Nonce biaisés dans le chiffrement de Goldwasser Micali

A∼Z

July 28, 2022

1 Symbole de Jacobi

Étant donné un entier n et un nombre x, on désire savoir si x est un résidu quadratique mod n, ie s'il existe un entier a tel que $a^2 \equiv x \mod n$.
Dans le cas où n est premier, ce problème est résolu par ce qu'on appelle le symbole de Legendre.

Définition 1.1 (Symbole de Legendre)**.** Si p est premier impair, on appelle symbole de Legendre de x modulo p, noté $\left(\frac{x}{p}\right)$, la quantité

$$\left(\frac{x}{p}\right) = x^{(p-1)/2} \mod p.$$

On n'a pas besoin d'étudier le cas $p = 2$ puisque tout entier y est un carré.
Comme $\varphi(p) = p - 1$, $x^{(p-1)/2}$ est un nombre d'ordre au plus 2. En fait, ce nombre est égal à 1 si et seulement si x est un carré non nul (puisqu'alors il est d'ordre au plus $(p-1)/2$).

On veut généraliser le symbole de Legendre à des moduli non premiers. Pour cela, on remarque que si $n = \prod_i p_i^{\alpha_i}$, alors un carré mod n est un carré mod $p_i^{\alpha_i}$ pour tout i grâce au théorème des restes chinois. En particulier, si x est un carré mod n c'est un carré mod p_i.

Définition 1.2 (Symbole de Jacobi)**.** Si $n = \prod_i p_i^{\alpha_i}$ et x est un entier, on définit le symbole de Jacobi par

$$\left(\frac{x}{n}\right) = \prod_i \left(\frac{x}{p_i}\right)^{\alpha_i}.$$

Il n'y a bien sûr pas de conflit de notations avec le symbole de Legendre.

La discussion plus haut montre que si x est un carré mod n, alors $\left(\frac{x}{n}\right) = 1$. Ainsi, si $\left(\frac{x}{n}\right) = -1$ on sait que x n'est pas un résidu quadratique mod n. La réciproque est fausse: en effet, si x n'est pas un résidu quadratique modulo un nombre pair de diviseurs de n son symbole de Jacobi sera égal à 1. Grâce à la loi de réciprocité quadratique, on peut calculer efficacement le symbole de Jacobi, même sans connaître la factorisation de n.

2 Cryptosystème de Goldwasser-Micali

Comme tous les cryptosystèmes à clef publique, celui de Goldwasser-Micalli se base sur un problème considéré comme intractable; c'est à dire qu'on est incapable de le résoudre de manière efficace (en un temps sous-exponentiel en le nombre de bits du paramètre de sécurité) à l'heure actuelle.

Il s'agit ici du problème de la résiduosité quadratique: étant donné un entier n de factorisation inconnue et un nombre x de symbole de jacobi $\left(\frac{x}{n}\right) = 1$, peut-on déterminer si x est un résidu quadratique modulo n ?
Bien qu'il s'agisse d'un problème plus facile à résoudre que celui de la factorisation, il est considéré comme difficile.

2.1 Génération de clefs

Le paramètre de sécurité λ est le nombre de bits du modulus public n. On génère deux grands premiers p et q de $\lambda/2$ bits chacun, et on pose $n = pq$. p et q constituent la clé secrète.
On génère ensuite un entier x qui ne soit un résidu quadratique ni mod p ni mod q. n et x constituent la clé publique.

2.2 Chiffrement

Afin de chiffrer un message m, on va encoder un bit de m par l'information qu'un entier aléatoire est ou non un résidu quadratique mod n.
Pour ce faire, on écrit $m = \overline{b_1 b_2 \dots b_n}$ la décomposition en base 2 de m. Pour chaque i, on choisit un entier aléatoire y_i entre 0 et n, et on renvoie

$$c_i := y_i^2 x^{b_i}.$$

2.3 Déchiffrement

c_i est un résidu quadratique mod n si et seulement si $b_i = 0$. Connaissant la clé secrète (p, q), on obtient donc

$$b_i = \begin{cases} 0 \text{ si } \left(\frac{c_i}{p}\right) = 1 \\ 1 \text{ si } \left(\frac{c_i}{p}\right) = -1 \end{cases}$$

Bien que sémantiquement sécurisé, le cryptosystème de Goldwasser-Micali n'est que peu pratique à cause de la dilatation gigantesque des messages: le facteur d'expansion de la fonction de chiffrement est λ, où λ représente le paramètre de sécurité. Autrement dit, si un message m fait k bits, son chiffrement fait λk bits — c'est à dire beaucoup trop de bits.
NB: x est choisi tel qu'on ne puisse pas distinguer b_i par le symbole de jacobi. En effet,

$$\left(\frac{c_i}{n}\right) = \begin{cases} \left(\frac{y_i^2}{n}\right) = 1 & \text{si } b_i = 0 \\ \left(\frac{y_i^2 x}{n}\right) = \left(\frac{x}{n}\right) = \left(\frac{x}{p}\right)\left(\frac{x}{q}\right) = 1 & \text{si } b_i = 1. \end{cases}$$

3 Importance de la sécurité cryptographique de la source d'aléatoire

Il est vital que les *nonces* y_i soient choisis aléatoirement de manière uniforme, sans relation algébrique quelconque les liant. Considérons à titre d'exemple le challenge Small Fortune du DiceCTF@HOPE 2022, écrit par willwam845.

```
from Crypto.Util.number import *
from gmpy2 import legendre

flag = bytes_to_long(open("flag.txt", "rb").read())

p, q = getPrime(256), getPrime(256)
n = p * q

x = getRandomRange(0, n)
while legendre(x, p) != -1 or legendre(x, q) != -1:
    x = getRandomRange(0, n)

def gm_encrypt(msg, n, x):
    y = getRandomRange(0, n)
    enc = []
    while msg:
        bit = msg & 1
        msg >>= 1
        enc.append((pow(y, 2) * pow(x, bit)) % n)
        y += getRandomRange(1, 2**48)
    return enc

print("n =", n)
print("x =", x)
print("enc =", gm_encrypt(flag, n, x))
```

script.py

La sortie est donnée dans `output.txt`; le but est de récupérer la chaîne de caractère qu'est le flag.
On remarque tout de suite que $y_{i+1} = y_i + k_i$, où k_i est un nombre aléatoire choisi de manière non-uniforme dans $\mathbb{Z}/n\mathbb{Z}$. L'introduction de ce biais va se révéler suffisante pour récupérer les k_i, en déduire les y_i, et obtenir les bits b_i du flag.
Pour ce faire, nous allons utiliser la méthode de Coppersmith.

Théorème 3.1 (Coppersmith). *Soit $P \in \mathbb{Z}/n\mathbb{Z}[X]$ un polynôme de degré δ. On peut trouver en temps polynomial en $\max(\log N, 2^\delta)$ toutes les racines x de P telles que $|x| < N^{1/\delta}$.*

Une preuve est donnée dans [JM06].

Supposons que l'on connaisse les deux premiers bits du message, b_0 et b_1. On sait alors que $y_0^2 = c_0 x^{-b_0}$ et que $(y_0 + k_0)^2 = c_1 x^{-b_1}$.

```
R.<y, k> = Zmod(n)[]

bits = 1, 0 # bruteforced: small_roots below will only find solutions for the right values
p1 = y^2 - enc[0]/R(x)^bits[0]
p2 = (y + k)^2 - enc[1]/R(x)^bits[1]

# res_y(p1, p2) is a polynomial in k that is 0 iff p1(y,k) and p2(y,k) have a common root y
f = p1.sylvester_matrix(p2).det().univariate_polynomial()
k0 = min(f.small_roots(2^48))
y0 = -gcd(p1.univariate_polynomial(), p2.specialization({k:k0})).constant_coefficient()
```

Connaissant y_i, on obtient k_i et b_{i+1} en lançant l'algorithme de coppersmith sur $(y_i + k_i)^2 - c_{i+1}/x^{b_{i+1}}$: il n'y aura de solution que pour la bonne valeur de b_{i+1}. On réitère avec $y_{i+1} = y_i + k_i$.

```
R.<k> = Zmod(n)[]

y = y0
m = [bits[0]]
for ci in enc[1:]:
    for bi in range(2):
        f = (y + k)^2 - ci/R(x)^bi
        r = f.small_roots(2^48)
        if r:
            y += min(r)
            m.append(bi)
            break
    else:
        print('NO SOLUTION FOUND')
        break

m = sum(b << i for i, b in enumerate(m))
print(int(m).to_bytes(m.nbits()//8 + 1, 'big'))
```

References

[JM06] Ellen Jochemsz and Alexander May. "A Strategy for Finding Roots of Multivariate Polynomials with New Applications in Attacking RSA Variants". In: *Advances in Cryptology – ASIACRYPT 2006*. Ed. by Xuejia Lai and Kefei Chen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 267–282. ISBN: 978-3-540-49476-8.