Cryptographie - ReverseTheFlow - 200 points

Wail TAHMAOUI, Kévin DUVERGER

Table des matières

1 Résolution ReverseTheFlow :

Introduction. Ce challenge nous donne donc 2 fichiers : le fichier `cipher_image_lfsr.py` qui représentent probablement un chiffrement à base de LFSR, et le fichier `cipher_image` qui est l'image chiffrée.
Vous pouvez voir ici que nous ne connaissons pas la clé et c'est ce qu'il va falloir trouver. Pour ce faire, il faut savoir que le LFSR est sensible à une attaque à clair connu : on pourra utiliser la signature de l'image ! Pour savoir le type d'image, on peut aller voir le code du fichier de chiffrement qui nous montre qu'une `png` semble avoir été utilisée.

Rappel du fonctionnement d'un LFSR. Le LFSR permet de créer un keystream infini bit par bit en partant d'une seed initiale. Dans la suite, pour avoir formaliser un peu mieux l'attaque, nous appellerons donc l la longueur de la seed et s_i la suite des bits générés par le LFSR (en partant de s_0).
Le bit s_l est donc calculé à partir de tous les bits précédents, l'idée étant d'avoir $s_l = a_0 \times s_0 + a_1 \times s_1 + ... + a_{l-1} \times s_{l-1}$. Ces $(a_0, a_1, ..., a_{l-1})$ sont un autre paramètre du LFSR et ils vont décider quels bits précédents prennent part au calcul du nouveau bit (c'est ce qui est appelé `taps` dans le code donné).

Partie 1 - Trouver la seed. Vu que c'est une image PNG, l'en-tête est `0x89504E470D0A1A0A` (ce que vous pouvez retrouver en allant sur `https://en.wikipedia.org/wiki/List_of_file_signatures`). Le chiffrement consiste en un simple XOR entre le texte clair et le keystream généré par le LFSR, c'est à dire $C = M \oplus K$. On peut donc retrouver le début du keystream avec $K = C \oplus M$!
En faisant cela sur les 64 premiers bits du texte chiffré, on obtient donc `80000001800000FD` comme premiers bits du keystream. Malheureusement, nous ne connaissons pas la taille de la seed, nous allons donc la bruteforcer !

Partie 2 - Trouver a. Pour trouver les a_i, l'idée sera d'utiliser l'équation du LFSR et les bits que l'on connaît déjà pour avoir autant d'équations qu'il y a de a_i (il y en a autant que la taille de la seed). Cela veut dire qu'avec nos 64 bits de keystream, la seed aura une taille maximale de 32 bits.
Voici donc la liste d'équations que l'on peut obtenir :

$$\begin{cases} s_l = a_0 \times s_0 + a_1 \times s_1 + \cdots + a_{l-1} \times s_{l-1} \\ s_{l+1} = a_0 \times s_1 + a_1 \times s_2 + \cdots + a_{l-1} \times s_l \\ s_{l+2} = a_0 \times s_2 + a_1 \times s_3 + \cdots + a_{l-1} \times s_{l+1} \\ \vdots \\ s_{l+(l-2)} = a_0 \times s_{l-2} + a_1 \times s_{l-1} + \cdots + a_{l-1} \times s_{l+(l-3)} \\ s_{l+(l-1)} = a_0 \times s_{l-1} + a_1 \times s_l + \cdots + a_{l-1} \times s_{l+(l-2)} \end{cases}$$

Et on peut le transformer en matrice :

$$\begin{pmatrix} s_l \\ s_{l+1} \\ s_{l+2} \\ \vdots \\ s_{l+(l-2)} \\ s_{l+(l-1)} \end{pmatrix} = \begin{pmatrix} s_0 & s_1 & s_2 & \cdots & s_{l-1} \\ s_1 & s_2 & s_3 & \cdots & s_l \\ s_2 & s_3 & s_4 & \cdots & s_{l+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{l-2} & s_{l-1} & s_l & \cdots & s_{l+(l-3)} \\ s_{l-1} & s_l & s_{l+1} & \cdots & s_{l+(l-2)} \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{l-2} \\ a_{l-1} \end{pmatrix}$$

Il ne reste donc plus qu'à inverser la matrice pour récupérer les a_i !

Code de la solution. Étant donnée une taille `LFSR_SIZE`, voici le code pour récupérer la seed et les a_i :

```python
import galois
import numpy as np

def computeNextLFSRBit(A, K) :
    return sum([ A[j] & K[j] for j in range(0, LFSR_SIZE) ]) % 2

# Chargement
fichier = open("cipher_image", "rb")
data = fichier.read()
fichier.close()
knownBytes = bytes.fromhex("89504E470D0A1A0A")

# Bits de la clé
Kbits = []
for i in range(0, 8) :
    Kbits += [ int("{:08b}".format(data[i] ^ knownBytes[i])[j]) for j in range(0, 8) ]

# On forme la matrice
GF2 = galois.GF(2)
mat = [ [] for i in range(0, LFSR_SIZE) ]
```

```python
for i in range(0, LFSR_SIZE) :
    mat[i] = Kbits[i:i+LFSR_SIZE]
# Inversion
try :
    invMat = np.linalg.inv(GF2(mat))
except np.linalg.LinAlgError :
    continue

# On peut maintenant calculer le vecteur des a_i
A = [ 0 for i in range(0, LFSR_SIZE) ]
for i in range(0, LFSR_SIZE) :
    for j in range(0, LFSR_SIZE) :
        A[i] ^= int(invMat[i][j]) & Kbits[LFSR_SIZE + j]

# Une fois qu'on a les A[i], on va tenter de re-calculer la suite
computedKbits = [ Kbits[i] for i in range(0, LFSR_SIZE) ]
for i in range(0, 64) :
    computedKbits.append(computeNextLFSRBit(A, computedKbits[i:i+LFSR_SIZE]))

# Si tout est OK, on a probablement la bonne taille
if Kbits == computedKbits :
    print("Taille qui marche : " + str(LFSR_SIZE))
    print("Seed : " + str(Kbits[0:LFSR_SIZE]))
    print("A    : " + str(A))
```

En réalisant un brute-force sur `LFSR_SIZE`, on trouve que 31 semble marcher !

Finalisation. Pour finir, il ne reste plus qu'à générer la totalité du keystream et le xorer avec l'image :

```python
# Etape 1 : calcul du keystream
index = 64 - LFSR_SIZE
while len(computedKbits) != 8 * len(data) :
    computedKbits.append(computeNextLFSRBit(A, computedKbits[index:index+LFSR_SIZE]))
    index += 1

# Etape 2 : transformation en bytes
toXorWith = b""
for i in range(0, len(data)) :
    value = sum([ computedKbits[8 * i + j] * (2 ** (7 - j)) for j in range(0, 8) ])
    toXorWith += value.to_bytes(1, "big")

# Etape 3 : on dechiffre
decrypted = b""
for i in range(0, len(data)) :
    decrypted += (toXorWith[i] ^ data[i]).to_bytes(1, "big")

# Etape 4 : on sauvegarde
fichier = open("solved.png", "wb")
fichier.write(decrypted)
fichier.close()
```

Ce qui nous permet de déchiffrer l'image et récupérer le flag :