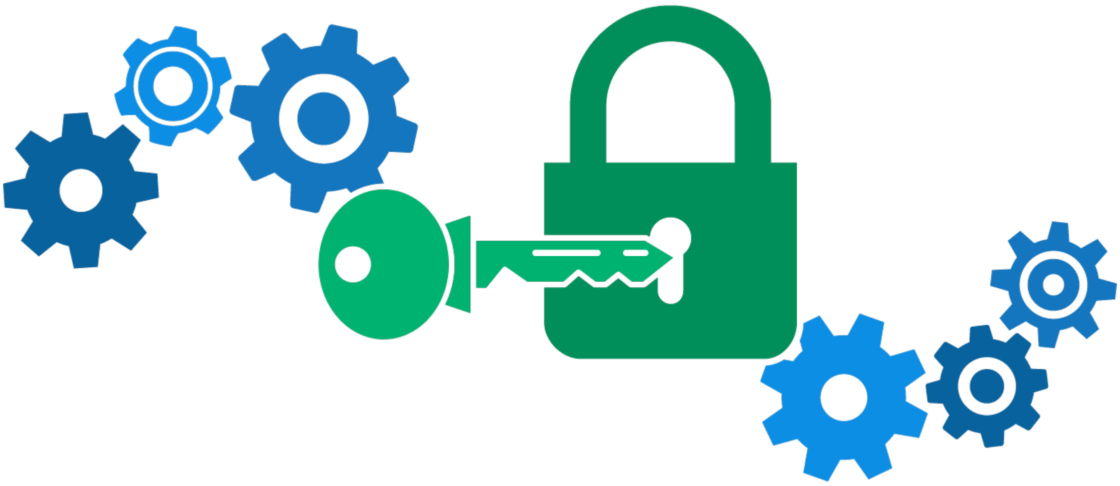
Université de Versailles Saint-Quentin-en-Yvelines

Devoir maison

Cryptographie Symétrique



Enseignant : Christina Boura

Auteur : Salsabil Dafrane
Théo Lefebvre
Soufiane Chikar

19 Mars 2020 - 7 avril 2020

1 Générateur de type Geffe pour le chiffrement à flot

Solution 1.1 :

Cf : Fichiers sources (.c) et en-tête (.h)

```
/* Représente un registre ayant une taille de 16 bits */
struct Register
{
    int value[R_SIZE];
};
typedef struct Register REGISTER;

/* Représente un lfsr qui possède un registre et des coefficients de rétroaction */
struct lfsr
{
    REGISTER L;
    int coefficient[R_SIZE];
};
typedef struct lfsr LFSR;

/* Représente une sous-clé de 16 bits qui compose la clé principale K,
sert à l'initialisation d'un registre */
struct subkey
{
    int value[S_SIZE];
};
typedef struct subkey SUBKEY;

/* Représente la clé principale de 48 bits, composé de 3 sous-clés */
struct key
{
    SUBKEY subkey1;
    SUBKEY subkey2;
    SUBKEY subkey3;
};
typedef struct key KEY;

/* Représente la suite chiffrante, ayant une taille
définie parl'utilisateur et un ensemble de valeur */
struct ciphersuite
{
    int size;
    int* value;
};
typedef struct ciphersuite CIPHERSUITE;

/* Représente le générateur de Geffe, composé d'une clé K de 48 bits,
de 3 lfsr et de la fonction de filtrage */
struct geffe
{
    KEY key;
    LFSR lfsr1;
    LFSR lfsr2;
    LFSR lfsr3;
    int F[8];
};
typedef struct geffe GEFFE;

/* Construit le générateur, produit la suite chiffrante,
réalise les corrélations avec F(x0x1x2) et permet de réaliser l'attaque sur celui-ci */
void Generate(char* _filtrageArg[], char* _keyArg, int _n);
```

Solution 1.2 :

La première chose à faire afin de calculer la corrélation entre la sortie du générateur et la sortir de chaque LFSR, est d'établir un tableau en disposant toutes les combinaisons possible des bits de sorties des 3 LFSR et ce que produit la fonction de filtrage avec ces combinaisons. Ici chaque bit peut prendre 2 valeur : 0 ou 1, étant donnée qu'il y a 3 bit de sortie (un par LFSR) on à 2^3 = 8 combinaisons possibles.

x0	x1	x2	F(x0x1x2)
0	0	0	f_0
0	0	1	f_1
0	1	0	f_2
0	1	1	f_3
1	0	0	f_4
1	0	1	f_5
1	1	0	f_6
1	0	1	f_7

Corrélation de x_1 avec f_i = 50%.

A partir du tableau on compte le nombre de fois où un bit d'entrée x_i correspond à un bit de sortie f_i. Pour x_1, on imagine qu'il soit égale à f_i 4 fois sur les 8 cas possibles (cf. couleur du tableau), avec $i_{max} = 7$. On calcule dont le pourcentage d'égalité entre x_1 et F(x0x1x2), ce qui donne (4/8)*100 = 50%. On a donc x_1 qui à une corrélation de 50% avec la sortie du générateur s_i.

La formule générale reviens à :

$$\text{Corrélation } x_{iF} = \frac{\text{nombre de fois ou } x_i \text{ est egale à } f_i}{i_{max} + 1} * 100$$

Solution 1.3 :

Pour effectuer une attaque de type "diviser pour régner" contre ce générateur il suffit de réaliser les étapes suivantes :
- Effectuer les corrélations entre les 2^3 possibilités de bits d'entrée x0x1x2 avec le résultat de la fonction de filtrage (f_0 à f_n).
- Ensuite diviser le générateur en 3 parties indépendantes.
- Faire une recherche exhaustive (force brut) de la clés k0 du registre L0, la clés sera la bonne lorsque la corrélation entre les bits x0 de la clés k0 avec la suite s sera la même que la corrélation de base.
- On effectue le même chose pour les registres L1 et L2.

Solution 1.4 :

L'attaquant a besoin d'une suite chiffrante de 16 bits seulement pour mettre l'attaque en oeuvre, car il lui faut autant de bit de la suite chiffrante que la taille d'une sous clés.

- **complexité en mémoire :** Pour une recherche exhaustive on teste toutes les possibilités des 3 LFSR ce qui nous donne 2^{48}. Pour une attaque par corrélation, on cherche l'initialisation des 3 LFSR indépendament des autres et on doit stocker leur valeur pour vérifier que ces 3 initialisations redonne bien la suite chiffrante, donc 2^{16} + 2^{16} + 2^{16}.
- **complexité en temps :** $2^1$6, car c'est le nombre de bits de la suite chiffrante générée.
- **comparaison :** L'attaque diviser pour régner à une complexité en temps beaucoup plus faible que la recherche exhaustive de la clé : (2^{16} + 2^{16} + 2^{16}) < 2^{48}.

Solution 1.5 :

Un programme a été implémente en C pour répondre à la question.

```c
/* Calcul la corrélation entre une sous-clé ki
(parmis les 2^16 possibilités de sous-clés) avec la suite chiffrante */
float isGoodKey(SUBKEY* _subkey, CIPHERSUITE* _suite);

/* Permet de tester toute les valeurs possibles pour une sous-clé,
en cheangant la valeur de celle-ci (2¹ = 65 536 combinaisons) */
void brutForce(SUBKEY* _subkey, int* i);

/* Initialise une sous-clé avec une valeur par défault
avant le début de la recherche exhaustive */
void subkeyAsDefaultValue(SUBKEY* _subkey);

/* Test si la corrélation trouver avec la fonction isGoodKey()
est égale à cele trouver avec la fonction corrélation,
si oui renvoi la sous-clé actuel sinon test avec une autre sous-clé */
SUBKEY findSubkey(float _correlation, CIPHERSUITE* _suite);

/*/ Lance une attaque par corrélation de type
diviser pour régner pour récuperer les 3 sous-clés indépendamment */
KEY attaque(float _correlation[3], CIPHERSUITE* _suite);
```

Solution 1.6 :

Une fonction F qui rend l'attaque contre ce générateur la plus difficile aura une corrélation de 50% avec chaque bits d'entrées. L'attaquant aura donc 50% de chance de trouver la bonne clef, mais aussi 50 % de chance de trouver la mauvaise en comparant les bits x0 puis x1 puis x2 avec le bit de sortie. Voici un exemple de fonction de filtrage qui rend l'attaque le plus difficile possible F = { 0, 0, 1, 1, 1, 1, 0, 0} . Sur ce tableau, on remarque donc une correspondance avec le bit x_i et celui de la fonction de filtrage F de 50 %.

x0	x1	x2	F(x0x1x2)
0	0	0	0
1	0	0	0
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	0	1	0
1	1	1	0

2 Un chiffrement par bloc faible

Solution 2.1 :

On sait que

x_0^L = (0x45019824)$_{16}$ = (0100 0101 0000 0001 1001 1000 0010 0100)$_2$
x_0^R = (0x51023321)$_{16}$ = (0101 0001 0000 0010 0011 0011 0010 0001)$_2$

k_0 = (0x01020304)$_{16}$ = (0000 0001 0000 0010 0000 0011 0000 0100)$_2$
k_1 = (0x98765432)$_{16}$ = (1001 1000 0111 0110 0101 0100 0011 0010)$_2$

En observant le chiffrement de Feistel sur la Figure 1 on obtient :

$$x_1^L = ((x_0^L \oplus x_0^R) <<< 7) \oplus k_0$$
$$x_1^R = ((x_0^R \oplus x_1^L) <<< 7) \oplus k_1$$

Soit $x_1^L = (((0001\ 0100\ 0000\ 0011\ 1010\ 1011\ 0000\ 0101)_2 <<< 7) \oplus k_0)$
$\equiv (((0x1403AB05)_{16} <<< 7) \oplus k_0)$

$x_1^L = (((0000\ 0000\ 1101\ 0111\ 1000\ 0001\ 1000\ 1110)_2) \oplus k_0)$
$\equiv (((0x1D5828A)_{16}) \oplus k_0)$

$x_1^L = (0000\ 0000\ 1101\ 0111\ 1000\ 0001\ 1000\ 1110)_2$
$\equiv (0xD7818E)_{16}$

Par conséquent

$x_1^R = (((0101\ 0001\ 1101\ 0101\ 1011\ 0010\ 1010\ 1111)_2 <<< 7) \oplus k_1)$
$\equiv (((0x51D5B2AF)_{16} <<< 7) \oplus k_1)$

$x_1^R = (1110\ 1010\ 1101\ 1001\ 0101\ 0111\ 1010\ 1000)_2 \oplus k_1$
$\equiv (0xEAD957A8) \oplus k_1$

$x_1^R = (0111\ 0010\ 1010\ 1111\ 0000\ 0011\ 1001\ 1010)_2$
$\equiv (0x72AF039A)_{16}$

On a donc déterminer le couple (x_1^L, x_1^R) = **(0xD7818E, 0x72AF039A)**$_{16}$

Un programme a été implementé en C pour cette question.

```c
int * chiffrement_par_bloc(
            int x0L,
            int x0R,
            int k0,
            int k1,
            int * chiffree){
     /* chiffree est tableau qui récupère les sous clefs */
    chiffree[0] = rotation_gauche_7bits(x0L ^ x0R) ^k0;

    chiffree[1] = rotation_gauche_7bits(x0R ^ T0) ^ k1;

    return chiffree;

}
```

Solution 2.2 :

Voici ci-dessous le chiffrement sur un tour sous forme d'équations où les clef k_0 et k_1 seront nos inconnues :

$$\begin{aligned} x_1^L &= ((x_0^L \oplus x_0^R) <<< 7) \oplus k_0 \\ x_1^R &= ((x_0^R \oplus x_1^L) <<< 7) \oplus k_1 \end{aligned}$$

Or, on sait que $C = A \oplus B \equiv B = A \oplus C$

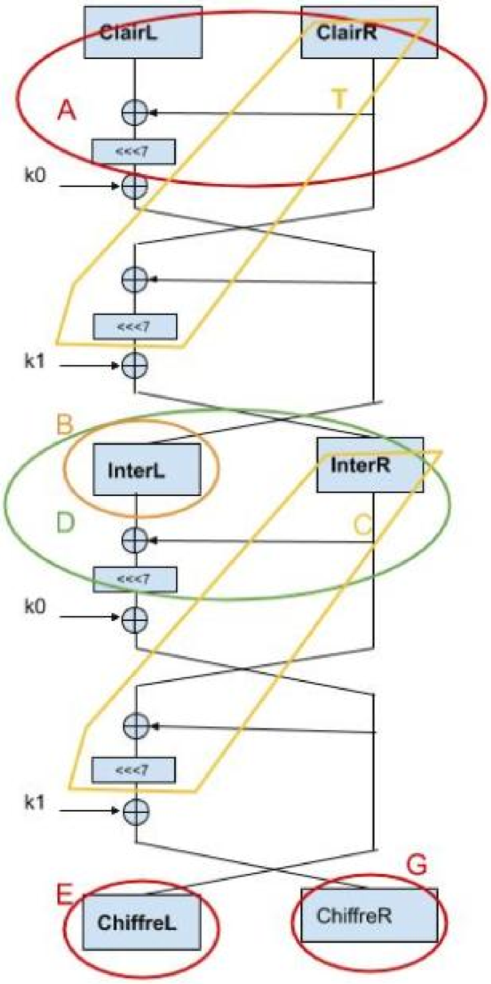
On a donc :

$$\begin{aligned} k_0 &= ((x_0^L \oplus x_0^R) <<< 7) \oplus x_1^L \\ k_1 &= ((x_0^R \oplus x_1^L) <<< 7) \oplus x_1^R \end{aligned}$$

Un programme a été implémenter en C, qui prend en entrée des textes (claires/chiffrés) choisis par l'attaquant sur 1 tour de Feistel et renvois la clé secrète (k_0,k_1).

```
int * cryptanalyse_1_tour(
                     int clairL,
                     int clairR,
                     int chiffreL,
                     int chiffreR,
                     int * keys){

   /* k0 = x1L  ( ( x0L  x0R) <<< 7 ) */
   /* k1 = x1R  (( x0R  x1L) <<< 7 )  */

    keys[0] = chiffreL ^ rotation_gauche_7bits(clairL ^ clairR);

    keys[1] = chiffreR ^ rotation_gauche_7bits(clairR ^ chiffreL);

    return keys;
}
```

Solution 2.3 :



Premièrement :

on sait que pour 1 tour, on a ::
k0 = A ^ B
k1 = interR ^ T

Au second tour , on a :

k0 = D ^ E
k1 = C ^ G

on a donc comme égalité :

pour k0 : A ^ B = D ^ E
pour k1 : interR ^ T = C ^ G

pour k0 : A ^ B ^ D ^ E = 0
pour k1 : interR ^ T ^ C ^ G = 0

pour k0 : A ^ E = B ^D
pour k1 : interR ^ C = T ^ G

pour k0 = A ^ E
pour k1 = T ^ G

On possède G, E, T et A

ça veut dire que pour 12 tours la clef (k0,k1) est :

k0 = (clairL ^clairR)<<<7) ^ chiffreL

k1 = ((clairR ^ (((clairL ^clairR)<<<7) ^k0))<<<7) ^ chiffreR

$$k_0 = ((x_0^L \oplus x_0^R) <<< 7) \oplus x_{12}^L$$
$$k_1 = ((x_0^R \oplus (((x_0^L \oplus x_0^R) <<< 7) \oplus k0)) <<< 7) \oplus x_{12}^R$$

Solution 2.4 :

Un programme a été implémenté en C pour retourner la clef(k_0,k_1), en entrant un couple (clair, chiffré) sur 12 tours.

```
int * cryptanalyse(int clairL,
                   int clairR,
                   int chiffreL,
                   int chiffreR,
                   int * keys){
    /* Déclaration de variable temporaire pour faire les calcul */
    int k0, k1;

    k0 = rotation_gauche_7bits(clairL ^clairR) ^ chiffreL;

    k1 = ( rotation_gauche_7bits(clairL^clairR) ^k0 ) ^ clairR;
    k1 = rotation_gauche_7bits(k1) ^ chiffreR;

    /* tableau retournant la clef(k0,k1) */
    keys[0] = k0;
    keys[1] = k1;
    return keys;
}
```

Solution 2.5 :

Ajoutez plus de tours ne rendra pas le chiffrement plus solide, car la clef (k_0,k_1) utilisée entre chaque tour est identique. Quelques soit le nombre de réitération du chiffrement par bloc, l'attaquant pourra récupérer la sous-clef k_0 puis en déduire la sous-clef k_1 en appliquant un système d'équation dessus, avec le couple (clair/chiffré) qu'il possède.

Solution 2.6 :

Pour obtenir une amélioration du chiffrement $E_{k0,k1}$, de façon que notre attaque ne s'applique, il faut générer une clef aléatoire entre chaque tour.
Pour N tours on doit avoir une clef (k0,k1) qui se modifie N fois.