TP : RSA

Esteban Becker

14/04/2023

Table des matières

Objectif

Ce TP est à but uniquement pédagogique, plusieurs faiblesses de sécurité sont à garder en tête. L'objectif du TP est d'implémenter l'algorithme RSA en utilisant les notions vu en CM et en TD. Dans le rendu, 3 codes sont présents. Un avec les fonctions de base, un second avec le reste chinois et un dernier gérant les messages de taille supérieure à n avec les blocs.

1 Exponentiel modulaire

Dans cette partie nous allons implémenter la fonction d'exponentiel modulaire qui prend en paramètre un nombre, son exposant et le module.

```
    def home_mod_expnoent(x,y,n): #exponentiation modulaire
    """
    param x: int le nombre a elever
    param y: int l'exposant
    param n: int le modulo

    return: int le resultat de x^y mod n
    """

    #Nous allons utiliser l'algorithme d'exponentiation modulaire rapide

    carre=x

    resultat=1

    while y>0:

        #afin de ne pas utiliser de la memoire inutilement nous allons a chaque iteration calculer si y est pair ou impair puis le diviser par 2
        if y%2==1:

            resultat=(resultat*carre) % n

        carre = (carre*carre) % n

        y=y//2

    return resultat
```

Ici on se sert de la décomposition binaire de l'exposant. Et a chaque étape on met au carré la variable carre et on prend le reste de la division par n. Ceci permet de limiter la taille des nombres à manipuler. Ensuite, à chaque étape on divise par 2 l'exposant, et s'il est impair on multiplie le résultat par le carré. Au lieu de décomposer le nombre y en binaire et de le stocker en mémoire, nous le décomposons en parallèle de l'exécution pour limiter l'empreinte mémoire de la fonction.

2 Algorithme d'Euclide étendu

Pour l'algorithme d'Euclide étendu, nous allons juste reprendre l'algorithme vu en TD. La fonction prend deux nombres entiers. De plus, de par le fonctionnement de RSA, y est supérieur à b, du coup, il nous suffit de calculer la suite v.

```
    def home_ext_euclide(y,b): #algorithme d'euclide etendu pour la recherche de l'exposant secret
    """
```

```
    param y: int le premier nombre entier superieur a b
    param b: int le deuxieme nombre entier

    return: int l'exposant secret'
    """

    #Ici nous allons calculer en parallele le pgcd de y et b et la relation de Bezout

    dividend=y
    diviseur=b
    quotient=y//b
    reste=y%b

    #initialisation des variables
    v=[0,1]

    i=0

    while reste!=0:

        i=i+1

        #Il ne faut pas calculer la relation de bezout pour le premier quotient, en effet v0 v1 sont initialises.
        if i>=1:
            v.append(v[i-1]-quotient*v[i])

        dividend=diviseur
        diviseur=reste
        quotient=dividend//diviseur
        reste=dividend%diviseur

    return v[-1]%y
```

3 Amélioration

3.1 Algorithme de hachage

La fonction MD5 étant une fonction faible, il faut la remplacer par une autre fonction. J'ai choisi de la remplacer par la fonction sha-256. Cette fonction est plus robuste que MD5 et est utilisée dans de nombreux protocoles de sécurité. Cependant, la taille du hash généré est de 256 bits, ce qui est trop grand par rapport à notre clé, en effet il faut respecter la condition suivante : $M < n$ en sachant que $n = p * q$. Pour résoudre cela nous allons augmenter la taille de p et q à 100 nombres.

La nouvelle limite de taille de message est donc de 65 caractères.

3.2 Algorithme du reste chinois

L'algorithme du reste chinois permet de réduire le temps de calcul de la fonction d'exponentiel modulaire. En effet, il permet de calculer le résultat de la fonction modulo p et modulo q puis de calculer le résultat modulo n. Cela permet de réduire le temps de calcul de la fonction exponentielle modulaire. Il ne peut être utilisé que pour le déchiffrement du message quand on connait les deux nombres premiers p et q.

```
    def home_CRT(c,q,p,d,n):
    """
```

```
    Fonction utilisant le reste chinois pour dechiffrer le message
    param c: int le message chiffre
    param q: int le premier facteur de n
    param p: int le deuxieme facteur de n
    param d: int l'exposant prive
    param n: int le modulo

    return: int le message dechiffre
    """

    if(q>p):
        q,p=p,q

    dp=d%(p-1)
    dq=d%(q-1)

    q_inv=home_ext_euclide(q,p)

    mp=home_mod_expnoent(c,dp,p)
    mq=home_mod_expnoent(c,dq,q)

    h=(q_inv*(mp-mq))%p

    m=(mq+h*q)%n

    return m
```

4 Blocking

Afin de pouvoir utiliser l'algorithme RSA avec des messages long, il faut découper le message en plusieurs blocs. Pour effectuer cela, j'ai défini la taille de chaque bloc à 20 [1].

L'algorithme de bourrage donné dans le tp fonctionnant avec des bytes, il faut convertir le message en bytes :

```
    num_sec=num_sec.to_bytes((num_sec.bit_length() + 7) // 8, 'little')
```

Ensuite il faut découper le message en messages de taille j avec $j \leq k/2$:

```
    j=k//2
    for i in range(0,len(num_sec),j):
        m_bloc.append(num_sec[i:i+j])
    print(m_bloc)
```

Maintenant, il faut bourrer chaque bloc :

```
    for i in range(len(m_bloc)):
        j=len(m_bloc[i])
        x=random.randbytes(k-j-3)
        m_bourrer.append(b'\x00\x02'+x+b'\x00'+m_bloc[i])
```

À partir de là il faut chiffrer chaque bloc un à un.

```
    for i in range(len(m_bourrer)):

        chif=home_mod_expnoent(int.from_bytes(m_bourrer[i],'little'),ea,na)
        m_chif.append(chif.to_bytes((chif.bit_length() + 7) // 8, 'little'))
```

1. Bien qu'avec les nombres premiers choisis nous pouvons avoir des blocs plus grands, la taille est volontairement limitée pour illustrer rapidement plusieurs blocs lors de l'exécution

Enfin il faut déchiffrer chacun des blocs un à un et les concaténer pour obtenir le message déchiffré :

```
    for i in range(len(m_chif)):
        dechif=home_CRT(int.from_bytes(m_chif[i],'little'),x1a,x2a,da,na)
        dechif=dechif.to_bytes((dechif.bit_length() + 7) // 8, 'little')

        i=len(dechif)

        while dechif[i-1]!=0:
            i=i-1

        message=message + "".join(dechif[i:].decode())
```

Pour la signature rien ne change, en effet sha-256 peut gérer des messages de toutes tailles.

5 Conclusion

Ce tp m'a permis de comprendre le fonctionnement de l'algorithme RSA et de son implémentation en python. J'ai pu voir les limites de cet algorithme et les améliorations possibles. J'ai pu voir aussi l'importance de la taille des nombres premiers utilisés pour la génération des clés, des limites de l'algorithme d'exponentiation modulaire. J'ai particulièrement apprécié ce tp car il met en pratique les notions vues en cours et permet de comprendre des concepts très théoriques.