# Implementing Ball Collision Decoding

Justin Cruz

December 18, 2019

## 1 Introduction

Cryptography is a study that focuses on the study of creating and solving codes, through the use of binary digits, namely 0 and 1. In this field, Cryptography is essential in how our world communicates with each other, as it protects and raises standards to promote a secure environment of communication. While Cryptography focuses on protecting the way our world shares information, it also includes the study on extracting information through the use of algorithms. In our case, we will be focusing on Ball Collision Decoding.

## 2 Abstract

This paper will discuss and analyze the implementation of Ball Collision Decoding through the computer language of *Python*. We will go over the necessary parameters needed to perform our algorithm to understand how the algorithm works. We will then summarize the steps using our parameters to understand the relationship between the parameters and their contribution to the algorithm. Finally we will analyze our attempt of implementing the Ball Collision Decoding algorithm.

## 3 Parameters

**Note:** All parameters and information regarding the Ball Collision Decoding can be found on page 5 of *Ball-collision decoding* by Bernstein, Lange, and Peters.

**Constants:** $n$ , $k$ , $w \; \varepsilon \; \mathbf{Z}$ and $0 \leq k \leq n$

**Parameters:** $p_1, p_2, q_1, q_2, k_1, k_2, l_1, l_2 \; \varepsilon \; \mathbf{Z}$ with $0 \leq k_1, 0 \leq k_2$, $k = k_1 + k_2, 0 \leq p_1 \leq k_1, 0 \leq p_2 \leq k_2, 0 \leq q_1 \leq l_1, 0 \leq q_2 \leq l_2$, and $0 \leq w - p_1 - p_2 - q_1 - q_2 \leq n - k - l_1 - l_2$

**Input**: $H \; \varepsilon \; \mathbf{F}_2^{(n-k) \; x \; n}$ and $s \; \varepsilon \; \mathbf{F}_2^{(n-k)}$

**Output:** zero or more vectors $e \; \varepsilon \; \mathbf{F}_2^n$ with $He = s$ and $\mathrm{wt}(e) = w$

## 3.1 Defining Constants

Given a $\mathbf{G}_{k \; x \; n}$ generator matrix, our constants $k$ denotes the number of rows, while $n$ denotes the number of columns. From the $k$ number of rows, we have a variety of bounded numbers, namely $p_1, p_2, q_1, q_2, k_1, k_2, l_1, l_2$. For now, we define these bounded variables as random numbers generated from their respective boundaries as we define them later. In addition we have $w$, which denotes the weight of our error pattern, $e$. We define $w$ as the total amount of 1's inside any code word. For this algorithm, $w$ will be the particular weight for $e$. The error pattern $e$ is a binary code word of length $n$ which will be the primary focus of our decoding algorithm.

## 3.2 Defining Parameters

From our information set $\mathbf{Z}$, we have $p_1, p_2, q_1, q_2, k_1, k_2, l_1, l_2 \quad \varepsilon \quad \mathbf{Z}$. Given a generator matrix $\mathbf{G}_{k \; x \; n}$, we define the *information set*,{1,2,...,n}, as any $k$ numbers from the set $[n]$, which index the columns of $\mathbf{G}$, that satisfy having a determinant of 0. From the *information set*, we partition $k$ into sizes of $k_1$ and $k_2$. Similarly, one the *information set* has been established, we define the *partition set*, {1,2,...,n}\Z, as the rest of the numbers in [n] that were not a part of the *information set*, such that each number in {1,2,...,n}\Z indexes the column number in $\mathbf{G}$. Since the *partition set* is the remaining column numbers from the *information set*, the *partition set* has a size of $n-k$. With a size $n-k$, we partition $n-k$ into sizes $l_1$, $l_2$, and $n-k-l_1-l_2$. In addition, $p_1, p_2, q_1, q_2$ will be bounded by their respective k and l terms.

### 3.3 Defining Input:

$H \varepsilon \mathbf{F}_2^{(n-k)\ x\ n}$ binary parity check matrix which is derived from $\mathbf{G}$. The *parity check matrix* will be used to index sub matrices $A_1$ and $A_2$ that will appear later in the algorithm. The *syndrome*, $s$, is defined as the column vector of the product $He = s$. The syndrome will be used later in the algorithm to serve as a part of a conditional statement for weight.

# 4 Overview

**Note:** All steps for the Ball Collision Decoding can be found on page 5 of *Ball-collision decoding* by Bernstein, Lange, and Peters.

We will breakdown the steps of the BC algorithm as follows:

**1.** Create an information set Z.
**2.** Partition the size of Z into sizes $k_1$ and $k_2$.
**3.** Create the partition set {1,2,...,n}\Z and partition the size into $l_1$, $l_2$, and $n - k - l_1 - l_2$
**4.** Create a inverse matrix $U \varepsilon \mathbf{F}_2^{(n-k)\ x\ (n-k)}$ such that $U$ is found from taking the inverse of a sub matrix whose columns are indexed by the partition set. Then create $UH$ as $\binom{A_1}{A_2}$ , such that $A_1 \varepsilon \mathbf{F}_2^{(l_1+l_2)\ x\ k}$, $A_2 \varepsilon \mathbf{F}_2^{(n-k-l_1-l_2)\ x\ k}$
**5.** Create $Us$ as $\binom{s_1}{s_2}$ such that $s_1 \varepsilon \mathbf{F}_2^{l_1+l_2}$, $s_2 \varepsilon \mathbf{F}_2^{n-k-l_1-l_2}$
**6.** Create the Set $S$ containing all triples $(A_1x_0 + x_1, x_0, x_1)$ such that $x_0 \varepsilon \mathbf{F}_2^{k_1} \times \{0\}^{k_2}$, $wt(x_0) = p_1, x_1 \varepsilon \mathbf{F}_2^{l_1} \times \{0\}^{l_2}$, $wt(x_1) = q_1$.
**7.** Create the Set $T$ containing all triples $(A_1y_0 + y_1, y_0, y_1)$ such that $y_0 \varepsilon \{0\}^{k_1} \times \mathbf{F}_2^{k_2}$, $wt(y_0) = p_2, y_1 \varepsilon \{0\}^{l_1} \times \mathbf{F}_2^{l_2}$, $wt(y_1) = q_2$.
**8.** For each $(v, x_0, x_1) \varepsilon \mathbf{T}$ :
For each $y_0, y_1$ such that $(v, y_0, y_1) \varepsilon T$:
If $wt(A_2(x_0 + y_0) + s_2) = w - p_1 - p_2 - q_1 - q_2$ :
Output the vector $e \varepsilon \mathbf{F}_2^n$ whose Z-indexed components are $x_0 + y_0$ and whose remaining components are $(x_1 + y_1 || A_2(x_0 + y_0) + s_2)$

# 5 Implementation

We will look at the implementation of the BC algorithm step by step. For each step, we will interpret the Python standards and conventions, used to create the implemented Python code called **"ballCol"**, which will serve as the main function that performs all the steps, including its sub functions that will assist and sometimes serve as a step itself.

## 5.1 Step 1:

For this step, we must create the information set Z. As mentioned before, Z comes from extracting information about our generator matrix **G**. In Python terms, this means the given generator matrix will be used as a parameter for our Python function. We start by creating the function "**createInfo**".

```
def createInfo(gmat):

    # creating a matrix until we find
    # a info set where det != 0

    # determinant
    det = 0

    while(det == 0):

        # row size == k
        k = len(gmat)

        # column size == n
        n = len( gmat.transpose() )

        info_set = []              # Z-set , size n
        info_part = range(1,n+1)   # partitioned Z-set , size n-k

        while(k != 0):
            ii = random.choice(info_part)
            info_set += [ii]
            info_part.remove(ii) # removes number for no duplicates
            k -= 1

        # gmat tranpose matrix
        g_tran = gmat.transpose()

        # info set
        info = []

        # creating info matrix by getting each
        # column in info_set


        for col in info_set:
            info.append(g_tran[col-1])

        # transforms info into a matrix
        info_mat = np.array(info)

        # determinant of info_mat
        det = round(np.linalg.det(info_mat))

        detmod2 = det%2

        if( detmod2 != 0 ):
            return [ info_set , info_part ]
```

Figure 1: Python demonstration of creating the information set

**Note:** Throughout the entire code, we will assume any matrix that serves as an input, will be a "list of lists" that has been converted into an array.

From Figure 1, we see the source code of what creating the information set would look like. The input for the **createInfo** function will be an array/vector version of **G**. Using the input of **G**, called "*gmat*", we create the values k and n by setting k as the number of rows *gmat* contains, while n is the number columns, which can be interpreted as taking the transpose of *gmat*, and counting the number of rows it contains, equivalent to the number of columns *gmat* has.

Next, we define the empty information and partition sets as *"info_set"* and *"info_part"*. Then, we define *"info_part"* as the range of numbers from 1 through n. From *"info_part"*, we choose k-numbers from 1 through n, and for each number chosen, that number will be a part of *"info_set"*, giving us two distinct lists. Once we have our *"info_set"*, we create another empty list called *"info"*, where info will be later transformed into a matrix to test if its determinant will be nonzero. Once we create *"info"*, we grab columns from *"gmat"*, which were indexed by each element inside *"info_set"*. Once all the indexed columns have been extracted to *"info"*, we perform the determinant modulo 2. If determinant modulo 2 is nonzero, then we return both *"info_part"* and *"info_set"*. However, because each step is inside a while loop, the function will keep repeating, and initialize itself, until we find a set of columns that satisfy the condition.

```
def ballCol(gmat,Hmat,err):
    '''
    STEP 1
    '''

    #info set properties
    sets = createInfo(gmat)

    # Z-Set
    set_Z = sets[0]

    # Z-Partition
    part_Z = sets[1]

    '''
```

Figure 2: Step 1 inside the main ballCol function

Looking at Figure 2, we see that using the generator, parity check, and error as inputs, Step 1 uses *gmat* as an input for **createInfo**, which returns the desired partition and information set in a form of a list, where the first element of the return value is the information set and the second element is the partition. We then set the result of **createInfo** to a variable *sets*. Because *sets* has return type of "list", we define its first element as *set_Z* and the second element as *part_Z*. Hence, first step is done.

## 5.2 Step 2

```
'''
STEP 2
'''
# Partitioning k into size k1 & k2 , where k1 + k2 = k
k = len(set_Z)
k1 = random.randint(0,k)
k2 = k - k1
```

Figure 3: Step 2 inside ballCol, where k is separated into sizes $k_1, k_2$

Because Step 2 is just creating different sizes from a given parameter of k, we can perform this step inside **ballCol**. Once we get the value of k, we find $k_1$ by choosing a random number between 0 and k. Once $k_1$ is found, we simply subtract $k_1$ from k to give us $k_2$.

## 5.3 Step 3

```
STEP 3
'''
# Partitioning L into L1,L2, n-k-L1-L2 , where L1 + L2 = L
L = len(part_Z)
L1 = random.randint(0,L)
L2 = L - L1
n_k_l_2 = (k+L) - L1 - L2
```

Figure 4: Step 3 inside ballCol, where l is separated into sizes $l_1, l_2, n-k-l_1-l_2$

Similar to Step 2, Step 3 performs the same procedure, except the code calculates $l_1, l_2, n-k-l_1-l_2$.

## 5.4 Step 4

```
'''
STEP 4
'''
# Find a (n-k)*(n-k) U matrix such that U*V = Identity_(11 x 11)
# Creating V, st, V is all columns in part_Z

V = []

# adding each column in part_z to V
for col in part_Z:
    V.append(Hmat[:,col-1])

# Inverse Matrix U created from
arrV = np.array(V)
U = invMod(arrV)

# matrix U*H
UH = (np.mat(U))*(np.mat(H))%2

# creating A1 and A2
A = []
A_1 = []


for col in set_Z:
        A.append(UH[col-1])

Amat = np.array(A)

# creating A1
for row in xrange(0,L1+L2+1):
    A_1.append(Amat[row])
    Amat.remove(row)

A1 = np.array(A_1)

#creating A2
A2 = Amat
```

Figure 5: Step 4 inside ballCol, where U is found

In this step, we are calculating U by first creating V such that V will be a matrix where the columns of the parity check matrix are indexed by the partition set. Once V is created, U is found by taking the inverse modulo 2, using the function, **"invMod"**. Next, U and H are multiplied to obtain the matrix UH. Once UH is found, we are creating a matrix A such that its contents will be the columns of UH, indexed by the information set. Then, we create sub matrices $A_1$ and $A_2$.

```
# MODULAR INVERSE MATRIX

def invMod(mat):
    return np.linalg.inv(mat)%2
```

Figure 6: inverseMod function that takes the modulo 2 inverse of any matrix

## 5.5 Step 5

```
'''
STEP 5
'''
# Split U*s into s_1 and s_2, st, s = H*e
s = (np.mat(Hmat))*(np.mat(err))

US = (np.mat(U))*(np.mat(s))

USmat = np.array(US)

# creating s1 and s2

s1 = []

for row in xrange(0,L1+L2+1):
    s1.append(USmat[row])
    USmat.remove(row)

s_2 = USmat
s_1 = np.array(s1)
```

Figure 7: Step 6 in ballCol, where $s_1, s_2$ are created

In Step 6, we are creating $s$ by taking the product of $s = He$, then we take the product of $Us = U \times s$. Once we have $Us$, we create sub vectors of $Us$ to get $s_1, s_2$.

## 5.6 Step 6 & 7

```
def createS(k1,k2,L1,L2,A1):
    '''
    creates x0 and x1 terms
    '''
    #creating v
    v = []

    # creating x0
    x0 = []
    zeros_x0 = list(repeat(0, k2))

    for i in range(1<<k1):
        s=bin(i)[2:]
        s='0'*(k1-len(s))+s
        arr0 = (map(int,list(s)))

        #zeros_x0 = list(repeat(0, k2))

        if( sum(arr0) == 3 ):
            arr0.extend(zeros_x0)
            x0 += [arr0]
            vcode = (np.mat(A))*(np.mat(arr0.transpose()))
            v += [vcode]

    # creating x1
    x1 = []
    zeros_x1 = list(repeat(0, L2))

    for j in range(1<<L1):
        s=bin(j)[2:]
        s='0'*(L1-len(s))+s
        arr1 = (map(int,list(s)))

        #zeros_x1 = list(repeat(0, L2))

        if( sum(arr1) == 3 ):
            arr1.extend(zeros_x1)
            x1 += [arr1]

    v_sums = []
    count = 0
    while (count < len(x1)):
        v_sums += [np.array(v[count]) + np.array(x1[count])]
        count += 1

    return [v_sums ,x0 , x1]
```

Figure 8: createS function, creates all sets of triples

In order to perform Step 7, we must be able to obtain the set of all possible triples for S and T. To do this, we create a function called **createS** (createT for the set T, which has the exact same inputs). Our inputs for the function will be $k_1, k_2, l_1, l_2, A_1$, which are all defined within the main ballCol function. We use the help of a Python library called "itertools", which will output all possible binary combinations of size $k_1$. However, because we are looking for triples, we set a conditional statement to only extract the $x_0, x_1$ codes that have weight $= 3$. If the condition is met, then $k_2$ amount of zeros are appended to the respective $x_0, x_1$ codes. In addition to $x_0$, because Step

6 requires the product of $A_1x_0$, we take the matrix multiplication of $A_1$ and $x_0$ and add that result into a list called $v$. Once all $x_0, x_1$ are found, we take each value from $v$, and take the sum of each value in the list *x1*, and add those values inside a list called *v_sum*. Once everything is found, we return a list a lists, such that the first list inside the return statement will be *v_sum*, while the second list will be $x_0$, and the last list will be $x_1$. The same procedure is done for set T using the function, **createT**.

## 5.7 Step 8

```
'''
STEP 8
'''
x = createS(k1,k2,L1,L2,A1)
v_x = createS[0]
x0 = createS[1]
x1 = createS[2]

y = createT(k1,k2,L1,L2,A1)
v_y = createT[0]
y0 = createT[1]
y1 = createT[2]

w = sum(err)

for x_1 in x1:
    for y_1 in y1:
        for x_0 in x0:
            for y_0 in y0:
                p1 = sum(x_0)
                q1 = sum(x_1)
                p2 = sum(y_0)
                q2 = sum(y_1)
                x0y0 = x_0 + y_0
                x0y0_tran = x0y0.transpose()
                if ( sum( np.mat(A2)*np.mat(x0y0_tran) + s_2 ) == w - p1 - p2 ):
                    for col in set_Z:
                        err[col-1] = x_0 + y_0
                    for col in part_Z:
                        x1y1 = x_1 + y_1
                        err[col-1] = x1y1.append( np.mat(A2)*np.mat(x_0+y_0) + s_2 )

                    return error
```

Figure 9: Step 8 of ballCol, where all triples are met with the conditional

In Step 8, all the parameters from the return statements of both *createS* and *createT* are set as variables to prepare for the conditional statements. Because we are using information from $x_0, x_1, y_0, y_1$, we use a nested for loop to compare all the variables together for the condition. Although only $x_0$

and $y_0$ are the only values from set S and set T being used for addition, we are still considering the weights of $x_1$ and $y_1$. Hence, their information is stored before the conditional statement. In addition, the sum of $x_0$ and $y_0$ are transposed before performing matrix multiplication with $A_2$ because, **createS** and **createT** both have a return type of a nested list. Because each list is treated like a row vector, we transpose in order to have proper matrix multiplication conditions. Once the loop hits the if-statement, we are looking to see if the our conditions are met so we can return an updated version of the error pattern. If the conditions are successful, we update the error pattern, *"err"*, such that its Z-indexed components are $x_0 + y_0$ while the rest of the components, namely the partitioned components will be the concatenation of $(x_1 + y_1)$ and $A_2(x_0 + y_0) + s_2$.

# 6 Technical Problems and Solutions

## 6.1 Rounding Error

Throughout the creation of this code, there were a few minor issues that came across while creating specific parts of the projects. One issue that came up was modulo 2 computation. Because of the rules of binary algebra, the BC code must follow rules of modulo 2 operations. This problem arrived during the computation of Step 1. When we are computing the determinant of any matrix, we expect our answer to be in modulo 2. In Python, this is as simple as setting your answer to *det* % 2. However, when I tested Python's built in determinant function on all integers, I noticed that the numpy (Python library) version of computing its determinant had an error of 1 digit from its true value. For example, multiplying $250 \times 16$ would give 4000. However, Python was giving answers of 3999. With this issue, the determinant modulo 2 would drastically change its answer, ie) 3 % 2 ! = 4 % 2. To address this, I simply added a *round(det)* % 2 to get the correct answer before performing modulo 2.

## 6.2 Efficiency of Set S and Set T

The process of creating all possible triples for a given k value is taxing. By brute force, it is unreasonable to call k amount of for-loops in order to

consider three positions of a 1's inside a code word. To fix this, I used numpy's built in library called *itertools* in order to output all possible combinations of triples. By using this built in function, the only work that is needed on my end is to add a conditional statement that will take only vectors with weight 3. By using the *itertools* package, it is much efficient to use a library package that will generate all possible triple combinations than to hard code multiple for loops.

# 7 Conclusion

The Ball Collision Algorithm has shown that implementation over software requires some thought in to formulating a clever code. Because this paper was done using Python, some coding standards are not comparable to the Python's conventions. Unlike other software, Python requires us to manually code things to suit the needs of the BC algorithm. This can be seen when we define the columns of a generator as the number of rows of its transpose. In addition, we had to manually code a determinant modulo with a rounding operation to ensure that our calculations were correct. Along with manually coding basic operations, the issue of choosing how to operate with large data became significant once we started handling with creating combinations of binary tuples. Although we have shown that the Ball Collision Algorithm can be implemented via computer programming, Python may not be the optimal choice of language, simply because some of the basic standard operations required for Ball Collision decoding are not available. In addition, by not having these basic operations, implementing the Ball Collision Algorithm would require much more work aside from the required steps. With this caveat, Python programmers are required to put extra thought in how they implement this code.

# 8 Sources

Bernstein, Daniel J, et al. “Ball-Collision Decoding.” Ball-Collision Decoding, Cryptology EPrint Archive, 2010, cr.yp.to/codes/ballcoll-20101117.pdf.