Ball-collision Decoding Analysis
Linear Codes in the McEliece Cryptosystem

Kyle Yates

MATH 499 Fall 2019*

Abstract

Modern cryptography relies on computationally difficult problems. With the anticipated advancements of quantum computing in the near future, many cryptosystems are in jeopardy of being compromised. The McEliece cryptosystem, although rarely used in practice, is resistant to attacks even with the development of quantum computers.

The McEliece Cryptosystem tasks an adversary with the difficult decoding of a seemingly random long linear code. In 2011, Daniel J. Bernstein, Tanja Lange, and Christiane Peters published **Smaller decoding exponents: ball-collision decoding**, showing a speedup from previous decoding algorithms [1].

This project focuses on decoding long linear codes via the ball-collision decoding algorithm. We will implement the algorithm, test the algorithm on a variety of linear codes, and discuss feasibility of ball-collision decoding in breaking the McEliece Cryptosystem.

1 Introduction

1.1 Decoding Long Linear Codes

For specific structures of linear codes, efficient decoding algorithms are known. However, with our current knowledge this is not true for a general linear code. In terms of computational difficulty, syndrome decoding is NP-complete [2]. With a code of substantial enough length, even the best decoding algorithms become computationally infeasible.

1.2 The McEliece Cryptosystem

In 1978, Robert McEliece developed a cryptosystem based on the hardness of decoding long linear codes [3]. A long linear code with an efficient decoding algorithm is chosen (often a Goppa code), and is disguised via multiplication by a random non-singular matrix and a random permutation matrix. Messages are encrypted with this masked code. If this code is intercepted during transmission, the interceptor will not be able to distinguish the code from a random linear code and generally will not be able to efficiently decode the message. Without access to the private key, the interceptor will also not be able to convert the code into the original code for which an efficient decoding algorithm is known. The process for generating a matched public and private key is outlined below [4]

1. Choose a $k \times n$ generator matrix G for a (n, k)-linear code with error correcting capability t and an efficient known decoding algorithm.

2. Choose a random $k \times k$ invertible matrix S.

*Senior research project, supervised by Dr. J. Carmelo Interlando

3. Choose a random $n \times n$ permutation matrix P.
4. Calculate $\hat{G} = SGP$.
5. The public key is $(\hat{G}, t)$. The private key is (S, G, P).

The intended receiver of the message can now generate a public and private key pair. Users can encrypt messages with the public key and send the encrypted message to the receiver. The receiver can then decrypt the message with the private key. The encryption and decryption process is outlined below[4].

Encryption

1. Obtain the public key $(\hat{G}, t)$
2. Represent the message m as a binary string of length k
3. Choose a random binary error vector e of length n having $\text{wt}(e)$ within the error correcting capability t
4. Compute the ciphertext $c = m\hat{G} + e$
5. Send the encrypted message c to the intended receiver

Decryption

1. Compute $\hat{c} = cP^{-1}$
2. Use an efficient decoding algorithm to decode $\hat{c}$ to $\hat{m}$
3. Compute $m = \hat{m}S^{-1}$, the original message

1.3 The Ball-collision Decoding Algorithm

If a third-party adversary were to intercept a message encrypted with the McEliece cryptosystem during transmission, they would be tasked with decoding a seemingly random linear code. Ball-collision decoding can be used in attempting to decode this linear code [1]. The algorithm is outlined below. Slight adjustments in notation from the original paper have been made in this report, which will be discussed further in section 2.3.

CONSTANTS: $n, k, w \in \mathbb{Z}$ with $0 \leq w \leq n$ and $0 \leq k \leq n$
PARAMETERS: $p_1, p_2, q_1, q_2, \ell_1, \ell_2 \in \mathbb{Z}$ with
$0 \leq k_1, 0 \leq k_2, k_1 + k_2 = k, 0 \leq p_1 \leq k_1, 0 \leq p_2 \leq k_2, 0 \leq q_1 \leq \ell_1, 0 \leq q_2 \leq \ell_2$, and
$0 \leq w - p_1 - p_2 - q_1 - q_2 \leq n - k - \ell_1 - \ell_2$
INPUTS: Parity check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ and syndrome $s \in \mathbb{F}_2^{n-k}$
OUTPUT: Zero or more vectors $e \in \mathbb{F}_2^n$ such that $He = s$

1. Choose a uniform random information set Z
2. Choose a uniform random partition of Z into sizes k_1 and k_2
3. Choose a uniform random partition of $\{1, 2, ..., n\} \setminus Z$ into parts of sizes ℓ_1, ℓ_2 and $n - k - \ell_1 - \ell_2$

4. Find an invertible $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$ such that the columns of UH indexed by $\{1, 2, ..., n\} \setminus Z$ form an identity matrix of size $(n-k) \times (n-k)$. Write the columns of UH indexed by Z as $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$ with $A_1 \in \mathbb{F}_2^{(\ell_1+\ell_2)\times k}$ and $A_2 \in \mathbb{F}_2^{(n-k-\ell_1-\ell_2)\times k}$

5. Write Us as $\begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$ with $s_1 \in \mathbb{F}_2^{\ell_1+\ell_2}$ and $s_2 \in \mathbb{F}_2^{n-k-\ell_1-\ell_2}$

6. Compute the set S containing all possible triples $(A_1x_0 + x_1, x_0, x_1)$ where $x_0 \in \mathbb{F}_2^{k_1}||\{0\}^{k_2}$, $\mathrm{wt}(x_0) = p_1$, $x_1 \in \mathbb{F}_2^{\ell_1}||\{0\}^{\ell_2}$, and $\mathrm{wt}(x_1) = q_1$

7. Compute the set T containing all possible triples $(A_1y_0 + y_1 + s_1, y_0, y_1)$ where $y_0 \in \{0\}^{k_1}||\mathbb{F}_2^{k_2}$, $\mathrm{wt}(y_0) = p_2$, $y_1 \in \{0\}^{\ell_1}||\mathbb{F}_2^{\ell_2}$, and $\mathrm{wt}(y_1) = q_2$

8. For each (v, x_0, x_1) in S:

   For each y_0, y_1 with $(v, y_0, y_1) \in T$:

   If $\mathrm{wt}(A_2(x_0 + y_0) + s_2) = w - p_1 - p_2 - q_1 - q_2$:

   Output the vector $e \in \mathbb{F}_2^n$ with $x_0 + y_0$ as the Z indexed components, and $(x_1 + y_1||A_2(x_0 + y_0) + s_2)$ as the remaining components.

1.4 Notation

Notation will be introduced in sections to which it is relevant, but is also listed here for convenience. The following is a list of mathematical notation used in this paper.

$\mathbb{Z}$ denotes the set of integers

$\emptyset$ denotes the empty set

$|S|$ denotes the cardinality of a set S

$\mathbb{F}_2$ denotes a finite field of order 2

$\mathrm{GF}(2^m)$ denotes a finite field of order 2^m

An $[n, k]$ code is a code of length n and dimension k

An $[n, k, d]$ code is a code of length n, dimension k, and minimum distance d

t denotes the error correcting capability of a code, equivalent to $\lfloor \frac{d-1}{2} \rfloor$

e denotes an error vector

s denotes a syndrome of a code

G denotes the generator matrix of a code

H denotes the parity-check matrix of a code

Z denotes an information set of a code

$v \in \mathbb{F}_2^a$ denotes a binary vector v of length a

$\mathrm{wt}(v)$ denotes the Hamming weight of a vector v

$v||w$ denotes concatenation of vectors v and w

$\{0\}^a$ denotes the zero vector of length a

$M \in \mathbb{F}_2^{a\times b}$ denotes a binary matrix M of length a and dimension b

2 Implementation of the Ball-collision Decoding Algorithm

The primary goal of this project is to implement and test performance of the ball-collision decoding algorithm. This section will discuss the implementation of the ball-collision decoding algorithm.

2.1 Magma Computational Algebra

Magma Computational Algebra is a mathematical software geared towards algebraic topics. This includes linear codes. Implementation of the ball-collision decoding algorithm in this report is assumed in Magma.

Several Magma-specific functions are utilized in this implementation of the algorithm. These includes functions for finding parity-check matrices, checking linear independence, finding subsets, and calculating Hamming weights.

2.2 Code

In this section, we will discuss the Magma code for the ball-collision decoding algorithm. We will move through the algorithm one step at a time, discussing the code for each step if necessary. The more difficult tasks in steps 6 to 8 of matching matrix dimensions and sets S and T will be discussed separately in sections 2.3 and 2.4. Note that steps 6 and 7 of the original algorithm are switched here, but the end result is exactly the same.

A full version of the code without intermediate steps or explanations can be found in the Appendix.

PARAMETER SETUP:

```
time for x:=1 to 1 do

n:= ;
k:= ;
w:= ;

k1:= ;
k2:=k-k1;
l1:= ;
l2:= ;

p1:= ;
p2:= ;
q1:= ;
q2:= ;

H:= ;
s:= ;
```

This is the basic parameter setup. The beginning `for` loop is to provide an easily accessible break point if the algorithm succeeds, as well as the opportunity to time the algorithm.

1. Choose a uniform random information set Z

```
H:=Transpose(H);

Z:={};
Zc:={};
Zcv:={};
RS:={1..n};

repeat
  ran:=Random(RS);
  Zct:=Zcv join {H[ran]};
  if IsIndependent(Zct) eq true then
    Zcv:=Zct;
    Zc:=Zc join {ran};
  end if;
  RS:=RS diff {ran};
until #Zc eq n-k;

Z:={1..n} diff Zc;

H:=Transpose(H);
```

   To build a random information set Z, we first start with a set $Z^* = \emptyset$ and choose a random column vector w of H (or a random row vector w of H^T). If w is linearly independent to every element in Z^*, set $Z^* = Z^* \cup \{w\}$. If not, we discard w. We repeat this process until $|Z^*| = n - k$. Z is the set containing column indices of all the remaining vectors of the code which are not in Z^*.

2. Choose a uniform random partition of Z into sizes k_1 and k_2

```
Fk1:=RandomSubset(Z, k1);
Fk2:=Z diff Fk1;
```

3. Choose a uniform random partition of $\{1, 2, ..., n\} \setminus Z$ into parts of sizes ℓ_1, ℓ_2 and $n - k - \ell_1 - \ell_2$

```
Fl1:=RandomSubset(Zc, l1);
Zi:= Zc diff Fl1;
Fl2:=RandomSubset(Zi, l2);
Fnkl1l2:= Zi diff Fl2;
```

4. Find an invertible $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$ such that the columns of UH indexed by $\{1, 2, ..., n\} \setminus Z$ form an identity matrix of size $(n - k) \times (n - k)$. Write the columns of UH indexed by Z as $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$ with $A_1 \in \mathbb{F}_2^{(\ell_1+\ell_2)\times k}$ and $A_2 \in \mathbb{F}_2^{(n-k-\ell_1-\ell_2)\times k}$

```
Z:=Sort(SetToSequence(Z));
Zc:=Sort(SetToSequence(Zc));
V:=Matrix(GF(2),n-k,n-k,[]);

for i:=1 to n-k do
```

```
  for j:=1 to n-k do
    V[i,j]:=H[i,Zc[j]];
  end for;
end for;

U:=V^-1;
UH:=Transpose(U*H);
A:=Matrix(GF(2),k,n-k,[]);

for i:=1 to #Z do
  A[i]:=UH[Z[i]];
end for;

A:=Transpose(A);
A1:=ExtractBlock(A, 1, 1, l1+l2, k);
A2:=ExtractBlock(A, l1+l2+1, 1,n-k-l1-l2, k);
```

To construct U, we take all the columns of H indexed by $\{1, ..., n\} \setminus Z$. The inverse of this is U. We then take the columns of UH indexed by Z as A, and then extract A_1 and A_2 accordingly.

5. Write Us as $\begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$ with $s_1 \in \mathbb{F}_2^{\ell_1+\ell_2}$ and $s_2 \in \mathbb{F}_2^{n-k-\ell_1-\ell_2}$

```
Us:=U*s;
s1:=ExtractBlock(Us, 1, 1, l1+l2, 1);
s2:=ExtractBlock(Us, l1+l2+1, 1,n-k-l1-l2, 1);
```

6. Compute the set T containing all possible triples $(A_1y_0 + y_1 + s_1, y_0, y_1)$ where $y_0 \in \{0\}^{k_1}||\mathbb{F}_2^{k_2}$, $\mathrm{wt}(y_0) = p_2$, $y_1 \in \{0\}^{\ell_1}||\mathbb{F}_2^{\ell_2}$, and $\mathrm{wt}(y_1) = q_2$

```
T:=[];
ct:=1;
T1:=Subsets({1..k2}, p2);
T2:=Subsets({1..l2}, q2);

for a in T1 do
  y0:=Matrix(GF(2),k,1,[]);

  if #T1 ne 0 then
    for b in a do
      y0[k1+b,1]:=1;
    end for;
  end if;

  for c in T2 do
    y1:=Matrix(GF(2),l1+l2,1,[]);
    if #T2 ne 0 then
      for d in c do
        y1[l1+d,1]:=1;
```

```
      end for;
    end if;

    v:=A1*y0+y1+s1;

    if #T eq 0 then
      Append(~T,<v,{<y0,y1>}>);
    else
      for h:=1 to ct do
        if T[ct][1] eq v then
          T[ct][2]:=T[ct][2] join {<y0,y1>};
break h;
        else
          Append(~T,<v,{<y0,y1>}>);
          ct:=ct+1;
        end if;
      end for;
    end if;
  end for;
end for;
```

7. Compute the set S containing all possible triples $(A_1x_0 + x_1, x_0, x_1)$ where $x_0 \in \mathbb{F}_2^{k_1}||\{0\}^{k_2}$, $\mathrm{wt}(x_0) = p_1$, $x_1 \in \mathbb{F}_2^{\ell_1}||\{0\}^{\ell_2}$, and $\mathrm{wt}(x_1) = q_1$

```
S:=[];
ct:=1;
S1:=Subsets({1..k1}, p1);
S2:=Subsets({1..l1}, q1);

for a in S1 do
  x0:=Matrix(GF(2),k,1,[]);

  if #S1 ne 0 then
    for b in a do
      x0[b,1]:=1;
    end for;
  end if;

  for c in S2 do
    x1:=Matrix(GF(2),l1+l2,1,[]);

    if #S2 ne 0 then
      for d in c do
        x1[d,1]:=1;
      end for;
    end if;

    v:=A1*x0+x1;
```

8. For each (v, x_0, x_1) in S:

For each y_0, y_1 with $(v, y_0, y_1) \in T$:

If $\text{wt}(A_2(x_0 + y_0) + s_2) = w - p_1 - p_2 - q_1 - q_2$:

Output the vector $e \in \mathbb{F}_2^n$ with $x_0 + y_0$ as the Z indexed components, and $(x_1 + y_1 || A_2(x_0 + y_0) + s_2)$ as the remaining components.

```
for r:=1 to #T do
  if T[r][1] eq v then
    for dl in T[r][2] do
      cl:=Vector(A2*(x0+dl[1])+s2);
      if Weight(cl) eq w-p1-p2-q1-q2 then

        t1:=x0+dl[1];
        t2:=x1+dl[2];
        t3:=A2*(x0+dl[1])+s2;

        R:=Matrix(GF(2),n,1,[]);
        for i in Z do
          R[i]:=t1[Index(Z, i)];
        end for;

        for i:=1 to l1+l2 do
          R[Zc[i]]:=t2[i];
        end for;

        for i:=1 to n-k-l1-l2 do
          R[Zc[i+l1+l2]]:=t3[i];
        end for;

        print Vector(R);
        break x;

      end if;
    end for;
  end if;
 end for;
end for;

end for;
```

The algorithm will output every vector e fulfilling $He = s$ if p_1, p_2, q_1, and q_2 are chosen correctly. Since McEliece encryption requires that $\text{wt}(e)$ is within the error-correcting capability t, e will be unique. Thus, we can halt the algorithm once e is found, as no further solutions will exist.

2.3 Dimensions

One of the trickier parts to implementing the algorithm is ensuring dimensions of matrices and vectors match. This is apparent in steps 6 to 8 of the algorithm. Constructing sets S and T

will be discussed from an efficiency perspective in the next section. However, the computations needed for triples of these sets are can be prone to discrepancies in dimensions matching.

In [1], the following notations are used for defining some specific matrices and vectors

$$A_1 \in \mathbb{F}_2^{(\ell_1+\ell_2)\times k},\ A_2 \in \mathbb{F}_2^{(n-k-\ell_1-\ell_2)\times k}$$
$$s_1 \in \mathbb{F}_2^{\ell_1+\ell_2},\ s_2 \in \mathbb{F}_2^{n-k-\ell_1-\ell_2}$$
$$x_0 \in \mathbb{F}_2^{k_1},\ x_1 \in \mathbb{F}_2^{\ell_1}\ , y_0 \in \mathbb{F}_2^{k_2},\ y_1 \in \mathbb{F}_2^{\ell_2}$$

The algorithm constructs triples in S via $(A_1y_0 + y_1 + s_1, y_0, y_1)$. Problems can arise in this computation depending on how we interpret notation. If we take $x_0 \in \mathbb{F}_2^{k_1}$ as a vector of length k_1, then the multiplication A_1x_0 is not always defined. A_1 is a $\ell_1 + \ell_2$ by k matrix, so A_1x_0 is only valid in the special case $k_1 = k$. If instead we take $x_0 \in \mathbb{F}_2^{k_1}$ as a vector of length k with variations in the first k_1 digits and 0's in the remaining k_2 digits, then A_1x_0 is valid. The result a vector of length $\ell_1 + \ell_2$. For the addition $A_1x_0 + x_1$ to make sense, $x_1 \in \mathbb{F}_2^{\ell_1}$ must then be of length $\ell_1 + \ell_2$. x_1 will have variations in the first ℓ_1 digits and 0's in the remaining ℓ_2 digits by the same logic as x_0's structure.

Similarly, the algorithm constructs triples in T via $(A_1y_0 + y_1 + s_1, y_0, y_1)$. $y_0 \in \mathbb{F}_2^{k_2}$ must be of length k for the multiplication A_1y_0 to be valid. Take $y_0 \in \mathbb{F}_2^{k_2}$ as a vector of length k with 0's in the first k_1 digits and variations in the remaining k_2 digits. A_1y_0 then results in a vector of length $\ell_1 + \ell_2$. For $A_1y_0 + y_1 + s_1$ to be valid, take $y_1 \in \mathbb{F}_2^{\ell_2}$ as a vector of length $\ell_1 + \ell_2$ with 0's in the first ℓ_1 digits and variations in the remaining ℓ_2 digits.

This interpretation for the dimensions of these various subspaces is based on an earlier version of the ball-collision decoding paper[1]. This earlier version uses the notation $x_0 \in \mathbb{F}_2^{k_1} \times \{0\}^{k_2}$ for these types of elements. This report uses $x_0 \in \mathbb{F}_2^{k_1}||\{0\}^{k_2}$ as the notation for these types of elements, with $||$ denoting concatenation of two vectors.

2.4 Sets S and T

The most operationally taxing portion of the algorithm is constructing and comparing sets S and T. For constructing S, we need to compute every possible vector $x_0 \in \mathbb{F}_2^{k_1}||\{0\}^{k_2}$ of weight p_1, every possible vector $x_1 \in \mathbb{F}_2^{\ell_1}||\{0\}^{\ell_2}$ of weight q_1, and the triple $(A_1x_0 + x_1, x_0, x_1)$ for the combination of any x_0 and x_1. Similarly, for constructing T, we need to compute every possible vector $y_0 \in \{0\}^{k_1}||\mathbb{F}_2^{k_2}$ of weight p_2, every possible vector $y_1 \in \{0\}^{\ell_1}||\mathbb{F}_2^{\ell_2}$ of weight q_2, and the triple $(A_1y_0 + y_1 + s_1, y_0, y_1)$ for the combination of any y_0 and y_1. Once our sets are constructed, we must compare the first element of each triple in S with the first element of each triple in T looking for matches.

As specified in [1], the sizes of S and T are $\binom{k_1}{p_1}\binom{\ell_1}{q_1}$ and $\binom{k_2}{p_2}\binom{\ell_2}{q_2}$ respectively. With big parameters, the sizes of S and T explode. Storing these sets outright will eventually be infeasible. To achieve the best runtime for the algorithm available to us, we need to minimize storage of these sets. Rather than computing and storing every triple (v, x_0, x_1) and (v, y_0, y_1) for both S and T, we will instead store only one set. Then, we will calculate terms as needed, and discard them if they do not fulfill the required conditions.

[1] https://cr.yp.to/codes/ballcoll-20101117.pdf

Constructing T

We construct T as follows. Let $T = \emptyset$. $\forall y_0 \in \{0\}^{k_1}||\mathbb{F}_2^{k_2}, y_1 \in \{0\}^{\ell_1}||\mathbb{F}_2^{\ell_2}$ do

1. Calculate $v = A_1 y_0 + y_1 + s_1$.
2. Set $T(v) = T(v) \cup \{(y_0, y_1)\}$. If $T(v)$ has not yet been defined, set $T(v) = \{(y_0, y_1)\}$.
3. If $T(v) \nsubseteq T$, set $T = T \cup \{T(v)\}$.

T will then be stored in the following form

$$T = \{T(v_1), T(v_2), ..., T(v_m)\}$$

Where $T(v_1), T(v_2), ..., T(v_m)$ are subsets of T such that

$$T(v) = \{(y_0, y_1) : Ay_0 + y_1 + s_1 = v\}$$

Constructing and Checking S

We construct S and check it against T as follows. $\forall x_0 \in \mathbb{F}_2^{k_1}||\{0\}^{k_2}, x_1 \in \mathbb{F}_2^{\ell_1}||\{0\}^{\ell_2}$ do

1. Calculate $v = A_1 x_0 + x_1$
2. If $T(v) \nsubseteq T$, discard this pair x_0, x_1 and go back to step 1. Else, continue.
3. $\forall (y_0, y_1) \in T(v)$, check if $\text{wt}(A_2(x_0 + y_0) + s_2) = w - p_1 - p_2 - q_1 - q_2$. If true, output $e \in \mathbb{F}_2^n$ with $x_0 + y_0$ as the Z indexed components, and $(x_1 + y_1||A_2(x_0 + y_0) + s_2)$ as the remaining components.

Once the vector e is found, no more computation is necessary and we can end the program. By discarding x_0 and x_1 as we go and storing only unique v's in T, there is less computation and storage that the algorithm must perform.

3 The Golay Code

Now that the ball-collision decoding algorithm has been implemented, we can begin to test it. The Golay Code is one of the few known non-trivial perfect codes, meaning that it achieves the Hamming bound [5]. The [23,12,7] code G_{23} is defined as follows

$$G_{23} = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1&1&0&0&0&1\\
0&1&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&1&0&0&1\\
0&0&1&0&0&0&0&0&0&0&0&0&1&1&0&1&0&0&1&0&1&0&1\\
0&0&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&1&1&1&0&1&1\\
0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&0&1&1&0&1&1&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&1&1&0&0&1&1&0&1&1&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&1&1&0&0&1&1&0&1&1&0&1\\
0&0&0&0&0&0&0&1&0&0&0&0&1&0&1&1&0&1&1&1&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&1&1&0&1&1&1&1&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&1&1&0&1&1&1&1\\
0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1&1&0&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1&1&0&0&0&1&1
\end{bmatrix}$$

The code can be extended by a parity check bit to the nearly perfect [24,12,8] code G_{24}. However, we will only look at G_{23}.

G_{23} is a well-known code, making it a good starting point for our study of the ball-collision decoding algorithm.

3.1 Hamming Bound

Let C be an $[n, k, d]$ code with error correcting capability $t = \left\lfloor \frac{d-1}{2} \right\rfloor$. The Hamming bound is an upper bound for the number of codewords in C, namely [5]

$$|C| \leq \frac{2^n}{\sum_{i=0}^{t} \binom{n}{i}}$$

A code C is perfect is it attains the Hamming bound [5]

$$|C| = \frac{2^n}{\sum_{i=0}^{t} \binom{n}{i}}$$

G_{23} is a $[23, 12, 7]$ code with $t = 3$. It achieves the Hamming bound and is therefore a perfect code, as

$$|G_{23}| = 2^{12} = \frac{2^{23}}{\binom{23}{0} + \binom{23}{1} + \binom{23}{2} + \binom{23}{3}}$$

3.2 Ball-collision Decoding on the Golay Code

The small size of the Golay Code does not provide for much insight regarding the efficiency of ball-collision decoding, as any calculations in the algorithm will be in a computationally trivial range. The Golay code will rather be useful in studying the mechanics of ball-collision decoding and analyzing success probabilities.

Example - Algorithm Mechanics

Let Z be the following information set of the Golay code

$$Z = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

Consider the error vector e with $\mathrm{wt}(e) = 3$

$$e = (0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)$$

And the corresponding partitions of e for parameter values $k_1 = k_2 = 6$, $\ell_1 = \ell_2 = 3$

$$e = (\underbrace{\underbrace{[0\ 0\ 0\ 1\ 1\ 0]}_{k_1}\ \underbrace{[0\ 0\ 0\ 0\ 0\ 0]}_{k_2}}_{Z}\ \underbrace{\underbrace{[0\ 0\ 0]}_{\ell_1}\ \underbrace{[0\ 0\ 0]}_{\ell_2}\ \underbrace{[0\ 1\ 0\ 0\ 0]}_{n-k-\ell_1\ell_2}}_{\{1,2,..,n\}\setminus Z})$$

Because $\text{wt}(e) \leq t$, there is only one e satisfying $He = s$. e has a weight of 2 in the partition corresponding with k_1, a weight of 0 in the partitions corresponding with k_2, ℓ_1, ℓ_2, and a weight of 1 in the partition corresponding with $n - k - \ell_1 - \ell_2$. The algorithm searches for an error vector of weights p_1, p_2, q_1, q_2 in these respective partitions. Hence, the algorithm will succeed when $p_1 = 2$, $p_2 = q_1 = q_2 = 0$, and will fail otherwise.

Success Probabilities

The success probability b for any parameters of ball-collision decoding is given via the counting argument [1]

$$b = \binom{n}{w}^{-1} \binom{n-k-\ell_1-\ell_2}{w-p_1-p_2-q_1-q_2} \binom{k_1}{p_1} \binom{k_2}{p_2} \binom{\ell_1}{q_1} \binom{\ell_2}{q_2}$$

The success probability for the Golay code is

$$b = \binom{23}{w}^{-1} \binom{11-\ell_1-\ell_2}{w-p_1-p_2-q_1-q_2} \binom{k_1}{p_1} \binom{k_2}{p_2} \binom{\ell_1}{q_1} \binom{\ell_2}{q_2}$$

For example purposes, consider the parameter values $k_1 = 8, k_2 = 4, \ell_1 = 2, \ell_2 = 3$. We will search for the same weight distribution in our partitions as outlined in the previous example, with $p_1 = 2$, $p_2 = q_1 = q_2 = 0$. For a random vector e with $\text{wt}(e) = w = 3$, our success probability is

$$b = \binom{23}{3}^{-1} \binom{6}{1} \binom{8}{2} \binom{4}{0} \binom{2}{0} \binom{3}{0} = \binom{23}{3}^{-1} \binom{6}{1} \binom{8}{2} \approx 9.5\%$$

We will now test the algorithm with these outlined parameters in an attempt to match the theoretical success probability. Consider multiple iterations of ball-collision decoding, each with a random error vector e with $\text{wt}(e) = 3$. Results are as follows:

Iterations	Successful attempts	Success rate	Runtime (seconds)
1	0	0%	0.000
10^1	1	10%	0.000
10^2	9	9%	0.063
10^3	97	9.7%	0.516
10^4	930	9.3%	5.031
10^5	9486	9.49%	55.422

This matches the desired success probability after a large amount of iterations.

4 Long Linear Codes

The Golay code serves as a good starting point for understanding the mechanics and success probability of ball-collision decoding. To evaluate practicality and efficiency, we must move to bigger codes.

The codes introduced in this section are general linear codes, and thus do not have efficient decoding algorithms. They are not usable in the McEliece Cryptosystem. For an adversary trying to break the McEliece cryptosystem however, the intercepted codes will seem to be codes such as these.

4.1 Ball-collision Decoding on Random Linear Codes

Let's look at some random linear codes. We will look at a $[100, 50]$ linear code. This code is too large to write explicity in this paper. The randomness also does not allow us to write it using a generating polynomial such as Goppa codes do. We will need to generate this code using functions in Magma.

Generate a random code C and calculate the minimum distance via

```
C:=RandomLinearCode(GF(2), 100, 50);
d:=MinimumDistance(C);
```

In our case $d = 13$ and $t = 6$. We will test the algorithm's performance on this random linear code.

Stern's algorithm

Consider the following parameter values

$$k_1 = k_2 = 25, \ell_1 = \ell_2 = 4$$
$$p_1 = p_2 = q_1 = q_2 = 0$$

Stern's algorithm coincides with special case of ball-collision decoding [1]. Namely, $p_1 = p_2, q_1 = q_2 = 0, k_1 \approx k_2$. The above parameter values give us Stern's algorithm. Letting $\mathrm{wt}(e) = w = t = 6$, the success probability for one iteration of ball-collision decoding is

$$b = \binom{100}{6}^{-1} \binom{42}{6} \binom{25}{0} \binom{25}{0} \binom{4}{0} \binom{4}{0} \approx 0.44\%$$

The most computationally intensive portion of the algorithm is constructing and comparing sets S and T. Since $p_1 = p_2 = q_1 = q_2 = 0$, $|S| = |T| = 0$. This means the algorithm will run very quickly. The price we pay is having a small success probability. Consider multiple iterations of ball-collision decoding with these parameters, each with a random error vector e. Results are as follow:

Iterations	Successful attempts	Success rate	Runtime (seconds)
1	0	0%	0.000
10^1	0	0%	0.031
10^2	1	1%	0.266
10^3	4	0.4%	2.688
10^4	46	0.46%	27.703
10^5	454	0.45%	285.141

The successful attempts and runtimes expand linearly with the number of iterations. Consider now the same table with values $p_1 = p_2 = 1, q_1 = q_2 = 0$. Results are as follows:

Iterations	Successful attempts	Success rate	Runtime (seconds)
1	0	0%	0.000
10^1	1	10%	0.047
10^2	11	11%	0.422
10^3	159	15.9%	4.063
10^4	1575	15.8%	64.516

With the values of p_1 and p_2 being nonzero, we see a tradeoff between success rate and runtime. Though this is true in our case, a larger value of p_1 and p_2 will certainly not always infer a higher success rate.

Ball-collision Decoding vs. Stern's Algorithm

With our desire to minimize computational strain, the best parameter choices are usually comparable to the requirements of Stern's algorithm. As the values of ℓ_1, ℓ_2, q_1, q_2 inflate so does the size of set T. Parameter values for attacks on high-security systems coincide with small values of ℓ_1, ℓ_2, q_1, q_2. Stern's algorithm requires $q_1 = q_2 = 0$, and although this at times is the best known attack on a system, non-zero values of these parameters can be of slight advantage (such as the hypothetical attack on a 256-bit security system outlined in [1]).

5 Goppa Codes

Goppa codes are the standard codes used in the McEliece Cryptosystem. The way in which Goppa codes are structured provide bounds for both the dimension k and minimum distance d. With a well-chosen Goppa polynomial, both k and d can be a reasonable and sufficient size. Efficient decoding algorithms are known for Goppa codes as well, making them the perfect candidate for the McEliece Cryptosystem. This section will discuss the basics of constructing Goppa codes and the performance of the ball-collision decoding algorithm on Goppa codes of a substantial size. We will only consider binary Goppa codes.

5.1 Constructing Goppa Codes

Define $G(z)$ as the Goppa polynomial. This will be the polynomial that will generate our code. Let $m \in \mathbb{Z}$ be fixed. Then, $G(z)$ is a polynomial of degree r with coefficients from

GF(2^m). Let $L = \{\alpha_1, \cdots \alpha_n\}$ be a subset of GF(2^m) such that $\forall \alpha_i \in L, G(\alpha_i) \neq 0$. With any vector $a = (a_1, \cdots a_n)$ over GF(2), we have the rational function $R_a(z)$ as

$$R_a(z) = \sum_{i=1}^{n} \frac{a_i}{z - \alpha_i}$$

The Goppa code $\Gamma(L, G)$ consists of all vectors a such that $R_a(z) = 0 \mod G(z)$ [6].

Goppa codes are a subclass of alternant codes. Thus, they have the property that for a Goppa code $\Gamma(L, G)$ with $\deg(G(z)) = r$, the minimum distance d has a lower bound [6]

$$d \geq r + 1$$

and the dimension k has an upper and lower bound

$$n - mr \leq k \leq n - r$$

5.2 Efficient Decoding

As Goppa codes are a subclass of alternant codes, efficient decoding algorithms do exist. Efficient decoding algorithms for these codes are not discussed in this paper, but they are available in other resources [6].

5.3 Ball-collision decoding on Goppa Codes

Now we will test the performance of the ball-collision decoding algorithm on some big Goppa codes. The runtime of the algorithm will obviously differ depending on the machine it is run on and the way in which the algorithm is implemented. However, it still gives us a good idea of what the algorithm is capable of and how it will scale depending on parameter values and the amount of errors being corrected.

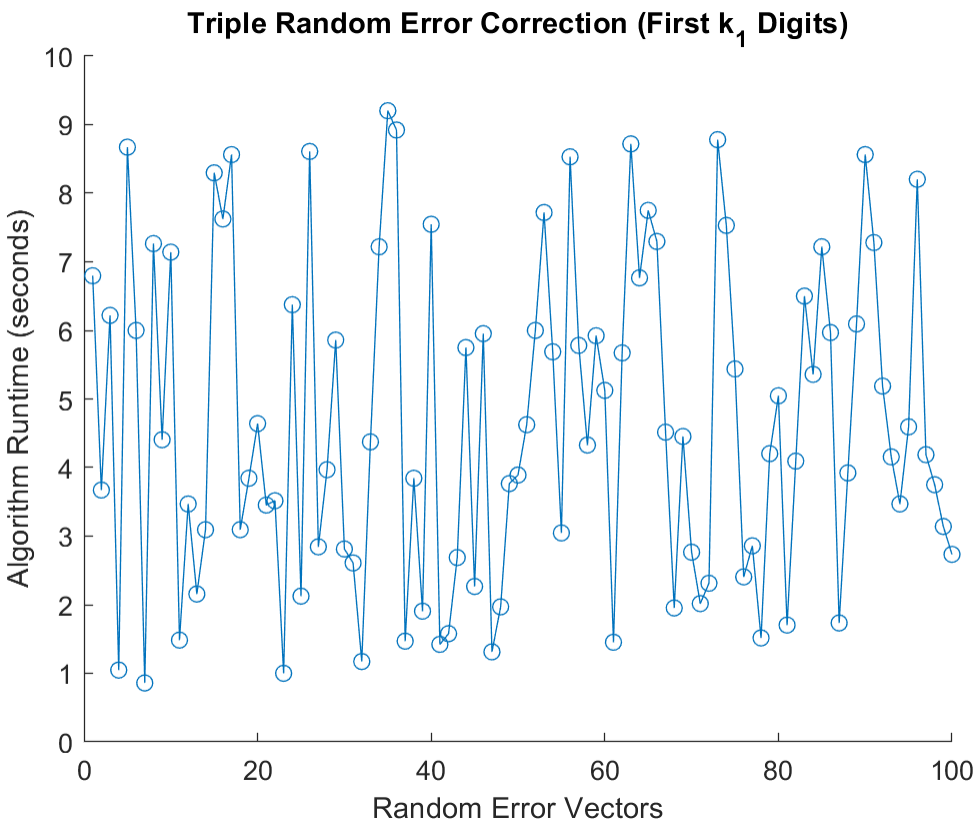
Our goal here will first be to analyze the ball-collision decoding algorithm's performance on correcting few errors, and then progress to see how many errors we are able to correct assuming a reasonable and non-trivial distribution of errors.

Example 5.3.1. (Triple Random Error Correction)
Consider the parameter values for the Goppa code of length $n = 2^9 - 1 = 511$ generated by $G(z) = z^{31} + \alpha z^2 + \alpha$

$$k_1 = k_2 = 116,\ \ell_1 = \ell_2 = 60$$

$$p_1 = 3,\ p_2 = q_1 = q_2 = 0$$

Consider an error vector e of length $n = 511$ with 3 random errors in the first k_1 digits indexed by an information set Z. With these parameter values, the algorithm will always succeed. We run the algorithm 100 times, each with a different random error vector e. Runtime results are plotted below.



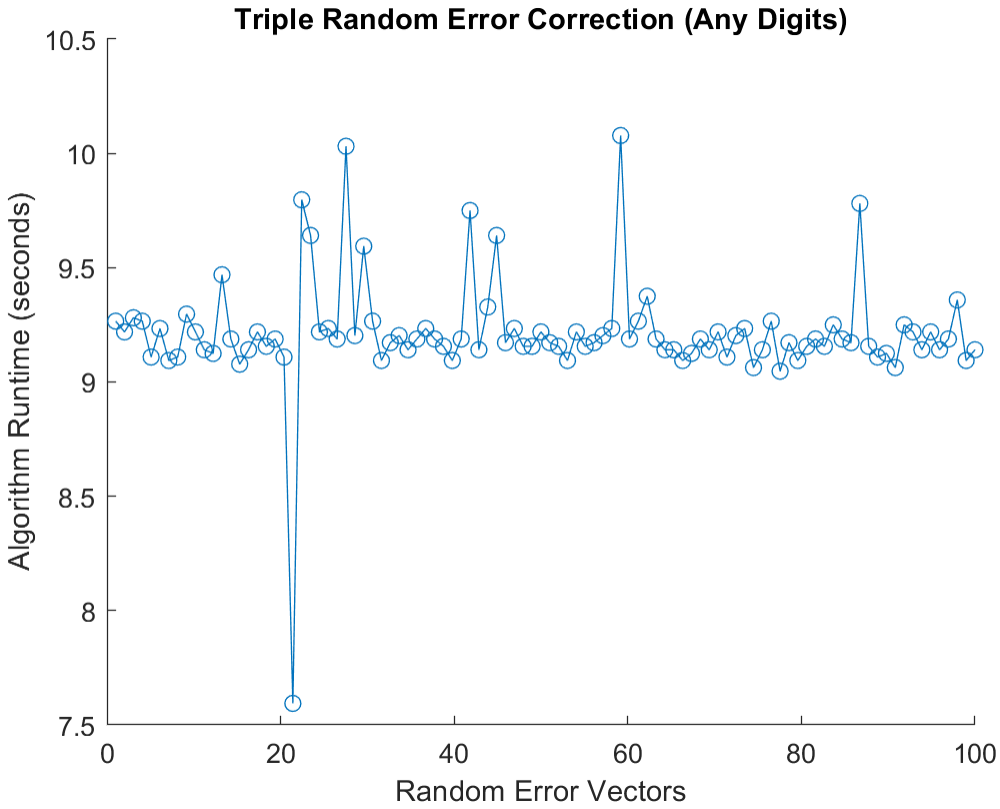
Since the algorithm is done as soon as the error vector is found, we have a variety of completion times. The average runtime is ≈ 4.6585 seconds.

Now, lets look at an error vector e of length $n = 511$ with 3 random errors in any digits. Consider the same code and the same parameters. The errors are not isolated to the first k_1 digits indexed by Z, so the algorithm is not guaranteed to succeed. The success probability is[1]

$$\binom{n}{w}^{-1}\binom{n-k-\ell_1-\ell_2}{w-p_1-p_2-q_1-q_2}\binom{k_1}{p_1}\binom{k_2}{p_2}\binom{\ell_1}{q_1}\binom{\ell_2}{q_2}$$

$$=\binom{511}{3}^{-1}\binom{159}{0}\binom{116}{3}\binom{116}{0}\binom{60}{0}\binom{60}{0}=\frac{253460}{22108415}\approx .0114$$

So, the algorithm has a $\approx 1\%$ chance of succeeding. Again, we run the algorithm 100 times, each with a different random error vector e. Runtime results are plotted below.



The average runtime for the above plot is ≈ 9.2221 seconds. As we can see, the runtime is much more consistent when the algorithm does not succeed, as it likely has to compute a similar amount of iterations no matter the error vector. The discrepancy around position 21 in the above plot with a runtime of ≈ 7.6 seconds is likely an error vector for which the algorithm succeeded. This matches the success probability we outlined above as well, since the algorithm succeeded for 1 out of 100 of our random error vectors.

Example 5.3.2. (Triple Burst Error Correction)
When all the errors in an error vector are contained within a small chunk of bits, it is considered a burst error [5]. For instance, an error vector

$$(0\ 0\ 0\ |1\ \underset{burst}{1\ 0\ 1}|\ 0\ 0\ 0)$$

would have a burst length of 4, as all errors happen within 4 positions of each other. For simplicity sake, we will only be considering bursts in which the burst length is exactly equal to the number of errors (ie, there are no 0's within our burst chunk).

Consider the same Goppa code and parameters as our previous example; the Goppa code of length $n = 2^9 - 1 = 511$ generated by $G(z) = z^{31} + \alpha z^2 + \alpha$ and parameter values

$$k_1 = k_2 = 116,\ \ell_1 = \ell_2 = 60$$

$$p_1 = 3,\ p_3 = q_1 = q_2 = 0$$

Consider an error vector e with a burst of length 3 containing 3 errors. The position of the errors will be $a, a+1$, and $a+2$ from $a=1$ to $a=110$. So, the error vectors we are introducing will look like

$$(1\ 1\ 1\ 0\ 0\ 0\ \cdots)$$
$$(0\ 1\ 1\ 1\ 0\ 0\ \cdots)$$
$$(0\ 0\ 1\ 1\ 1\ 0\ \cdots)$$
$$(0\ 0\ 0\ 1\ 1\ 1\ \cdots)$$

and so on. All errors are in the first k_1 digits indexed by Z, and the algorithm will always succeed. We run the algorithm for the varying values of a and plot the runtime results below.

As we can see, there seems to be a fairly linear progression with our burst position. With our implementation of the algorithm, error vectors with all the errors concentrated in the first bits of the vector seem to be found the quickest.

Capacities of ball-collision decoding

We'll now move to bigger error correction. Consider the Goppa code of length $n = 2^9 - 1$ generated by $G(z) = z^{34} + \alpha z^7 + \alpha$ with dimension $k = 205$ and error correcting capability $t \geq 17$. Set the following parameters

$$k_1 = 103, k_2 = 102, \ell_1 = \ell_2 = 15$$

Our corresponding partitions then make up the following percentages of the code in terms of size

$$k_1 \approx k_2 \approx 20\%$$
$$\ell_1 = \ell_2 \approx 3\%$$
$$n - k - \ell_1 - \ell_2 \approx 54\%$$

We'll use this in finding the best success probabilities for a given distribution of errors.

10 errors

Consider the values

$$p_1 = p_2 = 2, q_1 = q_2 = 0$$

and a random error vector of weight $\mathrm{wt}(e) = 10 = w$. The success probability for a single iteration of ball-collision decoding is

$$\binom{511}{10}^{-1}\binom{276}{6}\binom{103}{2}\binom{102}{2}\binom{15}{0}\binom{15}{0} \approx 5.13\%$$

Based on our values, we are assuming the position of our errors roughly matches the percentages of each partition. In other words, 20% of our errors are in the partition of k_1, 20% of our errors are in the partition of k_2, and 60% of our errors are in the partition of $n - k - \ell_1 - \ell_2$. This means our success probability for a single iteration of ball-collision decoding will be nearly maximized.

In our 20 simulated attempts, the algorithm succeeded exactly once, matching our success probability with a runtime of 46.000 seconds.

Cherry-picked Errors

For larger error corrections, we will be introducing error vectors with a structure that matches our parameter values p_1, p_2, q_1, q_2, assuring that the algorithm will always succeed in finding the error vector. We will still discuss the success probability as necessary, but it is more advantageous for us to look at only the cases that will succeed.

Our errors will be picked in a roughly similar distribution to our distribution of partitions, with slightly more errors in the $n - k - \ell_1 - \ell_2$ partition. This will slightly lessen the success probability, but will drastically lessen runtime. All further examples will use the code generated by the Goppa polynomial $n = 2^9 - 1 = 511$ generated by $G(z) = z^{31} + \alpha z^2 + \alpha$ and parameter values

$$k_1 = k_2 = 116,\ \ell_1 = \ell_2 = 60$$

11 Errors

Consider an error vector with 11 errors. Consider 2 errors in the partition corresponding to k_1, 2 errors in the partition corresponding to k_2, and 7 errors in the partition corresponding to $n - k - \ell_1 - \ell_2$. The success probability is

$$\binom{511}{11}^{-1}\binom{276}{7}\binom{103}{2}\binom{102}{2}\binom{15}{0}\binom{15}{0} \approx 4.3\%$$

Run the algorithm with $p_1 = p_2 = 2, q_1 = q_2 = 0$.

Successful Decoding Time: 50.594 seconds

12 Errors

Consider an error vector with 11 errors. Consider 2 errors in the partition corresponding to k_1, 2 errors in the partition corresponding to k_2, and 8 errors in the partition corresponding to $n - k - \ell_1 - \ell_2$. The success probability is

$$\binom{511}{12}^{-1}\binom{276}{8}\binom{103}{2}\binom{102}{2}\binom{15}{0}\binom{15}{0} \approx 3.5\%$$

Run the algorithm with $p_1 = p_2 = 2, q_1 = q_2 = 0$.

Successful Decoding Time: 45.844 seconds

13 Errors

Consider an error vector with 11 errors. Consider 3 errors in the partition corresponding to k_1, 2 errors in the partition corresponding to k_2, and 8 errors in the partition corresponding to $n - k - \ell_1 - \ell_2$. The success probability is

$$\binom{511}{13}^{-1}\binom{276}{8}\binom{103}{3}\binom{102}{2}\binom{15}{0}\binom{15}{0} \approx 3.1\%$$

Run the algorithm with $p_1 = 3, p_2 = 2, q_1 = q_2 = 0$.

Successful Decoding Time: 309.938 seconds

14 Errors

Consider an error vector with 14 errors. Consider 3 errors in the partition corresponding to k_1, 2 errors in the partition corresponding to k_2, and 9 errors in the partition corresponding to $n - k - \ell_1 - \ell_2$. The success probability is

$$\binom{511}{14}^{-1}\binom{276}{9}\binom{103}{3}\binom{102}{2}\binom{15}{0}\binom{15}{0} \approx 2.6\%$$

Run the algorithm with $p_1 = 3, p_2 = 2, q_1 = q_2 = 0$.

Successful Decoding Time: 433.547 seconds

15 Errors

Consider an error vector with 15 errors. Consider 3 errors in the partition corresponding to k_1, 2 errors in the partition corresponding to k_2, and 10 errors in the partition corresponding to $n-k-\ell_1-\ell_2$. The success probability is

$$\binom{511}{15}^{-1}\binom{276}{10}\binom{103}{3}\binom{102}{2}\binom{15}{0}\binom{15}{0} \approx 2.1\%$$

Run the algorithm with $p_1 = 3, p_2 = 2, q_1 = q_2 = 0$.

Successful Decoding Time: 179.578 seconds

6 Conclusions

In this paper, we implemented the ball-collision decoding algorithm and successfully corrected 15 errors in a Goppa code of length $2^9 - 1$ in reasonable time, coming within 2 errors of the lower bound for this code's error correcting capability. This is far from feasible in breaking the McEliece scheme. However, we can still justify recommended parameter choices of Goppa codes in the McEliece Cryptosystem from the data collected.

6.1 Iteration Cost

The cost of a single iteration of ball-collision decoding is [1]

$$\begin{aligned}
&\frac{1}{2}(n-k)^2(n+k) + (\ell_1+\ell_2)\left(\sum_{i=1}^{p_1}\left(\binom{k_1}{i}\right) + \sum_{i=1}^{p_2}\left(\binom{k_2}{i}\right) - k_1\right) \\
&+\min\{1,q_1\}\binom{k_1}{p_1}\sum_{i=1}^{q_1}\left(\binom{\ell_1}{i}\right) + \min\{1,q_2\}\binom{k_2}{p_2}\sum_{i=1}^{q_2}\left(\binom{\ell_2}{i}\right) \\
&+2(w-p_1-p_2-q_1-q_2+1)(p_1+p_2)\binom{k_1}{p_1}\binom{k_2}{p_2}\binom{\ell_1}{q_1}\binom{\ell_2}{q_1}2^{-\ell_1-\ell_2}
\end{aligned}$$

In the case of our 15 error correction, this cost is

$$\begin{aligned}
&\frac{1}{2}(306)^2(716) + (30)\left(\sum_{i=1}^{3}\left(\binom{103}{i}\right) + \sum_{i=1}^{2}\left(\binom{102}{i}\right) - 103\right) \\
&+110\binom{103}{3}\binom{102}{2}\binom{15}{0}\binom{15}{0} \\
&= 100244687508 \approx 10^{11}
\end{aligned}$$

Since the algorithm ends as soon as the error vector is found, this is the upper bound for the number of operations for the given parameters.

6.2 McEliece Parameters

Our analysis of ball-collision decoding, though far from comprehensive, aids us in choosing secure parameters for McEliece. Bernstein, Lange, and Peters recommend codes with parameters [3178,2384,68] for 128-bit security and codes with parameters [6944,5208,136] for 256-bit security [1].

Goppa codes opt for a tradeoff of minimum distance and dimension, with higher degree Goppa polynomials guaranteeing a larger minimum distance with the consequence of a lower dimension and vice versa, as

$$d \geq r + 1 \quad \text{and} \quad n - mr \leq k \leq n - r$$

With ball-collision decoding, heavy computation occurs in sets S and T with $\binom{k_1}{p_1}\binom{k_2}{p_2}\binom{\ell_1}{q_1}\binom{\ell_2}{q_2}$ choices of (x_0, x_1, y_0, y_1) [1]. Parameter restrictions allow for us to pick low values of ℓ_1, ℓ_2, q_1, q_2 to lessen operations. Parameter restrictions do not allow much leeway in our choices for p_1, p_2, k_1, k_2 as $k_1 + k_2 = k = |Z|$. If k is of sufficient size, then the success probability for small values of p_1, p_2 will be impractical, as we would be assuming most of the errors occurr in $\{1, ..., n\} \setminus Z$ which has size $|\{1, ..., n\} \setminus Z| = n - k$. Thus, we disregard standard coding theory practice of having $\frac{k}{n} \approx \frac{1}{2}$, as larger k values with a still reasonable error correcting capability are most beneficial to us. This aligns with the security parameters proposed in the ball-collision decoding paper, with $\frac{2384}{3178} \approx \frac{3}{4}$ and $\frac{5208}{6944} \approx \frac{3}{4}$.

References

[1] Daniel J. Bernstein, Tanja Lange, Christiane Peters, *Smaller decoding exponents: ball-collision decoding*, 2011.5.27

[2] Matthieu Finiasz, *NP-completeness of Certain Sub-classes of the Syndrome Decoding Problem*, 2009.12.02

[3] Suanne Au, Christina Eubanks-Turner, Jennifer Everson, *The McEliece Cryptosystem*, 2003.09.17

[4] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *Handbook of Applied Cryptography*, 1996

[5] D.C. Hankerson, Gary Hoffman, D.A. Leonard, Charles C. Lindner, K.T. Phelps, C.A. Rodger, J.R. Wall, *Coding Theory and Cryptography: The Essentials* Second Edition, Chapman & Hall/CRC Pure and Applied Mathematics, 2000.08.04

[6] F.J. MacWilliams, N.J.A. Sloane, *The Theory of Error-Correcting Codes*, Volume 16, North-Holland Mathematical Library, 1983.01.01

Appendix

Full Code for the Ball-collision Decoding Algorithm

```
time for x:=1 to 1 do

n:= ;
k:= ;
w:= ;

k1:= ;
k2:=k-k1;
l1:= ;
l2:= ;

p1:= ;
p2:= ;
q1:= ;
q2:= ;

H:= ;
s:= ;

H:=Transpose(H);

Z:={};
Zc:={};
Zcv:={};
RS:={1..n};

repeat
  ran:=Random(RS);
  Zct:=Zcv join {H[ran]};
  if IsIndependent(Zct) eq true then
    Zcv:=Zct;
    Zc:=Zc join {ran};
  end if;
  RS:=RS diff {ran};
until #Zc eq n-k;

Z:={1..n} diff Zc;

H:=Transpose(H);

Fk1:=RandomSubset(Z, k1);
Fk2:=Z diff Fk1;

Fl1:=RandomSubset(Zc, l1);
```

```
Zi:= Zc diff Fl1;
Fl2:=RandomSubset(Zi, l2);
Fnkl1l2:= Zi diff Fl2;

Z:=SetToSequence(Z);
Zc:=SetToSequence(Zc);
V:=Matrix(GF(2),n-k,n-k,[]);

for i:=1 to n-k do
  for j:=1 to n-k do
    V[i,j]:=H[i,Zc[j]];
  end for;
end for;

U:=V^-1;
UH:=Transpose(U*H);
A:=Matrix(GF(2),k,n-k,[]);

for i:=1 to #Z do
  A[i]:=UH[Z[i]];
end for;

A:=Transpose(A);
A1:=ExtractBlock(A, 1, 1, l1+l2, k);
A2:=ExtractBlock(A, n-k-(n-k-l1-l2)+1, 1,n-k-l1-l2, k);

Us:=U*s;
s1:=ExtractBlock(Us, 1, 1, l1+l2, 1);
s2:=ExtractBlock(Us, n-k-(n-k-l1-l2)+1, 1,n-k-l1-l2, 1);

T:=[];
ct:=1;
T1:=Subsets({1..k2}, p2);
T2:=Subsets({1..l2}, q2);

for a in T1 do
  y0:=Matrix(GF(2),k,1,[]);

  if #T1 ne 0 then
    for b in a do
      y0[k1+b,1]:=1;
    end for;
  end if;

  for c in T2 do
    y1:=Matrix(GF(2),l1+l2,1,[]);
    if #T2 ne 0 then
      for d in c do
```

```
        y1[l1+d,1]:=1;
      end for;
    end if;

    v:=A1*y0+y1+s1;

    if #T eq 0 then
      Append(~T,<v,{<y0,y1>}>);
    else
      for h:=1 to ct do
        if T[ct][1] eq v then
          T[ct][2]:=T[ct][2] join {<y0,y1>};
break h;
        else
          Append(~T,<v,{<y0,y1>}>);
          ct:=ct+1;
        end if;
      end for;
    end if;
  end for;
end for;

S:=[];
ct:=1;
S1:=Subsets({1..k1}, p1);
S2:=Subsets({1..l1}, q1);

for a in S1 do
  x0:=Matrix(GF(2),k,1,[]);

  if #S1 ne 0 then
    for b in a do
      x0[b,1]:=1;
    end for;
  end if;

  for c in S2 do
    x1:=Matrix(GF(2),l1+l2,1,[]);

    if #S2 ne 0 then
      for d in c do
        x1[d,1]:=1;
      end for;
    end if;

    v:=A1*x0+x1;

    for r:=1 to #T do
```

```
      if T[r][1] eq v then
        for dl in T[r][2] do
          cl:=Vector(A2*(x0+dl[1])+s2);
          if Weight(cl) eq w-p1-p2-q1-q2 then

            t1:=x0+dl[1];
            t2:=x1+dl[2];
            t3:=A2*(x0+dl[1])+s2;

            R:=Matrix(GF(2),n,1,[]);
            for i in Z do
              R[i]:=t1[Index(Z, i)];
            end for;

            for i:=1 to l1+l2 do
              R[Zc[i]]:=t2[i];
            end for;

            for i:=1 to n-k-l1-l2 do
              R[Zc[i+l1+l2]]:=t3[i];
            end for;

            print Vector(R);
            break x;

          end if;
        end for;
      end if;
    end for;
  end for;
end for;

end for;
```