Rapport TP1 RS40

Alexandre BARTHELME

Mai 2023

Sommaire

1 Introduction

Le TP 1 consiste en la découverte des bases du chiffrement par RSA, de la mise en place des algorithmes vu en cours, de l'application par un programme python et par exemple des notions de chiffrement, déchiffrement et signature.

Nous voyons aussi dans le TP la notion de bourrage et le théorème du reste chinois

L'ensemble du code, du sujet et des annexes sont disponible sur mon GitHub : en cliquant ici.

2 L'exponentiation modulaire

2.1 Algorithme et définition

La premère partie du TP consiste en la création et l'amélioration de certaines fonctions du code initial. La fonction $home_mod_expnoent(x, y, n)$ permet qui permet de réaliser l'exponentiation modulaire. L'exponentiation modulaire calcule $x^y mod(n)$, il s'agit d'un algorihme rapide pour calculer de grande valeur en exposant. Cette algorithme converti en base 2 l'exposant et en fonction de la valeur du bit dans qui parcours tout l'exposant en binaire, on réalise plusieurs opérations qui manipule x, la valeur de base et n le modulo. Pour réaliser le code, je me suis basé sur l'algorithme fourni en cours :

Algorithm 1 Exponentiation modulaire simple

Require: x : base, n : modulo, y : exposant
Ensure: $x^y mod(n)$
R1 ← 1
R2 ← x
while $y > 0$ **do**
  if $ymod2 == 1$ **then**
    R1 = (R1*R2)mod(n)
  end if
  R2 = (R2*R2)mod(n)
  y= y//2
end while
Return R1

2.2 Programme

Avec cette algorithme, voici le code associé et commenté :

```
def home_mod_expnoent(x,y,n): #exponentiation modulaire
    """
    param x: base
    param y: exposant
    param n: modulo

    return: L'expoentiation modulaire de x^y modulo n
    """
    R1 = 1
    R2 = x
    while(y>0):             #tant que y est positif
        if (y%2==1):        #si le bit est a 1
            R1 = R1*R2
            R1 = R1%n
        R2 = R2*R2
        R2 = R2%n
        y = y//2          #on decale d'un bit
    return R1
```

Il est possible de l'essayer avec un exemple de TD en calculant $100^{11} mod(319) = 265$. Via ce code, on obtient bien 265.

3 Algorihme d'euclide généralisé

L'algorithme d'Euclide généralisé est un algorithme qui permet de calculer $d = pgcd(a, b)$, ainsi que 2 entiers u et v tels que $au + bv = d$.
Grâce à cette algorithme, nous pouvons obtenir d, la clé privé secrète en se basant sur le pgcd et les coefficients de Bézou. Pour le code relatif à cette algorithme, il faut calculer le PGCD et le coefficient v de Bézou.

```
def home_ext_euclide(a, b): #recherche du pgcd et de la cle
secrete
    """
    param a: un nbr entier
    param b: un autre nombre entier

    return: la cle secrete d
    """
    save_a=a        #sauvegarde de a pour le calcul de la cle
secrete
    quotient=a//b      #quotient et reste pour le calcul du
pgcd
    reste=a%b
    i=0
    v=[0,1]
    #v0=0 et v1=1 ,variables par defaut pour le calcul de
Bezout
    while reste!=0:
        i=i+1
        if i>=1:
             #calcul de la relation de bezout v
            v.append(v[i-1]-quotient*v[i])
        a=b
        b=reste
        quotient=a//b
        reste=a%b

    return v[-1]%save_a
    #retourne la cle secrete d
    #(valeur de v pour le dernier reste non nul modulo a)
```

Maintenant, via ce programme nous pouvons obtenir la clé secrète d en utilisant l'exposant public (clef publique) et ϕ (obtenu avec un calcul entre p et q)

4 RSA

4.1 Hashage

Par défaut dans le code fourni, le hashage du message utilise la fonction MD5. Elle s'avère relativement basique et ne permet pas une sécurité optimale. Par conséquent il est nécessaire de la remplacer par SHA - 256, algorithme de Hashage très reconnu.

Pour remplacer MD5 par SHA-256 on doit modifier ces deux lignes de code :

```
 Bhachis0=hashlib.md5(secret.encode(encoding='UTF-8',
errors='strict')).digest() #MD5 du message

 #remplacer par :

 Bhachis0=hashlib.sha256(secret.encode(encoding='UTF-8',
errors='strict')).digest() #sha256 du message
```

Cependant, malgré cette modification, le code ne fonctionnera pas. En effet, les deux nombres premiers p et q (les éléments de la clé) sont trop petits pour être utiliser avec SHA 256. Il est donc nécessaire de générer deux nouveaux nombres premiers.
Pour ma part, j'ai pris deux nombres suffisamment grand pour repousser la limite de caractère (qui était de 10 à l'origine) à environ 80 caractères.
Bien entendu, plus p et q seraient grand, plus le message peut être long, mais pour l'exemple 80 caractères est amplement suffisant.

4.2 Théorème du reste Chinois

4.3 Algorithme

Outre le problème de sécurité lorsqu'on utilise exponentiation modulaire, le calcul est extrêmement long et complexe. Il est donc intéressant d'utiliser le théorème du reste chinois qui va alléger le code et la complexité.
Le théorème peut être résumé comme l'algorithme ci-dessous

Algorithm 2 Théorème du reste chinois

Require: p,q : Nombre premiers,
msgc : message crypté,
n : exposant modulaire,
d : exposant de la clé (d)

Ensure: $m = c^d mod(n)$

if $p > q$ **then**
  Inverser p et q
end if
Calculer $d_p = dmod(p-1)$ et $d_q = dmod(q-1)$
Calculer q_1 avec l'exponentation modulaire
Calculer m_p et m_q avec l'exponentation modulaire
Déterminer h puis m le message final
Renvoyer le message

4.4 Code

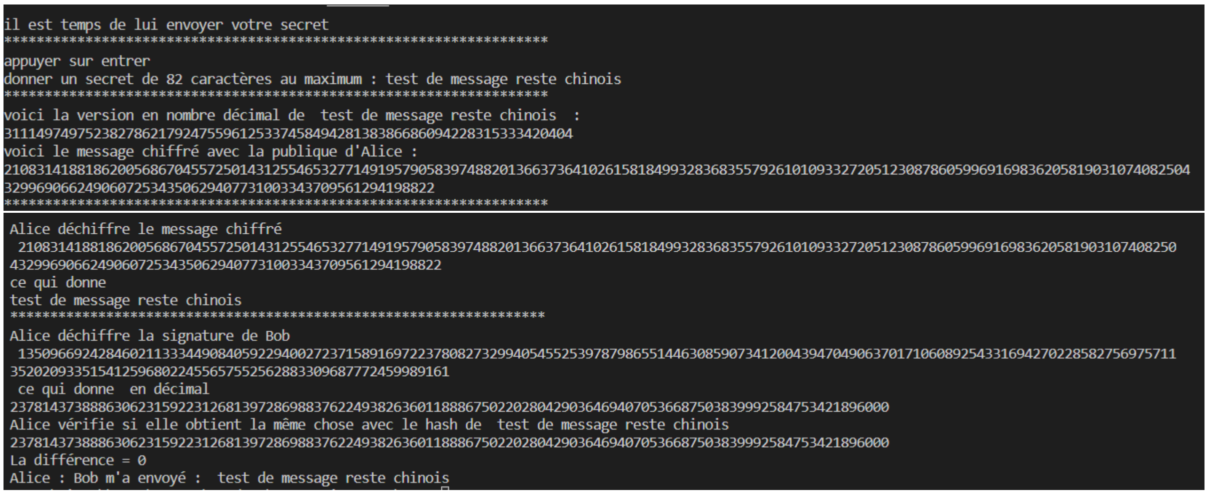
Voici le code associé à cet algorithme pour tout le théorème :

```
def home_crt(p,q,msgc,nmod,d): #theoreme reste chinois
    if (p>q) :          # p doit etre > q
        p,q = q,p       #sinon on inverse

    #calcul de dp,dq,q1,mp,mq,h puis le message m :
    dp= d%(p-1)
    dq= d%(q-1)
    q1= home_ext_euclide(q,p)
    mp= home_mod_expnoent(msgc,dp,p)
    mq= home_mod_expnoent(msgc,dq,q)
    h= (q1*(mp-mq))%p
    m= (mq+h*q)%nmod

    return m
```

Maintenant il est possible d'utiliser cette fonction pour le déchiffrement :

```
    #Remplacer ca :
    dechif=home_int_to_string(home_mod_expnoent(chif, da, na)
   )
    #Par ca :
    dechif=home_int_to_string(home_crt(x1a,x2a,chif,na,da))
```

A partir de là, le fichier est utilisable, on peut tester de chiffrer un message



et de voir s'il est bien déchiffrer et cohérent avec ce qui a été donné :

5 Découpage par blocs

Cependant, il y a encore un autre moyen de réaliser le chiffrement et déchiffrement qui permettrait de ne plus être limité par la taille de la clé. Pour chiffrer et déchiffrer des messages plus grand tout en gardant la même taille de clé, il faut découper le message initial en "bloc". Ces blocs de taille fixe, seraient "bourrés" s'ils ne peuvent pas être complet.

Pour ce faire, il faut déjà découper le message en bloc, via une fonction qui créer des blocs de taille 10 (défini personnellement), qui sont des tableaux contenant les caractères du mot. Chaque bloc ne peut contenir plus de 50% du message (variable j ci dessous) Création de bloc :

```
 def home_create_block(msg): #pour creer les blocs de 10
caracteres
 """
 param msg : message a decouper en blocs de 10 caracteres

 return : blocs de 10 caracteres
 """
 j= k//2
 i=0
 msgblock=[]
 while (i<len(msg)):
```

```
        msgblock.append(msg[i:i+j])
        i=i+j
    return msgblock
```

msgbloc contient maintenant les blocs du message initiale.
Maintenant, nous souhaitons ajouter du bourrage dans chaque bloc, avec précisément ces valeurs avant le bloc de message : 00,02 puis 3 caractères aléatoires, suivi du message fini par 00.

Pour ce faire on va récupérer chaque bloc, générer les caractères aléatoires et insérer dans le bloc en question en ajoutant les valeurs 00,02 et 00 .

```
def home_bourrage(msgblock): #pour bourrer les blocs de 10 caracteres
    """
    param msgblock : blocs de 10 caracteres
    return : blocs de 10 caracteres bourres
    """

    for msg in msgblock: #pour chaque bloc

        alea=''
        for i in range(k-len(msg)-3):
            alea=alea+chr(random.randint(0,255)) #on ajoute des caracteres aleatoires

        n_msg= chr(0)+chr(2)+alea+chr(0)+msg #on ajoute le bourrage
        msgblock[msgblock.index(msg)]=n_msg #on remplace le bloc par le bloc bourre

    return msgblock
```

Chaque *msgblock* contient maintenant un morceau de message initial et le bourrage rajouté.
Enfin, pour récupérer le message initial on fait l'opération inverse de façon à commencer par la fin du bloc pour accéder le plus rapidement au morceau de message qui nous intéresse :

```python
 def home_deblock(msgblock): #pour debourrer les blocs de 10 caracteres

 for msg in msgblock:        #pour chaque bloc
     for i in range(len(msg),0,-1): #on parcourt le bloc de la fin au debut
         if msg[i-1]==chr(0): #si on trouve un caractere nul
             msgblock[msgblock.index(msg)]=msg[i:] #on remplace le bloc par le bloc debourre
             break #on sort de la boucle
 return ''.join(msgblock) #on retourne le message debourre
```

Pour que le tout soit fonctionnel, il faut que le message qu'entre l'utilisateur appel la fonction de création et de bourrage de bloc (de même pour le passage en décimal) :

```python
block = home_bourrage(home_create_block(secret)) #creation des blocs de 10 caracteres a partir du secret (bourrage inclut)

num_sec=[]  #liste des blocs en nombre decimal
for msg in block:   #transformation des blocs en nombre decimal
    num_sec.append(home_string_to_int(msg)) #ajout du bloc en nombre decimal dans la liste
print(num_sec)
```

Enfin, pour le chiffrement, on utilise l'exponentiation modulaire pour chaque bloc :

```python
chif=[]
for a_num_sec in num_sec:
    chif.append(home_mod_expnoent(a_num_sec, ea, na))
print(chif)
```

Pour le déchiffrement, on réalise exactement l'inverse

```python
dechif=[]
for a_chif in chif:
    dechif.append(home_int_to_string(home_mod_expnoent(a_chif, da, na)))
print(dechif)
dechif=home_deblock(dechif)
print("Alice debloque le message \n",dechif)
```

A partir de là le chiffrement et déchiffrement par bloc est fonctionnel et peuvent être testé avec des messages plus long.

6 Conclusion

Ce TP présente les bases du fonctions des différentes méthodes et algorithmes de chiffrements et de hashages. Il est très intéressant pour mettre en pratique ce qui a été vu en cours et qui pouvait paraître très flou au début. Le fait de mettre en pratique cette théorie avec Python rend le TP accessible malgré la complexité des fonctions demandées. Le TP est un défi à aborder mais satisfaisant au final.