2EL1740 : Algèbre & Cryptologie

CentraleSupélec - 2A

Challenge 2

27 septembre 2024

CentraleSupélec

Raphaël PAIN DIT HERMIER
Alexis LOMBARD-GAILLARD
Edward LUCYSZYN

Table des matières

1 Question 1

Dans cette question, notre tâche est de casser un chiffrement de Merkle-Hermann avec certaines informations données.

On rappelle les informations données. Notre cible possède comme clé publique :

$$\overline{S} = [9689, 9535, 8457, 7071, 4145, 8136, 6121, 2091, 4028, 8364, 6423, 7918].$$

On possède aussi ces statistiques à partir de 900 messages chiffrés (de type $C = m \cdot \overline{S}$) :

$$\text{min : 8, mean : 4976.2, median : 4981, standard dev. : 2964.9, max : 10141.}$$

et notre objectif est de retrouver la clé (S, p, r).

1.1 Analyse du problème, pistes et hypothèses

On remarque que la taille de la clé publique est de taille 12 et que la valeur maximale de chiffrés obtenue par statistiques est de l'ordre de 10^4, ce qui nous donne approximativement l'ordre de p. Ainsi un décodage par "brute force" consistant à tester les valeurs de p, r qui correspondraient à la clé publique trouvée aurait un temps de calcul négligeable par rapport à 2^{128}. Un décodage par brute force est donc faisable étant donné la taille des données en jeu, néanmoins il serait utile de l'optimiser le plus que possible afin d'effectuer le moins de calculs inutiles et ainsi d'économiser du temps.

Tout d'abord, on peut supposer qu'en tant qu'observateurs, les messages chiffrés que l'on collecte sont choisis de manière aléatoire, et même on peut supposer que $m \xleftarrow{\$} \{0,1\}^{12}$ est choisi aléatoirement de manière uniforme (12 est la taille de la clé publique, donc celle de la clé secrète).
Cela peut nous amener alors à supposer que $C \xleftarrow{\$} \{0, \dots, p-1\}$ suit une loi uniforme sur $\{0, \dots, p-1\}$, pourvu que les valeurs possibles de C sont "assez bien réparties" dans $\{0, \dots, p-1\}$. La validité de cette hypothèse peut être corroborée par le fait qu'une loi uniforme sur $\{0, \dots, p-1\}$ possède une moyenne $\mathbb{E} = \dfrac{p-1}{2}$ ainsi qu'une variance $\mathbb{V} = \dfrac{p^2-1}{12}$. Pour p de l'ordre de 10141 (le max des données obtenues), on obtient une moyenne $\mathbb{E} \approx 5070$ et un écart-type $\sigma \approx \dfrac{10141}{2\sqrt{3}} \approx 2927.5$, ce qui est de l'ordre des statistiques de départ.
Enfin, on supposera que les messages $C_1, ..., C_{900}$ sont pris de manière indépendante entre eux.

1.2 Détail de la stratégie

Ces hypothèses étant formulées, on va maintenant pouvoir s'intéresser au maximum de nos données, notons $M = \max_{i \in \{1,\dots,900\}} C_i$ variable aléatoire sur $\{0, \dots, p-1\}$. On sait dans tous les cas que $M < p$ par définition du chiffrement de Merkle-Hermann.
Puis, pour $n \in \mathbb{N}^*$:

$$\begin{aligned}
\mathbb{P}(M + n \geq p) &= 1 - \mathbb{P}(M < p - n) \\
&= 1 - \mathbb{P}(\forall i \in \{1, \dots, 900\}, C_i < p - n) \\
&= 1 - \prod_{i=1}^{900} \mathbb{P}(C_i < p - n) \text{ (par indépendance des } C_i) \\
&= 1 - \left(\frac{p-n}{p}\right)^{900} \text{ (loi uniforme)} \\
&\approx 1 - \left(1 - \frac{n}{M}\right)^{900}
\end{aligned}$$

dès lors que M est de l'ordre de p, et $n \ll M$.
Ainsi on prendra $n \in \mathbb{N}^*$ tel que $p \in \{M+1, \dots, M+n\}$ avec une probabilité $1-\varepsilon$, ε étant une précision arbitraire.

Par exemple, pour $n = 100$ (et $M = 10141$), on obtient une précision $\varepsilon \approx 0.00013$, qui est une précision convenable pour une première tentative de décodage.
Notre stratégie est alors la suivante : on teste pour les valeurs de $p \in \{M+1, \dots, M+n\}$ premier et $0 < r < p$ si $S = r^{-1} \cdot \overline{S}[p]$ est une suite surcroissante, renvoyant à la fin tous les triplets (S, r, p) qui satisfont cette propriété. On propose alors le code suivant :

```
import numpy as np
import time as t

Sbar = np.array([9689, 9535, 8457, 7071, 4145, 8136, 6121, 2091, 4028, 8364, 6423, 7918])
size = np.size(Sbar)

borneinf = 10141
def is_prime(p):
    for k in range(2,int(np.sqrt(p)+1)):
        if p%k ==0:
            return False
    return True

def givelistp(borneinf,bornesup):
    listp = []
    for p in range(borneinf,bornesup):
        if is_prime(p):
            listp.append(p)
    return listp

listp = givelistp(borneinf,borneinf+100)

def is_surcroissante(S):
    nS = np.size(S)
    sumS = 0
    for k in range(nS-1):
        sumS += S[k]
        if S[k+1] <= sumS:
            return False
    return True

def CrackMH(Sbar,listp):
    start = t.time()
    listpossibleS = []
    for p in listp:
        for r in range(1,p):
            rminus1 = pow(r, -1, p)
            S = np.sort((rminus1*Sbar)%p)
            if is_surcroissante(S):
                listpossibleS.append((S,p,r))
                print("Trouve ! ")
                print(t.time()-start, (S,p,r))
    return listpossibleS

listS = CrackMH(Sbar,listp)
```

Le code renvoie un unique triplet $(S, p, r) = ([3, 4, 11, 20, 39, 79, 158, 316, 633, 1264, 2529, 5090], 10151, 9997)$ en tournant pendant moins de 1 seconde. Il est donc très probable (d'après les calculs préliminaires précédents) que ce soit l'unique triplet qui marche.

1.3 Confirmation des statistiques

On se propose de vérifier que l'unique triplet (S, p, r) obtenu satisfait bien aux statistiques de départ et aux hypothèses énoncées. Pour ce faire, on vérifiera :

— que les valeurs minimale, médiane et maximale sont atteignables par la clé privée obtenue,

— que les statistiques de nos données, particulièrement les valeurs moyenne, médiane et d'écart-type sont reproduisibles par des tirages aléatoires indépendants $m \xleftarrow{\$} \{0,1\}^{12}$.

On propose alors le code suivant :

```
def decode(S,p,r,C):
    n = np.size(S)
    m = [0]*n
    u = (pow(r, -1, p)*C)%p
    for k in range(n-1,-1,-1):
        if u>=S[k]:
            u-=S[k]
            m[k] = 1
    if u==0:
        return m
    return False


def statsMH(S,p,r,N):
    Sbar = (r*S)%p
    print(Sbar)
    size = np.size(Sbar)
    Messages = np.random.randint(2, size=(N, size))
    ListC = np.matmul(Messages,Sbar)%p
    print(np.size(Messages),np.size(Sbar),np.size(ListC))
    return np.min(ListC),np.mean(ListC),np.median(ListC),np.std(ListC),np.max(ListC)

N = 900

for H in listS:
    (S,p,r) = H
    for C in [8,4981,10141]:
        print(decode(S,p,r,C))
    print(statsMH(S,p,r,N))
```

On obtient alors que :

— 8, 4981 et 10141 sont obtenus par les messages respectifs $(0,0,0,0,0,0,0,0,1,1,0,1)$, $(1,1,1,1,0,1,1,1,0,1,1,0)$ et $(0,1,0,0,0,0,1,0,0,1,1,0)$,

— Lorsque l'on fait tourner la fonction statsMH(S,p,r,900), cette dernière renvoie le quintuplet $(9, 4937.35, 4919.5, 2901.391064596812, 10146)$. Par complétude, elle renvoie dans une autre instance le quintuplet $(7, 5151.4366666666665, 5265.5, 2953.1433015216553, 10136)$ et des valeurs similaires dans les autres instances observées. Les statistiques de départ sont donc consistantes avec la solution (S, p, r) trouvée.

Ces observations nous permettent de conclure avec très grande certitude que cette solution est bien la clé secrète de notre cible.

2 Question 2

L'objectif de l'exercice est de produire un nombre p véritablement premier entre 2^{38} et 2^{40} et son certificat de primalité de Pocklington (ainsi que tous les sous-certificats nécessaires). Puis, il faudra donner le plus petit entier $n > p$ qui est 7-lisse.

2.1 Recherche de p

Tout d'abord, rappelons le théorème de Pocklington. Soit n un entier. S'il existe des entiers a et q tels que :

1. q est premier, $q|(n-1)$, et $q > \sqrt{n} - 1$;
2. $a^{n-1} = 1 \bmod n$;
3. $\gcd(a^{(n-1)/q} - 1, n) = 1$;

alors n est premier.

La difficulté avec le test de primalité de Pocklington, est qu'il est nécessaire d'avoir la paire (q, a) qui constitué le certificat de primalité pour n. Comme il y a deux ordres de grandeur d'écart entre 2^{38} et 2^{40}, l'idée à appliquer ici serait fabriquer des nombres premiers de plus en plus grands, en parant d'un nombre premier q très petit.

Par exemple, on prend $a = 2$ et $q = 3$. Puis, on calcule $n = qR + 1$. On a bien $q|(n-1)$, et en prenant R suffisamment petit, on a aussi $q > \sqrt{n} - 1$. Maintenant, pour les deux dernières conditions, comme on a fixé a, il suffit d'incrémenter la valeur de R jusqu'à avoir $a^{qR} = 1 \bmod n$ et $\gcd(a^{(R} - 1, n) = 1$. Une fois qu'on a un tel R, on recommence avec le nouveau nombre premier n.

En parant de ce principe, nous obtenons l'algorithme suivant.

```
def fast_exponentiation(a, n, m):
    """
    Calcule a^n mod m en O(log n).
    """
    result = 1
    a %= m
    while n > 0:
        if n % 2 == 1:
            result = (result * a) % m
        a = (a * a) % m
        n //= 2
    return result


def gcd(a, b):
    """
    Calcule le PGCD de a et b.
    """
    while b:
        a, b = b, a % b
    return a


def generate_pocklington_prime(a, q, min, max):
    """
    Genere un nombre premier entre min et max en utilisant le test de Pocklington.
    """
    Q = []
    A = []
    N = []
    while q < min:
```

```
            R = 2
            a = 2
            n = q*R + 1
            while not (q + 1)**2 - n > 0 or fast_exponentiation(a, n - 1, n) != 1 or \\
            gcd(fast_exponentiation(a, int((n - 1)/q), n) - 1, n) != 1:
                R += 1
                n = q*R + 1
            else:
                Q.append(q)
                A.append(a)
                N.append(n)
                print(a, q, R, n)
                q = n

    if q > max:
        return None
    else:
        return [(N[i], A[i], Q[i]) for i in range(len(Q))]

print(generate_pocklington_prime(2, 3, 2**38, 2**40))
```

L'avantage de prendre un R petit, donc en commençant les tests avec $R = 2$, est d'être à peu près sûr de respecter la condition $q > \sqrt{n} - 1$, mais aussi de ne pas augmenter trop rapidement la valeur des nombres premiers que nous trouvons. A l'origine, nous voulions tester plusieurs valeurs de départ, comme $q = 3, 5, 7, 11, \ldots$, ou encore changer la valeur de a. Cependant, le test avec $a = 2$ et $q = 3$ nous a directement renvoyé un nombre véritablement premier entre 2^{38} et 2^{40}. La chaîne de certificats est :

1. 391577672677 est premier (a=2, q=32631472723) ;
2. 32631472723 est premier (a=2, q=5438578787) ;
3. 5438578787 est premier (a=2, q=2719289393) ;
4. 2719289393 est premier (a=2, q=169955587) ;
5. 169955587 est premier (a=2, q=9441977) ;
6. 9441977 est premier (a=2, q=1180247) ;
7. 1180247 est premier (a=2, q=590123) ;
8. 590123 est premier (a=2, q=22697) ;
9. 22697 est premier (a=2, q=2837) ;
10. 2837 est premier (a=2, q=709) ;
11. 709 est premier (a=2, q=59) ;
12. 59 est premier (a=2, q=29) ;
13. 29 est premier (a=2, q=7) ;
14. 7 est premier (a=2, q=3).

Nous choisirons donc $p = 391577672677$.

2.2 Recherche du plus petit nombre 7-lisse après p

Pour cela, il nous faut déjà un test pour vérifier qu'un nombre est 7-lisse. Pour ce faire, il est possible de diviser le nombre à examiner par $2, 3, 5, 7$ jusqu'à ce qu'il ne soit plus divisible par aucun de ces nombres premiers. Si le reste est de 1, alors le nombre est 7-lisse, sinon il ne l'est pas. L'algorithme Python est donc.

```
def is_7_smooth(n):
    """
    Teste si n est 7-lisse.
    """
    for p in [2, 3, 5, 7]:
        while n % p == 0:
```

```
            n //= p
    return n == 1
```

Puis, il suffit de partir de $p+1$ et d'incrémenter jusqu'à tomber sur un nombre 7-lisse.

```
def generate_7_smooth_number(p):
    """
    Genere le plus petit nombre 7-lisse strictement plus grand que p.
    """
    i = p + 1
    while not is_7_smooth(i):
        i += 1
    return i
```

Cela donne : 391820820480.