Université de Limoges
Faculté des Sciences et Techniques

Année Universitaire 2019 - 2020
version du 26 mai 2020

Projet de Cryptographie Avancée

Auteurs :

NGUYEN Viet Sang
LAGORCE Jérémy

Encadrant :

GABORIT Philippe

Master 1 - Informatique

Contents

1 Introduction

In this report, we present the implementation of ISD and MDPC. For both algorithms, we choose C as programming language. For ISD, we can decode with small parameters. For MDPC, instead of using matrices, we implement all computations in polynomials. Our program works well with the given parameters. Besides, we also test the program with different T and w to explore the impact of this parameters.

This report is organized as follows.

- Section 2 presents the implementation of ISD. We firstly recall the principle of ISD and its complexity. Then, we show the method to calculate an inverse of a matrix by Gauss-pivot. After that, we explain the implementation of ISD and show the results of experiment.
- Section 3 presents the implementation of MDCP. We firstly present the principle of Bit Flipping Algorithm. We then prove that the product of a circulant matrix with a vector can be computed by polynomials. We also prove that we can find the inverse of a matrix by inverting the associated matrix. After that, we show our implementation of MDPC and the results of experiment.
- Section 4 discuss the our results. We also propose some solutions to improve our current implementation.
- Section 5 concludes our work.

2 Algorithm ISD (Information Set Decoding)

2.1 Priciple of ISD

Suppose we have a matrix generator G. We aim to decode a codeword $y = xG + e$, where e represents a vector (size n) of error with weight t. Recall that y is a vector of size n, x is a vector of size k and size of matrix G is $(k \times n)$. In ISD, we randomly choose k columns among n columns in G. We have a new matrix M $(k \times k)$ with these k columns.

If we obtain k columns without errors, we can compute the inverse M^{-1} of M. We also obtain y_k by getting k corresponding positions in y. We calculate $x_k = y_k M^{-1}$, and then $e_k = y - x_k G$. If the weight of e_k is equivalent to t, the decoding is successful.

The complexity of IDS is the complexity of finding k columns without errors. There are $n - t$ columns without errors in G and the number of choices k columns is $\binom{n-t}{k}$ among $\binom{n}{k}$ possible choices. Besides, the complexity of inversing a matrix $(k \times k)$ is $\mathcal{O}(k^3)$. Therefore, the total complexity of ISD is $\mathcal{O}(k^3)\frac{\binom{n}{k}}{\binom{n-t}{k}}$.

2.2 Inverse of matrix by Gauss-elimination

The steps of inverting a binary matrix is shown in algorithm 1. Suppose that M is the input matrix. At first, we expand M with an identity matrix which forms a new $n \times 2n$ matrix MI. We then find the pivot for each row. It means the diagonal line must be all of bits 1. If any row does has a pivot (pivot line is still equal to -1), the matrix M is not invertible. If a pivot is found for that row, we can use the operations (swapping, addition) in binary to reduce other positions of that line to 0. In the end, the left half of MI is the inverse of M.

Algorithm 1 Inverse of binary matrix

```
    Input : matrix $M$ with size $(n \times n)$
    Output : matrix $M\_INV$ if $M$ is invertible
 1: $MI = (M \ I)$, where I is an identity matrix with size $(n \times n)$
 2: for $i = 0$ to $n - 1$ do
 3:     $pivot\_line = -1$
 4:     for $j = i$ to $n - 1$ do
 5:         if $MI\_ji \neq 0$ then
 6:             $pivot\_line = j$
 7:             break
 8:     if $pivot\_line = -1$ then
 9:         return false
10:     if $pivot\_line \neq i$ then
11:         swap_rows($pivot\_line$, $i$)
12:     for $j = i$ to $n - 1$ do
13:         if $j \neq i$ and $MI_{ji} \neq 0$ then
14:             for $k = 0$ to $2n - 1$ do
15:                 $MI_{jk} = (MI_{jk} + MI_{ik}) \pmod 2$
16: $M\_INV \leftarrow$ right half of $MI$
17: return true
```

2.3 Implementation of ISD

We use 2D static array in C as the structure of data for computation with matrices. First, we implement some supportive functions of calculations in matrices such as addition, subtraction, multiplication, etc.

In the **Encoding** (listing 1), we randomly generate x of size k (line 4). Error e of size n is also generated randomly satisfied that it contains t bits 1 (line 5). Then, we generate the generator matrix G following the standard form $G = [I_k|P]$, where I_k is the $k \times k$ identity matrix and P is the $k \times (n - k)$ random matrix in binary. Finally, we compute $y = xG + e$ (lines 8-9).

```c
1 int x[1][k], e[1][n], y[1][n];
2 int matG[k][n];
3
4 generate_message(k, x);
5 generate_error(n, t, e);
6 generate_matrix_generator(k, n, matG);
7
8 mul_matrix(1, k, n, x, matG, y);
9 add_matrix(1, n, y, e, y);
```

Listing 1: Implementation of **Encode**

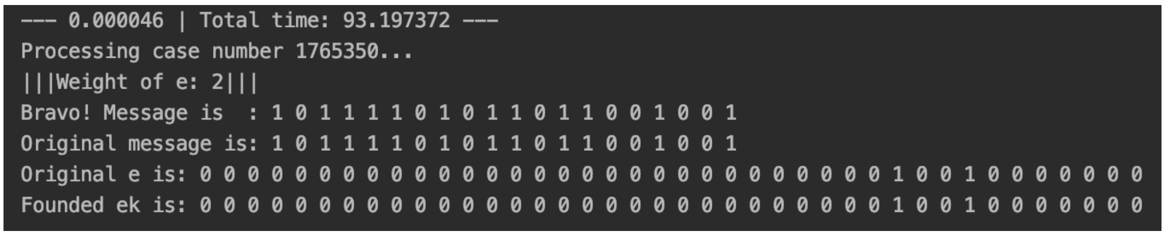
In the **Decoding** (listing 2), we randomly generate k numbers in range $[0, n-1]$ representing indexes of k chosen columns in G. We then generate matrix M with these k columns (line 8) and use Gauss-elimination to calculate the inverse of M (line 9). If M is invertible, we then get y_k from y which contains elements in k corresponding columns as M. We compute $x_k = y_k M^{-1}$ (line 12) and $e_k = y - x_k G$ (line 14). Finally, we check that if weight of e_k is t or not (line 16).

```c
int k_cols[k];
int matM[k][k], matINV[k][k];
int yk[1][k], xk[1][k], xkG[1][n], ek[1][n];
int w_e;    // weight of e
bool is_invertible;
while (1){
    random_k_cols(k, n, k_cols);
    get_submatrix(k, n, k_cols, matG, matM);
    is_invertible = inverse_matrix(k, matM, matINV); // inverse of M
    if (is_invertible){
        get_submatrix(k, n, k_cols, y, yk);          // yk
        mul_matrix(1, k, k, yk, matINV, xk);         // xk = yk.M^(-1)
        mul_matrix(1, k, n, xk, matG, xkG);          // xk.G
        sub_matrix(1, n, y, xkG, ek);                // ek = y - xk.G
        w_e = calculate_weight(n, ek);
        if (w_e == t){
            printf("Bravo! Message is  : ");
            display_matrix(1, k, xk);
            break;
        }
    } else
        printf("Not invertible\n");
}
```

Listing 2: Implementation of **Decode**

2.4 Results

At first, we test with small parameters such as $n = 40, k = 20, t = 2$ (figure 1). Our program usually needs less than 2 minutes to decode sucessfully. It try about 2 million iterations of choosing k columns before reaching the correct one.



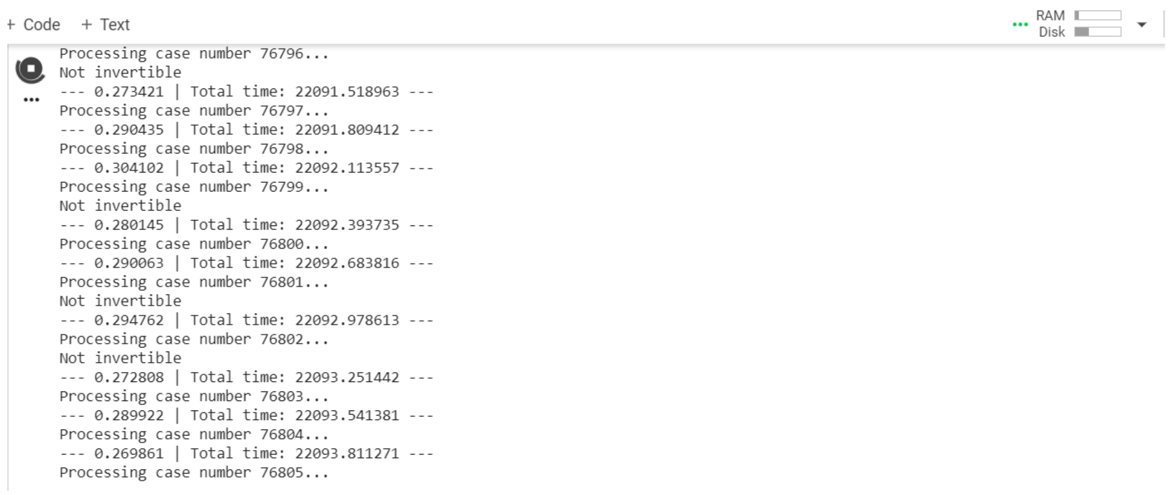
```
--- 0.000046 | Total time: 93.197372 ---
Processing case number 1765350...
|||Weight of e: 2|||
Bravo! Message is  : 1 0 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1
Original message is: 1 0 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1
Original e is: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0
Founded ek is: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0
```

Testing with $n = 40, k = 20, t = 2$

When testing with larger parameters $n = 400, k = 200, t = 20$, it takes about 0.02 seconds per iteration. We run in 8 hours and traverse about 1.5 million iterations. However, we still cannot decode the message.

In theory, when using parameters $n = 1000, k = 500, t = 10$, the probability of sucessful decoding is increase since t is decreased while size of matrix G is larger. Nevertheless, larger matrices lead to slower computation (0.28 seconds per iteration, figure 2). We run the program in 6 hours and still cannot decode. All tests are run on a device with 2.6 GHz of processor.

3 System of encryption MDPC



```
Processing case number 76796...
Not invertible
--- 0.273421 | Total time: 22091.518963 ---
Processing case number 76797...
--- 0.290435 | Total time: 22091.809412 ---
Processing case number 76798...
--- 0.304102 | Total time: 22092.113557 ---
Processing case number 76799...
Not invertible
--- 0.280145 | Total time: 22092.393735 ---
Processing case number 76800...
--- 0.290063 | Total time: 22092.683816 ---
Processing case number 76801...
Not invertible
--- 0.294762 | Total time: 22092.978613 ---
Processing case number 76802...
Not invertible
--- 0.272808 | Total time: 22093.251442 ---
Processing case number 76803...
--- 0.289922 | Total time: 22093.541381 ---
Processing case number 76804...
--- 0.269861 | Total time: 22093.811271 ---
Processing case number 76805...
```

Testing with $n = 1000, k = 500, t = 10$

3.1 Principle of Bit Flipping Algorithm

Algorithm 2 is the idea of Bit Flipping. At first, the syndrome of the noisy codeword is calculated, this is eH^T. The positions in the noisy codeword that participate in an "abnormally high number" of unsatisfied parity check equations (this value is called the decoding threshold) are considered to be likely in error, and therefore they are flipped. The syndrome is then recomputed. With high probability, this process decreases the total number of unsatisfied parity check equations. This process is repeated until the syndrome becomes zero (decoding success) or a maximum number of iterations is reached (decoding failure).

In BIKE-2, we apply this algorithm in **Decaps**. Parameter s is computed in step 1 of Decaps. Parameter H is the concatenation of h_0 and h_1, $H = (h_0^T\ h_1^T)$.

Algorithm 2 Bit Flipping

```
   Require : H $\in \mathbb{F}_2^{(n-k)\times n}$, s $\in \mathbb{F}_2^{(n-k)}$
   Ensure : eH$^T$ = s
1: e $\leftarrow$ 0
2: s' $\leftarrow$ s
3: while $s' \neq 0$ do
4:     $\tau$ = threshold $\in$ [0,1], found according to some predefined rule
5:     for $j = 0$ to $n-1$ do
6:         if |h$_j \star$ s'| $\geq \tau$|h$_j$| then
7:             e$_j \leftarrow$ e$_j$ + 1   mod 2
8:     s' $\leftarrow$ s - eH$^T$
9: return e
```

h$_j$ denotes the j-th column of H, as a row vector, '$\star$' denotes the componentwise product of vectors, and |h$_j \star$ s| is the number of unchecked parity equations involving j.

3.2 Product a circulant matrix with a vector by calculation on polynomials

There exists a natural ring isomorphism, which we denote φ, between the binary $r \times r$ circulant matrices and the quotient polynomial ring $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$. The circulant matrix A whose first row is $(a_0, a_1, ..., a_{r-1})$ is mapped to the polynomial $\varphi(A)$.

Suppose that we have vector $\mathbf{v} = (v_0, v_1, ..., v_{r-1})$. At first, we calculate the product of this vector with the circulant matrix of $\varphi(A) = a_0 + a_1X + ... + a_{r-1}X^{r-1}$.

$$\mathbf{b} = \mathbf{v}A = (v_0, v_1, ..., v_{r-1}) \begin{pmatrix} a_0 & a_1 & \dots & a_{r-1} \\ a_{r-1} & a_0 & \dots & a_{r-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \dots & a_0 \end{pmatrix} = (b_0, b_1, ..., b_{r-1})$$

In the vector of result,

$$\begin{aligned} b_0 &= v_0a_0 + v_1a_{r-1} + ... + v_{r-1}a_1 \\ b_1 &= v_0a_1 + v_1a_0 + ... + v_{r-1}a_2 \\ &... \\ b_{r-1} &= v_0a_{r-1} + v_1a_{r-2} + ... + v_{r-1}a_0 \end{aligned}$$

We set $\varphi(\mathbf{v}) = v_0 + v_1X + ... + v_{r-1}X^{r-1}$. We calculate $\varphi(\mathbf{v})\varphi(A)$ as follows.

$$\mathbf{b}' = \varphi(\mathbf{v})\varphi(A) = (v_0 + v_1X + ... + v_{r-1}X^{r-1})(a_0 + a_1X + ... + a_{r-1}X^{r-1})$$

Since all calculations is in the cyclic polynomial ring $\mathbb{F}_2[X]/(X^r - 1)$, we can form the result of $\varphi(\mathbf{v})\varphi(A)$ as a polynomial $\mathbf{b}' = \varphi(\mathbf{v})\varphi(A) \pmod{X^r - 1} = b'_0 + b'_1X + ... + b'_{r-1}X^{r-1}$.

In the polynomial $\mathbf{b}'$, we have the elements as follows.

$$\begin{aligned} b'_0 &= v_0a_0 + v_1a_{r-1}X^r + ... + v_{r-1}a_1X^r = v_0a_0 + v_1a_{r-1} + ... + v_{r-1}a_1 \\ b'_1X &= v_0a_1X + v_1a_0X + ... + v_{r-1}a_2X = (v_0a_1 + v_1a_0 + ... + v_{r-1}a_2)X \\ &... \\ b'_{r-1}X^{r-1} &= v_0a_{r-1}X^{r-1} + v_1a_{r-2}X^{r-1} + ... + v_{r-1}a_0X^{r-1} = (v_0a_{r-1} + v_1a_{r-2} + ... + v_{r-1}a_0)X^{r-1} \end{aligned}$$

As we can see, $b_0 = b'_0$, $b_1 = b'_1$, ..., $b_{r-1} = b'_{r-1}$. Therefore, we can conclude that $\varphi(\mathbf{v})\varphi(A) = \varphi(\mathbf{v}A)$

Advantages of using polynomial for encryption

Using polynomials for encryption saves so much space of memory. For example, if the size of vector $\mathbf{v}$ is 4800 and the size of matrix A is (4800×4800), we need $4800^2 + 2 \times 4800 = 23,049,600$ memory cells (one more time 4800 to store the result). Meanwhile, if we use the two polynomials to calculate the product, we only need $4800 \times 3 = 14,400$ memory cells which is ≈ 1600 times smaller.

3.3 System of encryption MDPC

MDPC corresponds to BIKE-2 [1] and Bit Flipping Algorithm in the decapsulation. There are three phases in this system including **KeyGen**, **Encaps**, **Decaps** which are described as follows.

KeyGen

- Input: λ, the target quantum security level.
- Output: the sparse private key (h_0, h_1) and the dense public key h.

0. Given λ, set the parameters r, w. However, in this project, λ is not given. Instead, r, w are given at the beginning. More specially, $r = 4813$ and $w = 39 * 2 = 78$.

1. Generate polynomials h_0, h_1 in the cyclic polynomial ring $\mathbb{F}_2[X]/(X^r - 1)$ satified that weights of the vectors $|h_0| = |h_1| = w/2$.
2. Compute $h \leftarrow h_1 h_0^{-1}$. It needs to multiply two polynomials and compute the inverse of a polynomial. For the inverse of a polynomial, there are two ways to compute. The first one is to use the Extended Euclidean Algorithm and the second one is to use the associated matrices of polynomials.

Encaps

- Input: the dense public key h.
- Output: the encapsulated key K and the cryptogram c.

1. Sample $(e_0, e_1) \in \mathcal{R}^2$ such that weight $|e_0|+|e_1| = t$. It means to generate two polynomials e_0 and e_1 in the cyclic polynomial ring $\mathbb{F}_2[X]/(X^r - 1)$ satified the condition of weight. In this project, number of errors is $t = 78$.
2. Compute $c \leftarrow e_0 + e_1 h$. It needs the operations of multiplication and addition between two polynomials in this step.
3. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$, where $\mathbf{K} : \{0,1\}^n \to \{0,1\}^{l_k}$ is the hash function used by encapsulation and decapsulation, and l_k is the desired symmetric key length (typically 256 bits). In this project, we use SHA256.

Decaps

- Input: the sparse private key (h_0, h_1) and the cryptogram c.
- Output: the decapsulated key K or a failure symbol $\perp$.

1. Compute the syndrome $s \leftarrow c h_0$. It only needs to multiply two polynomials in this step.
2. Try to decode s (noiseless) to recover an error vector (e'_0, e'_1). In this step, we use the Bit Flipping Algorithm which requrires two parameters. The first one is s which is computed in step 1. The second one is H which is block-circulant of index 2, $H = (h_0^T \ h_1^T)$ (size of $r \times 2r$).
3. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$. This is similar to step 3 in Encaps. It aims to verify the decoded result given by Bit Flipping Algorithm.

3.4 Calculate inverse of a polynomial by associated matrix

Suppose that we have two polynomials $\varphi(f)$, $\varphi(g)$ and their associated matrix are F, G. We have the following polynomials $\varphi(f) = f_0 + f_1 X + ... + f_{r-1} X^{r-1}$ and $\varphi(g) = g_0 + g_1 X + ... + g_{r-1} X^{r-1}$. We denote that $h(X) = f(X)g(X)$ and $h(X) = h_0 + h_1 X + ... + h_{r-1} X^{r-1}$.

$\varphi(g)$ is the inverse polynomial of $\varphi(f)$ iff $\varphi(X)\varphi(X) = 1 \pmod{X^r - 1}$. In other words, $\varphi(h) = 1$. It means

$$\begin{aligned} h_0 &= f_0 g_0 + f_1 g_{r-1} + ... + f_{r-1} g_1 = 1 \\ h_1 &= f_0 g_1 + f_1 g_0 + ... + f_{r-1} g_2 = 0 \\ &... \\ h_{r-1} &= f_0 g_{r-1} + f_1 g_{r-2} + ... + f_{r-1} g_0 = 0 \end{aligned}$$

Consider the associated matrices of $\varphi(f)$ and $\varphi(g)$,

$$FG = \begin{pmatrix} f_0 & f_1 & \dots & f_{r-1} \\ f_{r-1} & f_0 & \dots & f_{r-2} \\ \vdots & \vdots & \ddots & \vdots \\ f_1 & f_2 & \dots & f_0 \end{pmatrix} \begin{pmatrix} g_0 & g_1 & \dots & g_{r-1} \\ g_{r-1} & g_0 & \dots & g_{r-2} \\ \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & \dots & g_0 \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & \dots & h_{r-1} \\ h_{r-1} & h_0 & \dots & h_{r-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \dots & h_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

We can say that $FG = I$, where I is an identity matrix with size $(r \times r)$. So $G = F^{-1}$. Therefore, to find $g(X)$, we can calculate F^{-1} and the first row of the result represents coefficients of polynomial $g(X)$.

3.5 Implementation of MDPC

At first, we present our structure of data and some supportive functions. This includes the representation of a polynomial and operations on polynomials such as multiplication, addition, subtraction, etc.

Representation of a polynomial

We use a static array to represent a polynomial. For example, $r = 5$, vector $h_0 = (1, 1, 1, 0, 0)$ represents the polynomial $\varphi(h_0) = 1 + X + X^2$.

```
int h0[r];
int h1[r];
// Similar to e0, e1, h, c, etc.
```

Operations on polynomials

We implement basic operations with two polynomials including *addition*, *multipication*, *subtraction*, *wise-product*. We notice that in binary, the results of addition and subtraction are the same. At the beginning, we also implement some supportive functions to convert a polynomial to its circulant matrix and vice versa. However, we do not use these functions in the final solution because we utilise calculations with polynomials instead of matrices.

We use computations with polynomials in all operations in MDPC. As mentioned above, it requires $4813^2 = 23,164,969$ memory cells when we allocate memory for a square matrix $r \times r$ with $r = 4813$. This number is so enormous that our computer cannot allocate. Therefore, it is necessary to compute with polynomials instead of matrices.

From a polynomial, for example h, we can get a certain row hi as well as column hj in the associated matrix of h_0. Algorithm 3 shows the steps to get row hi. To get column, we use the same algorithm, but line 2 becomes $k = i - j$. In summary, it has rules in indexes which we can base on to get a certain row or column in circulant matrix from its original polynomial.

Algorithm 3 get_row_in_circulant_matrix

```
    Input : $h$, $r$, $i$
    Output : row i-th $hi$ in the cirulant matrix
1: for $j = 0, 1, ..., r-1$ do
2:     $k = j - i$
3:     if $k < 0$ then
4:         $k = r + k$
5:         $hi_j = h_k$
6: return $hi$
```

The most important function in which we replace computations in matrices by polynomials is to find the inverse of a polynomial. In this function, we apply Extended Euclidean Algorithm to to calculate the inverse of a given polynomial in the cyclic polynomial ring $\mathbb{F}_2[X]/(X^r - 1)$. Polynomial v is an inverse of polynomial b iff $au + bv = 1$, where $a = X^r - 1$. For example, $r = 5$, we find the inverse v of $b = 1 + X + X^2$. When applying Extended Euclidean Algorithm, if there exists the remider 1, we have that v is the desired inverse. Otherwise, if the reminder is 0, polynomial b is not invertible.

	u	v	q
$a = X^r - 1$	1	0	
$b = X^2 + X + 1$	0	1	$X^3 + X^2 + 1$
X	1	$X^3 + X^2 + 1$	X
$X + 1$	X	$X^4 + X^3 + X + 1$	1
1	$X + 1$	$X^4 + X^2 + X$	

Main program of MDPC

In the **GenKey** (listing 3), we generate h_0 and h_1 satified that their weights are both $w/2$. In BIKE-2, if parameter w is large enough, the probability of having inverse of h_0 is near 1. However, in practice, we carefully check whether h_0 has inverse or not (lines 5-9). We then multiply h_1 and h_0^{-1} to compute h (lines 11-12).

```
1  int h0[r], h1[r];
2  int h0_inv[r]; // inverse of h0
3  generate_message_with_weight(r, w/2, h1);
4  
5  bool is_invertible = false;
6  while (!is_invertible){
7      generate_message_with_weight(r, w/2, h0);
8      is_invertible = inverse_poly(r, h0, h0_inv);
9  }
10 
11 int h[r]; // h = h1.h0^(-1)
12 mul_poly(r, h1, h0_inv, h); // h should be dense of bits 1
```

Listing 3: Implementation of **GenKey**

In **Encaps** (listing 4), we fisrtly generate e with length of $2r$ and t bits 1 (line 3). We then split e into $e0$ and $e1$ (lines 5-8). After that, we compute $c = e_0 + e_1 h$ by multiplication and addition with polynomials (line 10-12). Finally, we compute the hash of (e_0, e_1) (lines 15-18).

```c
int e[2*r], e0[r], e1[r];

generate_message_with_weight(2*r, t, e);
// Split e (2*r) to e0 (r) and e1 (r) which ensures |e0| + |e1| = t
for (int i=0; i<r; i++){
    e0[i] = e[i];
    e1[i] = e[i + r];
}

int c[r]; // c = e0 + e1.h
mul_poly(r, e1, h, c);
add_poly(r, e0, c, c);

// K = hash(e0, e1) = hash_decaps
unsigned char hash_encaps[SHA256_DIGEST_LENGTH];
char e0e1[2*r];
convert_int_to_char(2*r, e, e0e1);
SHA256(e0e1, 2*r, hash_encaps);
```

Listing 4: Implementation of **Encaps**

In **Decaps** (listing 5), we firstly compute $s = ch_0$ (lines 1-2). Secondly, we apply Bit Flipping Algorithm to recover e' (denoted by `e_bf` in code). Finally, we compute the hash of `e_bf` and compare with the hash in **Encaps**. The most important part in **Decaps** is the implementation of Bit Flipping Algorithm which is discussed soon after.

```c
 1  int s[r]; // s = c.h0
 2  mul_poly(r, c, h0, s);
 3
 4  int e_bf[2*r]; // result e of Bit Flipping algo
 5  bit_flipping(r, s, h0, h1, e_bf, T, LIMIT);
 6
 7  // K = hash(e0', e1') = hash_decaps
 8  unsigned char hash_decaps[SHA256_DIGEST_LENGTH];
 9  char e_bf_chars[2*r];
10  convert_int_to_char(2*r, e_bf, e_bf_chars);
11  SHA256(e_bf_chars, 2*r, hash_decaps);
```

Listing 5: Implementation of **Decaps**

Listing 6 shows the implementation of Bit Flipping Algorithm. We follow the steps of algorithm as follows. This takes s and H as parameters. s is computed in step 1 of **Decaps**. We initialize $e \leftarrow 0$ (lines 2-3) and $s' \leftarrow s$ (`s1` on lines 5-7). Instead of using matrix H directly, we pass h_0 and h_1 as parameters and use them to retrieve desired rows or columns in H. We notice that H is the horizontal concatenation of circulant matrices h_0^T and h_1^T. It means H^T is the vertical concatenation of circulant matrices h_0 and h_1. Therefore, we can retrieve column j-th in H by retrieving row j-th in H^T (lines 16-20). Having hj and s', we calculate the component-wise product $|hj \star s'|$ (line 22), then compare its weight to the threshold T and update e (lines 24-26). After that, we compute $s' \leftarrow s + eH^T$ (lines 29-30). We also use h_0 and h_1 to multiply with e instead of matrix H^T.

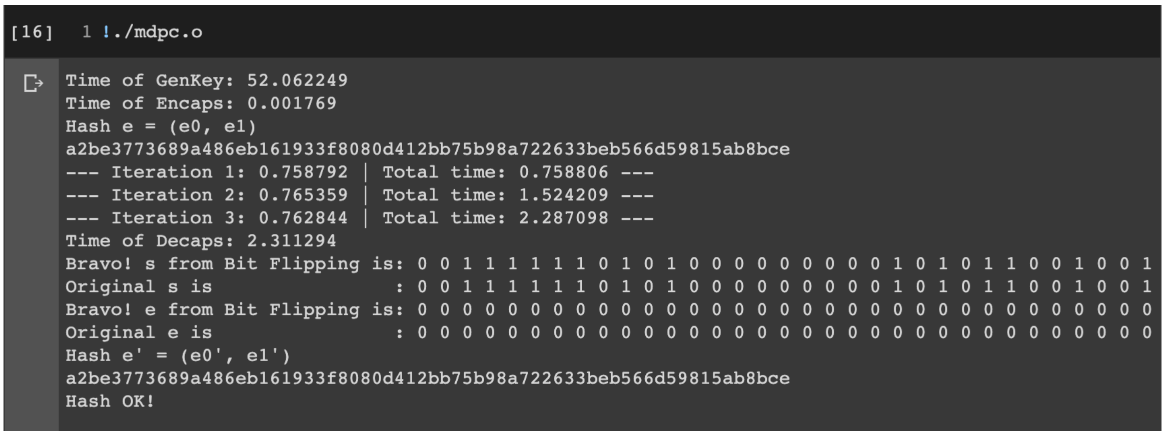
```c
void bit_flipping(int r, int s[r], int h0[r], int h1[r], int e[2*r],
 ↪int T, int LIMIT){
    for (int i=0; i<2*r; i++)
        e[i] = 0;       // e <- 0

    int s1[r];          // s' in algo, s' <- s
    for (int i=0; i<r; i++)
        s1[i] = s[i];

    int hjs1[r];        // hj * s'
    int hj[r];
    int weight_hjs1;    // |hj * s'|
    int eHT[r];         // e.H^T

    while (calculate_weight(r, s1) != 0){
        for (int j=0; j<2*r; j++){
           // column hj in H is row hj in HT
            if (j < r)
                get_row_in_circulant_matrix(r, j, h0, hj);
            else
                get_row_in_circulant_matrix(r, j-r, h1, hj);

            product_wise(r, hj, s1, hjs1); // hjs1 = hj * s1

            weight_hjs1 = calculate_weight(r, hjs1);
            if (weight_hjs1 >= T)
                e[j] = (e[j] + 1) % 2;
        }

        mul_e_HT(r, e, h0, h1, eHT);     // e.H^T
        sub_poly(r, s, eHT, s1);         // s1 = s - e.H^T

        if (count == LIMIT)
            break;
    }
}
```

Listing 6: Implementation of **Bit Flipping Algorithm**

3.6 Results

We test with parameters $r = 4813, w = 39 \times 2, t = 78, T = 26$ (notations as presented in BIKE-2) and the result is shown in figure 3. It takes 45.5 seconds for **GenKey**, 0.0015 seconds for **Encaps** and 2.22 seconds for **Decaps**. As we can see, **Encaps** costs the most time when comparing to the two other phases mainly because of the step of inverting matrix. For these parameters, it always needs 3 iterations to successfully decapsulate.

We also explore the impact of the threshold T by testing this parameter with different values. As we can see in table 1, time of **GenKey** and **Encaps** are nearly the same with different T. This is reasonable because T does not affect in this two phases. However, there are significant differences in **Decaps** and number of iterations in Bit Flipping Algorithm. If we gradually increase T, the number of iterations are also increase. And the turning point may be $T = 29$, the number of iterations dramatically increases and we stop the program at 1500 iterations. It



```
[16]   1 !./mdpc.o

Time of GenKey: 52.062249
Time of Encaps: 0.001769
Hash e = (e0, e1)
a2be3773689a486eb161933f8080d412bb75b98a722633beb566d59815ab8bce
--- Iteration 1: 0.758792 | Total time: 0.758806 ---
--- Iteration 2: 0.765359 | Total time: 1.524209 ---
--- Iteration 3: 0.762844 | Total time: 2.287098 ---
Time of Decaps: 2.311294
Bravo! s from Bit Flipping is: 0 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1
Original s is                : 0 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1
Bravo! e from Bit Flipping is: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Original e is                : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Hash e' = (e0', e1')
a2be3773689a486eb161933f8080d412bb75b98a722633beb566d59815ab8bce
Hash OK!
```

Testing with $n = 4813, w = 39, t = 78, T = 26$

costs about 20 minutes.

Threshold T	22	23	24	25	26	27	28	29
GenKey Time (s)	52.60	51.02	53.18	52.88	52.06	53.63	52.98	54.28
Encaps Time (s)	0.0018	0.0017	0.0018	0.0019	0.0017	0.0018	0.0019	0.0014
Decaps Time (s)	2.30	1.53	1.53	2.30	2.31	3.06	5.36	> 1146
Iterations of Bit Flipping	3	2	2	3	3	4	7	> 1500

Table 1: Testing with $r = 4813, w = 39 \times 2, t = 78$ and changing T

Parameter w is also have a great impact to the algorithm. Table 2 shows the results of testing with different w. As we can see, $w = 41 \times 2$ give nearly the same result with $w = 39 \times 2$. However, $w = 45 \times 2$ and $w = 49 \times 2$ cause the huge number of iterations in Bit Flipping algorithm.

Weight w	39×2	41×2	45×2	49×2
GenKey Time (s)	52.06	50.42	51.56	54.64
Encaps Time (s)	0.0017	0.0020	0.0017	0.0018
Iterations of Bit Flipping	3	3	> 1500	> 1500

Table 2: Testing with $r = 4813, t = 78, T = 26$ and changing w

4 Discussion and Future Work

Time of running also depends on the language which we choose to implement. At the beginning, we practice ISD with Python. However, it costs about 1.3 seconds per iteration. Meanwhile, implementation of C costs only 0.25 seconds. This is because Python is an interpreted language, so the Python code will be interpreted to the language of lower level before executed. Instead, C directly uses system calls.

In the structure of data, it is more ideal to use a smaller type of data such as `boolean` since all computations work in binary. A variable of `boolean` takes 1 byte of space, while a variable of `int` takes 2 bytes (32-bit compiler) or 4 bytes (64-bit compiler).

It is reasonable for the result of ISD that we cannot decode in a considered amount of time. As the testing with small parameters $n = 40, k = 20, t = 2$, it needs about 2 millions of iterations to successfully decode. Therefore, when we try with $n = 400, k = 200, t = 20$, the number of iterations will be enormous while the speed of computation is slower because of larger matrices. Besides, the given complexity of ISD bases on the average number of tries needed to find an invertible M. However, the success of decoding depends on not only M but also the weight of e_k.

5 Conclusion

In this project, we implement the ISD and MDPC in C. For the ISD, our program works well with small parameters. We also test with bigger parameters, however, the program cannot decode after many hours of running. For the MDPC, our program successfully works with the suggested parameters. We also do experiment to see the impact of T and w in this algorithm. Moreover, we give some discussion for our obtained results as well as future work.

6 References

[1] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, et al. “BIKE: Bit Flipping Key Encapsulation”. In: *NIST* (2017).