

TP N°4

Cryptographie Asymétrique (Partie 01)

Implémentation, Optimisation RSA-CRT et Attaques sur le PLD

Exercice 0-1 : Euclide étendu et inversion modulaire

1. Ecrire une fonction `EuclidEtendu(a,b)` qui retourne u,v calculé à partir de l'algorithme de Euclide étendu tels que : $a.u + b.v = \text{pgcd}(a,b)$
2. En déduire une fonction `invMod(a,b)` qui retourne l'inverse de a modulo b (vérifier que $a \wedge b = 1$)
3. Retrouver le résultat de la question 2 via la fonction python : `pow(a,p,m)`

Exercice 0-2 : Théorèmes des restes chinois

Ecrire une fonction python `TRC(LR, LM)` qui prend en paramètres une liste des restes LR et une liste de modulo LM (tels que les modules $LM[i]$ doivent être premiers entre eux deux à deux) et qui résout le système : $x \equiv LR[i] \pmod{LM[i]}$

Exercice 1 : Mise en œuvre, Attaque et Optimisation du système RSA

Objectif : Manipuler les paramètres réels du cryptosystème RSA, comprendre la vulnérabilité des clés courtes et implémenter l'optimisation par le Théorème des Restes Chinois (TRC).

I. Génération, Chiffrement et Déchiffrement manuel

Pour cette partie, nous utilisons des valeurs pédagogiques (clés courtes) :

○ $p=9973, q=11251, e=65537$

1. Calcul des clés :
 - Calculez $n = p \times q$. (Vous devriez obtenir 112206223)
 - Calculez $\phi(n) = (p-1)(q-1)$.
 - Calculez l'exposant privé d ($e^{-1} [\phi(n)]$).
2. Opérations :
 - Soit le message $M = 132596$ Calculez le message chiffré $C = M^e \pmod{n}$.
 - Vérifiez le bon fonctionnement en calculant $M' = C^d \pmod{n}$. Vérifiez que $M'=M$.

II. Attaque par Force Brute et Pollard-Rho

Dans un scénario réel, l'attaquant ne possède que la clé publique (n,e) . Sur une clé d'une petite taille (environ 27 bits), la factorisation n'est pas difficile.

1. Écrivez une fonction qui prend en entrée $n = 112206223$ et retrouve p et q par force brute.

2. Faire le même travail via l'algorithme de Pollard-Rho :

```
fonction pollard_rho(n):  
    choisir x aléatoire  
    y ← x  
    d ← 1  
    tant que d = 1:  
        x ← (x2 + 1) mod n  
        y ← (y2 + 1) mod n  
        y ← (y2 + 1) mod n  
        d ← gcd(|x - y| , n)  
    si d = n:  
        échec  
    sinon  
        retourner d
```

3. En déduire la clé privée sachant que $e = 65537$

4. Comparer de temps d'exécutions des deux méthodes

III. Optimisation du déchiffrement par le TRC

Le déchiffrement classique $M = C^d [n]$ est lourd car d et n sont grands. L'optimisation RSA-CRT permet de diviser ce calcul en deux exponentiations plus petites en utilisant les facteurs p et q .

Algorithme à implémenter : Au lieu de calculer $M = C^d [n]$, suivez ces étapes :

1. Réduction :

- $dp = d \bmod (p-1)$
- $dq = d \bmod (q-1)$

2. Calculs partiels :

- $mp = C^{dp} \bmod (p)$
- $mq = C^{dq} \bmod (q)$

3. Reconstruction (Formule de Garner) :

- Calculez $q_{inv} = q^{-1} \bmod p$
- $M = mq + q \cdot (q_{inv} \cdot (mp - mq) [p])$

Valeurs pour vérification (avec $M=132596$ et les p,q de la partie 1) :

- Vous devez calculer dp et dq et trouver les mêmes résultats intermédiaires avant de recomposer M .

IV. Mesure de performance :

Soient deux nombres p et q de ~ 80 bits chacun (soit $n \approx 160$ bits).

- p : 821664027145220167960399
 - q : 1116629742743296920375667
1. Ecrire une fonction python qui implémente l'algorithme MILLER RABIN et qui valide la primalité de ces nombres (sinon, proposer d'autres nombres)

Pour générer des nombres premiers de longueur ~ 80 bits :

```
def generer_premier_80bits():  
    while True:  
        #  $2^{79} < n < 2^{80}$   
        n = random.getrandbits(80) | (1 << 79) # Force bit poids fort à 1  
        n |= 1 # force n à être impair  
        if TstPremier_RabinMiller(n):  
            return n
```

2. Comparez le temps d'exécution de 100 déchiffrements avec la méthode standard ($\text{pow}(C, d, n)$) et 100 déchiffrements avec votre fonction `dechiffrement_CRT`.
3. Calculez le gain de vitesse. (Un gain de 3 à 4 fois est attendu).

Exercice 2 : Analyse de Complexité (Baby-Step Giant-Step)

Pour montrer pourquoi le PLD est considéré comme "difficile", tentez une attaque générique :

1. Implémentez l'algorithme Baby-Step Giant-Step (BSGS) pour retrouver le secret d de Bob à partir de β, a et p .
2. Mesure de performance :
 - Testez l'algorithme sur $p = 23549, a=2, \beta = 13245$
 - Testez-le ensuite sur un nombre premier p de 30 bits, 40 bits ...
 - Observez l'augmentation du temps de calcul et de la consommation mémoire (stockage de la table "Baby-Steps").

Exercice 3 : Chiffrement ElGamal et Attaque sur l'Aléa (Nonce)

Objectif : Implémenter le cryptosystème ElGamal et démontrer mathématiquement pourquoi l'utilisation d'un même secret éphémère pour deux messages différents permet à un attaquant de casser la confidentialité sans résoudre le PLD.

I. Implémentation du Chiffrement ElGamal

Paramètres du système (Valeurs précises) : $p = 31847, a = 5$, Bob choisit (clé privée) $d = 789$

1. Génération de la clé publique : Calculez la clé publique de Bob $\beta = \alpha^d [p]$
2. Chiffrement (Alice) :

- Alice veut envoyer le message $M1 = 9876$.
- Elle choisit un nombre aléatoire (secret éphémère) $i = 888$.
- Calculez le couple $(c1, c2)$ tel que $c1 = \alpha^i [p]$ et $c2 = M1 \cdot \beta^i \pmod{p}$.

3. Déchiffrement (Bob) :

- Implémentez la fonction de déchiffrement : $M = c2 \cdot (c1^d)^{-1} [p]$
- Vérifiez que Bob retrouve bien $M1 = 9876$.

II. Simulation de l'Attaque (Réutilisation de i)

Scénario : Alice commet l'erreur d'utiliser le même secret éphémère $i = 888$ pour envoyer un second message $M2$ à Bob. Un attaquant (Ève) intercepte les deux transmissions.

Données interceptées par l'attaquant :

- Transmission 1 : ($c1 = 15682, c2 = 12754$) — Alice a envoyé $M1=9876$.
- Transmission 2 : ($c1' = 15682, c2' = 20560$) — Alice a envoyé un message $M2$ inconnu.

En observant $c1$ et $c1'$, Ève sait immédiatement que l'aléa i a été réutilisé.

Calcul de l'attaque :

- Sans chercher à calculer i ou d (ce qui serait difficile si p était grand), Ève peut utiliser la relation :
$$c2' \cdot c2 = M2 \cdot M1 \pmod{p}, \text{ c\`ad } M2 = M1 \cdot c2' \cdot (c2)^{-1} \pmod{p}$$
- Implémentez ce calcul en Python pour retrouver $M2$
Indice : Utilisez l'inverse modulaire de $c2$ modulo p .
- Réaliser un scénario qui illustre cette attaque