Coding Theory-Based Cryptopraphy: McEliece Cryptosystems in Sage

Christopher Roering
College of Saint Benedict/Saint John's University

Coding Theory-Based Cryptography: McEliece Cryptosystems in Sage

An Honors Thesis

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for Distinction

in the Departments of Mathematics and Computer Science

by

Christopher Roering

May 2013

PROJECT TITLE: Coding Theory-Based Cryptography: McEliece Cryptosystems in Sage

Approved by:

Dr. Sunil Chetty
Assistant Professor of Mathematics, Project Advisor

Dr. Lynn Ziegler
Professor of Computer Science, Project Advisor

Dr. Bret Benesh
Professor of Mathematics, Department Reader

Dr. Robert Campbell
Assistant Professor of Mathematics, Department Reader

Dr. Robert Hesse
Chair, Department of Mathematics
Associate Professor

Dr. Jim Schnepf
Chair, Department of Computer Science
Associate Professor

Dr. Tony Cunningham
Director, Honors Thesis Program

Abstract

Unlike RSA encryption, McEliece cryptosystems are considered secure in the presence of quantum computers. McEliece cryptosystems leverage error-correcting codes as a mechanism for encryption. The open-source math software Sage provides a suitable environment for implementing and exploring McEliece cryptosystems for undergraduate research. Using our Sage implementation, we explored Goppa codes, McEliece cryptosystems, and Stern's attack against a McEliece cryptosystem.

Acknowledgements

I would like to thank Dr. Sunil Chetty and Dr. Lynn Ziegler for being my advisors and Dr. Bret Benesh and Dr. Robert Campbell for reading through my thesis and providing feedback and comments. I would also like to thank John Miller and Josh Trutwin for setting up and reserving a Linux box for my research.

Contents

1 An Introduction to Cryptography

Whenever we transmit sensitive information over a public channel, there is potential that an eavesdropper could intercept this message and steal this sensitive information. Fortunately, modern-day transmissions are protected using the principles of cryptography, the study of transmitting secure messages that may be intercepted by an adversary. The main idea behind cryptography is that the sender selects a message that he or she would like to transmit, applies some sort of encryption process, and transmits this encrypted message across the channel. The receiver obtains the encrypted message, also referred to as a ciphertext, then uses a known decryption process to recover the sender's original message. If an adversary intercepts an encrypted message, he or she will be unable to recover the original message without knowledge of the secret decryption process. The encrypting and decrypting processes usually involve the use of a piece of secret information, known as a private key, which can be used to perform these tasks. One should think of encryption and decryption as inverse operations of one another.

In order to further discuss cryptography, a few important distinctions and and clarifications need to be made. Cryptography is the study of transmitting secure messages, and a cryptosystem is an implementation of a particular cryptographic algorithm. Cryptanalysis refers to the study of finding and exploiting weaknesses in a particular cryptosystem. These are commonly referred to as attacks against a cryptosystem.

Cryptosystems are typically classified as being one of two types: private or public key. There are benefits and drawbacks to each type of cryptosystem, and we will briefly discuss each below. In the following discussion, we will suppose that two friends, Alice and Bob, would like to securely communicate with one another using binary vectors of length n.

1.1 Private Key Cryptography

Although the ultimate focus of this thesis is on public key cryptography, it is important to understand the basics of private key cryptography as well. In private key cryptography, also known as symmetric key cryptography, Alice and Bob share a single piece of information (known as the private key) which is the secret necessary to perform both the encryption and decryption processes. In the case of private key cryptography, a naive observer has virtually no knowledge of the cryptosystem (this will be slightly different in public key cryptography). The only information available to an attacker is intercepted ciphertexts. A quick example of a private key cryptosystem is given below.

1.1.1 Example: One-time Padding Scheme

In one example of a one-time padding scheme, Alice and Bob both agree on a single, random n-bit binary vector p (known as the pad). In this case, p is the private key shared by Bob and Alice. When Alice would like to transmit a message to Bob, she performs the component-wise modulo two addition of p and her message m (note that binary addition (mod 2) is the same as XOR, exclusive or) and transmits the result to Bob. This addition constitutes the entirety of the encryption process. When Bob receives Alice's encrypted message, he uses the same pad p and performs the same component-wise addition of p to the received message. The result is

$$(m \text{ XOR } p) \text{ XOR } p = m,$$

the original message. Since only Alice and Bob know the secret pad, any third party that intercepts the encrypted message will have a difficult time deducing the original message.

One of the primary disadvantages to private key cryptography relates to the difficulty of keeping private keys synchronized. In order to protect the cryptosystem from attacks, the private key is often frequently changed, and the process of agreeing on a private key may need to take place in person. Furthermore, increasing the number of users in this cryptosystem also increases the chances that the system will be broken; that is, private key cryptosystems do not scale well. These difficulties are not present in public key cryptography.

1.2 Public Key Cryptography

Public key cryptography, also known as asymmetric key cryptography, takes a different approach to encrypting and decrypting. In public key cryptography, Alice and Bob each maintain a distinct private key and also a distinct public key. A public key is a piece of information that is published for all parties to see. Thus, Bob and Alice each publish a public key, but they also each keep a single piece of information secret - only Alice knows Alice's private key, and only Bob knows Bob's private key. Note that an individual's public and private key are typically related in some way that facilitates and enables the decryption process.

Suppose Alice wishes to send Bob a message. Alice begins by looking up Bob's public key. Alice then uses Bob's public key to encrypt her message and transmits the result to Bob. Bob receives Alice's transmitted message and uses his secret private key to decrypt the encrypted message and recover Alice's original message. Notice that the information available to an attacker has increased substantially; the attacker now has knowledge of a public key (which is related to the private key used for decryption) in addition to the ciphertexts. In order to ensure security, it must therefore be difficult to derive the private key from the public key. A well-known implementation of public key cryptography, RSA, is provided as an example.

1.2.1 Example: RSA Public Key Encryption

RSA is named after its inventors, Rivest, Shamir, and Adelman [13]. A whole paper could be devoted to exploring RSA, but we will only provide an overview here to illustrate our point. In order for Bob to set up his portion of an RSA cryptosystem, he begins by selecting two large, prime integers, p and q, and computes the product $n = pq$. Bob then picks e coprime to $\phi(n) = (p-1)(q-1)$ such that $1 < e < \phi(n)$, where $\phi(n)$ is Euler's totient function. Bob then publishes as the public key the pair (n, e), but keeps the pair (p, q) as his private key. Notice that, as we had previously suggested, the public key and private key are related, as $n = pq$.

When Alice wants to transmit a secure message to Bob, she begins by looking up Bob's public key from a database containing everyone's public keys. She uses the information (n, e) and encrypts her message m (now represented as an integer) as $m^e \bmod (n)$. Since it is a hard problem to 'undo' modular exponentiation [14], a third party that intercepts this encrypted message will have a difficult time recovering the original message without knowledge of p and q. When Bob receives the message, he uses his secret knowledge of p and q to 'undo' the exponentiation. We will gloss over the specific details here, but the interested reader is referred to [13]. RSA is considered secure due to the difficulty of factoring n into p and q to facilitate the computation of $\phi(n)$. Since RSA is based on the difficulty of factoring large integers, this cryptosystem will be weak in the presence of a quantum computer, as an efficient algorithm for factoring integers on quantum computers is given in [14].

Notice the advantage intrinsic to public key cryptosystems − key management is relatively simple. There is no need to meet to exchange private keys, and the system scales well because new members can be added without impacting existing users. Another noteworthy advantage is the ability to 'sign' messages to ensure the sender is who he/she claims. We will not discuss this in detail here, but the interested reader is referred to [13] for more details. One downside to public key cryptography is that an attacker has more information that can be used for cryptanalysis.

Now that we have a basic understanding of how public and private key cryptosystems work, we can shift our focus to the intended topic of this paper: McEliece cryptosystems.

1.3 McEliece Cryptosystems

A McEliece cryptosystem is a public key cryptosystem that leverages error-correcting codes as a method of encryption. Initially proposed by Robert J. McEliece in [7], this cryptosystem has a number of advantages which will be discussed later in detail. Among these advantages is the ability of McEliece cryptosystems to resist cryptanalysis, especially in a quantum computer setting. This property is especially of interest because some modern-day public key cryptosystems, including the aforementioned RSA, can be broken using quantum computers. It is for this reason that McEliece cryptosystems are important in both theory and real-world application. But before we can delve into a thorough discussion of McEliece cryptosystems, we must first discuss and understand the notion of error-correcting codes.

2 An Introduction to Coding Theory

The field of coding theory encompasses the study of information preservation, information compression, and more. In this section, we will focus our attention on the information preservation (error-detection and error-correction) aspect of coding theory. Specifically, we will cover the topics that are most relevant to understanding the implementation of a McEliece cryptosystem. Following the example of Walker in [17], we will begin by defining a code.

Definition 2.1. *A **code** $\mathcal{C}$ over a finite field $\mathcal{A}$ is a subset of $\mathcal{A}^n := \mathcal{A} \times \mathcal{A} \times \cdots \times \mathcal{A}$ (n copies). The **length** of $\mathcal{C}$ is defined as n. The field $\mathcal{A}$ is referred to as the **alphabet** of $\mathcal{C}$.*

Example 2.2. *The set $\Gamma = \{000, 101, 011\}$ is a code over alphabet $\{0, 1\}$ with length 3.*

Although we will restrict our definition of an alphabet to be a finite field and will exclusively consider the case $\mathcal{A} = \mathbb{F}_2$ for the remainder of this thesis, the choice of $\mathcal{A}$ is somewhat arbitrary. Walker mentions in [17] that $\mathcal{A}$ could even be a ring in some cases. But with our definition of a code in mind, we can now discuss some terminology that will be used when working with codes.

Definition 2.3. *A* ***word*** *w is an element of the set $\mathcal{A} \times \mathcal{A} \times \cdots \times \mathcal{A}$ (n copies).*

Definition 2.4. *A* ***codeword*** *c in a code $\mathcal{C}$ is word such that $c \in \mathcal{C}$.*

The difference between a word and a codeword in a code is subtle. A codeword is simply an element of the code. A word can be thought of as any element that has the right length and alphabet to be a codeword, but is not necessarily an element of the code. The reason for this distinction will become clearer later in this paper.

Example 2.5. *Using the same Γ as in the first example, neither* 014 *(wrong alphabet) nor* 0001 *(wrong length) are words in Γ. The string* 010 *is a word in Γ, but not a codeword in Γ. The string* 011 *is both a word and a codeword in Γ.*

We now have terminology that we can use to describe the notion of a code as well as terms to describe what it means to be a member of the code. We now turn our attention to describing methods of classifying the codes.

Definition 2.6. *If a code $\mathcal{C}$ is a vector subspace of $\mathcal{A}^n$, $\mathcal{C}$ is called a* ***linear code*** *with alphabet $\mathcal{A}$.*

Definition 2.7. *The* ***dimension k*** *of a linear code $\mathcal{C}$ is defined as the dimension of $\mathcal{C}$ as a vector space over $\mathcal{A}$.*

Example 2.8. *The code $\Gamma = \{110, 011, 101, 000\}$ is a linear code with dimension $k = 2$ because the code (viewed as a vector space) has basis $\{110, 011\}$.*

The sole focus of this paper will be on linear codes. Recall that the dimension of a vector space can be defined as the number of elements in a basis for the space. Then if $\mathcal{A} = \mathbb{F}_q$ (a field with q elements) and $\dim_{\mathbb{F}_q} \mathcal{C} = k$, we have the relationship $q^k = |\mathcal{C}|$ (the cardinality of $\mathcal{C}$). Also, we would like to emphasize the point that since $\mathcal{C}$ can be viewed as a vector subspace, the vector addition and subtraction of any two words in $\mathcal{C}$ is also a member of $\mathcal{C}$. This point will be important in later discussions and proofs.

Let us now consider a measure that can describe the 'spacing' of a given linear code. We will do so by first defining a metric known as Hamming distance.

Definition 2.9. *Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n) \in \mathcal{A}^n$. The* ***Hamming distance*** *from x to y is defined as:*

$$d_{\text{ham}}(x, y) := |\{i | x_i \neq y_i\}|.$$

Although Walker leaves it as an exercise, we will formally prove that the Hamming distance indeed defines a metric. Note that we will denote the ith index of a vector x by x_i. This notation will be used in the following proof and in the remainder of this paper (unless otherwise specified).

Lemma 2.10. *The Hamming distance defines a metric on $\mathcal{A}^n$. That is, for $x, y, z \in \mathcal{A}^n$:*

(a) $d_{\text{ham}}(x, y) \geq 0$, *with* $d_{\text{ham}}(x, y) = 0$ *iff* $x = y$

(b) $d_{\text{ham}}(x, y) = d_{\text{ham}}(y, x)$

(c) $d_{\text{ham}}(x, y) + d_{\text{ham}}(y, z) \geq d_{\text{ham}}(x, z)$.

Proof. First, we show that $d_{\text{ham}}(x, y) \geq 0$ with $d_{\text{ham}}(x, y) = 0$ if and only if $x = y$. It is trivial to show that the Hamming distance cannot be negative, so the first part follows naturally. We also know that $d_{\text{ham}}(x, y) = 0$ if and only if $x_i = y_i$ for every $0 \leq i \leq n$, which is true if and only if $x = y$.

For the second part, we know that $d_{\text{ham}}(x, y) = d_{\text{ham}}(y, x)$ because the equal sign is symmetric ($x_i = y_i$ if and only if $y_i = x_i$ and similarly for $x_i \neq y_i$).

The third part is a bit more complicated. We want to show that

$$d_{\text{ham}}(x, y) + d_{\text{ham}}(y, z) \geq d_{\text{ham}}(x, z).$$

Consider the inequality component-wise. That is to say, for every $0 \leq j \leq n$, $x_j = z_j$ means that the right hand side of the inequality is zero and the inequality holds trivially. In the case that $x_j \neq z_j$, we also know the inequality holds because y_j cannot be the same as both x_j and z_j. Since the inequality holds component-wise, it holds vector-wise. Thus, the Hamming distance is indeed a metric. □

The notion of a Hamming weight is directly related to the idea behind Hamming distance.

Definition 2.11. *The **Hamming weight** of a word w is defined as $d_{\text{ham}}(w, 0^n)$.*

In most contexts for discussion, we will drop the Hamming descriptor and use the terms distance and weight to describe these ideas. To explore a further relationship between weight and distance in the context of a code, we also need the following terminology.

Definition 2.12. *The **minimum distance d** of a linear code $\mathcal{C}$ is the minimum of the distances between codewords. Formally, this can be described as*

$$d = \min\{d_{\text{ham}}(x, y) | x, y \in \mathcal{C}\}$$

With this definition in mind, Walker leaves the following as an exercise for the reader. As before, we will explicitly prove the result here.

Lemma 2.13. *The minimum distance d of a linear code $\mathcal{C}$ with $\mathcal{A} = \mathbb{F}_2$ is equivalent to the lowest weight of all the codewords in $\mathcal{C}$.*

Proof. Suppose the minimum distance d of a code occurs between the words w and w'. Consider the sum of these words, $z = w + w'$. Since these words are a distance d apart, we know that z has weight d because the (mod 2) addition of w and w' makes the indices at which the words agree vanish. Thus, we have constructed a codeword of weight d. We must now demonstrate that there is no word with weight less than d. Suppose for a contradiction that such a word (call it z') exists. But then the distance between z' and 0^n has weight less than d, a contradiction. □

This is a useful lemma because it is easier to search for the lowest-weight codeword than to compute the distance matrix for this code and then find the minimum of these distances. We now have the terminology to discuss the typical classification of codes.

Definition 2.14. *An $[n, k, d]$ code $\mathcal{C}$ is a linear code with length n, dimension k, and minimum distance d.*

We will use this notation to quickly provide information about a particular linear code. When appropriate, we will truncate the notation to $[n, k]$ and drop the minimum distance information. We can now turn our attention to a few different aspects of linear codes that will be useful in implementing a cryptosystem.

2.1 Properties of Linear Codes.

Now that we have a general way of classifying different linear codes, we can begin to discuss what these properties mean in practical applications.

Generator and Parity-Check Matrices. We mentioned earlier that a linear code is a k-dimensional vector subspace of $\mathcal{A}^n$. This means that codewords in $\mathcal{C}$ are a linear combinations of k basis vectors. Thus, one method of defining a code is through the use of a generator matrix (often denoted by G). This matrix would have k rows and n columns, and codewords would be generated by multiplying row vectors of length k on the right by the generator matrix.

Example 2.15. *The matrix $G = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$ is a generator matrix for an $[n = 5, k = 3]$ linear code $\mathcal{C}$ with $\mathcal{A} = \mathbb{F}_2$.*

To clarify, the elements of $\mathcal{C}$ are precisely the span of the row-space of the generator matrix. An alternative method of defining a code is achieved through the use of a parity-check matrix. A parity-check matrix (often denoted by H), is a matrix with $(n - k)$ rows and n columns whose right null-space defines the linear code $\mathcal{C}$.

Example 2.16. *The matrix $H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$ is a parity check matrix with $(n - k) = 2$ rows and $n = 5$ columns. Thus, H is a parity-check matrix for a $[n = 5, k = 3]$ linear code $\mathcal{C}$ with $\mathcal{A} = \mathbb{F}_2$.*

For a given linear code, notice that in order for the row-space of G to coincide with the null-space of H, it must be the case that $HG^T = 0$. It is an exercise in linear algebra to convert from a generator matrix to a parity-check matrix and vice-a-versa. It is left to the reader to verify that $HG^T = 0$ for the previous two examples and that these matrices therefore yield the same linear code $\mathcal{C}$.

Notice that if we think of our codewords as vectors of length n that encode messages of length k, each codeword contains $(n - k)$ redundant entries and k information symbols. We follow Niebuhr and define the information rate as in [9].

Definition 2.17. *The **information rate** R is defined as $R = k/n$.*

The information rate describes what percentage of the symbols of a message encode actual information. In an ideal situation, we want to maximize the information rate, but in reality, we must also balance this with the desired minimum distance of a code. In other words, the farther apart that we want codewords to be, the more redundant symbols we will require. Walker goes on to discuss just how good a code can be by discussing bounds on linear codes. But a further discussion of the bounds on codes would distract from the task at hand. We will shift our discussion to the error-correcting and detecting properties of linear codes.

2.2 Error Detection and Error Correction

The minimum distance of a code is responsible for perhaps the most interesting property of linear codes. Linear codes can be used to determine whether or not an error occurred in a transmission, and when carefully constructed, can even correct these errors. For the following paragraphs, suppose we are transmitting codewords over a 'noisy' channel that could potentially flip one or more bits in our transmission and that we have an $[n, k, d]$ linear code $\mathcal{C}$.

Error Detection. Suppose we transmit a word $c \in \mathcal{C}$ over the noisy channel and $0 < t < d$ errors occur in the transmission (i.e, t of the n bits were flipped from 1 to 0 or 0 to 1). Since the codeword nearest to c is a distance d away and the received word c' is a distance $0 < t < d$ from c, the received word cannot be a codeword. Thus, the receiver of the transmission will know an error occurred in transmission. Notice the complication that arises when $t \geq d$ errors; it is possible that enough errors have occurred that the received word is indeed the codeword that is a distance d from c. In this case, no error would be detected. Thus, a linear code can detect any $t < d$ bit errors.

Error Correction. In the previous case, we only knew that an error had occurred. We would like to be able to correct the errors as well and determine the transmitted codeword. To see how this works, suppose that $t \leq \lfloor \frac{d-1}{2} \rfloor$ errors occurred. In the same manner as before, we can detect when an error occurs. But now, we can also find the unique nearest codeword. Conceptually, we can define a ball of radius $\lfloor \frac{d-1}{2} \rfloor$ around each codeword. Since the codewords are a distance $d > 2\lfloor \frac{d-1}{2} \rfloor$ apart, we know these balls do not intersect. Thus, each word can be assigned to a unique ball and corresponding codeword. It is in this manner that we can correct the errors incurred during transmission.

It is simple enough to understand how the error-detection may be implemented (check for errors through parity-check matrix multiplication), but the process of locating the nearest codeword for error-detection can be extremely expensive. The brute force approach involves computing a given word's distance from every other element of $\mathcal{C}$ until a codeword of the desired distance from the received word is found. Fortunately, efficient decoding algorithms exist that operate in significantly faster time. In the next section, we will shift our attention to understanding and implementing one of these algorithms, as efficient decoding is a crucial step in the McEliece cryptosystem. But before doing so, we must discuss the class of linear codes to which this specialized decoding algorithm applies.

2.3 Goppa Codes

In McEliece's initial proposal of his cryptosystem, he suggests the use of Goppa codes. This class of codes is of particular interest because some choices of linear codes can lead to insecure cryptosystems [9]. Goppa codes, on the other hand, have thus far been resistant to cryptanalysis. In this section, we discuss the construction of a Goppa code using Bernstein's polynomial approach in [2]. As we proceed, we keep in mind our goal of implementing Goppa codes for the purpose of a McEliece cryptosystem.

2.3.1 Defining a Goppa Code

To begin constructing a Goppa code Γ, one begins by selecting an integer $m \geq 3$ and an integer t satisfying $2 \leq t \leq (2^m - 1)/m$. The value of t will dictate the number of errors which our Goppa code can correct. Fix an integer n that satisfies $mt + 1 \leq n \leq 2^m$. The choice $n = 2^m$ is typical when constructing a Goppa code for a McEliece cryptosystem, but Bernstein points out in [2] that varying the value of n can yield a better security/efficiency trade-off.

Proceed by selecting distinct elements $a_1, a_2, \dots, a_n$ in the finite field $\mathbb{F}_{2^m}$. Notice that when $n = 2^m$, the sequence of elements is the entirety of $\mathbb{F}_{2^m}$. These elements are often referred to as 'code locators'. The last parameter to be chosen is a degree-t polynomial g that is irreducible in $\mathbb{F}_{2^m}[x]$. This $g \in \mathbb{F}_{2^m}[x]$ is known in this context as a Goppa polynomial. Although some g are known to lead to weak cryptosystems (see [6] for one such example), picking an optimal g is an open problem in the field of coding theory.

Now that we have discussed each of the parameters necessary for constructing a Goppa code, we can construct the code using Bernstein's polynomial view of a Goppa code.

Goppa Code (polynomial view). Define the polynomial $h = \Pi_i(x - a_i) \in \mathbb{F}_{2^m}[x]$. In the typical case $n = 2^m$, we have that $h = x^n - x$. Then the set

$$\Gamma = \left\{ c \in \mathbb{F}_2^n \;\middle|\; \sum_i c_i \frac{h}{x - a_i} \bmod g \equiv 0 \right\}$$

is an $[n, \geq n - mt]$ Goppa code. We are also assured by Bernstein in [2] that this code is capable of correcting up to t errors, and he consequently indicates that the minimum distance of Γ is at least $2t + 1$. Notice that the codewords of Γ are exactly the linear combinations of the polynomials $h/(x - a_i)$ that sum to a multiple of g.

To construct a parity-check matrix for this code, one can think of each of the polynomials $h/(x - a_1) \bmod g$, $h/(x - a_2) \bmod g$, $\dots$, $h/(x - a_n) \bmod g$ as an mt-bit column vector by mapping each of the m-bit coefficients of the degree-t polynomials to a single column vector. The matrix with these bit vectors as its columns forms a parity check matrix for Γ with mt rows and n columns. The process of mapping these polynomials to a binary matrix will be explained later in further detail.

2.3.2 Implementing a Goppa Code in Sage

Selecting the values of m, n, and t are simple exercises in choosing parameters satisfying appropriate bounds. But initializing finite fields, polynomial rings, and irreducible polynomials in these rings are not trivial matters. For these functionalities, we turn to Sage [15], the open-source mathematical software application freely available from `http://www.sagemath.org`. Sage has some of these functionalities built-in already, and in other instances, Sage provides the mechanisms to implement solutions. For the remainder of the paper, Sage code will be intertwined with our explanation of theoretical topics. In general, a topic will be described, and the code for implementation in Sage will follow. The full code for the Sage classes used to implement a McEliece cryptosystem can be found in the appendix of this paper.

Finite Fields and Polynomial Rings. Sage has built-in functionalities for constructing finite fields and polynomial rings. To construct a finite field in Sage, use the GF() function and pass as parameters the size of the desired field and a generator symbol. For the case $n = 2^m$, one can easily access all the elements of the finite field by using the elements() or list() functions. Likewise for polynomial rings, one calls the PolynomialRing() function and passes as parameters the field in which the coefficients reside and a symbol for the indeterminate. These objects also provide support for quotient rings.

```
F_2m = GF(n,'Z');
PR_F_2m = PolynomialRing(F_2m,'X');
```

Goppa Polynomial. To construct the Goppa code, we need an irreducible, degree-t polynomial from $\mathbb{F}_{2^m}[x]$. After constructing the polynomial ring $\mathbb{F}_{2^m}[x]$ in Sage, one can employ the random_element() and is_irreducible() functions in tandem to generate a pseudo-random, irreducible Goppa polynomial g.

```
while 1:
        irr_poly = PR_F_2m.random_element(t);
        if irr_poly.is_irreducible():
                break;
g = irr_poly;
```

Generator and Parity-Check Matrices Sage obviously includes support for matrices, but we are more interested in Sage's ability to transform a matrix with entries from a polynomial ring into a matrix which represents these polynomials as binary vectors. In [12], Risse outlines a method of using existing Sage functions to perform such a task. The method leverages Sage's bin() function which converts mathematical objects into a reasonable/useful binary representation. After constructing either a generator or a parity-check matrix, Sage includes functions that make moving from one to the other relatively simple. Since a linear code can be defined by the right kernel of a parity-check matrix, Sage's right_kernel() function for matrices is a simple way of changing representations.

```
#Construct the binary parity-check matrix for the Goppa code.
#Do so by converting each element of F_2^m to its binary
#representation.
H_Goppa = matrix(F2,m*H_check_poly.nrows(),H_check_poly.ncols());
```

```
for i in range(H_check_poly.nrows()):
      for j in range(H_check_poly.ncols()):
            be = bin(eval(H_check_poly[i,j].int_repr()))[2:];
            be = '0'*(m-len(be))+be; be = list(be);
            H_Goppa[m*i:m*(i+1),j] = vector(map(int,be));
```

Now that we have an idea of how to implement a Goppa code in Sage, we can continue our exploration of McEliece cryptosystems by discussing a specific decoding algorithm for Goppa codes.

3 Efficient Decoding Methods

The McEliece cryptosystem is constructed using the NP-hard problem [1] of finding a codeword with minimal Hamming distance to a given word. Since the recipient of an encrypted message has knowledge of the Goppa polynomial used to construct the code, an efficient decoding method is possible. With this knowledge, the sender can efficiently locate the codeword that is within a Hamming distance of t from the encrypted message. One method of performing this efficient decoding is Patterson's Algorithm.

3.1 Patterson's Algorithm

McEliece's inital implementation proposed using Patterson's algorithm to perform efficient decoding. Patterson's algorithm and its proof are described in full detail in [10]. Pseudocode for the algorithm (in the context of linear code decoding) is provided below, following procedures outlined in [2] and [12]. Definitions of terms used will follow after the algorithm is presented.

Algorithm 1 Patterson's Algorithm

```
Input: Syndrome $s(x) \in \mathbb{F}_{2^m}[x]$
Output: Polynomial $\sigma(x) \in \mathbb{F}_{2^m}[x]$
  Set $T(x) = s^{-1}(x) \bmod g(x)$
  if $T(x) = x$ then
    $\sigma(x) = x$
  else
    Set $R(x) = \sqrt{T(x) + x} \bmod g(x)$
    Apply lattice-basis reduction to the lattice generated by the vectors $(R(x), 1)$ and
    $(g(x), 0)$ to obtain a minimum vector, $(\alpha, \beta)$
    Set $\sigma(x) = \alpha^2(x) + x\beta^2(x)$
  end if
```

The roots of $\sigma(x)$ correspond to the locations of errors in a given word. Although perhaps simple in appearance, a few key concepts are important in understanding and implementing Patterson's decoding algorithm in Sage. These concepts include syndrome calculation, an efficient method for calculating the inverse and square root of a polynomial in a quotient ring, and the process of lattice-basis reduction.

3.1.1 Syndrome Calculation

In the literature, the term syndrome is often used to refer to different objects in different contexts, so we will explicitly define it here. A *syndrome*, $s(w) \in \mathbb{F}_{2^m}[x]/g(x)$, for a vector $w \in \mathbb{F}_2^n$, is defined as

$$s(w) = \sum_{i=1}^{n} w_i/(x - a_i) \quad \text{in the field } \mathbb{F}_{2^m}[x]/g(x)$$

where the a_i correspond to the code-locators of the particular Goppa code. Notice that this calculation could be viewed as a matrix multiplication $w * S$ over $\mathbb{F}_{2^m}[x]/g(x)$, where S is a vector containing the values $1/(x - a_i)$. Our Sage implementation for syndrome calculation uses this approach; once the Goppa code is fixed, a SyndromeCalculator matrix is stored and used to quickly compute syndromes for various w.

```
#Construct the polynomial h
h = PR_F_2m(1);
for a_i in codelocators:
        h = h*(X-a_i);

#Construct the syndrome calculator
SyndromeCalculator = matrix(PR_F_2m, 1, len(codelocators));
for i in range(len(codelocators)):
        SyndromeCalculator[0,i] = (X - codelocators[i]).inverse_mod(g);
```

3.1.2 Inverses in a Polynomial Quotient Ring

In order to implement Patterson's decoding algorithm in Sage, we need an efficient method of calculating the inverse of a given polynomial $p(x)$ to the modulus $g(x)$. In order to accomplish this, we will leverage the Euclidean algorithm. Since we chose $g(x)$ to be irreducible over $\mathbb{F}_{2^m}[x]$, it follows that $\gcd(g(x), p(x)) = 1$ as long as we avoid all $p(x)$ that are multiples of $g(x)$. Consequently, there exist polynomials $u(x), v(x) \in \mathbb{F}_{2^m}[x]$ such that

$$u(x)g(x) + v(x)p(x) = 1.$$

As mentioned before, such polynomials can be found using the Euclidean algorithm. If we consider this equation mod $g(x)$, it simplifies to

$$u(x)g(x) + v(x)p(x) \equiv v(x)p(x) \equiv 1 \bmod g(x)$$

so $v(x)$ is the inverse of $p(x)$ mod $g(x)$. Fortunately, we can leverage Sage's already-implemented xgcd function to perform the Euclidean algorithm and consequently compute the inverse of functions to the modulus $g(x)$.

```
def _g_inverse(self, p):
        # returns the g-inverse of polynomial p
        (d,u,v) = xgcd(p,self.goppa_polynomial());
        return u.mod(self.goppa_polynomial());
```

3.1.3 Square Roots in a Polynomial Quotient Ring

In general, it is difficult to determine the square root of a given polynomial $p(x) \in \mathbb{F}_{2^m}[x]$. So to be more specific, we are looking for an efficient algorithm to calculate the square root of a polynomial with a particular structure, that is, a polynomial of the form $p(x) = T(x) + x$. Risse's paper [12] outlines a method for calculating this square root, but a more explicit explanation will be given here to ease the procedure's translation to Sage. The process is as follows.

Calculate the square root of $x \in \mathbb{F}_{2^m}[x]/g(x)$. This is another instance of a special-case square root. As Risse describes, this can be accomplished by first splitting $g(x)$ into even and odd parts (even and odd being a reference to the degree of each term) such that

$$g(x) = g_0^2(x) + xg_1^2(x),$$

where $g_0(x)$ and $g_1(x)$ are the term-by-term square roots of the even and odd terms of $g(x)$, respectively. We are able to do this because $\mathbb{F}_{2^m}[x]$ has characteristic 2. Thus, $g_0(x)$ is the polynomial that results from taking square roots of the even terms, and $xg_1(x)$ is that which results from the odd terms. In his explanation, Risse also includes code for a Sage function that can be used to accomplish the splitting of polynomials into even and odd parts as desired.

```
def _split(self,p):
        # split polynomial p over F into even part po
        # and odd part p1 such that p(z) = p2 (z) + z p2 (z)
        Phi = p.parent()
        p0 = Phi([sqrt(c) for c in p.list()[0::2]]);
        p1 = Phi([sqrt(c) for c in p.list()[1::2]]);
        return (p0,p1);

(g0,g1) = self._split(g);
sqrt_X = g0*self._g_inverse(g1);
```

To be assured our above code accomplishes what we expect, we must now prove the following.

Lemma 3.1. *The square root of $x \in \mathbb{F}_{2^m}[x]/g(x)$ is given by $g_0(x)g_1^{-1}(x)$.*

Proof. Recall that

$$g(x) = g_0^2(x) + xg_1^2(x).$$

Then

$$\begin{aligned} g(x) - g_0^2(x) &\equiv xg_1^2(x) \bmod g(x) \\ g_0^2(x) &\equiv xg_1^2(x) \bmod g(x) \\ g_0^2(x)g_1^{-2}(x) &\equiv x \bmod g(x) \\ (g_0(x)g_1^{-1}(x))^2 &\equiv x \bmod g(x) \end{aligned}$$

Which is what we wished to prove. □

Now that the square root of $x \in \mathbb{F}_{2^m}[x]$ is known, it is now possible to efficiently calculate $\sqrt{T(x) + x} \in \mathbb{F}_{2^m}[x]$.

Compute the square root of $T(x) + x \in \mathbb{F}_{2^m}[x]/g(x)$. Begin as we did in the previous section: split $T(x) + x$ into odd and even parts, $T_0(x)$ and $T_1(x)$, such that

$$T(x) + x = T_0^2(x) + xT_1^2(x)$$

I again claim the following:

Lemma 3.2. *Let $R(x) = T_0(x) + g_0(x)g_1^{-1}(x)T_1(x)$. Then $R(x) \equiv \sqrt{T(x) + x} \bmod g(x)$.*

Proof. Square $R(x)$, keeping in mind that the calculations are simplified by the fact that $\mathbb{F}_{2^m}[x]$ has characteristic 2:

$$R^2(x) = T_0^2(x) + (g_0(x)g_1^{-1})^2T_1^2(x) \equiv T_0^2(x) + xT_1^2(x) \equiv T(x) + x \bmod g(x)$$

which is what we wanted to demonstrate. □

Using this procedure, we can efficiently compute the $\sqrt{T(x) + x} \in \mathbb{F}_{2^m}[x]$.

```
(T0,T1) = self._split(self._g_inverse(syndrome_poly) - X);
R = (T0 + sqrt_X*T1).mod(g);
```

3.1.4 Lattice-Basis Reduction

A thorough discussion of lattice-basis theory is beyond the scope of this paper, so here we will only discuss that which is necessary to implement Patterson's decoding algorithm in Sage. To begin, we first need to define what we mean by a lattice.

Definition 3.3. *A **lattice** L is a finitely generated Abelian group.*

In our case, the Abelian group is ordered pairs of polynomials from $\mathbb{F}_{2^m}[x]$ under component-wise addition. Put another way, our lattice is the integral combination of two fixed polynomial tuples. In our case, the fixed polynomial tuples are $(R(x), 1)$ and $(g(x), 0)$.

In [2], Bernstein uses the term 'norm' to describe the size, in a sense, of a polynomial in this group. To use consistent notation, we will refer to the norm of a polynomial in the same way as Bernstein:

Definition 3.4. *The **norm** of a polynomial $p(x) \in \mathbb{F}_{2^m}[x]$ is given by $|p(x)| = 2^{\deg p(x)}$ if $p(x) \neq 0$ and 0 if $p(x) = 0$. The **length** of $(\alpha, \beta) \in \mathbb{F}_{2^m}[x]$ is defined as $|\alpha^2 + x\beta^2|$.*

The idea behind lattice-basis reduction is somewhat analogous to the Euclidean algorithm; subtract integer multiples of the shorter vector from the longer vector until the resulting vector is of minimal length. The formal algorithm is as follows. See [2] for Bernstein's proof of why exactly one vector satisfies $|(\alpha, \beta)| \leq 2^t$.

Algorithm 2 Lattice-Basis Reduction for Patterson's Algorithm

```
Input: Parameter $t$ and two vectors $(a, b)$ and $(c, d)$ that form a basis for the lattice.
       Without loss of generality, suppose $|(a, b)| \geq |(c, d)|$.
Output: A vector $(\alpha, \beta)$, with $|(\alpha, \beta)| \leq 2^t$.

  Set $(\alpha_0, \beta_0) = (a, b) - \lfloor a/c \rfloor (c, d)$
  if $|(\alpha_0, \beta_0)| > 2^t$ then
    $(\alpha_1, \beta_1) = (c \bmod \alpha_0, 1 - \lfloor c/\alpha_0 \rfloor * \beta_0)$
  else
    Set $(\alpha, \beta) = (\alpha_0, \beta_0)$ and terminate.
  end if
  Set $i = 1$
  while $|(\alpha_i, \beta_i)| > 2^t$ do
    Set $\alpha_{i+1} = \alpha_{i-1} \bmod \alpha_i$
    Set $\beta_{i+1} = \beta_{i-1} - \lfloor \alpha_{i-1}/\alpha_i \rfloor * \beta_i$
    $i = i + 1$
  end while
  Set $(\alpha, \beta) = (\alpha_i, \beta_i)$
```

```
def _norm(self,a,b):
       #This is the way in which Bernstein indicates
       #the norm of a member of the lattice is
       #to be defined.
       X = self.goppa_polynomial().parent().gen();
       return 2^((a^2+X*b^2).degree());

def _lattice_basis_reduce(self, s):
       g = self.goppa_polynomial();
       t = g.degree();

       a = []; a.append(0);
       b = []; b.append(0);
       (q,r) = g.quo_rem(s);
       (a[0],b[0]) = simplify((g - q*s, 0 - q))

       #If the norm is already small enough, we
       #are done. Otherwise, intialize the base
       #case of the recursive process.
       if self._norm(a[0],b[0]) > 2^t:
              a.append(0); b.append(0);
              (q,r) = s.quo_rem(a[0]);
              (a[1],b[1]) = (r, 1 - q*b[0]);
       else:
              return (a[0], b[0]);

       #Continue subtracting integer multiples of the shorter vector from
       #the longer until the produced vector has a small enough norm.
       i = 1;
       while self._norm(a[i],b[i]) > 2^t:
              a.append(0); b.append(0);
              (q,r) = a[i-1].quo_rem(a[i]);
              (a[i+1],b[i+1]) = (r, b[i-1] - q*b[i]);
              i+=1;

       return (a[i],b[i]);
```

3.1.5 Algorithmic Complexity of Patterson's Decoding Algorithm

In order to be assured that Patterson's decoding algorithm is a suitable decoding mechanism for a McEliece cryptosystem, we must consider its algorithmic complexity. Notice that the most computationally intensive steps of our algorithm − taking the inverses and square roots along with the lattice-basis reduction − each utilize the Euclidean algorithm. Thus, understanding the complexity of the this algorithm will be important in our analysis. A reader unfamiliar with the Euclidean algorithm is referred to [4].

In order to analyze the complexity of the Euclidean algorithm as it applies to polynomials, we will begin by considering the worst-case runtime for integers and adapting the results. As Buhler and Wagon point out in [4], the worst case runtime for two integers $a > b$ is when the quotient at each iteration is one. In this case, the sequence of remainders follows the Fibonnaci numbers. Buhler and Wagon therefore conclude that the algorithm takes $\log a$ steps in the worst case since the ratio of two consecutive Fibonnaci numbers approximates the golden ratio.

The worst case for polynomials $a(x)$ and $b(x)$ (with $\deg a(x) \geq \deg b(x)$) follows naturally from this argument. For polynomials, the worst case occurs when the degrees of the remainder polynomials follow the Fibonnaci sequence. By a therefore similar argument, the Euclidean algorithm takes $\log(\deg a(x))$ steps in this case. With this complexity analysis in mind, we can now evaluate the overall complexity for Patterson's decoding algorithm.

Inverses mod g(x). We use the Euclidean algorithm to take inverses, so this step takes $\log(\max(\deg s(x), \deg g(x)))$ time.

Square roots mod g(x). We again leverage the Euclidean algorithm to take square roots, so this step takes $\log(\max(\deg T(x), \deg g(x)))$.

Lattice-basis Reduction. Since our version of lattice-basis reduction is essentially the Euclidean algorithm, we consider this step to take $\log(\max(\deg R(x), \deg g(x)))$ time. Since we terminate the algorithm when the length of the resulting pair of polynomials is less than a given threshold rather continuing until the quotient reaches zero, our bound in this case might be an overestimate. Even so, a logarithmic bound is sufficient for our purposes.

Other steps involved in decoding. After completing Patterson's decoding algorithm, we must also find the roots of $\sigma(x)$ to determine the error locations. Our brute-force search for the roots of $\sigma(x)$ takes n steps by evaluating $\sigma(x)$ at each of the code locators. Thus, depending on the size of n, our search for roots may be the most time-consuming due to the low complexity for the other steps. This will be important to keep in mind when we compare our theoretical analysis with our experimental results.

Now that we know how to implement an efficient decoding algorithm for Goppa codes, we can turn our attention to a full implementation of a McEliece cryptosystem.

4 The McEliece Cryptosystem

The McEliece cryptosystem leverages the error-correcting codes previously discussed as a mechanism for encryption. The idea is that one can intentionally add errors to a codeword to obscure/encrypt the message. The effectiveness of the cryptosystem depends on two hard problems in the field of coding theory: the general decoding problem and the problem of finding codewords of a given weight.

Problem 4.1 (**Nearest Codeword Problem**)**.** *Let C be an $[n,k]$ linear code over $\mathbb{F}_q$ and let $y \in \mathbb{F}_q^n$. Find a codeword $x \in C$ where $d(y,x)$ is minimal.*

In other words, it is difficult to determine the codeword x which is closest to y. This will be the basis for encryption in a McEliece cryptosystem.

Problem 4.2 (**Finding weight w codewords**)**.** *Let C be an $[n,k]$ linear code over $\mathbb{F}_q$ and let $w \in \mathbb{N}$. Find a codeword $x \in C$ where x has weight w.*

The first problem is important for the decryption to be difficult enough to constitute a cryptosystem, and the second is important in that it assures us that a closely related problem is also difficult (we will discuss this in further detail in the section on Stern's attack). But how difficult are these problems? In particular, we need to be assured that they are hard enough to constitute the basis of a secure cryptosystem. Fortunately, in [1], the following theorem is proven:

Theorem 4.3. *The general decoding problem and the problem of finding weights are both NP-hard.*

With this assurance that the problems upon which the McEliece cryptosystem are based are indeed sufficiently difficult, we turn to the details of the system itself. To set up a McEliece cryptosystem, one must select an $[n,k]$ linear code, $\mathbf{G}$, capable of correcting t errors (via an efficient decoding method - Patterson's algorithm in the case of Goppa codes). A disguised binary generator matrix for $\mathbf{G}$ serves as the public key (we will discuss how this 'disguise' is constructed in the next section). The encryption process entails multiplication by the public key and the addition of an error vector to the result. Without knowledge of $\mathbf{G}$, the previous problems highlight the difficulty of decryption. But with knowledge of $\mathbf{G}$ and an efficient decoding algorithm, the receiver can efficiently decrypt the message.

Encoding vs. Encrypting. One important distinction that must be observed is the difference between the words 'encode' and 'encrypt'. The word 'encode' refers to the process of adding redundancy to a word to enable error-correction. The term 'encryption' refers the the process of obscuring a message to be sent. In other words, encoding is a function of a linear code, while encrypting is a function of a cryptosystem. The same applies to the terms 'decode' and 'decrypt'. Admittedly, this difference is rather subtle in a McEliece cryptosystem. But with this distinction in mind, we will now discuss the cryptosystem in further detail.

4.1 A Summary of the McEliece Cryptosystem

Suppose Alice would like to be able to send Bob encrypted messages using a McEliece cryptosystem. Bob must first choose a linear code, generated by a matrix G, with an efficient decoding algorithm, $\mathcal{D}$, for correcting t errors. He also randomly chooses two matrices, S and P. Bob then publishes $G' = SGP$ and t and keeps the rest of the information secret. Alice can now look up Bob's public key and send a message m (a length n binary row vector) by transmitting to Bob the encrypted message:

$$y = mG' + e$$

where e is a random row vector of weight t that Alice re-generates each time she encrypts a message. Upon receiving this message, Bob can decrypt Alice's message by doing

$$\mathcal{D}(yP^{-1}) = \mathcal{D}([mG' + e]P^{-1}) = \mathcal{D}(mSGPP^{-1} + eP^{-1}) = \mathcal{D}(mSG + e') = mSG$$

where $e' = eP^{-1}$. Bob can then solve the resulting matrix for m by doing a multiplication on the right by $(SG)^{-1}$ or another more efficient method of his choice.

We will now discuss the encryption and decryption processes in more detail.

4.2 McEliece Public Key Encryption

To encrypt a message m, the sender transmits:

$$y = mG' + e$$

where G' is the disguised generator matrix for $\mathbf{G}$ and e is a random error vector with weight t. The 'disguise' is necessary because the standard generator matrix for $\mathbf{G}$ can reveal crucial information about $\mathbf{G}$ to an attacker. In order to disguise the generator matrix G for $\mathbf{G}$, we leverage matrix multiplication by the following matrices.

Scrambler Matrix. The scrambler matrix is an invertible matrix that plays a primary role in disguising the structure of G. Notice the effect of S on a particular message m; multiplying m on the left by S before multiplication by G produces a different codeword than mG. We can think of this as a method of hiding the structure of G. In [12], Risse provides a method for generating such a scrambler matrix. The process is as follows: generate an empty binary matrix of the appropriate size, then randomly flip bits in this matrix until the matrix's rank is equivalent to the number of rows (a condition that ensures the matrix is invertible).

```
# Set up the random scrambler matrix
S = matrix(GF(2),k,[random()<0.5 for _ in range(k^2)]);
while (rank(S) < k) :
      S[floor(k*random()),floor(k*random())] +=1;
```

Permutation Matrix. The permutation matrix provides another level of obscurity to the public key. A permutation matrix is a matrix with exactly one non-zero bit in every row and column. In other words, a permutation matrix can be obtained by permuting the columns of an identity matrix. Risse provides a function for computing this permutation matrix as well. The function randomly selects a column, and then sets a random bit in the column that complies with the restrictions of a permutation matrix (only a single one per row and column). Notice that the permutation matrix is clearly invertible by construction.

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

Public Key G'. Thus, the public key published by the receiver is the binary matrix $G' = SGP$ and t, the number of errors which are to be added to an encoded message. According to McEliece in [7], the astronomical number of choices for S and P make an attempt to recover G from G' infeasible.

4.3 McEliece Private Key Decryption

The receiver receives the encrypted message $y = mG' + e$ and must use y along with his private key to recover the message m. With knowledge of S and P, the matrix multiplication can be undone as described previously, and the decoding function $\mathcal{D}$ can be used to remove the error vector.

Private Key. The matrices S and P along with the decoding function $\mathcal{D}$ constitute the private key of a McEliece cryptosystem.

4.4 Advantages and Disadvantages of the McEliece Cryptosystem

If McEliece cryptosystems are so secure, why are they seldom used in modern-day transmissions? In this section, we briefly highlight a few of the advantages and disadvantages of the McEliece cryptosystem to explore why this is the case.

Advantages. The McEliece cryptosystem has a few distinguishing advantages in comparison with other public key cryptosystems. One advantage is that the cryptosystem incorporates an element of randomness in every encryption to improve security; e is a randomly generated error vector. RSA and other modern day cryptosystems do not incorporate such randomness in the encryption process. This element of randomness contributes to the McEliece cryptosystem's current status as a strong candidate for secure post-quantum computer encryption [3]; RSA and other modern-day encryption methods can be easily broken using a quantum computer [14].

Disadvantages. The primary disadvantage of a McEliece cryptosystem lies in the size of the public key. The public key for this cryptosystem (an n x $(n-k)$ matrix) can be expensive to store, especially on a low-capacity device such as a smartphone. This is a major reason why McEliece cryptosystems are not more widely implemented. Further research needs to be done to find ways to compress the size of the public key. See [8] for one possible approach.

5 Attacking the McEliece Cryptosystem

There are two basic approaches to attacking the McEliece encryption: the attacker can either attempt to recover the linear code $\mathbf{G}$ or find a different way to decrypt messages on a ciphertext-by-ciphertext basis. For our discussion of attacks against the McEliece cryptosystem, we will assume that an attacker has access to ciphertexts and the public key.

5.1 Stern's Attack

Stern's attack, originally presented in [16], attempts to decrypt encrypted messages by transforming the nearest codeword problem into one which is easier to solve. Instead of finding the codeword z closest to a given word $y = z + e$, Stern's attack searches for the minimum-weight codeword in a slightly larger code; namely, the code $\mathbf{G} + \{0, y\}$ (this code is formed by appending y as the bottom row of $\mathbf{G}$'s generator matrix). The codeword with minimum weight in this code will be $y - z = e$, the error vector. The attack shares its name with the algorithm which it employs; Stern's algorithm is used to construct codewords of a desired weight. In our case, we are searching for a weight-t codeword in $\mathbf{G} + \{0, y\}$. Stern's probabilistic algorithm operates as follows:

Algorithm 3 Stern's Algorithm for finding weight-w codewords

```
Input: $H$, an $n-k$ by $n$ parity-check matrix for a linear code
       $w$, the weight of the desired codeword
       $p$ and $l$, parameters for Stern's Algorithm
Output: A codeword, $c$, of weight $w$
  1. Randomly select $n-k$ non-singular columns of $H$ and row-reduce them to the
     identity matrix.
  2. Divide the remaining $k$ columns into two subsets, $X$ and $Y$.
  3. Eliminate all but $l$ of the rows of the columns in $X \cup Y$.
  4. For every size $p$ subset, $A$, of $X$, calculate the $l$-bit vector, $\pi(A)$, that is the sum of
     the $p$ columns of $A$. Perform a similar calculation for the size $p$ subsets, $B$, of $Y$.
  5. For every instance $\pi(A) = \pi(B)$ (known as a collision), calculate the $(n-k)$-bit vector,
     $q$, formed by summing the columns in $A \cup B$ (reconsidering the previously deleted rows).
  6. If $q$ has weight $w - 2p$, then form the codeword $c$ of length $n$ and weight $w$ in the
     following manner. Mark as ones the indices of $c$ corresponding to the $2p$ indices of the
     columns in $A \cup B$ and those corresponding to the columns of the identity matrix
     that can be used to form $q$.
  7. If none of the collisions result in a vector of weight $w - 2p$, restart at step 1.
```

This algorithm can be used to decrypt a ciphertext by searching for $c = e$, a codeword of weight t. In order to be assured that Stern's attack produces a correct result, we look to the following lemmas. Other literature cites these facts without a formal proof, but we will explicitly prove these results here.

Lemma 5.1. *The error vector, e, is a codeword in the code $\boldsymbol{G} + \{0, y\}$.*

Proof. Recall that $y = c + e$. We know that for some x, $xG = c$. The generator matrix G' of the code $\mathbf{G} + \{0, y\}$ has the same rows as G, but with y appended to the bottom. But this means that

$$(x, 1)\ G' = (x, 1) \begin{pmatrix} G \\ \hline y \end{pmatrix} = xG + y = c + y = e.$$

Thus, e is a codeword in $\mathbf{G} + \{0, y\}$. □

In order to be assured that Stern's algorithm will not return anything other than the error vector with $w = t$, we apply the following lemma.

Lemma 5.2. *The error vector, e, is the only weight-t codeword in the code $\boldsymbol{G} + \{0, y\}$.*

Proof. Suppose that we have two vectors, e and $e' \in \mathbf{G} + \{0, y\}$, both with weight t. Our goal is to show that $e = e'$. First, recognize that $e \in \mathbf{G} + \{0, y\}$ if and only if $e \in \mathbf{G}$ or $e + y \in \mathbf{G}$. Recall that the minimum distance of $\mathbf{G}$ is $2t + 1$. But this means that $e \notin \mathbf{G}$ because e has weight $t \leq 2t + 1$. Thus, it must be that $e + y \in \mathbf{G}$. Similarly, $e' + y \in \mathbf{G}$. Since these are both codewords in $\mathbf{G}$, it follows that $(y + e) + (y + e') = e + e' \in \mathbf{G}$. But since e and e' both have weight t, $e + e'$ has weight at most $2t$. Because the minimum distance of $\mathbf{G}$ is $2t + 1$, this is only possible if $e + e' = 0$, which implies that $e = e'$. □

Now that we have these lemmas, we can turn our attention to coding Stern's Attack in Sage. Sage has many existing functions which can be leveraged to implement these steps.

Calculating H, the parity-check matrix. Since we are assumed to have access to the public key, calculating a parity-check matrix for the code $\mathbf{G} + \{0, y\}$ can be accomplished in Sage in the following manner. We begin by using Sage's stack function to append y to the bottom of G. After stacking G on top of y, Sage's function for computing the right kernel of this matrix can be used to yield a parity-check matrix for the code.

```
#Calculate a parity check matrix for the code G + {0,y}, where y = mG + e
H = (PK.stack(y)).right_kernel().basis_matrix();
```

Select $n - k$ non-singular columns of H and row-reduce them to I_{n-k}. Sage has the ability to both select subsets of the columns of a matrix and to row-reduce them, but it does not have the built-in ability to row-reduce a subset of the columns of the matrix . Sage's echelon functions can only reduce the first $n - k$ columns of our matrix. To implement this step in Sage, we take a unique approach. Since the right null-space of a matrix is not changed during row-reduction, we can simply use a permutation matrix, P, to permute the columns of H and row-reduce the first $n - k$ columns. However, in order to avoid altering the result, we will need to undo this permutation at the termination of the algorithm. After performing this initial permutation, we check whether or not the selected columns are linearly independent by row-reducing the first $n-k$ columns and comparing the result to the identity matrix.

```
singular = true;
P_Stern = 0; #initialize the permutation matrix
H_Prime = 0; #initialize the permuted parity-check matrix

#Search for (n-k) linearly independent columns.
while singular:
        P_Stern = _GetRandomPermutationMatrix(H_Stern.ncols());
        H_Prime = H_Stern*P_Stern; #permute the matrix
        H_Prime.echelonize(); #row-reduce the first n-k columns

        #If the selected n-k columns do not row-reduce, select a different combination of columns
        if H_Prime.submatrix(0,0,n_k,n_k) == I_n:
                singular = false;
```

Divide the remaining columns into two subsets, X and Y. Sage has a class called Subsets that will compute all size p subsets of a given set. This makes the implementation of this step rather simple. Since the column of a matrix is not hashable (as is required by the Subsets function), we instead compute the subsets of the set of column indices instead of the columns themselves.

```
Subsets_of_X_Indices = Subsets(X_Indices,p);
Subsets_of_Y_Indices = Subsets(Y_Indices,p);
```

Eliminate all but l of the rows in the columns in $X \cup Y$. Sage has built-in functions for deleting rows from a matrix, but we will not delete the rows of the original matrix because the original, unaltered matrix will be useful in later steps. Thus, we first make a copy of H and then delete the rows from the copy. We use this matrix to calculate the $\pi(A)$.

```
#initialize and populate the set of n_k-l rows that will be deleted from H_Prime to leave l rows
Z = set();
while len(Z) < n_k-l:
        Z.add(randint(0,n_k-1)); #H.nrows()=n_k, but indices start at 0

Z = list(Z); Z.sort(); #Make Z a sorted list

#A copy of H_Prime with only the l selected rows (in Z) remaining
H_Prime_l = copy(H_Prime);
H_Prime_l = H_Prime_l.delete_rows(Z);
```

Calculate $\pi(A)$ for every size p subset, A, of X. This is easily implemented in Sage by summing the columns represented in each subset and storing the results of the summation in a list that parallels the one containing the subsets.

```
#Initialize the lists containing each subset's sum
pi_A = list(); pi_B = list();

#Calculate pi(A) for each subset of X
for i in range(Subsets_of_X_Indices.cardinality()):
        column_sum = 0;
        for j in range(p):
                column_sum = column_sum + H_Prime_l.submatrix(0,Subsets_of_X_Indices[i][j],H_Prime_l.nrows(),1);

        pi_A.append(column_sum);

#Calculate pi(B) for each subset of Y
for i in range(Subsets_of_Y_Indices.cardinality()):
        column_sum = 0;
        for j in range(p):
                column_sum = column_sum + H_Prime_l.submatrix(0,Subsets_of_Y_Indices[i][j],H_Prime_l.nrows(),1);

        pi_B.append(column_sum);
```

Sum the columns in $A \cup B$ for each collision. We locate each of the collisions by comparing each of the $\pi(A)$'s against all of the $\pi(B)$'s. When we locate a collision, the computation of of the $(n-k)$-bit vector is simplified by the fact that we stored column indices, so we can simply sum the desired columns of H.

```
#Check each pi(A) value against every pi(B) value to check for collisions
for i in range(len(pi_A)):
        for j in range(len(pi_B)):

        #If a collision occurs, calculate the (n-k)-bit vector computed
        #by summing those columns whose indices are in A U B
        if pi_A[i] == pi_B[j]:
                sum = 0; #initialize the sum vector

                for k in (Subsets_of_X_Indices[i]):
                        sum = sum + H_Prime.submatrix(0,k,H_Prime.nrows(),1);
                for k in (Subsets_of_Y_Indices[j]):
                        sum = sum + H_Prime.submatrix(0,k,H_Prime.nrows(),1);
```

Forming the desired codeword. If the vector formed in step 5 has weight $w - 2p$, we can form the codeword of weight w. To see why this makes sense, consider what we have done. We found $2p$ columns whose sum is an $(n-k)$-bit vector with weight $w - 2p$. By marking as ones the $2p$ column indices in the set $A \cup B$ and the columns of the identity-matrix portion of H corresponding to the sum of the columns in $A \cup B$, this word will have weight $2p + (w - 2p) = w$. Simple looping procedures can be used in Sage to appropriately mark these entries as ones. Recall that we had permuted the columns of H to perform row reduction, so we must also undo this permutation before returning the constructed word.

```
if _GetColumnVectorWeight(sum) == (w-2*p):
        codeword_found = true;
        #Since the sum vector has the appropriate weight, the codeword of weight w can now be calculated
        #And mark the appropriate positions of the codeword as ones
        weight_w_codeword = matrix(GF(2),H_Stern.ncols(),1);
        for index in range(n_k):
                if sum[index,0]==1:
                        weight_w_codeword[index,0] = 1;
        for k in Subsets_of_X_Indices[i]:
                        weight_w_codeword[k,0] = 1;
        for k in Subsets_of_Y_Indices[j]:
                        weight_w_codeword[k,0] = 1;

        #Undo the permuting done when selecting n-k linearly independent columns
        weight_w_codeword = weight_w_codeword.transpose()*(~P_Stern);
        return copy(weight_w_codeword);
```

Appropriate choices for parameters p and l. In [11], some guidance is offered regarding the choice of p and l. Although they do not explicitly specify a general rule for the choice of p, they assert that l is optimized when $l \approx \log \binom{k/2}{p}$. We will follow this heuristic in our exploration, but further guidance on an appropriate choice for both p and l is a topic of further research later in this paper.

Bernstein, Lange, and Peters discuss a potential improvement to Stern's algorithm. The interested reader is referred to [3] for further details because we will not consider this potential improvement here - this is considered a topic for future research. But now that we have an understanding of how each of the steps in Stern's original algorithm can be implemented in Sage, we will proceed to examine the algorithmic complexity of this attack to evaluate its feasibility and effectiveness.

5.1.1 Algorithmic Complexity of Stern's Attack

Although the exact derivation of the algorithmic complexity of Stern's algorithm is beyond the scope of this paper, we will give an overview here based on [16]. Analyzing Stern's attack is difficult due to the random choices present in various steps of the algorithm. Notice that the most expensive steps in one iteration of Stern's attack are the first, fourth, and fifth steps, which correspond to the Gaussian elimination, $\pi(A)$ calculations, and the computations completed when a collision is detected, respectively.

Gaussian Elimination. Stern suggests that the Gaussian elimination performed in the first step of the algorithm requires

$$\frac{1}{2}(n-k)^3 + k(n-k)^2$$

bit operations. Notice that this is the cost for a single Gaussian elimination, and the elimination is only useful if it yields the identity matrix. Thus, it is possible this step may need to be repeated multiple times in order to complete a single iteration of Stern's Attack. This does not affect the formal derivation of complexity, but it is something to keep in mind when we compare our derivation to the actual implementation of the algorithm.

Calculating the $\pi(A)$. In this step, we compute the $\pi(A)$ for each subset of both X and Y. In a single computation, we perform the binary addition of p vectors of length l. Thus, one individual computation takes lp additions. But we do this computation for every size-p subset of X, and we repeat the whole process for the computations of $\pi(B)$. Thus, the total number of binary additions in this step is given by

$$2lp\binom{k/2}{p}$$

Notice that the $\binom{k/2}{p}$ term introduces a factorial into the complexity. This will be important to remember when we optimize p later in this paper.

Computations Resulting From Collisions. The first difficulty in analyzing this step is estimating the number of collisions which will occur. Stern begins by making the simplifying assumption that the π function has a uniform distribution. Thus, we can assume the probability of a single collision is $\binom{k/2}{p}/2^l$. In other words, we assume that there is a $1/2^l$ chance that the produced vector is of the desired form. But the algorithm also needs to compare the sums produced by $\pi(A)$ to all of the sums computed by $\pi(B)$, so the complexity for this step is roughly:

$$\binom{k/2}{p}^2/2^l$$

At each collisions, we must also compute the sum of $2p$ vectors of length $(n-k)$. This yields

$$2p(n-k)\binom{k/2}{p}^2/2^l$$

bit operations. Note that the analysis of this step relies on the the uniform distribution of the π function, which is not necessarily a realistic assumption. This is another factor to keep in mind when we turn our attention to comparing our derivation with our actual implementation.

Combining these three most expensive steps, we arrive at the following complexity for one iteration of Stern's attack:

$$\frac{1}{2}(n-k)^3 + k(n-k)^2 + 2lp\binom{k/2}{p} + 2p(n-k)\binom{k/2}{p}^2/2^l$$

At first glance, it may seem like increasing l and keeping p small would improve runtime by decreasing the number of operations performed in the second and third terms, but this is not necessarily the case. Due to the probabilistic nature of Stern's attack, the value of the parameters must be balanced between the number of operations performed at each step and the probability of encountering a successful configuration. Thus, a more careful approach is required to optimize the actual runtime of the attack. We will consider optimizing the parameters p and l in the following section.

6 Experimental Results

Now that we have a working implementation of a McEliece cryptosystem in Sage and a good understanding of its properties, we can begin to explore and experiment with McEliece cryptosystems. Before reading this section, it may be beneficial to consult the full Sage classes in the appendix of this paper, but it is by no means necessary. The topics we will explore include evaluating the ability of our cryptosystem to handle large parameters and optimizing the parameters of Stern's attack. This is by no means an exhaustive list of experimental topics; there is a high potential for further research in this domain.

6.1 Experimental Runtime of Patterson's Decoding Algorithm

Although Patterson's algorithm is in theory an appropriate tool to accomplish decoding (and therefore decrypting) received words, it is important to assure ourselves that our Sage implementation can decrypt messages in a reasonably fast manner and that it will scale well with increased parameter sizes. To explore the decryption time, we used a Python script to construct McEliece cryptosystems with a variety of Goppa codes and timed the decryption of randomly generated messages. A sample of this script can also be found in the appendix of this paper.

6.1.1 Experimental Parameters

Recall that to construct a Goppa code for a McEliece cryptosystem using Bernstein's construction in [2], we must provide a parameter m and choose an appropriately bounded n, t, and corresponding irreducible polynomial. Since small values of m produce less interesting codes, we will begin by choosing $m = 6$ and increase this value as desired. We use the standard choice of $n = 2^m$. To remain as consistent as possible, we choose values of t that are close to the middle of the required bounds. Bernstein points out that values of t close to the edges of the bounds produce small codes anyway. After selecting these parameters, a number of irreducible, degree-t polynomials will be generated and tested to provide a representative sample size for our experiments. In general, the practice of utilizing multiple polynomials is continued throughout our experiments.

6.1.2 Experimental Results

Since $m \in \{10, 11, 12\}$ yields a cryptosystem that is considered secure for practical purposes, we constructed cryptosystems using a few of these values and many values below as well. We hoped to gain a sense of the realistic order of operations for our implementation of Patterson's algorithm, the most time-consuming step in the decoding process. Below is a table that holds the average times (in CPU seconds) taken to decrypt an encrypted message.

Table 1: Patterson's Algorithm

$m = 6$	$m = 7$	$m = 8$	$m = 9$	$m = 10$	$m = 11$
0.022	0.050	0.113	0.297	0.905	2.965

One benefit to using Patterson's algorithm as our decoding mechanism that is not evident in the above table is the consistency of the runtimes. In other words, examining the raw data produced by the script indicates that there is little to no deviation between decryption times with fixed m. This is a strong benefit because we have a good idea of how much time to allot to a decryption.

But notice the mildly disturbing trend that the table suggests: our implementation of Patterson's algorithm appears to have exponential order. We suspect this because our runtime roughly doubles with an incremented value of m. Although exponential order algorithms are in general considered too expensive to consider, we argue that in this instance, exponential order is sufficient. This is the case because we only need relatively low values of m to reach cryptographic size. Thus, the exponential nature of our decryption process is sufficient for our purposes.

Note that the reason for this exponential growth is likely due to the brute-force search for the roots of $\sigma(x)$ to determine the error locations of a received word. Remember that increasing m by one corresponds to doubling the size of n, the number of code locators. This therefore doubles the time required for the brute-force search. Finding a faster way to compute the roots of $\sigma(x)$ would decrease the runtime for Patterson's decoding algorithm.

6.2 Stern's Attack and Optimization of Parameters

Stern's attack was our primary method of 'breaking' the McEliece cryptosystem. It is important for us to understand at which parameters sizes our attack is feasible and evaluate which parameter values render the attack infeasible. Although we can certainly expect our attack to be workable at intermediate m, Bernstein points out that setting m =10, 11 or 12 is typical for cryptographic applications. Thus, we do not expect Stern's attack to scale nearly as well as Patterson's algorithm. As before, it may be wise to consult the data collection script in the appendix of this paper before proceeding.

6.2.1 Experimental Parameters

Recall that Stern's algorithm takes as parameters a parity-check matrix for the desired code, the weight of the sought-after word, and parameters p and l. Parameter p determines the number of columns to be summed for one calculation of $\pi(A)$ or $\pi(B)$, and l determines the number of rows of each column to be used in this calculation. Since little formal guidance is provided for optimal choices of these parameters, there is a high potential for experimentation in this realm. Our focus here will be two-fold: find the best p for our given choices of m, and find the corresponding optimal value of l. Upon initial inspection, it seems as though the values of these two parameters may be independent, but we will see whether or not this is indeed the case. However, it is important to note that we must also select a value of l to use in conjunction with our experimental p (and vice a versa). Peters suggests in [11] that an appropriate value for l is $l \approx \log \binom{k/2}{p}$. Since we were unable to deduce a reasonable explanation for this suggestion, we will focus on exploring values of l later. For the following discussion of p, we proceeded with Peters' suggestion for l with varying p, choosing the floor of this value as needed to avoid fractions.

6.2.2 Optimizing Parameter p

Preliminary testing revealed that the value of p may have a more significant impact on time than the value of l, so we begin by optimizing this parameter. In [3], Bernstein performs his optimizations with $p = 2$ or $p = 3$, so we began by checking the value of $p = 1$ and built up as was reasonable. After running the data collection script to experiment with $p = 1$ and $p = 2$, the following average runtimes (in CPU seconds) were obtained.

Table 2: Optimizing p

Value of m	$p = 1$	$p = 2$
6	0.427	25.132
7	1.747	117.403
8	593.137	41497.963

First, we point out that our values of m were restricted to 6, 7, and 8 due to their manageability and workable runtimes. Second, notice the extreme difference in times between the $p = 1$ and $p = 2$ results. We did not pursue $p = 3$ or higher due to computational feasibility. For the remainder of our experimentation with Stern's attack, we fixed $p = 1$. Notice the practical implication of $p = 1$; $\pi(A)$ is now the sum of a single column. In other words, collisions are now detected by comparing a column to a column (with l rows, of course).

6.2.3 Optimizing Parameter l

As we mentioned before, Peters suggests that $l \approx \log \binom{k/2}{p}$ is an optimal choice. But how well does this estimate translate to improved runtime? Furthermore, since Peters' optimal value of p differs from our implementation, it is not unreasonable to expect a different optimal l. As before, we begin our search with her suggestion for l, but we also consider values near this suggestion. We will also keep in mind Canteaut and Chabaud's optimal parameters in [5] for a slightly modified version of Stern's attack and see with whose results ours agree. Initially, we considered values within one of Peters' conjecture, but after detecting the potential for a trend, we increased our search in the direction of improvement. Below are the average runtimes (in CPU seconds) for various l:

Table 3: Optimizing l

Value of m	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 6$	$l = 7$	$l = 8$	$l = 9$
6	0.276	0.195*	0.126	0.165	0.224	-	-	-
7	-	3.934	3.344*	2.943	1.743	2.728	-	-
8	-	-	-	426.417	449.100*	275.820	283.518	313.430

The values in the table flagged with asterisks indicate Peters' suggested optimal value of l (using the floor convention described previously) for a particular m. Also, recall that a few data sets were run prior to the one that generated the above table; in those data sets, we found that values of l which were higher than the suggested values yielded faster runtimes for Stern's algorithm. This shifted the emphasis of our search to values of l greater than the suggestion.

Notice the subtle pattern indicated in the above table. Although increasing l indeed initially decreases the runtime of Stern's attack, in each of the above cases, l gets large enough that the runtimes actually increase. This suggests that there might be a range of l which can yield fast results, but performance gains reverse after escaping these bounds. This agrees in principle with our earlier discussion of balancing the computations per step and the probability of success at each step. Nevertheless, our results suggest that $l \approx \log \binom{k/2}{p}$ may not be the optimal choice of l for our specific Sage implementation. Reasons for this difference could include the fact that Peters used a different programming language, or her suggestion might be best-applied to the optimizations she discusses later on in her notes. Ultimately, our results for optimal l (and consequently p) agree precisely with the optimal parameters suggested in [5].

6.3 Comparing Proper Decryption and Stern's Attack

At the core of our experimental research is the question of whether or not decryption using Patterson's algorithm (hereafter referred to as 'proper decryption') is significantly faster than using Stern's attack to recover a particular message. Specifically, we are interested in how well these runtimes scale when parameters approach practical cryptographic size. Based on our results, it is clear that proper decryption scales much better than Stern's attack. At small values of m, Stern's attack appears to be quite feasible; at $m = 6$, human perception can not necessarily even detect the difference in runtimes. But as the size of parameter m increases, the runtime of Stern's attack increases drastically − even with a value of l and p that appears optimal for our implementation. Furthermore, given the rate at which Stern's attack increases with increasing m (both in theory and reality), we can reasonably conjecture that the attack will be rendered infeasible with appropriate parameter sizes. Patterson's algorithm, on the other hand, scales reasonably well with increased parameter size.

7 Conclusions

The goal of this thesis has been to explore the properties of a McEliece cryptosystem and create an implementation in Sage that enables further research. We started out by introducing the terminology and tools necessary to implement a McEliece cryptosystem, and then looked for opportunities to implement these tools in Sage. Fortunately, Sage made implementing these tools relatively easy, since many of the foundational tools − including fields and polynomial rings − had been implemented previously. The only disadvantage to Sage is that the functions are not necessarily optimized for the purposes of a McEliece cryptosystem, and Python is not considered a relatively fast programming language. Nonetheless, our Sage implementation should provide an excellent environment for future explorations of McEliece cryptosystems.

Furthermore, using Goppa codes in conjunction with Patterson's algorithm as the decoding mechanism for decryption appear to be good choices. Goppa codes are relatively easy to understand and construct, and Patterson's decoding algorithm scales reasonably well to cryptographic parameter sizes. Stern's attack, on the other hand, is a good example of an attack on the McEliece cryptosystem that does not scale to cryptographic parameters. Even with our optimized choices of p and l, the attack quickly becomes infeasible.

8 Opportunities for Further Research

Here is a list of topics which highlights some of the opportunities for future research. This is by no means an exhaustive list, but it could serve as a starting point for further theses.

- A McEliece cryptosystem depends on an error-correcting code with an efficient decoding algorithm. In our exploration, we exclusively utilized Goppa codes. Is there a better choice of linear codes for a McEliece cryptosystem?
- Is Patterson's algorithm the best choice of decoding algorithm for Goppa codes? Or do faster algorithms exist?
- Find a faster root-finding algorithm to speed up decoding.
- Does a Goppa polynomial with a particular structure yield better codes for McEliece cryptosystems? Is there an optimal Goppa polynomial?
- In our exploration of a McEliece cryptosystem, we only explored and implemented Stern's attack, one of a plethora of cryptanalytic approaches. Is a different attack more effective against a McEliece cryptosystem?
- One of the major drawbacks to a McEliece cryptosystem is the substantial size of its public key. In what ways can we reduce the size of the public key?
- In [3], Bernstein presents a potential improvement to Stern's attack. How much of a performance gain will these improvements yield for our implementation?
- Although Sage provides many of the tools necessary for implementing a McEliece cryptosystem, Python is a somewhat slower language. Implementing a McEliece cryptosystem using a faster language, such as C or C++, or optimizing the existing Python code might allow for experimentation with higher parameter sizes.

References

[1] Berlekamp, Elwyn R., McEliece, Robert J., and van Tilborg, Henk C. On the inherent intractability of certain coding problems.
IEEE Transactions on Information Theory, 1978. pp. 384-386
`http://authors.library.caltech.edu/5607/1/BERieeetit78.pdf`

[2] Bernstein, Daniel J. List decoding for binary Goppa codes.
Third International Workshop, IWCC, 2011. pp. 62-80
`http://cr.yp.to/codes/goppalist-20110303.pdf`

[3] Bernstein, Daniel J., Lange, Tanja, and Peters, Christiane. Attacking and defending the McEliece cryptosystem.
Post-quantum cryptography: Second International Workshop, 2008. pp. 31-46
`http://cr.yp.to/codes/mceliece-20080807.pdf`

[4] Buhler, Joe and Wagon, Stan. Basic algorithms in number theory.
Algorithmic Number Theory, MSRI Publications, 2008. pp. 25-68
`http://library.msri.org/books/Book44/files/02buhler.pdf`

[5] Canteaut, Anne and Chabaud, Florent. A new algorithm for finding minimum-weight words in a linear code: application to McElieces cryptosystem and to narrow-sense BCH codes of length 511.
IEEE Transactions on Information Theory, 1998. pp. 367-378
`https://www.rocq.inria.fr/secret/Anne.Canteaut/Publications/Canteaut_Chabaud98.pdf`

[6] Loidreau, Pierre and Sendrier, Nicolas. Weak keys in the McEliece public-key cryptosystem.
IEEE Transactions on Information Theory, 2001. pp. 1207-1211
`http://perso.univ-rennes1.fr/pierre.loidreau/articles/ieee-it/Cles_Faibles.pdf`

[7] McEliece, Robert J. A public-key cryptosystem based on algebraic coding theory.
JPL DSN Progress Report, 1978. pp. 114-116
`http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF`

[8] Misoczki, Rafael and Barreto, Paulo. Compact McEliece keys from Goppa codes.
Selected Areas in Cryptography, LNCS 5867. Springer, 2009. pp. 376-392
`http://eprint.iacr.org/2009/187.pdf`

[9] Niebuhr, Robert. Application of algebraic-geometric codes in cryptography.
`http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Robert_Niebuhr.diplom.pdf`

[10] Patterson, Nicholas J. The algebraic decoding of Goppa codes.
IEEE Transactions on Information Theory, 1975. pp. 203-207.

[11] Peters, Christiane, Bernstein, Daniel J., and Lange, Tanja. A successful attack on the McEliece cryptosystem with original parameters.
Applied Computational Algebraic Geometric Modelling, S^3CM, 2009. Presentation notes.
`http://christianepeters.files.wordpress.com/2012/10/20090717-s3cm.pdf`

[12] Risse, Thomas. How sage helps to implement Goppa codes and McEliece PKCSs.
Home page of Thomas Risse, 2011.
`http://www.weblearn.hs-bremen.de/risse/papers/ICIT11/526_ICIT11_Risse.pdf`

[13] Rivest, Ron, Shamir, Adi, and Adleman, Leonard. A method for obtaining digital signatures and public-key cryptosystems.
Communications of the ACM, 1978. pp. 120-126
`http://people.csail.mit.edu/rivest/Rsapaper.pdf`

[14] Shor, Peter. Algorithms for quantum computation: discrete logarithms and factoring.
Symposium on Foundations of Computer Science, 1994.
`http://www.csee.wvu.edu/~xinl/library/papers/comp/shor_focs1994.pdf`

[15] Stein, William A, et al. Sage Mathematics Software (Version X.Y.Z),
The Sage Development Team, 2013.
`https://www.sagemath.org`

[16] Stern, Jacques. A method for finding codewords of small weight.
Coding theory and applications, Volume 388 of *Lecture Notes in Computer Science*, 1989.

[17] Walker, Judy. *Codes and Curves.*
Amer Mathematical Society, 2000.
`http://www.math.unl.edu/~jwalker7/papers/rev.pdf`.

A Sage Code

```
class GoppaCode:

        def __init__(self, n, m, g):
                t = g.degree()
                F2 = GF(2);
                F_2m = g.base_ring(); Z = F_2m.gen();
                PR_F_2m = g.parent(); X = PR_F_2m.gen();

                #Initialize the code locators
                codelocators = [];
                for i in range(2^m-1):
                        codelocators.append(Z^(i+1));
                codelocators.append(F_2m(0));

                #This is the same h to which Bernstein
                #refers in his polynomial view of a Goppa code
                h = PR_F_2m(1);
                for a_i in codelocators:
                        h = h*(X-a_i);

                #Gamma is a list of the polynomials
                #used to determine membership in the code
                gamma = [];
                for a_i in codelocators:
                        gamma.append((h*((X-a_i).inverse_mod(g))).mod(g));

                #Calculate the parity-check matrix (as polynomials)
                #In other words, the first column of this matrix
                #has entries that correspond to the coefficients of
                #the first polynomial of gamma.
                H_check_poly = matrix(F_2m, t, n);
                for i in range(n):
                        coeffs = list(gamma[i]);
                        for j in range(t):
                                # Check to make sure the coefficient exists
                                #(It may not if the polynomial is not of
                                #degree t)
                                if j < len(coeffs):
                                        H_check_poly[j,i] = coeffs[j];
                                else:
                                        H_check_poly[j,i] = F_2m(0);

                #Construct the binary parity-check matrix for the Goppa code.
                #Do so by converting each element of F_2^m to its binary
                #representation.
                H_Goppa = matrix(F2,m*H_check_poly.nrows(),H_check_poly.ncols());
                for i in range(H_check_poly.nrows()):
                        for j in range(H_check_poly.ncols()):
                                be = bin(eval(H_check_poly[i,j].int_repr()))[2:];
                                be = '0'*(m-len(be))+be; be = list(be);
                                H_Goppa[m*i:m*(i+1),j] = vector(map(int,be));

                #Construct the generator matrix for our code by computing
                #a basis for the null-space of H_Goppa. The null-space is,
                #by definition, the codewords of our code.
                G_Goppa = H_Goppa.right_kernel().basis_matrix();

                #Construct the syndrome calculator. This will be used
                #to simplify the calculation of syndromes for decoding.
                SyndromeCalculator = matrix(PR_F_2m, 1, len(codelocators));
                for i in range(len(codelocators)):
                        SyndromeCalculator[0,i] = (X - codelocators[i]).inverse_mod(g);
        #Remember these values
        self._n = n;
        self._m = m;
        self._g = g;
        self._t = t;
        self._codelocators = codelocators;
        self._SyndromeCalculator = SyndromeCalculator;
        self._H_Goppa = H_Goppa;
        self._G_Goppa = G_Goppa;

def Encode(self, message):
        #Encoding a k-bit binary message done by
        #multiplication on the right by the generator matrix.
        return (message*self._G_Goppa);

def _split(self,p):
        # split polynomial p over F into even part po
        # and odd part p1 such that p(z) = p2 (z) + z p2 (z)
        Phi = p.parent()
        p0 = Phi([sqrt(c) for c in p.list()[0::2]]);
        p1 = Phi([sqrt(c) for c in p.list()[1::2]]);
        return (p0,p1);

def _g_inverse(self, p):
        # returns the g-inverse of polynomial p
        (d,u,v) = xgcd(p,self.goppa_polynomial());
        return u.mod(self.goppa_polynomial());

def _norm(self,a,b):
        #This is the way in which Bernstein indicates
        #the norm of a member of the lattice is
        #to be defined.
        X = self.goppa_polynomial().parent().gen();
        return 2^((a^2+X*b^2).degree());

def _lattice_basis_reduce(self, s):
        g = self.goppa_polynomial();
        t = g.degree();

        a = []; a.append(0);
        b = []; b.append(0);
        (q,r) = g.quo_rem(s);
        (a[0],b[0]) = simplify((g - q*s, 0 - q))

        #If the norm is already small enough, we
        #are done. Otherwise, intialize the base
        #case of the recursive process.
        if self._norm(a[0],b[0]) > 2^t:
                a.append(0); b.append(0);
                (q,r) = s.quo_rem(a[0]);
                (a[1],b[1]) = (r, 1 - q*b[0]);
        else:
                return (a[0], b[0]);

        #Continue subtracting integer multiples of
        #the shorter vector from the longer until
        #the produced vector has a small enough norm.
        i = 1;
        while self._norm(a[i],b[i]) > 2^t:
                a.append(0); b.append(0);
                (q,r) = a[i-1].quo_rem(a[i]);
                (a[i+1],b[i+1]) = (r, b[i-1] - q*b[i]);
                i+=1;

        return (a[i],b[i]);
```

```
def Decode(self, word_):
        #We will decode codewords using Patterson's Algorithm.
        g = self._g;
        word = copy(word_);
        X = g.parent().gen();

        #Compute the syndrome necessary for Patterson's Algorithm.
        synd = self._SyndromeCalculator*word.transpose();
        syndrome_poly = 0;
        for i in range (synd.nrows()):
                syndrome_poly += synd[i,0]*X^i

        #Take the necessary square root.
        (g0,g1) = self._split(g); sqrt_X = g0*self._g_inverse(g1);
        (T0,T1) = self._split(self._g_inverse(syndrome_poly) - X);
        R = (T0 + sqrt_X*T1).mod(g);

        #Perform lattice basis reduction.
        (alpha, beta) = self._lattice_basis_reduce(R);

        #Construct the error-locator polynomial.
        sigma = (alpha*alpha) + (beta*beta)*X;

        #For every root of the error polynomial,
        #correct the error induced at the corresponding index.
        for i in range(len(self._codelocators)):
                if sigma(self._codelocators[i]) == 0:
                        word[0,i] += 1;

        return word;

#Accessors
def generator_matrix(self):
        return (self._G_Goppa);

def goppa_polynomial(self):
        return (self._g);

def parity_check_matrix(self):
        return (self._H_Goppa);
```

```
class McElieceCryptosystem:

        def __init__(self, n, m, g):
                #Construct the Goppa code
                goppa_code = GoppaCode(n,m,g);
                assert goppa_code.generator_matrix().nrows() <> 0, "Generator Matrix is empty.";
                k = goppa_code.generator_matrix().nrows();

                # Set up the random scrambler matrix
                S = matrix(GF(2),k,[random()<0.5 for _ in range(k^2)]);
                while (rank(S) < k) :
                        S[floor(k*random()),floor(k*random())] +=1;

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

                #Remember these values
                self._m_GoppaCode = goppa_code;
                self._g = g;
                self._t = g.degree();
                self._S = S;
                self._P = P;
                self._PublicKey = S*(self._m_GoppaCode.generator_matrix())*P;

        #This is a help function which will be useful for encryption.
        def _GetRowVectorWeight(self,n):
                weight = 0;
                for i in range(n.ncols()):
                    if n[0,i] == 1:
                              weight = weight+1;
                return weight;

        def Encrypt(self, message):
                #Assert that the message is of the right length
                assert (message.ncols() == self.public_key().nrows()), "Message is not of the correct length.";

                #Get an error vector, ensuring that there are exactly t errors.
                err_vec = matrix(1,self.goppa_code().generator_matrix().ncols());
                while (self._GetRowVectorWeight(err_vec) < self.max_num_errors()):
                        err_vec[0, randint(1,self.goppa_code().generator_matrix().ncols()-1)] = 1;

                code_word = message*self.public_key();
                return copy(code_word + err_vec);

        def Decrypt(self, received_word):
                assert (received_word.ncols() == self.public_key().ncols()), "Received word is not of the
                     correct length.";

                #Strip off the permutation and decode the received word.
                message = received_word * ~(self._P);
                message = self.goppa_code().Decode(message);

                #Solve the system to determine the original message.
                message = (self._S*self.goppa_code().generator_matrix()).solve_left(message);

                return copy(message);
```

```
    #Accessors
    def public_key(self):
           return copy(self._PublicKey);

    def goppa_code(self):
           return copy(self._m_GoppaCode);

    def max_num_errors(self):
           return copy(self._t);
```

```
#SternsAlgorithm.sage

def _GetRandomPermutationMatrix(n):
        # Set up the permutation matrix
        rng = range(n); P = matrix(GF(2),n);
        for i in range(n):
                p = floor(len(rng)*random());
                P[i,rng[p]] = 1; rng=rng[:p]+rng[p+1:];
        return copy(P);

def _GetColumnVectorWeight(n):
        weight = 0;
        for i in range(n.nrows()):
                if n[i,0] == 1:
                        weight = weight+1;
        return weight;

def SternsAlgorithm(H, w, p, l):
        #H - a parity-check matrix for the code G
        #w - the weight of the codeword for which we are searching
        #p - the size of the subsets of columns (a parameter for Stern's Algorithm)
        #l - the size of the subsets of rows (a parameter for Stern's Algorithm)

        H_Stern = copy(H);
        codeword_found = false;

        #Begin Stern's Algorithm for finding a weight-w codeword of the code generated by H
        while (not codeword_found):
                #########################################################
                #Select n-k columns and row-reduce the resulting matrix
                #       To do this, randomly permute the columns
                #        of H_Stern and row-reduce. In other words, the first
                #       (n-k) columns will be our randomly selected columns
                #########################################################

                n_k = H_Stern.nrows();
                k = H_Stern.ncols() - n_k;
                I_n = identity_matrix(n_k);
                singular = true;
                P_Stern = 0; #initialize the permutation matrix
                H_Prime = 0; #initialize the permuted parity-check matrix

                #Search for (n-k) linearly independent columns.
                while singular:
                        P_Stern = _GetRandomPermutationMatrix(H_Stern.ncols());
                        H_Prime = H_Stern*P_Stern; #permute the matrix
                        H_Prime.echelonize(); #row-reduce the first n-k columns

                        #If the selected n-k columns do not row-reduce, select a different combination of
                             columns
                        if H_Prime.submatrix(0,0,n_k,n_k) == I_n:
                                singular = false;

                #########################################################
                #Select l of the k rows of H_Prime
                #       Z is a set of l row indices of H_Prime
                #########################################################

                #initialize and populate the set of n_k-l rows that will be deleted from H_Prime to leave l rows
                Z = set();
                while len(Z) < n_k-l:
                        Z.add(randint(0,n_k-1)); #H.nrows()=n_k, but indices start at 0

                Z = list(Z); Z.sort(); #Make Z a sorted list
```

```
##########################################################
#Form two subsets, X and Y, from the remaining k columns
#       X and Y will be stored as sets of column indices
##########################################################

#A copy of H_Prime with only the l selected rows (in Z) remaining
H_Prime_l = copy(H_Prime);
H_Prime_l = H_Prime_l.delete_rows(Z);

#Initialize the sets of indices of the columns of H_Prime_l
X_Indices = list(); Y_Indices = list();

#Assign a column indices to X or Y randomly (50/50 chance)
for i in range(k):
       if randint(0,1)==0:
              X_Indices.append(i+n_k);
       else:
              Y_Indices.append(i+n_k);


##########################################################
#For every size p subset A of X, calculate the l-bit
#vector formed by adding the columns of A.
#Call this vector pi(A). Perform a similar calculator
#for every size p subset B of Y.
##########################################################

#Generate the size-p subsets of X and Y,
#and initialize the lists containing each subset's sum
Subsets_of_X_Indices = Subsets(X_Indices,p);
Subsets_of_Y_Indices = Subsets(Y_Indices,p);
pi_A = list();
pi_B = list();

#Calculate pi(A) for each subset of X
for i in range(Subsets_of_X_Indices.cardinality()):
       column_sum = 0;
       for j in range(p):
              column_sum = column_sum +
                   H_Prime_l.submatrix(0,Subsets_of_X_Indices[i][j],H_Prime_l.nrows(),1);

       pi_A.append(column_sum);

#Calculate pi(B) for each subset of Y
for i in range(Subsets_of_Y_Indices.cardinality()):
       column_sum = 0;
       for j in range(p):
              column_sum = column_sum +
                   H_Prime_l.submatrix(0,Subsets_of_Y_Indices[i][j],H_Prime_l.nrows(),1);

       pi_B.append(column_sum);

##########################################################
#For each collision pi_A = pi_B, calculate the n_k-bit
#vector formed by summing the columns of A U B.
#If the sum has weight w-2p, we can construct the
#desired codeword
##########################################################

weight_w_codeword = 0; #initialize the codeword

#Check each pi(A) value against every pi(B) value to check for collisions
for i in range(len(pi_A)):
       for j in range(len(pi_B)):

       #If a collision occurs, calculate the n-k - bit vector computed by summing the
       #entirety of the columns whose indices are in A U B
              if pi_A[i] == pi_B[j]:
```

```
sum = 0; #initialize the sum vector
for k in (Subsets_of_X_Indices[i]):
       sum = sum + H_Prime.submatrix(0,k,H_Prime.nrows(),1);
for k in (Subsets_of_Y_Indices[j]):
       sum = sum + H_Prime.submatrix(0,k,H_Prime.nrows(),1);

if _GetColumnVectorWeight(sum) == (w-2*p):
       codeword_found = true;

       #Since the sum vector has the appropriate weight, the codeword of
            weight w can now be calculated
       #Initialize the codeword
       weight_w_codeword = matrix(GF(2),H_Stern.ncols(),1);

       #Mark the appropriate positions of the codeword as ones
       for index in range(n_k):
              if sum[index,0]==1:
                     weight_w_codeword[index,0] = 1;
       for k in Subsets_of_X_Indices[i]:
              weight_w_codeword[k,0] = 1;
       for k in Subsets_of_Y_Indices[j]:
              weight_w_codeword[k,0] = 1;
       #Undo the permuting done when selecting n-k linearly independent
            columns
       weight_w_codeword = weight_w_codeword.transpose()*(~P_Stern);

       return copy(weight_w_codeword);
```

```
#DataCollectionScript.py
#!/opt/sage-5.6/sage -python

import time
from sage.all import *

############################################
#Functions & Classes to Import
############################################
load("./SAGE Classes/GoppaCode.sage")
load("./SAGE Classes/McElieceCryptosystem.sage")
load("./SAGE Classes/SternsAlgorithm.sage")

def GetRandomMessage(message_length):
	message = matrix(GF(2), 1, message_length);
	for i in range(message_length):
		message[0,i] = randint(0,1);
	return message;

def SternsAttack(crypto, encrypted_message, t, p, l):

	#Attacker has knowledge of PK and y (and presumably t), and is looking for a codeword of weight w
	PK = copy(crypto.public_key());
	y = encrypted_message;

	#Calculate a parity check matrix for the code G + {0,y}, where y = mG + e
	H = (PK.stack(y)).right_kernel().basis_matrix();
	w = t;

	#Find a weight w codeword
	weight_w_codeword = SternsAlgorithm(H, w, p, l);

	#Decrypt the message using the codeword found via Stern's Algorithm
	decrypted_message = PK.solve_left((y-weight_w_codeword));

	return decrypted_message;

############################################
#Beginning of Script
############################################

#The goal of the script will be to write the parameters and results to a .csv file format

#Table Headers
print "\nn,m,t,Patterson's Algorithm Wall Time,Patterson's Algorithm CPU Time,p,l,Stern's Attack Wall
     Time,Stern's Attack CPU Time,Polynomial";

for k in range(3):
	m = k+6;
	n = 2**m;
	t = floor((2+(2**m-1)/m)/2);

	F_2m = GF(n,'Z');
	PR_F_2m = PolynomialRing(F_2m,'X');

	for _ in range(10):
		while 1:
			irr_poly = PR_F_2m.random_element(t);
			if irr_poly.is_irreducible():
				break;
		crypto = McElieceCryptosystem(n,m,irr_poly);

		for i in range(5):
			#Get a random message and encrypt it.
			message = GetRandomMessage(crypto.goppa_code().generator_matrix().nrows());
			encrypted_message = crypto.Encrypt(message);
```

```
#Patterson's Algorithm Decoding
patterson_start_wall_time = time.time();
patterson_start_cpu_time = time.clock();
crypto.Decrypt(encrypted_message);
patterson_end_cpu_time = time.clock();
patterson_end_wall_time = time.time()
patterson_wall_time = patterson_end_wall_time - patterson_start_wall_time;
patterson_cpu_time = patterson_end_cpu_time - patterson_start_cpu_time;

H = (copy(crypto.public_key()).stack(encrypted_message)).right_kernel().basis_matrix();

#Stern's Attack
for j in range(2):
       p = j+1;
       k_2 = floor((H.ncols()-H.nrows())/2);
       l_min = floor(log(k_2,2))-1;

       for k in range(7):
              l = l_min + k;
              if (H.nrows() < l):
                     l = H.nrows();
              stern_start_wall_time = time.time();
              stern_start_cpu_time = time.clock();
              SternsAttack(crypto,encrypted_message,t,p,l);
              stern_end_cpu_time = time.clock();
              stern_end_wall_time = time.time()
              stern_wall_time = stern_end_wall_time - stern_start_wall_time;
              stern_cpu_time = stern_end_cpu_time - stern_start_cpu_time;

              #n,m,t,Iteration,Patterson's Algorithm,p,l,Stern's Attack,Polynomial";
              print "%s,%s,%s,%s,%s,%s,%s,%s,%s,%s" % (n,m,t, \
                                                   patterson_wall_time,patterson_cpu_time,\
                                                   p,l,stern_wall_time,stern_cpu_time,irr_poly);
```