RS40 - RSA Rapport

Flavian Theurel

Avril 2023

Table des matières

1 Introduction

Le TP a pour but d'illustrer le déploiement d'un ensemble de mécanismes cryptographiques pour sécuriser l'échange entre deux personnes, à savoir Alice et Bob. Bob envoie un message à Alice. Chaque individu dispose d'une paire de clés (publique, privée) pour le chiffrement RSA. Bob chiffre le message avec la clé publique d'Alice. Il envoie également la signature de l'empreinte numérique de son message avec sa clé privée. Pour finir Alice déchiffre le message reçu et vérifie la signature de Bob.

2 Fonctions utilitaires

2.1 Exponentielle modulaire

Cette fonction permet de réaliser l'exponentielle modulaire $x^y mod(n)$.

```
    def home_mod_exponent(x, y, n):  # exponentiation modulaire
        # x: le nombre a multiplier
        # y: la puissance
        # n: le modulo
        a = y
        r1 = 1
        r2 = x
        while a > 0:
            if a % 2 == 1:
                r1 = (r1 * r2) % n
            r2 = (r2 ** 2) % n
            a = a // 2
        return r1
```

Pour optimiser cette fonction, on utilise le modulo pour éviter de convertir le nombre y en binaire. En effet, on boucle sur la suite d'opération suivante :

— On teste si le bit considéré est égal à 1 ou non.
— Si oui, on assigne à r_1 la valeur $(r_1 \times r_2)mod(n)$.
— On assigne à r_2 la valeur $(r_2^2)mod(n)$.
— Pour finir, on garde la partie entière de la division de a par 2.

2.2 Algorithme d'Euclide étendu

Pour cette fonction, on utilise l'algorithme vu en cours. De plus, comme y est supérieur à b, il faut calculer la suite v. Dans notre cas, t correspondrait à v_0, nouvt à v_1, r et nouvr sont les deux premiers restes lorsque l'on déroule l'algorithme d'euclide.

```
    def home_ext_euclide(y, b):  # algorithme d'euclide etendu pour la recherche
    de l'exposant secret
        # y: le modulo
        (t, nouvt, r, nouvr) = (0, 1, y, b)
        while nouvr > 1:
            quotient = r // nouvr
            (r, nouvr) = (nouvr, (r - quotient * nouvr))
            (t, nouvt) = (nouvt, (t - quotient * nouvt))
        return nouvt % y
```

3 Améliorations

3.1 Fonction de hachage

La fonction MD5 étant faible, elle a été remplacée par la méthode SHA-256. En effet, SHA-256 est un standard fédéral de traitement de l'information (FIPS du NIST) depuis 2002. Elle produit un haché d'une longueur de 256 bits. Ainsi, la taille maximum du message est de 2^{64} bits. Cependant, il est nécessaire de changer les clés. p et q ont dorénavent une longueur de 100 chiffres. La taille maximale du message peut aussi être augmentée. Ici, je l'ai fixé arbitrairement à 65 caractères.

3.2 Théorème du reste chinois

Le théorème du reste chinois permet d'alléger le déchiffrement du message crypté. En effet, en permettant de calculer successivement le résultat du message modulo p puis q et pour finir $m = (m_q + h \times q) mod(n)$. Cela permet de réduire le temps de calcul. Il est toutefois bon de noter que le CRT ne peut être utilisé que pour le décodage du message lorsque l'on connait p et q.

```
    def home_crt(c, q, p, d, n):  # q < p
    # c: le message chiffre
    # q: le premier facteur de n
    # p: le deuxieme facteur de n
    # d: l'exposant prive
    # n: le modulo
    #
    # Retourne le message dechiffre

    # Verification q < p
    if q > p:
        q, p = p, q

    q_inv = home_ext_euclide(n, q)

    d_q = d % (q - 1)
    d_p = d % (p - 1)

    m_q = home_mod_exponent(c, d_q, q)
    m_p = home_mod_exponent(c, d_p, p)

    h = ((m_p - m_q) * q_inv) % p
    m = (m_q + h * q) % n

    return m
```

3.3 Découpage en blocs

3.3.1 Initialisation

La première étape est de définir la taille limite de chaque bloc notée k.

```
    k = 20  # limite arbitraire de la taille de chaque bloc
```

Il faut également penser à initialiser le générateur aléatoire qui servira pour la génération des valeurs du bourrage.

```
    #Initialisation du generateur aleatoire
    random.seed()
```

3.3.2 Conversion du secret en octets

On effectue la convertion en octets sur la valeur numérique de secret. *num_sec.bit_length()* retourne le nombre de bits nécessaires pour représenter *num_sec*. On ajoute 7 à cette valeur pour s'assurer que l'on arrondisse au nombre d'octets le plus proche. Ceci évite la troncature.

```
    # Convertir le secret en bytes
    bin_sec = num_sec.to_bytes((num_sec.bit_length() + 7) // 8, 'little')
```

3.3.3 Découpage en blocs

Ici, on ajoute à la liste *list_block* les blocs de message de taille $j = k//2$. En effet, il est demandé que le message ne contienne pas plus de 50% du message.

```
    # Decoupage en blocs
    j = k // 2
    list_block = []
    print("voici la version en blocs de " + str(j) + " octets")
    for i in range(0, len(bin_sec), j):
        list_block.append(bin_sec[i:i + j])
    print(list_block)
```

3.3.4 Bourrage des blocs

On générère $k - j - 3$ octets aléatoires non nuls. On constitue par la suite des blocs de la forme suivante : $00|02|x|00|m_i$.

```
    # Bourrage des blocs avec la forme suivante: 00|02|x|00|mi| ou x est un
    nombre aleatoire et mi est le message
    print("voici la version en bloc avec bourrage")
    list_fill = []
    for i in range(len(list_block)):
        j = len(list_block[i])
        x_rand = random.randbytes(k - j - 3)
        list_fill.append(b'\x00\x02' + x_rand + b'\x00' + list_block[i])
    print(list_block)
```

3.3.5 Chiffrement

On chiffre ensuite chaque bloc.

```
    list_chif = []

    print("voici le message chiffre avec la publique d'Alice : ")
    for i in range(len(list_fill)):
        chif = home_mod_exponent(int.from_bytes(list_fill[i], 'little'), ea, na)
        list_chif.append(chif.to_bytes((chif.bit_length() + 7) // 8, 'little'))
    print(list_chif)
```

3.3.6 Dechiffrement

On déchiffre le message chiffré reçu par bloc en utilisant le CRT.

```
    # On utilise le Theoreme du Reste Chinois pour dechiffrer le message de Bob
    # On dechiffre par bloc
    dechif = ""
    message = ""
    for i in range(len(list_chif)):
```

```
6         dechif = home_crt(int.from_bytes(list_chif[i], 'little'), x1a, x2a, da,
      na)
7         dechif = dechif.to_bytes((dechif.bit_length() + 7) // 8, 'little')
8
9         j = len(dechif)
10
11        while dechif[j - 1] != 0:
12            j = j - 1
13
14        message = message + "".join(dechif[j:].decode())
15    print(message)
16    dechif = message
```

L'implémentation du découpage en blocs ne modifie pas la technique de déchiffrement de la signature reçue.

De plus, grâce au découpage en blocs, la taille des messages n'est plus limitée.

4 Conclusion

Ce TP m'a permis de mieux comprendre le fonctionnement du RSA. Implémenter des améliorations possible était fort intéressant. Cependant, il faut garder un regard critique sur ces améliorations. En effet, ces dernières peuvent présenter des failles de sécurités. D'un point de vu global, j'ai aimé contribuer à l'implémentation d'une version simplifiée du RSA en mettant à profit les connaissances vues en cours.