Université de Limoges

Faculté des Sciences et Techniques

Mise en place d'une attaque exploitant les codes correcteurs (MDPC)

UE : Cryptographie Avancée

Rapport de Projet

Des étudiants :

AKAKPO Segno Prince
LAWSON ANANISSO Morel

Responsable :
M. Philippe GABORIT

Master 1 CRYPTIS INFO

France , Mai 2021

Table des matières

Table des figures

Chapitre 1

Introduction

Le projet se compose de 2 parties . Dans un premier temps , il est demander l'implémentation de l'algorithme ISD(pour Information Set Decoding) et dans une seconde partie , il sera réaliser , le système de chiffrement MDPC.Il s'agit donc dans ce projet d'implémnter et de mettre en oeuvre une attaque par décodage d'ensemble d'infromations dans la première partie du Projet . Le présent rapport présente donc les différentes implémentations (codes sources et leurs explications) pour répondre aux exigences du projet.

Algorithme de ISD

Définitions de terme

- **n et k** : nombre entier pris comme paramètre de l'alogorithme
- **H** : matrice aléatoire de dimension (n - k , n)
- **H'** ou A : sous matrice de H
- **w** ou poids : nombre de 1 dans la matrice
- **X** : matrice aléatoire de petit poids
- **X'** : sous matrice de X

1.1 Principe de l'algorithme ISD

Le principe vu en cours se base sur la recherche d'une matrice x qui est le produit de 2 matrices, la transposé de H d'une part et de S de l'autre part. En d'autre terme l'algorithme repose sur le problème de la capacité à trouver X tel que $H * x^t = s?$, connaissant le s. Le problème en apparence facile devient beaucoup plus difficile quand on fixe le X avec un certain poids prédéfinis, généralement x est pris de manière à avoir x = n/9 .

H' étant une matrice carrée , nous avons la certitude qu'il existe une unique solution à cette équation et c'est bien le x qui nous sera retourner.

Alors la probabilité de trouver le x équivalent devient : $Binom(n - w, n - k - w)/Binom(n, n - k)$

1.1.1 Procédé dévéloppé pour trouver X

Dans le but de trouver le X , nous faisant une suite d'opération qui à la fin , nous retournera la valeur voulue.Afin de trouver X , on deduit une matrice H' extraite de H telle que H' soit de taille (n-k , n-k). Nous devrions pour la suite du code nous assurer de l'inversibilité de H'(fonction que nous verrons dans la partie consacrée à l'inversion). X , nous allons deduire une sous matrice X', telle que w(x) = w(x'). en d'autre terme X' est telle qu'il faudra y ajouter des 0 afin d'obtenir le X que nous cherchons. Nous calculerons ensuite S en faisant $H' * X$ Ensuite , il sera question de calculer un certain X' , tel que X' = $H^-1 * S$ Afin de nous assurer de la bonne valeur de X' , nous calculerons son poids et verifirons si c'est le poids désiré , dans ce cas , on serait arrivé à notre but. Dans le cas contraire , on reprend la procédure depuis la génération du H'.

L'attaque peut se produire avec une compléxité en : **O((n-k)**$^3).binom(n, n-k)/binom(n-w, k)avec(westassezpet$

1.2 Inversion de Gauss

```c
int **inversionDouble(int **matrice , int dimension)
{
    float ** augmentedmatrix;
    augmentedmatrix = allocationf(augmentedmatrix,dimension,2*dimension );
    float temporary, r ;
    int i, j, k, temp;
    for(i=0;i<dimension; i++)
        for(j=0; j< dimension; j++)
                augmentedmatrix[i][j]=matrice[i][j];
    for(i=0;i<dimension; i++)
        for(j=dimension; j<2*dimension; j++)
            if(i==j%dimension)
                augmentedmatrix[i][j]=1;
            else
                augmentedmatrix[i][j]=0;

 /* using gauss-jordan elimination */
    for(j=0; j<dimension; j++)
    {
      temp=j;
     /* finding maximum jth column element in last (dimension-j) rows */
      for(i=j+1; i<dimension; i++)
        if(augmentedmatrix[i][j]>augmentedmatrix[temp][j])
                            temp=i;
        if(fabs(augmentedmatrix[temp][j])<minvalue)
             {
                printf("\n Elements are too small to deal with !!!");
                int **matriceResult;
                matriceResult = allocation(matriceResult,dimension,dimension);
                matriceResult = NULL;
                return matriceResult;
                //exit(0);
                //goto end;
             }
     /* swapping row which has maximum jth column element */
      if(temp!=j)
                for(k=0; k<2*dimension; k++)
                {
                temporary=augmentedmatrix[j][k] ;
                augmentedmatrix[j][k]=augmentedmatrix[temp][k] ;
                augmentedmatrix[temp][k]=temporary ;
                }
    /* performing row operations to form required identity matrix out of the input
   matrix */
    //TRY{
      for(i=0; i<dimension; i++)
            if(i!=j)
            {
                r=augmentedmatrix[i][j];
                for(k=0; k<2*dimension; k++)
                {
                    if (augmentedmatrix[j][j] != 0)
                    {
                        augmentedmatrix[i][k]-=(augmentedmatrix[j][k]/
   augmentedmatrix[j][j])*r ;
                    }
```

```
                }
            }
            else
            {
                r=augmentedmatrix[i][j];
                for(k=0; k<2*dimension; k++)
                {
                    if (r !=0)
                    {
                      augmentedmatrix[i][k] = abs((int)(augmentedmatrix[i][k]/r)%2);
                    }

                }
            }
    }
 /* Display augmented matrix */
    printf("\n La matrice augment e obtenu apr s la methode de gauss-jordan : \n\n") ;

 /* displaying inverse of the non-singular matrix */

    printf("\n\n\n La matrice inverse est : \n\n");
    int **matriceResult;
    matriceResult = allocation(matriceResult,dimension,dimension);
    for(i=0; i<dimension; i++)
    {
        for(j=dimension; j<2*dimension; j++)
        {
                       matriceResult[i][j%dimension] = abs( (int)augmentedmatrix[i][j]%2);
        }
        printf("\n");
    }

    return matriceResult;
}
```

Listing 1.1 – fonction inversionDouble

1.3 Partie 3 : programmer l'algorithme ISD

```
int **multiplication(int **matrice1 ,int  **matrice2 , int **matrice3 , int n , int m , int p)
 {

  int i , j , k ;
   // Affectation du r sultat de la multiplication    C
 for (i=0; i< n; i++)
     for (j=0; j<p; j++)
         {
          matrice3[i][j]=0;
          for (k=0; k< m; k++)
          {
            //result = allocation(result , n-k , 1);
            matrice3[i][j] += matrice1[i][k] * matrice2[k][j];
          }
```

```
            matrice3[i][j] = abs(matrice3[i][j]%2);
          }
 return matrice3;

}
```

Listing 1.2 – fonction multiplication

```
int poidHamming(int **matrice,int n ,int m)
{
    int poidHamming = 0 , i , j;

    for (i=0; i<n; i++)
        {
         for (j=0; j< m; j++)
              if (matrice[i][j] == 1)
              {
                poidHamming+=1;
              }

        }
        return poidHamming;
}
```

Listing 1.3 – fonction poidHamming

Cette fonction nous permet de calculer le poids de Hamming pour une matrice en entrée

```
int **matrice_X ;
    matrice_X = allocation(matrice_X , 1 , n);
    // total1 sert   connaitre le nombre de 1 dans x
    int total1 = 0;
    // generation al atoire de X avec un nombre de 1  gale  au poids cherch
    for(int r=0;r<1;r++){
        for(int c=0;c< n ;c++){
            matrice_X[r][c]=rand() & 1;

            if (matrice_X[r][c] == 1 )
                total1 += 1;
        }
    }
```

Listing 1.4 – Generation de X

Nous procédons de cette sorte pour pouvoir génerer les differentes valeurs que nous prenons de manière aléatoire.
NB : Pour les autres valeurs que nous generons , le code est pratiquement le même , donc dans un souci de resumer le rapport , nous ne montrerons pas toutes les générations (que vous pouvez bien sûr trouver dans le code source associé) .

Nous allons découper la partie où se deroule la partie principale de l'algorithme afin de mieux l'expliquer.

Ainsi en premier nous avons :

```
do{
        int ** temp_A;
        temp_A = allocation(temp_A,n-k,n-k);
        int rand_col=0;
        for(int p=0;p<n-k;p++)
        {
            rand_col = (rand() % (n ));
```

```
            temp_A[p]=temp_H[rand_col];

        }
        printf("\n");
        printf("Affichage du H (colonnes , lignes) \n");
        printf("\n");

        int ** extrait_H;
        extrait_H = allocation(extrait_H,n-k,n-k);
        // matrice transpos
        for(int i=0;i<(n-k);i++){
            for(int j=0;j<(n-k);j++){
                 extrait_H[j][i]=temp_A[i][j];
            }
        }
        printf("\n");
        printf("Affichage du H (lignes, colonnes) \n");
```

Listing 1.5 – fonction chiffrement

Dans cette partie , il s'agira de decouper d'une manière aléatoire la matrice A(encore nommée H') de H.

```
int ** inter ;
        inter = allocation(inter , n-k , 1 );
        int ** S ;
        inter = allocation(S , n-k , 1 );
         S = multiplication(extrait_H , transpose_matrice_X, inter , n-k , n-k , 1) ;

        //afficherMatriceXY(S,n-k,1);
        printf("\n");
        printf("Affichage de S \n");
        printf("\n");
/////////////////////////////////////////////////// Fin Calcul de S ///////////////////////////////////////////////////////////

/////////////////////////////////////////////////// Debut Calcul de H'^-1 ///////////////////////////////////////////////////////////

        int ** invers_H ;
        invers_H = allocation(invers_H,n-k,n-k);
        //invers_H doit doubler de taille normalement : n-k et 2(n-k)
        invers_H = inversionDouble(extrait_H,n-k);

        if (invers_H == NULL)
        {
            continue ;

        }
```

Listing 1.6 – fonction chiffrement

Dans cette partie , nous calculons la matrice S avec les paramètres que nous avons et nous calculons l'inverse de H'. Si l'inverse de H' n'existe pas , nous retournons dans la boucle while , ce qui veut dire refaire tout le calcul.

```
else {

 /////////////////////////////////////////////////// Fin Calcul de H'^-1 ///////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////// Debut Calcul de X'
  //////////////////////////////////////////////////////////////

                //afficherMatrice(invers_H,n-k);
                printf("Affichage de l'inversion \n");
                int matSize = n-k;
                int ** result;
                result = allocation(result , n-k , 1);
                // multiplication entre H' et S
                int ** extrait_X = multiplication(invers_H , S ,result,matSize,
  matSize,1);

                printf("\n");
                printf("\n");

                //afficherMatriceXY(extrait_X,matSize,1);

                printf("Affichage de x' \n");

/////////////////////////////////////////////////// Fin Calcul de X'
  //////////////////////////////////////////////////////////////

/////////////////////////////////////////////////// Debut Calcul du poid de X et
  comparaison ////////////////////////////////////////////////////////////////

                int w_x_derive = poidHamming(extrait_X,n-k,1);
```

Listing 1.7 – fonction chiffrement

Dans le cas où le H , trouver est inversible , nous continuons l'opération en recherchant cette fois ci le poids de X' , après l'avoir calculé. Dans le cas où le poids de X' ne correspond pas à ce qu'on désire , on retourne dans la boucle while et on re-execute tout depuis le debut.

```
if (w_x_derive == w_x)
                {
                    printf("bingo\n");
                    break;
                }
            }

    }
    while(1);
```

Listing 1.8 – fonction chiffrement

Dans le cas où le poids correspond , le programme s'arrête , car on aura retrouvé le X'.

Resultat lors de l'exécution du l'algorithme Notre exécution de l'algorithme n'a pu donner de resultat concluant (pas avant que nous soyons contraint d'y mettre terme) , nous n'avions pas pu le tourné aussi longtemps que possible. Nous l'avons arrêté , au bout de quelques heures , et donc nous somme dans le regret de ne pouvoir y ajoindre le temps d'exécution , dans les cas réel.

Principe de l'algorithme BitFlip

En premier , nous allons procédé à la définitions des termes.

1.3.1 Définition de Terme :

- **H : matrice creuse , matrice contenant beaucoup de 0 et peu de 1**
- **y : le mot recu (le mot sur lequel il faudra retourner les erreurs)**
- **Syndrome : c'est la multiplication entre la matrice H et la transposé de y (Hy^t)**
- **parity chechk : Comparaison entre les colonnes de H et y , cette operation renvoie un nombre qui est synomyme du nombre de 1 placé à au même endroit pour H et le syndrome.**
- **T : (le seuil) nombre entier naturel**
- **Alice et Bob : Personnage que nous allons utiliser pour illustrer les algorithmes que nous avons.**
- **Hash : fonction de hachage choisit par Alice et Bob**

Dans le cours nous sommes parties d'une matrice creuse H et d'une matrice y , l'algorithme de BitFlip retourne :

- **En premier l'algorithme de bitFlipping se charge de calculer le syndrome.**
- **Ensuite on procède à la parity check , sur toute les colonnes de H. Après cette operation , on a deux possibilité selon le cas :**
  - **On prends le nombre de parity check le plus élevé et on relève les colonnes où elles ont été trouvées ou soit**
  - **Au debut on se définit un seuil (T) , alors en calculant la parity check , on relève les colonnes dont le nombre de parity check est supérieur ou égale à ce seuil.**

  Après avoir fait cette opération , on change les bits de y qui correspondent aux colonnes de H.
  Ces étapes sont répétées jusqu'à ce que le syndrome soit égale à 0 , et alors les erreurs trouvées sont les bons , et on les retourne

1.4 Chiffrement MDPC

Dans le cas du chiffrement MDPC , nous utiliserons le deuxième cas 2 , c'est à dire que nous utiliserons un seuil T pour retourner les colonnes de H, qui ont le plus grand nombre de parity check.

Le Code Du MDPC vu en TP se décline sous 3 options à savoir :

- **Le partage de clé .**
  Le partage de clé se base sur l'algoritme de bitFlipping Dans le cas où Alice et Bob , veuillent échangé , sans pour autant se connaitre .
  Le procédé mise en oeuvre dans cette partie est :

— **Alice** Alice choisit aléatoirement un h0 et h1 aléatoire de taille (1,n) : Elle calcul l'inverse de $h0^{-1}$ puis multiplie les deux polynômes entre eux. Après cette operation Alice obtient le h qu'elle envoit à Bob.
NB : Au cas où h0 ne serait pas inversible , il faudra générer un nouveau h0 jusqu'a ce qu'on obtienne une inverse.

— **Bob** Bob choisit aussi de son côté 2 polynômes e0 et e1 . Il multiplie e1 par le polynôme h qu'il a recu de Alice. Il additionne le resultat avec son e0 pour trouver un C1.
A son tour , Bob renvoie le c1 à Alice .

— **Alice**
Après avoir recu le c1 envoyé par Bob , Alice le multiplie par son h0 pour obtenir une sortie s. Grâce à l'algorithme de bitFlipping et en passant h0 , h1 ,s , t et w en paramètre ,Alice peut recuperer les polynômes e0 et e1 que seul Bob detient .
A cette étape , Alice et Bob ont reussit à s'échanger un secret sans pour autant qu'un Attaquant puisse intercepter le secret partagé.

— **Le chiffrement**
Le secret que Alice et Bob ont reussit à partager est e0 et e1 , les deux décides d'un commun accord d'utilisé une fonction de hachage particulière . Alors chacun de son coté , peut procéder au hachage de (e0,e1).
La clé public de Alice est : $h = h1 * h0^{-1}$ Bob peut uiliser cette clé pour envoyé des message à Alice : Pour envoyer un message , Bob fait une addition modulo 2 entre (son message et la clé public de Alice). En d'autre terme , Bob fait un XOR de son message avec la clé public d'Alice. La clé privée est alors Hash(e0,e1).

— **Le dechiffrement** Cette partie est assez similaire à la première dans ce sens qu'il faudra toujours un XOR avec le messsage chiffré pour obtenir le message en claire.

1.5 Partie programmation du système MDPC

Dans cette partie nous presenterons les differentes fonctions implémenté en C pour pouvoir effechtuer chaque partie du chiffrement MDPC.

Nous présenterons en premier les programmes générales à toutes les parties puis , nous rentrerons sur les fonctions qui concernent chaque partie précisement.

Pour une faciliter de gestion de notre code et une uniformité, nous avons utilisé des tableaux à 2 dimmensions tout au long.

— **La fonction rot** : Elle est celle qui est utilisé , pour pouvoir mettre les polynômes sous forme de matrice cyclique. Dans ce rapport , nous ne presenterons que les fonctions qui ont été spécifié dans l'algoritme , pour les autres fonctions nécessaire pour le bon fonctionnement du programme , veuillez vous referez au fichier c qui accompagnera le rapport.
Elle a été programmer comme suit :

```c
int **rots(int **polynome,int **sortie)
{
  int k = 0;
  for (int i = 0 ; i < n ; i ++ )
  {
    k = k + 1 ;
    for(int j = 0 ; j < 2*n ; j ++ )
    {
      sortie[i][j] = polynome[0][k%n];
```

```
      k = k + 1 ;
    }
  }
  return sortie;
}
```

Listing 1.9 – fonction rots

— **La fonction transposer : La fonction transposer est celle utilisé pour pouvoir transposer une matrice. C'est à dire inverser les lignes et les colonnes.**

```
int **transposer(int **matrice , int **matrice_T , int ligne , int colonne)
{
  for(int i=0;i< ligne ;i++){
        for(int j=0;j< colonne ;j++){
              matrice_T[colonne][ligne]= matrice[ligne][colonne];
        }
    }

    return matrice_T ;
}
```

Listing 1.10 – fonction rot

La fonction bitFlipping

Dans la suite de notre code , nous avons pris l'opération (a,b) comme étant la concaténation des deux matrices a et b.

```
int **bitFlipping(int **ho,int **h1,int **s,int T,int t)
{
  int **u ;
  int **v ;
  int **u_v;
  // variable u et v conformement a celui d crit dans l'algo
  u = allocation(u,1,n);
  v = allocation(v,1,n);
  // variable qui contient la concatenation de u et v
  u_v = allocation(u_v,1 , 2*n);
  // initialisation de u et v
  for (int i = 0 ; i < n ; i ++)
  {
    u[0][i] = 0;
    v[0][i] = 0;
  }
  // partie de la concact nation (u,v)
  for (int i = 0; i < 1 ; i++){
    for(int j= 0 ; j < n ; j++)
      {
        u_v[i][j] = u[0][j];
      }
      for(int j= n ; j < 2*n ; j++)
      {
        u_v[i][j] = v[0][j];
      }
    }

  //declaration de la matrice H comme dans l'algo
  int **H ;
  // allocation de H
  H = allocation(H,n,2*n);
  // variable dont nous allons nous servir pour avoir -ho
```

```
  int ** negative_ho;
  negative_ho = allocation(negative_ho , 1 , n );

  // calcul de -ho
  for(int i = 0 ; i < n ; i ++ )
    negative_ho[0][i] = -1 * ho[0][i];

  //Partage du calcul de h en composante pour pouvoir l'impl menter.
  // Calcul de composante 1 : on fait rot (- ho)
  int **composante_H_1 = rot(negative_ho);
  // Calcul de composante 1  On fait aussi rot(h1) affin de pouvoir multiplier les 2 matrices
  int **composante_H_2 = rot(h1);

  int **T_composante_H_1 ;
  int  **T_composante_H_2;
  T_composante_H_1 = allocation(T_composante_H_1, n ,n );
  T_composante_H_2 = allocation(T_composante_H_2, n ,n );

  // transposer des 2 composantes conformement    l'algo
  T_composante_H_1 = transposer(composante_H_1,T_composante_H_1 , n , n);
  T_composante_H_2 = transposer(composante_H_2, T_composante_H_2 , n , n);

// concat nation de H (rot(  h   0 ) , rot(h 1 )^T)
  for (int i = 0; i < n ; i++)
  {
    for(int j= 0 ; j < n ; j++)
    {
      H[i][j] = T_composante_H_1[j][0];
    }
    for(int j= n ; j < 2*n ; j++)
    {
      H[i][j] = T_composante_H_1[j%n][0];
    }
  }

// calcul du syndrome
  int **syndrome ;
  syndrome = allocation(syndrome, 1 ,  n );

// allocation du syndrome avec S
  syndrome = s;
  int **sum;
  sum = allocation(sum,1,2*n);

// declaration de flipped_positions

  int **flipped_positions ;
  flipped_positions = allocation(flipped_positions , 1 , 2*n);
  int ** result ;
  result = allocation(result,1,2*n);

  while((poidHamming(u, 1 , n ) != t || poidHamming(v , 1, n) != t) & poidHamming(syndrome,1,n) != 0 )
  {

    sum = multiplication(syndrome,H,result,1,n,2*n);
    for (int i = 0 ; i < n ; i ++)
      flipped_positions[0][i] = 0;
```

```

    for(int i = 0; i < 2*n ; i++)
    {
      if (sum[0][i]>= T )
        {
          // XOR de flipped_positions avec 1
          flipped_positions[0][i] = (flipped_positions[0][i] + 1)%2;

        }
    }
    // XOR de u_v et de flipped_positions
    for (int j = 0 ; j < 2*n ; j ++)
      u_v[0][j] = (u_v[0][j] + flipped_positions[0][j])%2;

    int **T_flipped_positions;

    T_flipped_positions = allocation(T_flipped_positions , 2*n , 1);
    T_flipped_positions = transposer(flipped_positions, T_flipped_positions , 1 , 2*n );
    int **resu ;
    resu = allocation(resu, n , 1);

    // multiplication de H et la Transposer de flipped_positions
    int **Hflipped_positions = multiplication(H , T_flipped_positions , resu , n , 2*n , 1 );

    // Soustraction du syndrome et de Hflipped_positions
    for (int j = 0 ; j < 2*n ; j ++)
      syndrome[0][j] = syndrome[0][j] - Hflipped_positions[j][0];

  }
  //  Les parties qui viennent on   t   decoup  en plusieurs parties pour donner le resultat voulu par l'algorithme.
  int ** T_u_v;

  T_u_v = allocation(T_u_v , 2*n , 1);

  T_u_v = transposer(u_v , T_u_v , 1 , 2*n);

  int ** multiply ;

  multiply = allocation(multiply, n , 1);

  int **res;
  int **resume;
  resume = allocation(resume,1 , n);
  res = allocation(res , n , 1);

// multiplication entre H et Transpioser de u et v
  multiply = multiplication(H , T_u_v , res , n , 2*n , 1 );

  // soustration entre S et la partie pr c dente
  for (int j = 0 ; j < n ; j++ )
    resume[0][j] = s[0][j] - multiply[j][0];
  // a cette etape , on retourne le resultat.
  if (resume != 0)
  {
    return NULL;
  }
```

```
  else
  {
    return u_v;
  }
}
```

Listing 1.11 – fonction bitFlipping

- Partie Echange de clé
L'echange de clé se partage en plusieurs parties à savoir :

— **partie1, h :** $h1 * h0^-1$

```

int ** h1;
  int ** ho ;
  float **mat = NULL;
  //int **h ;
  //h = allocation(h , n , n );
  int **matrice_ho = NULL;
  int **matrice_result = NULL;
  matrice_result = allocation(matrice_result,n,n);
  int **result = NULL;
  result = allocation(result, n , n );
  int **inverse_matrice_ho = NULL;
  int **matrice_h1 = allocation(matrice_h1,n,n);
  h1 = allocation(h1,1,n);
  h1 = generation_aleatoire(w_x);

  mat = allocationf(mat,n,2*n);
  ho = allocation(ho,1,n);

  printf("\n");
  afficherMatriceXY(h1,1,n);
  printf("matrice de h1 \n");
  while(1)
  {

    ho = generation_aleatoire(w_x);
    afficherMatriceXY(ho,1,n);
    printf("matrice de ho \n");
    matrice_ho = rot(ho);
    inverse_matrice_ho = inversionDouble(matrice_ho,mat,matrice_result,n);
    if (inverse_matrice_ho != NULL)
      break;
  }
  afficherMatrice(inverse_matrice_ho,n);
  //exit(0);
  afficherMatrice(matrice_ho,n);
  printf("\n");
  printf("matrice_ho\n");
  matrice_h1 =rots(h1,matrice_h1);
  int **h = multiplication(matrice_h1,inverse_matrice_ho,result , n , n , n);
```

Listing 1.12 – implementation pour trouver h

— **partie2** , $c1 = e0 + h * e1$

```
int **partage_c1(int **h)
{


```

```
  int ** eo;
  int ** e1 ;
  int **res ;
  res = allocation(res , n ,n );
  eo = allocation(eo, 1 , n);
  e1 = allocation(e1,1,n);

  // generation al atoire de e0 et e1
  eo = generation_aleatoire(w_x);
  e1 = generation_aleatoire(w_x);

  //mettre e0 et e1 sous forme matricielle pour pouvoir faire la
   multiplication
  int **matrice_eo = rot(eo);
  int **matrice_e1 = rot(e1);

  int **he1 = multiplication(h , matrice_e1 , res , n , n ,n );

  int ** c1;

  // calcul de c1
  c1 = allocation(c1 ,n , n );

  for(int i = 0; i < n ; i++)
    for(int j = 0; j < n ; j++)
      c1[i][j] = matrice_eo[i][j] + he1[i][j];

  return c1;



}
```

Listing 1.13 – fonction partage$_c$1

— **partie 3 ,** $s = h0 * c1$

```
int **getS(int **ho,int **c1)
{
  int **s ;
  s = allocation(s , n , n );
  int **matrice_ho = rot(ho);
  s = multiplication(matrice_ho , c1 , s ,n,n,n);
  return s;
}
```

Listing 1.14 – fonction getS

— **partie 4 , appel de la fonction bitFlipping**

```
int ** eoe1 = bitFlipping(ho,h1,S, t , w );
```

Listing 1.15 – appel de la fonction bitFlipping

Partie Chiffrement

```
int **chiffrement(int **m, int **eo, int **e1)
{

  int ** co;
  co = allocation(co ,n , n );
  Hash_eo_e1 = Hash(eo,e1);
```

```c
  for (int i = 0 ; i < n ; i++)
  {
    co = (m[0][i] + Hash_eo_e1)%2;
  }

  return co;
}
```

Listing 1.16 – fonction chiffrement

Partie Dechiffrement

```c

int **dechiffrement(int **c)
{
  int ** m;
  m = allocation(m ,n , n );
  int **Hash_eo_e1 = Hash(eo,e1);
  for(int i = 0 ; i < n ; i ++ )
  {
    m = (c[0][i] + Hash_eo_e1)%2;
  }

  return m;
}
```

Listing 1.17 – fonction dechiffrement

Conclusion

Au terme de ce projet , nous sommes particulièrement fier d'avoir touché du doight ce projet qui nous permet d'avoir un apercu des codes correcteurs d'erreur , et la manière dont elle sont utilisé etablir un chiffrement pouvant résister à l'ère post-quantique. Nous avons pu implementer l'ensemble des codes demandées même si nous avons des soucis d'execution , ce qui à fait que n'avons pu obtenir de resultat aussi satisfaisant que nous aurions bien voulu. Ce projet reste quand même un appel pour nous à approfondir ce domaine qui regorge de trouvaille aussi interessante les unes que les autres.

NB : Dans le zip du projet , on peut retrouver 2 fichiers du nom de : BitFlippingtest.c et cryptoISDtest.c , que nous avons ajouté et dont les poids sont petits et qui contiennent des affichages ecran pour pouvoir faire executer le code et voir la manière dont notre algorithme tourne.

Bibliographie

[1] Ayon Chakraborty. Matrix inversion by gauss jordan. *http ://hullooo.blogspot.com/2011/02/matrix-inversion-by-gauss-jordan.html*.

[2] 1997 F.Faber Feedback Copyright © 1993, 1996. Multiplication de deux matrices. *https ://www.ltam.lu/cours-c/solex75.htm*.

[3] Philippe Gaborit and Jean-Christophe Deneuville. Code-based cryptography.

[2] [1] [3]