RS40 - Réseaux et Cybersécurité niveau 1

BENEDUCI Marie

30/04/2023

Table des matières

1. Introduction

Ce document est un rapport d'analyse sur un TP réalisé en RS40 à l'UTBM. L'objectif de ce dernier est d'implémenter le déploiement d'un ensemble de mécanismes cryptographiques pour sécuriser l'échange entre deux parties : Alice et Bob. Le TP considère un message envoyé de Bob vers Alice, où chacun dispose d'une paire de clés (publique, privée) pour le chiffrement RSA. Bob chiffre le message avec la clé publique d'Alice. Il procède aussi à la signature de l'empreinte numérique du message avec sa clé privée. Alice reçoit le message le déchiffre et vérifie la signature de Bob.

Ce TP a été réalisé au travers d'un exercice guidé et à partir d'un code de base donné.

Vous trouverez dans ce rapport une explication et une analyse de chaque étape de programmation suivant le TP, suivi d'une description, explication et comparaison des différents programmes.

Le code source du projet ainsi que de ce rapport sont disponibles sur mon Github :

https://github.com/Mxrie2001/RS40-RSA

2. Implémentation des fonction manquantes

Dans un premier temps nous est demandé dans le tp d'impplementer 2 fonctions :

- home_mod_expnoent(x,y,n): La fonction qui permet de réaliser l'exponentiation modulaire x^y%n.
- home_ext_euclide(y,b) : La fonction permettant d'obtenir la clé secrète en utilisant l'algorithme d'Euclide étendu.

1. Exponentiation modulaire

Pour implémenter cette fonction, on se sert de l'algorithme en pseudo code vu en cours :

Algorithme 1 Calcul de $y = x^p \mod (n)$

Entrées: $n \geq 2, x > 0, p \geq 2$
Sortie: $y = x^p \mod (n)$

Début
$p = (d_{k-1}; d_{k-2}; \cdots; d_1; d_0)$ % Écriture de p en base 2
$R_1 \leftarrow 1$
$R_2 \leftarrow x$

Traitement
Pour $i = 0; \cdots; k-1$ **Faire**
  Si $d_i == 1$ **Alors**
    $R_1 \leftarrow R_1 \times R_2 \mod (n)$ % Calcul de la colonne 4 du tableau si le bit est 1
  Fin Si
  $R_2 \leftarrow R_2^2 \mod (n)$ % carré modulo n de la colonne 3 du tableau
Fin Pour

Notre objectif ici est de calculer rapidement une puissance modulo un nombre donné.

Voici le code :

```python
def home_mod_expnoent(x, y, n):  # exponentiation modulaire
    # Convertir y en binaire et stocker chaque bit dans une liste
    tab = []
    binaireY = bin(y) # convertir en binaire
    binaireY = binaireY[2::] # supprimer le '0b' au début

    # ajouter chaque bit de y à la liste tab
    for i in range(len(binaireY)):
        tab.append(binaireY[i])

    # renverser la liste tab pour lire les bits dans le bon ordre
    tab.reverse()

    # initialiser r1 et r2
    r1 = 1
    r2 = x

    # Parcourir les bits de y
    for i in range(0, len(tab)):
        if tab[i] == str(1): # si le bit est égal à 1
            r1 = (r1 * r2) % n # calculer r1
        r2 = (r2 ** 2) % n # calculer r2

    return r1 # renvoyer le résultat final
```

La fonction home_mod_expnoent prend en entrée trois paramètres : x, la base de la puissance, y, l'exposant et n, le modulo.

Dans un premier temps, la fonction convertit l'exposant y en binaire en utilisant la fonction bin(y) et en retirant les deux premiers caractères pour ne conserver que la représentation binaire. Les bits de l'exposant binaire sont ensuite stockés dans une liste tab pour être traités un à un.

Ensuite, la fonction utilise l'algorithme d'exponentiation modulaire pour calculer la puissance modulo n. Elle initialise deux variables r1 et r2 à 1 et x respectivement.

Puis, elle itère sur chaque bit de l'exposant y stocké dans la liste tab. Si le bit est égal à 1, la fonction multiplie la valeur de r1 par r2 modulo n. Enfin, elle calcule la valeur de r2 élevée au carré modulo n.

Finalement, la fonction retourne la valeur de r1 qui représente la valeur de x élevée à la puissance y modulo n.

A noter que cette méthode est efficace car elle ne nécessite pas de calculer la valeur de x élevée à la puissance y en entier avant d'appliquer la division modulo n. Cela permet de réduire considérablement le temps de calcul pour de grandes valeurs de y.

2. Algorithme d'Euclide étendu

Nous devons maintenant appliquer l'algorithme d'Euclide étendu afin d'obtenir la clef secrète par la suite.

Voici le code :

```python
def home_ext_euclide(y, b):
    # Initialisation de variables et listes
    q = []
    u = [1, 0]
    nouvr = y
    r = b

    # Boucle qui calcule le pgcd et les coefficients de Bézout
    while nouvr:
        # Effectue une division euclidienne pour obtenir le quotient et le reste
        quotient, nouvr, r = r // nouvr, r % nouvr, nouvr
        # Calcule les coefficients de Bézout
        u.append(u[-2] - quotient * u[-1])
        # Stocke le quotient
        q.append(quotient)

    # Calcul du résultat final
    return u[-2] % y
```

La fonction home_ext_euclide implémente l'algorithme d'Euclide étendu pour trouver l'inverse d'un élément b dans le corps modulaire y, c'est-à-dire, l'exposant secret permettant de résoudre l'équation b^x = c (mod y).

La fonction commence par initialiser les variables et listes nécessaires pour le calcul, à savoir q pour les quotients, u pour les coefficients de Bézout, et nouvr et r pour les valeurs à diviser.

Ensuite, la fonction entre dans une boucle qui calcule le pgcd et les coefficients de Bézout en alternance, jusqu'à ce que nouvr atteigne 0. Dans chaque itération de la boucle, la fonction calcule le quotient et le reste de la division euclidienne de r par nouvr, puis échange les valeurs de nouvr et r. Les coefficients de Bézout sont également mis à jour à chaque itération. Les valeurs de quotient sont stockées dans la liste q pour une utilisation ultérieure.

Finalement, la fonction calcule le résultat final, qui est le coefficient de Bézout u[-2] modulo `y`.

Les commentaires dans la fonction expliquent chaque étape en détails comme dans la fonction précedente.

3. Améliorations

1. Amélioration du processus de vérification de la signature de Bob

Le md5 étant une fonction faible, nous allons, afin d'améliorer le processus de vérification de la signature de Bob la remplacer par la fonction Sha-256.

Pour mieux comprendre, il faut savoir que MD5 (Message Digest 5) et SHA-256 (Secure Hash Algorithm 256 bits) sont 2 algorithmes de hachage cryptographiques largement utilisés pour sécuriser les données sensibles.

L'objectif principal de ces algorithmes est de prendre une entrée (message) de longueur variable et de la transformer en une sortie (digest) de longueur fixe. Cette sortie est censée être unique pour chaque message donné, de sorte qu'un petit changement dans le message entraîne une grande différence dans la sortie.

La principale différence entre MD5 et SHA-256 est la taille de leur sortie de hachage. MD5 produit une sortie de 128 bits, tandis que SHA-256 produit une sortie de 256 bits. Cela signifie que SHA-256 offre un niveau de sécurité plus élevé que MD5, car il est beaucoup plus difficile de trouver deux messages différents qui ont la même sortie de hachage SHA-256 que deux messages qui ont la même sortie de hachage MD5.

En outre, MD5 est considéré comme vulnérable aux attaques de collision, ce qui signifie que des messages différents peuvent avoir la même sortie de hachage MD5. Des attaques de collision réussies ont été réalisées sur MD5, ce qui a conduit à sa mise en garde. SHA-256, en revanche, est considéré comme résistant aux attaques de collision.

Ainsi dans le cadre de ce TP, nous avons dans un premier temps remplacer tous les md5() par sha256(). 2 lignes de code seront impactées :

```
print("On utilise la fonction de hashage SHA-256 pour obtenir le hash du message avec plus de
sécurité", secret)
Bhachis0 = hashlib.sha256(secret.encode(encoding='UTF-8', errors='strict')).digest()  # SHA-256 du
message
```

```
print("Alice vérifie si elle obtient la même chose avec le hash de ", dechif)
Ahachis0 = hashlib.sha256(dechif.encode(encoding='UTF-8', errors='strict')).digest()
```

Mais en lançant le code, une erreur… ça ne marche pas :

```
appuyer sur entrer
*******************************************************************
Alice déchiffre le message chiffré
 3197390600983130022589801363641349402332255684686107088479992
ce qui donne
coucou
*******************************************************************
Alice déchiffre la signature de Bob
 4898891740135611455789658393002033990744124037381998335392553 5
 ce qui donne  en décimal
8078874760564384163746408399836785350082529813661271013675327 1
Alice vérifie si elle obtient la même chose avec le hash de  coucou
23781725770691671994193429462436275602481784993520709009634127935033868910210669543518812088175589333142611008
La différence = 2378172577069167199419342946243627560248178499343992026202848409339640482621230169001798679003897662300585773 7
oups

Process finished with exit code 0
```

Ceci est normal car la taille du message est maintenant supérieure à la clé il faut donc augmenter la taille des clés. Pour cela nous utilisons le site web : https://bigprimes.org/ pour régénérer 4 nouvelles clés de 60 caractères.

```python
# voici les éléments de la clé d'Alice
# x1a = 2010942103422233250095259520183  # p
# x2a = 3503815992030544427564583819137  # q
x1a = 608374008321988961645676216446814912308108652191114995556897  # p nouvelle clef de 60 caracteres
pour sha256
x2a = 310862287259718118908416875730252948527603190886142270567957  # q nouvelle clef de 60 caracteres
pour sha256
na = x1a * x2a  # n
phia = ((x1a - 1) * (x2a - 1)) // home_pgcd(x1a - 1, x2a - 1)
ea = 17  # exposant public
da = home_ext_euclide(phia, ea)  # exposant privé
# voici les éléments de la clé de bob
# x1b = 9434659759111223227678316435911  # p
# x2b = 8842546075387759637728590482297  # q
x1b = 762807463949654769548656894998136037163904829497830908642209  # p nouvelle clef de 60 caracteres
pour sha256
x2b = 357925266421579046844087625375712068094818394193046496285147  # q nouvelle clef de 60 caracteres
pour sha256
nb = x1b * x2b  # n
phib = ((x1b - 1) * (x2b - 1)) // home_pgcd(x1b - 1, x2b - 1)
eb = 23  # exposants public
db = home_ext_euclide(phib, eb)  # exposant privé
```

Après cela le code marche de nouveau en le lançant.

2. Définition d’une nouvelle limite de caractère pour le message

La fonction de hachage ayant été changé, il est maintenant plus sécurisé d’envoyer des données. Après plusieurs recherches, nous pouvons dire que théoriquement la taille maximale de données à hacher avec la méthode SHA-256 est de 2^64-1 bits, soit 2^61-1 octets, soit environ 2.3 x 10^18 octets. Cela signifie qu'il est pratiquement impossible d'atteindre cette limite avec les technologies actuelles.

Dans le cadre général, pour une utilisation pratique, les données à hacher ne devraient pas dépasser quelques mégaoctets afin de garantir des temps de traitement raisonnables et une sécurité adéquate.

Notons que 1 mégaoctet (1 Mo) correspond à 1 048 576 octets. Le nombre de caractères dans 1 Mo dépend de l'encodage utilisé pour représenter ces caractères. Par exemple, si l'encodage est UTF-8, qui utilise généralement un octet pour représenter les caractères ASCII et jusqu'à 4 octets pour certains caractères non-ASCII, le nombre de caractères dans 1 Mo peut varier de 1 048 576 (pour des textes ne contenant que des caractères ASCII) à environ 262 144 (pour des textes contenant des caractères non-ASCII).

Pour ce TP, après plusieurs tests nous pouvons déduire que la taille maximale pour que le programme fonctionne est de 50 caractères. A noter que, la taille du message échangé entre Alice et Bob est limitée par la taille de la clé. Pour respecter cela, il faut modifier la fonction de limite de caractère et son appel ainsi :

```python
def mot50char():  # entrer le secret
    secret = input("donner un secret de 50 caractères au maximum : ")
    while (len(secret) > 51):
        secret = input("c'est beaucoup trop long, 50 caractères S.V.P : ")
    return (secret)
```

```python
x = input("appuyer sur entrer")
secret = mot50char()
```

3. Théorème du reste chinois

Dans la suite de ce TP, il nous faut modifier l'algorithme de RSA afin qu'il soit plus léger grâce au théorème du reste chinois. En nous appuyant sur le document fournit, nous pouvons coder la fonction utilisant le théorème chinois et son calcul préalable en suivant le pseudo code donné :

Algorithme de calcul $m = c^d \% n$ en utilisant CRT

Calcul préalable :

1- Avec $n = x_i x_j$ prendre $q = x_i$ et $p = x_j$ tel que $x_i < x_j$

2- Calculer q^{-1} dans $\mathbb{Z}_p$

3- Calculer $d_q = d\%(q\text{-}1)$ et $d_p = d\%(p\text{-}1)$

Ces calculs sont réalisés qu'une seule fois et les valeurs de q^{-1}, d_q et d_p sont gardées secrètement.

A la réception d'un message c, effectuer les opérations suivantes :

1- Calculer $m_q = c^{d_q}\%q$ et $m_p = c^{d_p}\%p$

2- Calculer $h = \left((m_p - m_q)q^{-1}\right)\%p$

3- Calculer $m = (m_q + h \times q)\%n$

Nous arrivons donc dans un premier temps à ce code pour le calcul préalable :

```python
def calculPréalable(xi, xj, d):
    # Initialisation de la variable n
    n = xi * xj

    # Si xi est inférieur à xj, q prend la valeur de xi et p prend la valeur de xj
    # Sinon, q prend la valeur de xj et p prend la valeur de xi
    if (xi < xj):
        q = xi
        p = xj
    else:
        q = xj
        p = xi
    # Appel de la fonction home_ext_euclide avec les arguments p et q
    qPrime = home_ext_euclide(p, q)

    # Calcul du reste de la division de d par q-1 et p-1, respectivement
    dq = d % (q - 1)
    dp = d % (p - 1)

    return (qPrime, dq, dp, q, p, n)
```

Et à celui-ci pour la suite, c'est-à-dire l'utilisation du théorème du reste chinois directement :

```python
def CRT(xi, xj, d, message):
    # Appel de la fonction calculPréalable pour déterminer les valeurs de qPrime, dq, dp, q, p et n
    (qPrime, dq, dp, q, p, n) = calculPréalable(xi, xj, d)

    # Calcul de mq et mp en utilisant la fonction home_mod_expnoent
    mq = home_mod_expnoent(message, dq, q)
    mp = home_mod_expnoent(message, dp, p)

    # Calcul de la variable h comme le reste de la division de ((mp-mq) multiplié par qPrime) par p
    h = ((mp - mq) * qPrime) % p

    # Calcul de la variable m comme le reste de la division de (mq plus h multiplié par q) par n
    m = (mq + h * q) % n

    return m
```

Pour ces 2 fonction, il s'agit d'une application directe du pseudo code.

Pour vérifier que cette méthode fonctionne bien, il nous faut l'appeler dans les résultats, et l'utiliser. Nous allons donc rajouter ces lignes pour comparer les 2 méthodes :

```python
print("voici la signature avec la clé privée de Bob du hachis")
signe = home_mod_expnoent(Bhachis3, db, nb)
print(signe)

print("voici la signature avec la clé privée de Bob du hachis avec le CRT")
signe2 = CRT(x1b, x2b, db, Bhachis3)
print(signe2)
```

```python
print("Alice déchiffre le message chiffré \n", chif, "\nce qui donne ")
dechif = home_int_to_string(home_mod_expnoent(chif, da, na))
print(dechif)

print("Déchiffrement par le CRT : ")
dechiffreCRT=CRT(x1a,x2a,da, chif)
print("Alice déchiffre  le message chiffré avec la clé de Bob \nCe qui donne : ")
print(home_int_to_string(dechiffreCRT))
```

```python
print("Alice déchiffre la signature de Bob \n", signe, "\n ce qui donne  en décimal")
designe = home_mod_expnoent(signe, eb, nb)
print(designe)

print("Alice déchiffre la signature CRT de Bob \n", signe2, "\n ce qui donne  en décimal")
designe2 = home_mod_expnoent(signe2, eb, nb)
print(designe2)
```

```python
print("La différence =", Ahachis3 - designe)
if (Ahachis3 - designe == 0):
    print("Alice : Bob m'a envoyé : ", dechif)
else:
    print("oups")

print("La différence pour le CRT =", Ahachis3 - designe2)
if (Ahachis3 - designe2 == 0):
    print("Alice : Bob m'a envoyé : ", dechif)
else:
    print("oups")
```

En exécutant le code, nous arrivons à ceci :

```
D:\UTBM\rs40\venv\Scripts\python.exe D:\UTBM\rs40\main.py
Vous êtes Bob, vous souhaitez envoyer un secret à Alice
voici votre clé publique que tout le monde a le droit de consulter
n = 273028064762549237576322909724747718817649424624692124090756463846977014021352348609587670320436644923038555256263969723
exposant : 23
voici votre précieux secret
d = 65289319834522643768251130586352715369437905888513334021702364657754631970932511185124788286861853221694858873205423175
******************************************************************
Voici aussi la clé publique d'Alice que tout le monde peut conslter
n = 189120535736336274040090719356905226986079606698141627139531464454971421246063257638980694904062265467767721267698549429
exposent : 17
******************************************************************
il est temps de lui envoyer votre secret
******************************************************************
appuyer sur entrer
donner un secret de 50 caractères au maximum : test ça marche tjrs?
******************************************************************
voici la version en nombre décimal de  test ça marche tjrs?  :
362240971284618683779959033786364107836689180020
voici le message chiffré avec la publique d'Alice :
171010006135638837204247919984384534356136158171831832257883575997121489718072905987188798085342812955983767694057260074
******************************************************************
On utilise la fonction de hashage SHA-256 pour obtenir le hash du message avec plus de sécurité test ça marche tjrs?
voici le hash en nombre décimal
23782013706520574306514066756512953708834594904145806157288294102994589631051771667011318696343954529481476160
voici la signature avec la clé privée de Bob du hachis
68747601128550014837845959839475475941955968426454144044625397720301911057958200600144608362048875345160422887388499817
voici la signature avec la clé privée de Bob du hachis avec le CRT
68747601128550014837845959839475475941955968426454144044625397720301911057958200600144608362048875345160422887388499817
```

```
*******************************************************************
Bob envoie
     1-le message chiffré avec la clé public d'Alice
 17101000613563883720424791998438453435613615817183183225788357599712148971807290598718879808534281295598376769405726 0074
     2-et le hash signé
 6874760112855001483784595983947547594195596842645414404462539772030191105795820060014460836204887534516042288738849981 7
*******************************************************************
appuyer sur entrer
*******************************************************************
Alice déchiffre le message chiffré
 171010006135638837204247919984384534356136158171831832257883575997121489718072905987188798085342812955983767694057260074
ce qui donne
test ça marche tjrs?
Déchiffrement par le CRT :
Alice déchiffre  le message chiffré avec la clé de Bob
Ce qui donne :
test ça marche tjrs?
*******************************************************************
Alice déchiffre la signature de Bob
 68747601128550014837845959839475475941955968426454144044625397720301911057958200600144608362048875345160422887388499817
 ce qui donne  en décimal
2378201370652057430651406675651295370883459490414580615728829410299458963105177166701131869634395452948147616 0
Alice déchiffre la signature CRT de Bob
 68747601128550014837845959839475475941955968426454144044625397720301911057958200600144608362048875345160422887388499817
 ce qui donne  en décimal
23782013706520574306514066756512953708834594904145806157288294102994589631051771667011318696343954529481476160
Alice vérifie si elle obtient la même chose avec le hash de  test ça marche tjrs?
23782013706520574306514066756512953708834594904145806157288294102994589631051771667011318696343954529481476160
La différence = 0
```

Le résultat final étant :

```
La différence = 0
Alice : Bob m'a envoyé :  test ça marche tjrs?
La différence pour le CRT = 0
Alice : Bob m'a envoyé :  test ça marche tjrs?

Process finished with exit code 0
```

Nous pouvons donc en déduire que tout marche.

4. Découpage du message par blocs et au bourrage

Pour cette partie, nous allons avoir besoin de plusieurs fonctions, que nous expliquerons au fur et à mesure :

- Dans un premier temps nous découpons le message en morceau d'une taille j aléatoire.

```
def créationBlocsMessage(message):
    # Initialisation de la liste de blocs et limite du k
    m = []

    if len(message) % 2 == 0:  # le message a une longueur
paire    limit = len(message) // 2
    else:  # le message a une longueur impaire
        limit = (len(message) - 1) // 2

    j = random.randint(2, limit)

    print( "\ntaille de j:",j,"\n")
    # Boucle pour couper le message
    while len(message) > j:
        mi = message[:j]
        m.append(mi)
        message = message[j:]

    # Ajout de la dernière sous-chaîne
    m.append(message)
    print("Messages : ", m)

    return m
```

Cette fonction prend en entrée une chaîne de caractères message et renvoie une liste de sous-chaînes de message, appelées "blocs de message". Elle utilise également la bibliothèque random de Python pour générer un nombre aléatoire j qui est utilisé pour découper la chaîne de caractères.

- Ensuite, nous allons transformer ces petits blocs de message en décimal pour la suite de l'exercice

```python
def blocDecimal(messages):
    messageDecimal = []
    for m in messages:
        messageDecimal.append(home_string_to_int(m))
    print("Version en nombre décimal des messages ", messageDecimal)
    return messageDecimal
```

Cette fonction appelle pour chaque bloc de string la fonction home_string_to_int implémentées plus haut qui renverra un tableau de decimal correspondant aux bouts de message.

- Il faut maintenant créer les blocs :

```python
def creationBloc(messages):
    blocks = []
    limitK = 60
    k = random.randint(5, limitK)
    print('taille de k :', k)
    for m in messages:
        sizeX = k-len(str(m)) - 4 # k − j − 3 octets sous forme de caracteres
        print('taille de x :', sizeX)

        nb_max = 10 ** sizeX - 1  # la valeur maximale de l'intervalle

        while True:
            x = secrets.randbelow(nb_max) + 1  # génère un nombre aléatoire dans l'intervalle (1, nb_max)
            if len(str(x)) == sizeX:  # vérifie que le nombre a x chiffres
                break

        print('x :', x)
        # Construire le bloc de la forme : 00||02||x||00||mi||
        block = '00' + '02' + str(x) + '00' + str(m)

        blocks.append(block)

    print('blocks :', blocks)
    return blocks
```

Cette fonction permet de créer des blocs de données à partir d'un ensemble de messages. Pour chaque message, la fonction génère un nombre aléatoire x qui sera utilisé pour remplir le bloc. Le bloc sera ensuite construit en concaténant différentes parties ensemble (données dans l'énoncé).

Plus précisément, la fonction commence par initialiser une liste vide appelée "blocks" qui contiendra tous les blocs de données générés. La variable limitK est initialisée à 60, ce qui sera la limite supérieure pour la taille des blocs. Elle devrait être limité à log2(n) mais je n'ai pas réussi à l'implémenter.

La boucle for itère sur chaque message passé en argument à la fonction. Pour chaque message, la fonction calcule la taille de la partie de remplissage du bloc sizeX en soustrayant la taille de la chaîne de caractères représentant le message len(str(m)) plus quatre octets supplémentaires. Ces quatre octets supplémentaires sont constitués de 00 et 02.

La variable nb_max est initialisée à la valeur maximale possible pour x. La fonction utilise alors une boucle while pour générer des valeurs aléatoires de x jusqu'à ce qu'un nombre de x correctement formaté soit obtenu. Le nombre x est obtenu en utilisant secrets.randbelow(nb_max) pour générer un nombre aléatoire dans l'intervalle (1, nb_max). La condition dans la boucle while vérifie que x a la bonne taille sizeX.

Le bloc final est créé en concaténant différentes parties : '00', '02', x, '00' et le message original m. Le bloc est ensuite ajouté à la liste "blocks".

Enfin, la fonction retourne la liste "blocks" contenant tous les blocs de données générés.

- Pour en finir avec l'envoie il faut crypter le message en utilisant le RSA

```python
def blocsRSA(blocks, ea, na):
    chiffblocks = []
    for m in blocks:
        chiff = home_mod_expnoent(int(m), ea, na)
        chiffblocks.append(chiff)

    print("voici les blocks message chiffré avec la publique d'Alice : \n", chiffblocks)
    return chiffblocks
```

Ici, la fonction appelée pour chaque élément de la liste block est home_mod_exponent (expliqué et implémentée plus haut dans le rapport). Cette dernière va donc chiffrer nos différents blocs et nous renvoyer une liste de blocks chiffrés.

- Passons maintenant au déchiffrage

```python
def blocsRSAinv(blockschiffree, da, na):
    dechiffblocks = []
    for m in blockschiffree:
        dechiff = home_mod_expnoent(m, da, na)
        dechiffblocks.append(dechiff)

    print("Alice déchiffre le bloc message chiffré rendu en decimal: \n", dechiffblocks)
    return dechiffblocks
```

Tout à l'inverse de précédemment nous allons maintenant récupérer la liste des blocs chiffrés et les déchiffrés avec la même fonction que précédemment à savoir home_mod_exponent mais avec des paramètres différents. Ce déchiffrement nous renverra la liste de blocs sous forme décimal.

- Pour finir, il nous faut retrouver le message initialement envoyé

```python
def dechiffrementRSAblocs(dechiffblocks):
    dechiffblocksMessage = []
    dechiffblocksMessageString = []
    for m in dechiffblocks:
        bloc_str = str(m)
        for i in range(len(bloc_str)):
            if bloc_str[i:i + 2] == '00':
                bloc_str = bloc_str[i + 2:]
                break
        dechiffblocksMessage.append(int(bloc_str))
    print("Alice obtient les message decimaux: \n", dechiffblocksMessage)

    for md in dechiffblocksMessage:
        blocstring = home_int_to_string(md)
        dechiffblocksMessageString.append(blocstring)
    print("Alice obtient les messages : \n", dechiffblocksMessageString)

    messageEnvoye = "".join(dechiffblocksMessageString)
    print("Pour finir, alice regroupe les differents blocs et recupere le message envoyé: \n",
messageEnvoye)

    return messageEnvoye
```

La fonction dechiffrementRSAblocs prend en entrée une liste de blocs chiffrés avec RSA, où chaque bloc est représenté sous forme d'un entier. La fonction a pour objectif de déchiffrer chaque bloc pour obtenir le message original, puis de regrouper les différents messages obtenus pour reconstituer le message envoyé.

En testant la méthode ici :

```python
print("*****************************************************************")
print("Méthode inspirée de PKCS#1v1.5 :\n" )
messagebloc = créationBlocsMessage(secret)
messageDecimalBloc = blocDecimal(messagebloc)
messageBlocksComplets = creationBloc(messageDecimalBloc)
messageBlocksCompletsChiffrees = blocsRSA(messageBlocksComplets, ea, na)
dechiffrageBlocks = blocsRSAinv(messageBlocksCompletsChiffrees, da, na)
dechiffrementRSAblocs(dechiffrageBlocks)
print("*****************************************************************")
```

Nous obtenons ce résultat :

```
Méthode inspirée de PKCS#1v1.5 :


taille de j: 2

Messages :  ['ça', ' m', 'ar', 'ch', 'e']
Version en nombre décimal des messages  [25063, 27936, 29281, 26723, 101]
taille de k : 10
taille de x : 1
x : 1
taille de x : 1
x : 3
taille de x : 1
x : 4
taille de x : 1
x : 8
taille de x : 3
x : 697
blocks : ['000210025063', '000230027936', '000240029281', '000280026723', '000269700101']
voici les blocks message chiffré avec la publique d'Alice :
 [33071144312863515348222891668544652150130459327047179358394773972501724683852312004393177398123034651200136585766566561, 46245734399259458643741084763137683184925765848827
Alice déchiffre le bloc message chiffré rendu en decimal:
 [210025063, 230027936, 240029281, 280026723, 269700101]
Alice obtient les message decimaux:
 [25063, 27936, 29281, 26723, 101]
Alice obtient les messages :
 ['ça', ' m', 'ar', 'ch', 'e']
Pour finir, alice regroupe les differents blocs et recupere le message envoyé:
 ça marche
*****************************************************************
```

Ce qui prouve bien que la méthode marche, le message initial est bien retrouvé !

4. Conclusion

Pour conclure, ce TP m’a permis de réaliser des fonctions illustrant des mécanismes cryptographiques pour sécuriser l’échange entre deux parties. Ce TP a été très intéressant pour moi, malgré le fait qu’il était assez flou dans un premier temps, après avoir fait plusieurs tests, j’ai pu mieux comprendre le principe afin d’appliquer les notions vues en cours et en TD.