Master 1 Informatique

Codage et Crypto

Rapport de projet : Crypter et Décripter

BARAULT Corentin, DE SOUSA Yvann

29 avril 2022

Table des matières

1 Contexte et nos Choix

Pour le langage utilisé, nous avons choisi Java pour plusieurs raisons :

— Java est un langage que nous avons souvent utilisé au cours de notre parcours scolaire, ainsi nous sommes assez familiers avec.

— Le langage est très populaire et possède de nombreuses bibliothèque permettant de nous aider si nécessaire

— Java gère de base la plupart des formes que peut prendre le message, ainsi que la gestion de la graine et de la clé.

2 Rappel du sujet

Comme indiqué précédemment, notre projet consiste à réaliser un programme informatique de chiffrement et de déchiffrement en utilisant le principe de masque jetable.
Le masque jetable est un algorithme de cryptographie inventé en 1917 et qui consiste à combiner le message que l'on souhaite crypter, avec une clé.

Cette clé doit être une suite binaire, aussi longue que le message à chiffrer lui-même, choisi de façon aléatoire, et à usage unique.
Toutes ces conditions sont censées permettre à cette méthode d'atteindre une sécurité théorique absolue.

La méthode de chiffrement est relativement simple et consiste à convertir le message en une suite binaire, prendre la clé qui est une suite binaire de même longueur, et appliquer à chaque bit du message l'opération XOR avec le bit de même rang de la clé.

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 1 – Table de logique de XOR

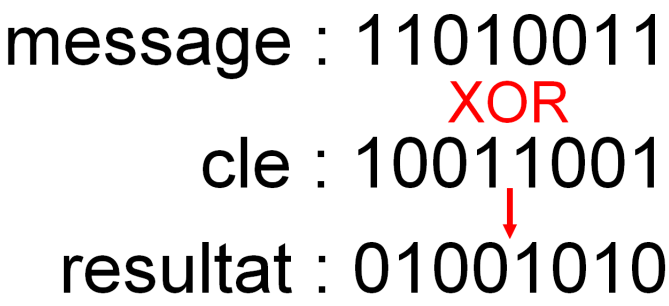

FIGURE 2 – Exemple de chiffrement d'un message

Quant à la méthode de déchiffrement, celle-ci est tout aussi simple et part du même principe, sauf qu'on prend le résultat obtenu auquel on applique l'opération XOR avec la clé.

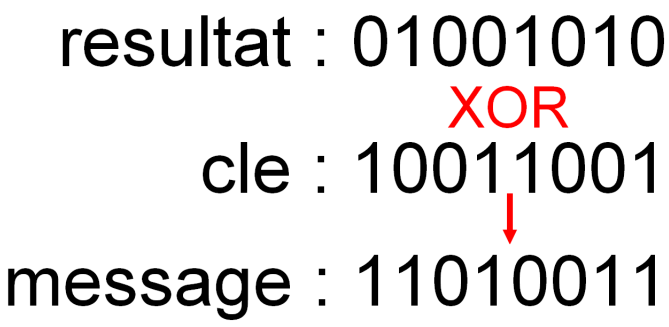

FIGURE 3 – Exemple de chiffrement d'un message

Comme dit précédemment, la clé doit être générée aléatoirement. Cependant, en informatique, l'aléatoire est pratiquement impossible à vraiment produire, ainsi nous sommes obligés d'utiliser ce que l'on appelle une "graine" pour générer une suite de chiffres aléatoire.
Cette précaution est obligatoire pour ne pas souffrir de problèmes de sécurité importants, ce qui gacherait l'entièreté du principe du programme.

Cette graine est une suite binaire que nous avons déjà créé, et ne correspond à aucun modèle précis.
À cette graine nous allons ensuite utiliser le principe de LFSR, c'est-à-dire d'abord utiliser la formule de récurrence suivante ;

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

FIGURE 4 – Formule de récurrence

Dans cette formule X0 est la graine, et a,c et m, des entiers choisis.

Utiliser cette formule nous permet d'obtenir les valeurs de X1 à XInfini si on le souhaite.
Le LFSR consiste ainsi à prendre les valeurs de 4 Xn spécifiques choisis, et de réaliser l'opération XOR sur ces 4 Xn afin d'obtenir la clé que l'on utilisera finalement.
Le choix du LFSR nous est attribué par défaut.

Exemple :

Exemple : Le LFSR

$$X_n = X_{n-1} \oplus X_{n-2} \oplus X_{n-4},$$

avec la graine

$$X_0 = 0, X_1 = 1, X_2 = 1, X_3 = 0$$

correspond à $m = 2$, $a_1 = 1$, $a_2 = 1$, $a_3 = 0$, $a_4 = 1$ et au $k = 4$. Les valeurs ainsi obtenues sont : $X_4 = 1$, $X_5 = 0$, $X_6 = 0$, $X_7 = 0$, $X_8 = 1$, $X_9 = 1$, ...

FIGURE 5 – Exemple d'utilisation du LFSR

Avec toute ces informations nous avons donc tout ce qu'il faut pour coder notre programme.

3 Méthode de codage

Le projet a été découpé en deux parties distinctes, ces deux parties possèdent toutes les deux des LFSR différents. Dans la première partie le LFSR est basique et correspond à cette équation ;

$$X_n = X_{n-5} \oplus X_{n-6} \oplus X_{n-15} \oplus X_{n-16}$$

Dans la deuxième partie, le sujet nous donne 4 LFSR différents à utiliser, et le résultat de ces 4 LFSR seront combiné avec l'opération XOR pour obtenir le LFSR final. Voici l'équation correspondante :

$$\begin{cases} v_n &= v_{n-5} \oplus v_{n-13} \oplus v_{n-17} \oplus v_{n-25} \\ x_n &= x_{n-7} \oplus x_{n-15} \oplus x_{n-19} \oplus x_{n-31} \\ y_n &= y_{n-5} \oplus y_{n-9} \oplus y_{n-29} \oplus y_{n-33} \\ z_n &= z_{n-3} \oplus z_{n-11} \oplus z_{n-35} \oplus z_{n-39} \end{cases}$$

Dans ce cas, la clé secrète est constituée de $25 + 31 + 33 + 39$ bits, et le bit secret engendré apres chaque itération est donné par $b = v_n \oplus x_n \oplus y_n \oplus z_n$ [1].

Ces deux parties ont été codées à part, cependant le principe que nous avons utilisé pour obtenir le LFSR final dans les deux cas est similaire.

3.1 Génération de la clé

Nous avons tout d'abord commencé par créer 5 variables correspondant à des graines. La première sera utilisée pour la première partie du sujet, tandis que les 4 autres pour la deuxième partie. Les fonctions utilisées pour la première partie auront toutes le suffixe "Courte",tandis que celles utilisés dans la deuxième partie auront le suffixe "Longue".

```
1
2 private String seed; // graine courte
3     private String seedLongueA; // graine longue 1
4     private String seedLongueB; // graine longue 2
```

```
5       private String seedLongueC; // graine longue 3
6       private String seedLongueD; // graine longue 4
```

Dans le constructeur de la classe nous avons associé à chaque seed une valeur correspondant à une suite binaire quelconque.

```
1
2   public GenerateurCle() {
3           this.seed = "11010111001001111";
4           this.seedLongueA = "01100010110100011101101011";
5           this.seedLongueB = "00011101010111110011010110110011011";
6           this.seedLongueC = "11110011010101011011001101010001111 0010";
7           this.seedLongueD = "
      1000001100101101000101101101011111110001000111000110110";
8           this.t = 17;
9           this.n = this.seed.length();
10          generer = seed;
11
12          genererLongueA = seedLongueA;
13          genererLongueB = seedLongueB;
14          genererLongueC = seedLongueC;
15          genererLongueD = seedLongueD;
16      }
```

A partir de là, il nous suffit donc d'appeler la fonction Step pour récupérer les valeurs du LFSR qui nous intéresse, ainsi que d'appliquer l'opérateur XOR sur les valeurs binaires ainsi récupérées.

Pour la première partie du sujet nous avons la fonction stepcourte qui s'occupe de faire ça.

```
1
2   public int stepCourte() {
3           byte a = new Integer(Character.getNumericValue(generer.charAt(n-5))).
      byteValue();
4           byte b = new Integer(Character.getNumericValue(generer.charAt(n-6))).
      byteValue();
5           byte c = new Integer(Character.getNumericValue(generer.charAt(n-15))).
      byteValue();
6           byte d = new Integer(Character.getNumericValue(generer.charAt(n-16))).
      byteValue();
7           if ((a ^ b ^ c ^ d) == 0) {
8               this.generer = generer.substring(1);
9               generer = generer + "0";
10              return 0;
11          } else {
12              generer = generer.substring(1);
13              generer = generer + "1";
14              return 1;
15          }
16      }
```

Tandis que pour la partie 2 du sujet nous avons 5 fonctions au total pour faire cela, 4 fonctions step (StepLongueA, StepLongueb, StepLongueC et Step LongueD) ainsi que la fonction

StepLongueComplete.
Les 4 premières s'occupent chacune d'une valeur du LFSR, et la dernière s'occupe de combiner les résultats pour obtenir le LFSR que nous voulons.

```
1
2   public int stepLongueD() {
3         byte LongueD1 = new Integer(Character.getNumericValue(genererLongueD.
      charAt(seedLongueD.length()-3))).byteValue();
4         byte LongueD2 = new Integer(Character.getNumericValue(genererLongueD.
      charAt(seedLongueD.length()-11))).byteValue();
5         byte LongueD3 = new Integer(Character.getNumericValue(genererLongueD.
      charAt(seedLongueD.length()-35))).byteValue();
6         byte LongueD4 = new Integer(Character.getNumericValue(genererLongueD.
      charAt(seedLongueD.length()-39))).byteValue();
7
8         if ((LongueD1 ^ LongueD2 ^ LongueD3 ^ LongueD4) == 0) {
9             this.genererLongueD = genererLongueD.substring(1);
10            genererLongueD = genererLongueD + "0";
11            return 0;
12        } else {
13            genererLongueD = genererLongueD.substring(1);
14            genererLongueD = genererLongueD + "1";
15            return 1;
16        }
17    }
18
19    public int stepLongueComplete() {
20        byte a = (byte) stepLongueA();
21        byte b = (byte) stepLongueB();
22        byte c = (byte) stepLongueC();
23        byte d = (byte) stepLongueD();
24        if ((a ^ b ^ c ^ d) == 0) {
25            return 0;
26        } else {
27            return 1;
28        }
29    }
```

Pour finir, les fonctions générales s'occupent de générer le LFSR basé sur les résultats obtenus à partir de la graine.

```
1  public String generateCourt(int k) {
2         generer = seed;
3         String var = "";
4         for (int i = 0; i < k; i++) {
5             var += stepCourte();
6         }
7         return var;
8     }
9
10    public String generateLongue(int k) {
11        genererLongueA = seedLongueA;
12        genererLongueB = seedLongueB;
13        genererLongueC = seedLongueC;
14        genererLongueD = seedLongueD;
15        String var = "";
```

```
16          for (int i = 0; i < k; i++) {
17              var += stepLongueComplete();
18          }
19          return var;
20      }
```

3.2 Crypter et décripter

Le sujet du projet nous demande de pouvoir crypter et décrypter tout type de fichiers, pour se faire ce que nous devons faire consiste à convertir tous les types de fichiers que l'on souhaite gérer en suite binaire.
C'est à ça que sert le fichier "Loader java" et sa fonction "Charger" ;

```
1  public byte[] Charger(String nom) throws IOException
2      {
3          File file = new File(nom);
4
5          byte[] b = new byte[(int) file.length()];
6          try {
7              FileInputStream fileInputStream = new FileInputStream(file);
8              fileInputStream.read(b);
9              return b;
10         } catch (FileNotFoundException e) {
11             System.out.println("File Not Found.");
12             e.printStackTrace();
13         }
14         return b;
15
16     }
```

À ce stade du projet nous avons tout ce qu'il nous faut pour générer la clé et le LFSR, nous avons le message que nous voulons chiffrer, il ne nous reste plus qu'a créé la fonction pour chiffrer et déchiffrer le tout.
Nous avons deux types de fonction de cryptage :
La première est "CrypteUnByte" qui sert a crypter un seul octet

```
1  public Byte crypteUnByteCourte(Byte message){
2
3          String clegenerer = generateCourt(8);
4          byte cle = (byte)Long.parseLong(clegenerer, 2);
5          byte resultat = (byte)(message ^ cle);
6          return resultat;
7      }
```

Tandis que la deuxième est "CrypteByteArray" qui comme son nom l'indique, crypte un array de byte en appelant la fonction CrypteUnByte dans une itération.

```
1
2   public byte[] crypteByteArrayCourte(byte[] message){
3
4          byte[] resultat = message;
5          for (int i = 0; i < message.length; i++) {
```

```
6               byte b = crypteUnByteCourte(message[i]);
7               resultat[i] = b;
8           }
9           return resultat;
10      }
```

Pour rappel, pour déchiffrer un message il faut simplement le recrypter avec la même clé. Ainsi, nous n'avons pas besoin de fonction spéciale pour décrypter étant donné que nos fonctions de chiffrement le font déjà.

4 Conclusion

Ce projet fut très intéressant à faire étant donné que c'est notre première approche concrète dans le monde du cryptage de données et de la sécurité informatique. Les objectifs qui nous ont été confiés par le sujet ont tous été remplis.