(z) + g_1$, which means that the last row starts with g_1. Now we can write our matrix M as

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ g_{t-1} & 1 & 0 & \dots & 0 \\ g_{t-2} & g_{t-1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \dots & 1 \end{bmatrix} \begin{bmatrix} 1 \\ \sigma_0(z) \\ \sigma_0(z)^2 \\ \vdots \\ \sigma_0(z)^{t-1} \end{bmatrix} = \begin{bmatrix} 1 \\ q_1 \\ q_2 \\ \vdots \\ q_{t-1} \end{bmatrix}. \tag{3.34}$$

We have now generalised the quotient for any $g(x, z)$. As we saw before, the matrix M does not change when we go from one element of the support to all n elements of the support, so the final parity check matrix H for the Goppa code is

$$H = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ g_{t-1} & 1 & 0 & \dots & 0 \\ g_{t-2} & g_{t-1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & \dots & 1 \\ \sigma_0(z) & \sigma_1(z) & \dots & \sigma_{n-1}(z) \\ \sigma_0(z)^2 & \sigma_1(z)^2 & \dots & \sigma_{n-1}(z)^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_0(z)^{t-1} & \sigma_1(z)^{t-1} & \dots & \sigma_{n-1}(z)^{t-1} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{g(\sigma_0(z))} & 0 & 0 & 0 \\ 0 & \frac{1}{g(\sigma_1(z))} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \frac{1}{g(\sigma_{n-1}(z))} \end{bmatrix} \tag{3.35}$$

This multiplication results in a $t \times n$ matrix in $\mathbb{F}_{2^m}$, which can be converted to a binary $(t \cdot m) \times n$ parity check matrix as shown in 3.18. Binary Goppa codes are different from the code in the previous chapter in the fact that we create the parity check matrix first. To generate code words, we should now turn the parity check matrix into a generator matrix. Recall from the previous chapter that we can turn a parity check matrix into a generator matrix and vice versa using the *systematic form*. Therefore, we should first turn H into its systematic form using *Gaussian Elimination* and then change it into a generator matrix. This is possible if and only if the right-most $(n - k) \times (n - k)$ block of the parity check matrix consists of linearly independent columns. This is the case in approximately 33% of parity check matrices like these, so on average it takes 3 trials to create a binary Goppa code from a random support.

In the previous chapter, we could use our parity check matrix to decode the syndromes, but that only works if there is 1 error and there will be a lot of errors in the binary Goppa code. Therefore, the next section shows a fast decoding algorithm for binary Goppa codes with multiple errors that uses the support σ and the modulus $g(x, z)$ to recover all t errors.

3.7 Decoding Binary Goppa Codes

Here we cover a decoding algorithm from 1975 created by Patterson [8]. Assume we receive a vector w that contains errors. As before, we can write w as a code word c plus some error vector e.

$$w = c + e \tag{3.36}$$

Let's call the summation part of the Goppa code in definition 3.4 $S(x, z)$. Since w can contain errors, we get the following equation for $S_w(x, z)$.

$$S_w(x, z) = \sum_{i=0}^{n-1} \frac{c_i + e_i}{x - \sigma_i(z)} \tag{3.37}$$

We know from definition 3.4 that

$$\sum_{i=0}^{n-1} \frac{c_i + e_i}{x - \sigma_i(z)} = \sum_{i=0}^{n-1} \frac{c_i}{x - \sigma_i(z)} + \sum_{i=0}^{n-1} \frac{e_i}{x - \sigma_i(z)} \equiv 0 + \sum_{i=0}^{n-1} \frac{e_i}{x - \sigma_i(z)} \mod g(x,z). \tag{3.38}$$

In order to find the errors that occurred during transmission, we calculate a so-called error locator polynomial. This error locator polynomial will be introduced first and then we will show how we can calculate it. The error locator polynomial $\epsilon(x,z)$ is a polynomial of the form

$$\epsilon(x,z) = \prod_{i, e_i \neq 0} (x - \sigma_i(z)), \tag{3.39}$$

where e_i means the bit at index i of the error vector, so the product only contains the term $(x - \sigma_i(z))$ if $e_i = 1$. If we have this polynomial, we can check for each i if replacing x with $\sigma_i(z)$ yields 0. If this is the case, then at some point in the product, there must have been $(\sigma_i(z) - \sigma_i(z)) = 0$, which means this i was in the error vector. We can reconstruct the error vector e by going through all i in this way.

Now to get this error locator polynomial, we will start off with a different representation of $S_w(x,z)$ which looks different, but is in essence the same. Let's first take a look at the sum of fractions in general. We can combine a sum of fractions into one single fraction. For example, let's take the sum of some random fractions $\frac{1}{2}$, $\frac{1}{3}$ and $\frac{1}{5}$. We can combine this into one fraction by multiplying all *denominators* as the common denominator and the *numerator* becomes a sum of each original numerator times all the other denominators. We can write our example fractions as

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{5} = \frac{1 \cdot 3 \cdot 5}{2 \cdot 3 \cdot 5} + \frac{1 \cdot 2 \cdot 5}{2 \cdot 3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{2 \cdot 3 \cdot 5} = \frac{1 \cdot 3 \cdot 5 + 1 \cdot 2 \cdot 5 + 1 \cdot 2 \cdot 3}{2 \cdot 3 \cdot 5} = \frac{\sum_{j \in \{2,3,5\}} 1 \cdot \prod_{i \neq j} i}{\prod_{i \in \{2,3,5\}} i}. \tag{3.40}$$

We see that the new numerator of $\frac{1}{2}$ is the multiplication of all the other denominators 3 and 5. In the same fashion, the numerators of $\frac{1}{3}$ and $\frac{1}{5}$ become the multiplication of the other denominators. If the numerators are binary variables instead of simply 1, we get the following.

$$\frac{c_2}{2} + \frac{c_3}{3} + \frac{c_5}{5} = \frac{c_2 \cdot 3 \cdot 5}{2 \cdot 3 \cdot 5} + \frac{c_3 \cdot 2 \cdot 5}{2 \cdot 3 \cdot 5} + \frac{c_5 \cdot 2 \cdot 3}{2 \cdot 3 \cdot 5} = \frac{c_2 \cdot 3 \cdot 5 + c_3 \cdot 2 \cdot 5 + c_5 \cdot 2 \cdot 3}{2 \cdot 3 \cdot 5} = \frac{\sum_{j \in \{2,3,5\}} c_j \cdot \prod_{i \neq j} i}{\prod_{i \in \{2,3,5\}} i} \tag{3.41}$$

What happens if one of the binary values is 0? If we take $c_2 = 1$, $c_3 = 1$, $c_5 = 0$, then

$$\frac{1}{2} + \frac{1}{3} + \frac{0}{5} = \frac{1 \cdot 3 \cdot 5}{2 \cdot 3 \cdot 5} + \frac{1 \cdot 2 \cdot 5}{2 \cdot 3 \cdot 5} + \frac{0 \cdot 2 \cdot 3}{2 \cdot 3 \cdot 5} = \frac{1 \cdot 3 \cdot 5 + 1 \cdot 2 \cdot 5}{2 \cdot 3 \cdot 5} = \frac{1 \cdot 3 + 1 \cdot 2}{2 \cdot 3}. \tag{3.42}$$

We see that the denominator that belongs to c_5, which in this case was 5, appeared in every term of the numerator and appeared in the denominator, so we could cancel it out. If we want to incorporate this cancellation in the product representation in 3.40, we can use c_i as the exponent. This looks like the following.

$$\frac{\sum_{j=2}^{5} c_j \cdot \prod_{i \neq j} i}{\prod_{i \in \{2,3,5\}} i} = \frac{\sum_{j \in \{2,3,5\}} c_j \cdot \prod_{i \neq j} i^{c_i}}{\prod_{i \in \{2,3,5\}} i^{c_i}} \tag{3.43}$$

Let's see why this is true. If $c_i = 0$, then the term i appears in all summands (entries of the sum) and in the denominator, which means we can cancel them out. To cancel this i in the product, we want it to become 1, because multiplication with 1 has no effect. This is exactly the case for i^{c_i}, because

$$i^{c_i} = \begin{cases} i & \text{if } c_i = 1 \\ 1 & \text{if } c_i = 0 \end{cases}. \tag{3.44}$$

To confirm this, let's apply it to the example fractions from 3.41 and again take $c_2 = 1$, $c_3 = 1$, $c_5 = 0$.

$$\frac{\sum_{j\in\{2,3,5\}} c_j \cdot \prod_{i\neq j} i^{c_i}}{\prod_{i\in\{2,3,5\}} i^{c_i}} = \frac{c_2 \cdot 3^{c_3} \cdot 5^{c_5} + c_3 \cdot 2^{c_2} \cdot 5^{c_5} + c_5 \cdot 2^{c_2} \cdot 3^{c_3}}{2^{c_2} \cdot 3^{c_3} \cdot 5^{c_5}} =$$
$$\frac{1 \cdot 3^1 \cdot 5^0 + 1 \cdot 2^1 \cdot 5^0 + 0 \cdot 2^1 \cdot 3^1}{2^1 \cdot 3^1 \cdot 5^0} = \frac{1\cdot 3\cdot 1 + 1 \cdot 2 \cdot 1}{2 \cdot 3 \cdot 1} = \frac{1 \cdot 3 + 1 \cdot 2}{2 \cdot 3} \tag{3.45}$$

Using this logic, we can write $S_w(x, z)$ from 3.37 as

$$S_w(x,z) = \sum_{i=0}^{n-1} \frac{e_i}{x - \sigma_i(z)} = \frac{\sum_{j=0}^{n-1} e_j \prod_{i\neq j}(x - \sigma_i(z))^{e_i}}{\prod_{i=0}^{n-1}(x - \sigma_i(z))^{e_i}}. \tag{3.46}$$

Note that the denominator of $S_w(x, z)$ is now the error locator polynomial from 3.39.

$$\epsilon(x,z) = \prod_{i,e_i\neq 0} (x - \sigma_i(z)) = \prod_{i=0}^{n-1} (x - \sigma_i(z))^{e_i} \tag{3.47}$$

There is actually an even directer connection between $S_w(x, z)$ and the error locator polynomial. To show this, we take the derivative of the error locator polynomial. As the error locator polynomial is a product of a lot of $(x-\sigma_i(z))$ terms, we have to apply the product rule. In general, the derivative of a product with respect to x is the following.

$$\frac{d}{dx}\Big[\prod_{i=1}^{k} f_i(x)\Big] = \sum_{i=1}^{k} \Big(\frac{d}{dx} f_i(x) \prod_{j\neq i} f_j(x)\Big) \tag{3.48}$$

If we apply this to the error locator polynomial as represented in 3.47, then

$$\frac{d}{dx}\Big[\prod_{i=0}^{n-1}(x - \sigma_i(z))^{e_i}\Big] = \sum_{i=0}^{n-1} \frac{d}{dx}\Big((x - \sigma_i(z))^{e_i}\Big) \prod_{j\neq i} (x - \sigma_j(z))^{e_i}. \tag{3.49}$$

In general, the derivative of $(x - \sigma_i(z))^{e_i}$ with respect to x is

$$\frac{d}{dx}\Big((x - \sigma_i(z))^{e_i}\Big) = e_i \cdot (x - \sigma_i(z))^{e_i - 1} \cdot \frac{d}{dx}(x - \sigma_i(z)) = e_i \cdot (x - \sigma_i(z))^{e_i-1} \cdot 1. \tag{3.50}$$

Note that the derivative of $(x - \sigma_i(z))$ with respect to x equals 1, because the derivative of x equals 1 and the derivative of $\sigma_i(z)$ with respect to x equals 0, as there is no x present. As all e_i are binary values, we can make the following distinction.

$$e_i \cdot (x - \sigma_i(z))^{e_i-1} = \begin{cases} 1 \cdot (x - \sigma_i(z))^0 = 1 & \text{if } e_i = 1 \\ 0 \cdot (x - \sigma_i(z))^{-1} = 0 & \text{if } e_i = 0 \end{cases} \tag{3.51}$$

which means that

$$\frac{d}{dx}\Big((x-\sigma_i(z))^{e_i}\Big) = e_i. \tag{3.52}$$

Now we can fill it into equation 3.49 and we get

$$\sum_{i=0}^{n-1}\frac{d}{dx}\Big((x-\sigma_i(z))^{e_i}\Big)\prod_{j\neq i}(x-\sigma_j(z))^{e_i} = \sum_{i=0}^{n-1} e_i \prod_{j\neq i}(x-\sigma_j(z))^{e_i}. \tag{3.53}$$

Coincidentally, this is the numerator of our $S_w(x,z)$ in 3.37. This means that we can write $S_w(x,z)$ in terms of our error locator polynomial as follows.

$$S_w(x,z) = \frac{\sum_{j=0}^{n-1} e_j \prod_{i\neq j}(x-\sigma_i(z))^{e_i}}{\prod_{i=0}^{n-1}(x-\sigma_i(z))^{e_i}} = \frac{\frac{d}{dx}\epsilon(x,z)}{\epsilon(x,z)} \tag{3.54}$$

Now we will use a trick to show how we can compute the error locator polynomial $\epsilon(x,z)$ from only $S_w(x,z)$. The first step is to write $\epsilon(x,z)$ in terms of even and odd powers of x, which can be done for any polynomial. For example, let's take $R(x) = x^7 + x^5 + x^4 + x^2 + x + 1$. We can write this as

$$R(x) = (x^4 + x^2 + 1) + (x^7 + x^5 + x) = (x^4 + x^2 + 1) + x(x^6 + x^4 + 1). \tag{3.55}$$

Note that we work with binary coefficients and thus $(x+1)^2 = x^2 + 1^2$. For more general coefficients $c, d \in \mathbb{F}_{2^m}$ it holds that $cx^2 + d = (\sqrt{c}x + \sqrt{d})^2$ and $\sqrt{c}$ and $\sqrt{d}$ exist in $\mathbb{F}_{2^m}$. If all powers are even, we can take the square root by dividing all powers by 2 and taking square roots of the coefficients. This means that we can rewrite $R(x)$ as

$$R(x) = (x^4+x^2+1) + x(x^6+x^4+1) = (x^2+x+1)^2 + x(x^3+x^2+1)^2 = A^2(x) + xB^2(x). \tag{3.56}$$

This way we can write any polynomial in terms of $A(x)$ and $B(x)$. If we take the derivative of $R(x)$, we get

$$\frac{d}{dx}R(x) = \frac{d}{dx}(A^2(x) + xB^2(x)) = 2\cdot A(x)\cdot\frac{d}{dx}A(x) + x\cdot\frac{d}{dx}B^2(x) + B^2(x)\cdot\frac{d}{dx}x \equiv$$
$$0 + 2\cdot B(x)\cdot\frac{d}{dx}B(x) + B^2(x) \equiv B^2(x) \tag{3.57}$$

This is because multiplying any polynomial by 2 is the same as multiplying all coefficients with 2, e.g. $2\cdot(x^2+x+1) = 2\cdot x^2 + 2\cdot x + 2\cdot 1$ and our coefficients are binary, which means that $2 \equiv 0$ and $2\cdot x^2 + 2\cdot x + 2\cdot 1 \equiv 0$. This is why in 3.57 all the multiplications by 2 are directly congruent to 0.

As this is applicable to any polynomial, we can also write $\epsilon(x,z)$ in terms of $A(x,z)$ and $B(x,z)$ ordered by powers of x. So

$$\epsilon(x,z) = A^2(x,z) + xB^2(x,z) \text{ and } \frac{d}{dx}\epsilon(x,z) = B^2(x,z). \tag{3.58}$$

Then we can write $S_w(x,z)$ as

$$S_w(x,z) = \frac{\frac{d}{dx}\epsilon(x,z)}{\epsilon(x,z)} \equiv \frac{B^2(x,z)}{\epsilon(x,z)} \implies \epsilon(x,z)\cdot S_w(x,z) \equiv B^2(x,z). \tag{3.59}$$

As we wrote $\epsilon(x,z)$ as $A^2(x,z) + xB^2(x,z)$, we can substitute and rewrite it in terms of $A(x,z)$ and $B(x,z)$ as follows.

$$\epsilon(x,z)\cdot S_w(x,z) \equiv B^2(x,z) \implies (A^2(x,z)+xB^2(x,z))S_w(x,z) \equiv B^2(x,z) \implies$$
$$A^2(x,z)+xB^2(x,z) \equiv \frac{1}{S_w(x,z)}B^2(x,z) \implies B^2(x,z)(x+\frac{1}{S_w(x,z)}) \equiv A^2(x,z) \implies$$
$$B(x,z)\sqrt{x+\frac{1}{S_w(x,z)}} \equiv A(x,z) \quad (3.60)$$

Let's take a step back and deduce what we know about $A(x,z)$ and $B(x,z)$. First of all, we introduced them in the definition of our error locator polynomial $\epsilon(x,z) = A^2(x,z) + xB^2(x,z)$. The assumption is that t or fewer errors were introduced. Following from 3.39, our error locator polynomial has a degree of t or lower, as we take the product of t elements containing an x. Note that we should not apply the modulus $g(x,z)$ on this polynomial, as that would reduce the polynomial to a lower degree when there are t errors. Then the degree of $A(x,z)$ is half the degree of the highest even power in $\epsilon(x,z)$ and the degree of $B(x,z)$ is half the degree of the highest odd power of $\epsilon(x,z)$ minus 1, as we divided the odd powers of $\epsilon(x,z)$ by x to get $B^2(x,z)$. As the highest possible degree is t, we get that

$$\text{the degree of } A(x,z) \leq \left\lfloor \frac{t}{2} \right\rfloor \text{ and the degree of } B(x,z) \leq \left\lfloor \frac{t-1}{2} \right\rfloor . \quad (3.61)$$

If we calculate $V(x,z) = \sqrt{x + \frac{1}{S_w(x,z)}}$ then $B(x,z)V(x,z) \equiv A(x,z) \bmod g(x,z)$ according to equation 3.60. Note that $S_w(x,z) \in \mathbb{F}_{2^m}[x]/g(x,z)$. We know that squaring a polynomial in $\mathbb{F}_{2^m}[x]/g(x,z)$ is the same as squaring each individual power of x and its coefficient. This means we can calculate the square root by taking each power of x separately and calculating the square root of the x power and the square root of the coefficient separately, because

$$\sqrt{a\cdot b} = \sqrt{a}\cdot\sqrt{b}. \quad (3.62)$$

As we know $S_w(x,z)$ and $g(x,z)$, we can calculate $V(x,z)$ and use the extended Euclidean algorithm on $V(x,z)$ and $g(x,z)$ so that we get the following equation in every round of the algorithm.

$$A'(x,z) = B'(x,z)V(x,z) + H(x,z)g(x,z) \quad (3.63)$$

We continue the algorithm until the degrees for $A'(x,z)$ and $B'(x,z)$ conform to the degrees in 3.61. Once we have found the right $A(x,z)$ and $B(x,z)$, we can calculate the error locator polynomial $\epsilon(x,z)$ according to equation 3.58. Now we apply the error locator polynomial and find the flipped bits as explained at the start of this section. Correcting the error simply requires flipping the flipped bits again.

Chapter 4

McEliece Cryptosystem

This chapter focuses on the first cryptographic system that could be used in combination with any linear code [6]. It first covers how encryption and decryption work and concludes with an analysis of the public and private information.

4.1 Encryption

In general, messages sent using a public key cryptographic system should be hard to decrypt without private information and easy to encrypt. If we want to use coding theory, we can have the following setting. Assume that G is the generator matrix of a code with an efficient decoding algorithm, such as the binary Goppa code in the previous section. Now we can multiply a message M by the generator matrix G to create a code word c and then add some arbitrary error vector e. The ciphertext T_{Mc}[1] then becomes

$$T_{Mc} = M \cdot G + e. \tag{4.1}$$

The fast decoding algorithm now allows the receiver to decrypt the message M efficiently. However, this would not provide any security, since an attacker could use the same algorithm and decode the message. We need to do something so that the fast decoding algorithm cannot be used. This is exactly what the *McEliece Cryptosystem* tries to achieve. The McEliece cryptosystem hides the nice code representation G by multiplying G by some matrices and publishing the result as the public key. One matrix, S_{Mc}, is random and invertible. Another matrix, P_{Mc}, is a matrix that permutes rows. An example of a P_{Mc} matrix is the following.

$$P_{Mc} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.2}$$

This becomes clear if we take some vector $v = \begin{pmatrix} a & b & c & d \end{pmatrix}$ and multiply it with P_{Mc}.

$$P_{Mc} \cdot v^T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} b \\ d \\ a \\ c \end{pmatrix} \tag{4.3}$$

Permutation matrices are always square and contain exactly one 1 in each row and column, which means that is has a number of 1s equal to its dimensions. Our example matrix has dimensions 4×4, so there are four 1s. We can now use S_{Mc} and P_{Mc} to randomly shuffle and permute G as follows. The result is our public key G'.

[1] Mc stands for McEliece, as we will use the same letters in the next section about the Niederreiter system

$$G' = S_{Mc} \cdot G \cdot P_{Mc} \tag{4.4}$$

This construction perfectly hides G. Since G is a $k \times n$ matrix, S_{Mc} is a $k \times k$ matrix and P_{Mc} is an $n \times n$ matrix. For the McEliece system, the ciphertext T_{Mc} is generated by substituting G' for G in 4.1.

$$T_{Mc} = M \cdot G' + e \tag{4.5}$$

4.2 Decryption

First we will cover how a legitimate receiver can decrypt the ciphertext. Attacks without this knowledge will be covered later on. To decrypt T'_o with knowledge of P_{Mc} and S_{Mc}, we first multiply T'_o by the inverse of permutation matrix P_{Mc}, which always exists. Then we get

$$T'_o \cdot P_{Mc}^{-1} = (M \cdot S_{Mc} \cdot G \cdot P_{Mc} + e) \cdot P_{Mc}^{-1} = MS_{Mc}G + eP_{Mc}^{-1}. \tag{4.6}$$

As P_{Mc}, and therefore also P_{Mc}^{-1}, only permutes the entries of e, the Hamming weight (number of 1s) of e does not change when it is multiplied by P_{Mc}^{-1}. As the number of errors stays the same, the decoding algorithm can remove e in exactly the same fashion as eP_{Mc}^{-1}.

In general, a decoding algorithm Δ retrieves the nearest code word c from a vector of the from $w = c + e$, where e is some error vector of weight t. We can then retrieve the message M from the code word. Recall that if the generator matrix is in systematic form, the message is the first k bits of the code word. As shown in the first chapter, regular decoding can be done by syndrome decoding to get the error vector (and therefore the code word) and taking the first k bits to retrieve the message, but some codes have specific fast regular decoding algorithms. For now, let's call the regular decoding algorithm Δ.

$$\Delta(MG + e) = M \tag{4.7}$$

If we use the decoding algorithm that belongs to the respective code on $T' \cdot P_{Mc}^{-1}$ from 4.6, we will retrieve $M \cdot S_{Mc}$. Note that $M \cdot S_{Mc}$ is a vector of length k. As S_{Mc} was created to be invertible, we can now retrieve M.

$$\Delta(T_{Mc} \cdot P_{Mc}^{-1}) \cdot S_{Mc}^{-1} = \Delta(MS_{Mc} \cdot G + eP_{Mc}^{-1}) \cdot S_{Mc}^{-1} = MS_{Mc} \cdot S_{Mc}^{-1} = M \tag{4.8}$$

4.3 Keys

4.3.1 General Codes

In the McEliece system for general codes, the *public information* is

1. The scrambled $k \times n$ generator matrix matrix G'
2. The plain text length k
3. The vector length n
4. The error correction capacity t
5. The exponent m

The *private information* is

1. The random invertible $k \times k$ matrix S_{Mc}
2. The $n \times n$ permutation matrix P_{Mc}
3. An efficient decoding algorithm for G

We have to add information on how to decode G in the private information, because there are many different codes, so it has to be clear which one was used and thus how to decode G. Note that it suffices to have just G' and t as the public key, as the rest can be deduced. However, we can do better than this for binary Goppa codes.

4.3.2 Binary Goppa Codes

Recall that our decoding algorithm needs $g(x,z)$ and the support σ to work. Instead of permuting the rows of the generator matrix, we can permute our support σ, because the n columns of the parity check matrix are then permuted, which results in the permutation of rows in the generator matrix, so by applying a permutation to σ, we have no need for P_{Mc}. An even simpler way however, is taking a random support. Additionally, converting the parity check matrix to systematic form implicitly replaces S_{Mc}. The great part is that we do not need to save the permutation matrix, because we can just keep the random support σ secret. Additionally, as G is now in systematic form, we can omit the identity part for the key. As $g(x,z)$ is not necessary for encryption, we can keep it secret too. This is essential, because it must be kept secret so that only the receiver can apply the decoding algorithm.

In the McEliece system with *binary Goppa codes*, the *public information* is

1. The $k \times n - k$ parity part of the perturbed generator matrix G
2. The plain text length k
3. The vector length n
4. The error correction capacity t
5. The exponent m

The *private information* is

1. the modulus $g(x,z)$
2. The random support σ

One caveat of the McEliece cryptographic system is that the code word contains a direct copy of the message, because the generator matrix is in systematic form. When the errors are introduced, this does become somewhat distorted, but if the message is meaningful, an attacker could try to recover the errors from just the context. Therefore, it is critical that only random strings are sent as messages using the McEliece cryptographic system. The next chapter focuses on a cryptographic system based on the McEliece system, which does not have this problem.

Chapter 5

Niederreiter Cryptosystem

Niederreiter suggested a cryptosystem that was based on the McEliece cryptosystem, but does not use the generator matrix for encryption and instead uses the parity check matrix [7]. The idea is that we send a syndrome instead of an erroneous vector. This chapter first covers how encryption and decryption works and concludes with an analysis on the private and public information.

5.1 Encryption

If H is the parity check matrix for some code, M is a message of length n and weight t (we will use this M in place of the error vector), S_N a random invertible matrix, P_N a permutation matrix and $H' = S_N \cdot H \cdot P_N$, then the cipher text T_N in the Niederreiter system is

$$T_N = H' \cdot M^T = S_N \cdot H \cdot P_N \cdot M^T. \tag{5.1}$$

So T_N is the syndrome. Additionally, since H is an $(n-k) \times n$ matrix, S_N is an $(n-k) \times (n-k)$ matrix and P_N is an $n \times n$ matrix.

5.2 Decryption

We decode the cipher text T'_N by first multiplying it by S_N^{-1}.

$$S_N^{-1} \cdot T'_N = S_N^{-1} \cdot S_N \cdot H \cdot P_N \cdot M^T = H \cdot P_N \cdot M^T \tag{5.2}$$

We can use a syndrome decoder Σ, which retrieves any e' from He'^T. How this is actually done depends on the code, but recall from the first chapter that we could simply use the parity check matrix to decode the syndrome for the example code.

$$\Sigma(He'^T) = e'^T \tag{5.3}$$

Recall from chapter 2 that regular decoding and syndrome decoding are the same problem, so decoding could also be done using a regular decoder. We can retrieve $P_N \cdot M$ from T'_N as follows.

$$P_N^{-1} \cdot \Sigma(S_N^{-1} \cdot T'_N) = P_N^{-1} \cdot \Sigma(H \cdot (P_N \cdot M^T)) = P_N^{-1} \cdot P_N \cdot M^T = M^T \tag{5.4}$$

This works, because the weight of M is at most t and the weight of $P_N \cdot M$ is the same as the weight of M.

5.3 Keys

5.3.1 General codes

In the Niederreiter system with general codes, the *public information* is the following.

1. The scrambled $(n-k) \times n$ parity check matrix H'
2. The plain text length n
3. The error correction capacity t
4. The exponent m (can be deduced using n, t and k)

The *private information* is

1. The random invertible $n-k \times n-k$ matrix S_N
2. The $n \times n$ permutation matrix P_N
3. Information on how to decode H

5.3.2 Binary Goppa Codes

Using binary Goppa codes gives an advantage when combined with the Niederreiter system. We implicitly create the parity check matrix when we create a binary Goppa code. Now we can apply the same reasoning we used in the McEliece system and take a random support σ and put the parity check matrix in systematic form. As the parity check matrix is now in systematic form, we can simply omit the identity matrix part, reducing the key size to an $(n-k) \times k$ binary matrix.

In the Niederreiter system with *binary Goppa codes*, the *public information* is

1. The $(n-k) \times k$ parity part of the perturbed parity check matrix H
2. The plain text length n
3. The error correction capacity t
4. The exponent m (can be deduced using n, t and k)

The *private information* is

1. The modulus $g(x, z)$
2. The random support σ

There are a few reasons to choose Niederreiter over McEliece. Firstly, the ciphertext in the is of length $n-k$ instead of k. Secondly, the ciphertext in the Niederreiter system does not contain a direct copy of the message, which allows for meaningful messages, provided they have weight t or lower. The next chapter looks at the security of the Niederreiter and McEliece cryptosystems and provides recommendations on how to use them.

Chapter 6

Security

This chapter starts by introducing the most famous attacks against both the Niederreiter and the McEliece cryptosystems to get an idea of how robust they are. Additionally, it provides some security notes for real-world applications.

6.1 Attacks

This section covers general attacks against both cryptographic systems, which gives an idea of how strong the systems are. The security of the cryptographic systems that were introduced in the previous chapters depends on the two assumptions that the systems are based on.

1. For an arbitrary linear code, it is hard to find the closest code word to an arbitrary vector.
2. Without knowing the secret parameters, decoding a code word is just as hard as decoding an arbitrary linear code.

The first requirement is conjectured to be NP-Complete for any code for both classic computers and quantum computers [1][5]. The second requirement however, depends on the code that is used. There are many codes for which this is not true, which is why using a secure code is extremely important. A few codes are known to be broken, such as Reed-Solomon codes, concatenated codes, Reed-Muller codes, several Algebraic Geometry (AG) codes, Gabidulin codes, several LDPC codes and some cyclic codes. Most of these codes try to reduce the key sizes, since key sizes highly depend on the code used, but these codes are often prone to efficient attacks, making them unsafe to use. Recently, quasi-cyclic (QC) moderate density parity-check (MDPC) codes have been proposed, which have much smaller key sizes. However, they have not been analysed as much as other codes, which makes it hard to trust them fully just yet. As binary Goppa codes have been analysed for decades, it is still the most trust-inspiring code.

6.1.1 Intuitive Brute Force Attack

As for safe codes, decoding a code word is just as hard as decoding an arbitrary code, let's take a look at how this is done. This attack is also known as a brute force attack, because it exhaustively tries all relevant entries. We will take the Niederreiter setting as our starting point and show that the attack works the same for the McEliece system. First we will cover the most intuitive attack. Assume we have access to

1. the parity check matrix H
2. a syndrome s of the form $s = He^T$

We now want to find e, which will give us the nearest code word as was shown in the chapter Coding Theory. The attack assumes no special structure on H, so, there is no particular fast decoding algorithm without more knowledge. As e is a binary vector where t entries are set to 1, $H'e^T$ is the addition of t columns of H'. This is clarified in the following example. Assume we have a random matrix R and an error vector with 2 entries set to 1.

$$Re^T = \begin{bmatrix} a1 & b1 & c1 & d1 \\ a2 & b2 & c2 & d2 \\ a3 & b3 & c3 & d3 \end{bmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} b1 + d1 \\ b2 + d2 \\ b3 + d3 \end{pmatrix} = \begin{pmatrix} b1 \\ b2 \\ b3 \end{pmatrix} + \begin{pmatrix} d1 \\ d2 \\ d3 \end{pmatrix} \tag{6.1}$$

We can see that this is simply an addition of the columns of the indices in e that are 1. As the second and fourth indices of e are set in our example, the result is the addition of the second and fourth column of R. In the same manner, we can take t columns from H at random and add them. If this is equal to our syndrome, we have found the indices that are set in e and therefore the vector e. This attack works exactly the same for the McEliece system. Assuming we have

1. the generator matrix G
2. some vector w' for which $w' = mG + e$

We can put G in systematic form if that is not yet the case and then generate the parity check matrix H as we saw in the chapter on Coding Theory. We then compute $Hw'^T = H(G^T m^T + e^T) = 0 + He^T = He^T$, because multiplication of a generator matrix by its respective parity check matrix results in the all-zeroes matrix. Now we can use the attack that was just covered. This attack exhaustively tries t out of n columns, which is why the cost is $\mathcal{O}(\binom{n}{t})$ additions of t vectors. However, the attack can be done in a systematic way by taking the same subset of columns and only changing one column in the next iteration. This reduces the cost to $\mathcal{O}(\binom{n}{t})$ additions of 1 column.

6.1.2 Information Set Decoding

The following attack was introduced in 1962 by Eugene Prange [10] and has seen a lot of modifications and optimisations over the years, but the core idea stays the same. Once again we will first show it for the Niederreiter system, which implictly creates an attack for the McEliece system. In the attack we have access to a parity check matrix H and the syndrome $s = He^T$ and we want to retrieve e. We also know that e should have weight t. Prange suggested to randomly permute the given parity check matrix H to some H'.

$$H' = HP \tag{6.2}$$

where P is the random permutation matrix. Recall from the previous section that a random permutation matrix for an $(n - k) \times n$ parity check matrix has the dimensions $n \times n$ and has one 1 in each row and column. We then put the matrix H' in systematic form using some matrix U. Then our matrix H' is

$$H' = UHP. \tag{6.3}$$

It is possible that our permuted matrix H' cannot be put in systematic form. In this case a new permutation matrix should be used. We can now multiply the syndrome s by our matrix U to get

$$Us = UHe^T = H'P^{-1}e^T. \tag{6.4}$$

We know that a permutation matrix does not change the weight of a vector, so $P^{-1}e^T$ also has weight t. We now hope that this permutation has placed all t errors in the last $n - k$ indices of $P^{-1}e^T$, which would mean that $P^{-1}e^T$ is now a vector of length n which contains the t errors in its last $n-k$ positions and zeroes in its first k positions. Us is then the multiplication of the last $n-k$ indices of $P^{-1}e^T$ and the right-most $n-k$ columns of H', which is the identity matrix, because H' is in systematic form. This is clarified in the following equation.

$$Us = H' \cdot (P^{-1}e^T) = \begin{bmatrix} X & | & I \end{bmatrix} \cdot \begin{pmatrix} 0 \\ \vdots \\ 0 \\ Us \end{pmatrix} \tag{6.5}$$

Here X is the left $(n-k) \times k$ submatrix of H', which is cancelled out because of the first k zeroes of $P^{-1}e^T$ and I_k is the $(n-k) \times (n-k)$ identity matrix. If the weight of Us is t, then the t errors are present in Us, so we can use the same tactic we saw in the chapter on Coding Theory to reconstruct the error vector. Namely, note that the length of Us is $n-k$. In order to reconstruct e, which has length n, we prepend k zeroes to Us. It then looks like $e' = (00\dots00)||(Us)^T$, where the double lines indicate concatenation. Note that s is a column vector, so Us is also a column vector. This means that e' also has weight t, as no 1s are added. As H' is in systematic form, we now know in the same way we saw in the chapter Coding Theory that e' is a vector that satisfies

$$Us = H'e'^T \implies s = U^{-1}H'e'^T = HPe'^T. \tag{6.6}$$

To get the error vector e that suffices $s = He^T$, we simply multiply e' by the permutation matrix P we used to construct H'. Following from equation 6.6, we then get

$$e^T = Pe'^T \implies e'^T = P^{-1}e^T \implies s = HPP^{-1}e^t = He^t. \tag{6.7}$$

This is exactly the vector e that we were looking for and therefore solves the problem. As the previous section showed, we can reduce the McEliece problem to the Niederreiter problem and then launch this attack for that cryptographic system too.

The cost of this attack can be analysed as follows. We find a solution if all t errors of e' are present in the identity part. As the identity part is of length $n-k$, the chance of this happening is $\mathcal{O}(\frac{\binom{n-k}{t}}{\binom{n}{t}})$. The cost of this attack is 1 divided by the probability of success and therefore of order $\mathcal{O}(\frac{\binom{n}{t}}{\binom{n-k}{t}})$

6.2 Security Notes

This section delves into the effect that the attacks have had on creating secure parameters and how the cryptographic systems should be used.

6.2.1 Key sizes

As stated before, the information set decoding attack has seen multiple speed-ups over the years, but the main term of the complexity of the keys necessary in the McEliece to achieve 2^b security has remained the same over the years, which is

$$\text{key size} = (c_0 + o(1))b^2(\log_2(b))^2,\ c_0 \approx 0.7418860694 \tag{6.8}$$

as b goes to infinity. In practice we cannot have infinite key sizes, but this is a good approximation for the key sizes necessary. The $o(1)$ denotes a value that approaches 0 as b approaches infinity. The speed-ups that were achieved with the attack papers over the years have increased this $o(1)$, so the attacks are actually more efficient, but it has very little effect on the key sizes that are necessary to achieve a certain level of security. For a large enough b, this can be an improvement of less than $0,001$ percent. However, there is one caviat. The number from 6.8 was based on a pre-quantum analysis. The McEliece system does not have any inherent periodicity properties that can be abused by *Shor's algorithm* [11], which breaks all popular cryptographic systems such as RSA and ECC. However, in general, brute force attacks can be sped up with *Grover's algorithm* [4], which instead of b bits security requires $2b$ bits security to combat. This changes equation 6.8 to

$$\text{post-quantum key size} = (c_0 + o(1))(2b)^2(\log_2(2b))^2 \approx 4(c_0 + o(1))b^2(\log_2(b))^2, \tag{6.9}$$

where c_0 is still the same constant. This requires approximately 4 times bigger key sizes. There have been a couple of propositions for security parameters that are guaranteed to give a certain level of post-quantum security for the McEliece and Niederreiter systems. These can be found in the Classic McEliece NIST submission [2].

6.2.2 How to use the Cryptographic Systems

The previous section covered the so-called school book version of the McEliece and Niederreiter systems. These have mathematical properties that can result in efficient attacks. For example, if we use the school book version of the McEliece system, we can have a situation where one person sends the same message to the same person more than once. If we add these messages we get

$$w' = w_1 + w_2 = mG' + e_1 + mG' + e_2 = (m + m)G + e_1 + e_2 = e_1 + e_2. \tag{6.10}$$

The messages are cancelled out, because in $\mathbb{F}_2$ the addition of the same message results in the all-zero vector. We now have knowledge about the error vectors. For example, we know that e_1 and e_2 both have weight t, so if the weight of w' is $2t$, we know that the error vectors had no coinciding indices with 1s and therefore all the indices that are 0 in w' are 0 in both e_1 and e_2. This means that for all these indices we know mG'. If we invert G' in the column corresponding to these indices, we know the values of m in those indices.

If the weight of w', which we will call W, is smaller than $2t$, we ignore all the indices that are 1 in w' and ignore those indices in G' as well. This results in a new generator matrix G'' and a new w'_1 and w'_2. For the resulting generator matrix and ciphertext, we know that the W errors are ignored, so $2t - W$ errors remain. As these remaining $2t - W$ errors were caused by the overlap of the error vectors, we know that remaining ciphertexts contain exactly $\frac{1}{2}(2t - W) = t - \frac{1}{2}W$ errors each. We solve the decoding problem for the resulting G'' and w'_1, which are a $k \times (n - W)$ generator matrix and a vector of length $n - W$ with $t - \frac{1}{2}W$ errors respectively. This problem is typically much easier to solve.

This is just one example of what can go wrong when a school book version is used. In general, cipher texts of school book cryptographic systems contain too many mathematical properties, which is why usually Key Encapsulation Methods are used. This is not only the case for code-based cryptographic systems, but for *all* cryptographic systems, even RSA and ECC.

The main idea of Key Encapsulation Methods is that instead of sending a message using the public-key cryptosystem, we send an encapsulation of a random string, so that there are no mathematical properties in the cipher texts anymore and no properties of the message (for example headers of emails) can be abused to learn parts of the message. When the random string is decoded by the other party, its hash can be used as a symmetric key for symmetric key encryption, which is much faster and does not have any restrictions on the message to be sent. Note that the attack we just introduced that abused sending the same message, does not work anymore when a KEM is used. The Classic McEliece submission to NIST [2] defined a Key Encapsulation Method using the Niederreiter system and binary Goppa codes. This method uses random strings for every message sent, which mitigates the above-mentioned attack.

Bibliography

[1] E. Berlekamp, R. McEliece, and H. Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384386, 1978. 31

[2] D. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang. Classic McEliece: conservative code-based cryptography. 2017. `https://classic.mceliece.org/`. 2, 34

[3] V. Goppa. A new class of linear error-correcting codes. *Problemy Peredachi Informatsii*, 6 no. 3:24–30, 1970. 9

[4] L. Grover. A fast quantum mechanical algorithm for database search. *28th Annual ACM Symposium on the Theory of Computing*, page 212, 1996. 34

[5] K. Kuo and C. Lu. On the hardnesses of several quantum decoding problems. `http://arxiv.org/abs/1306.5173`, Jul 2013. 31

[6] R. McEliece. A public-key cryptosystem based on algebraic coding theory. Technical report, NASA, 1978. `http://ipnpr.jpl.nasa.gov/progressreport2/42-44/44N.PDF`. 25

[7] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986. 29

[8] N. Patterson. The algebraic decoding of goppa codes. *IEEE Transactions on Information Theory*, 21:203–207, 1975. 20

[9] R. Pellikaan, X. Wu, S. Bulygin, and R. Jurrius. *Codes, cryptology and curves with computer algebra*. Cambridge University Press, 2017. 3

[10] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, IT-8:S5–S9, 1962. 32

[11] P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997. 33