RAPPORT TP1

RS40

(Réseaux et Cybersécurité)

Fait par : ***NOUTADIE DJOMGANG Willy Nelson***

Supervisé par :

M.ABBAS-TURKI Abdeljalil : Responsable UE

M.CHAH Badreddine : Responsable TP

Printemps 2025

SOMMAIRE

SOMMAIRE i
INTRODUCTION 1
Partie 1 : Implémentation de base du RSA 2
1. Fonction d'exponentiation modulaire rapide 2
2. Algorithme d'Euclide étendu 3
3. Fonction de hachage sha256 3
Part 2 : Attaque par canal auxiliaire et Optimisation via le CRT 4
1. Fonction home_mode_expnoent_trace 4
2. Optimisation avec le CRT 5
Part 3 : Bourrage 6
1. Les fonctions de bourrage 6
2. Observation du certificat de.springer.com 9

INTRODUCTION

Dans le cadre de notre unité d'enseignements RS40 intuitée Réseaux et Cybersécurités, nous avons réalisé notre premier TP qui pourtant sur le RSA. TP dans lequel nous avons mis en pratique les notions vues en cours accompagné de l'assistance de notre responsable de TP et aussi des documentations qui nous ont été fournies dans le cadre de ce TP. Notre rapport sera orienté sous 3 parties faisant ainsi références aux 3 parties de notre TP1.

Partie 1 : Implémentation de base du RSA

1. Fonction d'exponentiation modulaire rapide

Ceci a été faite grâce à la traduction du pseudo-code ci-dessous que nous avons vu en cours en langage python.

Pseudo-code: Exponentiation modulaire rapide

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

Nous avons ainsi défini la fonction ***home_mod_expnoent ()*** qui prend en paramètre les entiers **x, y, n** et calcul **(x^y) mod n**

```python
def home_mod_expnoent(x,y,n): #exponentiation modulaire
    R1 = 1
    R2 = x
    while y > 0:
        if y % 2 == 1:
            R1 = (R1 * R2) % n
        R2 = (R2 * R2) % n
        y = y // 2
    return R1
```

2. Algorithme d'Euclide étendu

```python
def home_ext_euclide(y,b): #algorithme d'euclide étendu pour la recherche de l'exposant secret
    (r, nouveau_r, t ,nouveau_t) = (y, b, 0, 1)
    while nouveau_r>1 :
        quotient = r//nouveau_r
        (r, nouveau_r) = (nouveau_r, r - quotient*nouveau_r)
        (t, nouveau_t) = (nouveau_t, t - quotient*nouveau_t)
    return nouveau_t%y
```

La fonction ***home_ext_euclide()*** sert à trouver l'inverse d'un nombre modulo un autre. Elle est utilisée pour calculer la clé privée RSA à partir de la clé publique.

r et ***nouveau_r*** : représentent les restes successifs lors de l'algorithme d'Euclide. On commence avec ***r = y et nouveau_r. = b***. À chaque étape, on met à jour ces valeurs pour descendre vers le PGCD.

t et nouveau_t : suivent les coefficients qui permettent de retrouver l'inverse modulaire. On commence avec ***t = 0 et nouveau_t = 1***. À chaque itération, ils sont mis à jour pour garder la trace de la combinaison linéaire.

quotient : c'est le résultat de la division entière de r par nouveau_r. Il sert à mettre à jour les restes et les coefficients.

À la fin, ***nouveau_t % y*** donne l'inverse modulaire de ***b modulo y***.

3. Fonction de hachage sha256

Dans l'intérêt d'augmenter la sécurité nous sommes passées du hachage en MD5 assez ancien en un hachage en sha256 qui est plus robuste et résistant aux attaques et nous avons aussi augmenter la longueur des nombres premiers de Alice et Bob

```python
print("On n'utilise plus la fonction de hashage MD5 pour obtenir le hash du message ",secret ," mais SHA256")
#Bhachis0=hashlib.md5(secret.encode(encoding='UTF-8',errors='strict')).digest() #MD5 du message
Bhachis0=hashlib.sha256(secret.encode(encoding='UTF-8',errors='strict')).digest() #sha256 du message
```

```python
print("Alice verifie si elle obtient la meme chose avec le hash de ",dechif)
#Ahachis0=hashlib.md5(dechif.encode(encoding='UTF-8',errors='strict')).digest()
Ahachis0=hashlib.sha256(dechif.encode(encoding='UTF-8',errors='strict')).digest() #sha256 du message
```

```python
#voici les elements de la cle d'Alice
x1a=2019621433216395452094094067048280932019396926270623159535551
#x1a=2010942103422233250095259520183 #p
x2a=3255555632959266166944547672834058475048441147380078493281 21
#x2a=350381599203054442756458381913 7 #q
na=x1a*x2a
phia=((x1a-1)*(x2a-1))//home_pgcd(x1a-1,x2a-1)
ea=17
da=home_ext_euclide(phia,ea)

#voici les elements de la cle de bob
x1b=765691443508358410243521431812663525934506973177182620044939
#x1b=9434659759111223227678316435911 #p
x2b=3288235531707810778065047865943942600756515948887927705172 43
#x2b=8842546075387759637728590482297 #q
nb=x1b*x2b
phib=((x1b-1)*(x2b-1))//home_pgcd(x1b-1,x2b-1)
eb=23
db=home_ext_euclide(phib,eb)
```

Part 2 : Attaque par canal auxiliaire et Optimisation via le CRT

1. Fonction home_mode_expnoent_trace

Tout comme notre fonction home_mode_exponoent faite précédemment, la fonction ***home_mode_expnoent_trace()*** réalise la même opération a la différence qu'elle affiche le déroulement de l'exponentiation modulaire. A chaque fois qu'une multiplication conditionnelle est effectuée (quand un bit de l'exposant est à 1), elle affiche un message indiquant le numéro de la multiplication. Cela permet de voir le nombre d'opérations réalisées lors du calcul de ***(x^y) % n***, ce qui est utile à des fins pédagogiques ou pour analyser l'efficacité de l'algorithme.

```
def home_mod_expnoent(x,y,n): #exponentiation modulaire
        R1 = 1
        R2 = x
        while y > 0:
            if y % 2 == 1:
                R1 = (R1 * R2) % n
            R2 = (R2 * R2) % n
            y = y // 2
        return R1

def home_mod_exponoent_trace(x,y,n): #exponentiation modulaire avec trace
        R1 = 1
        R2 = x
        count = 0 #compteur pour le nombre de multiplications coditionnelles
        while y > 0:
            if y % 2 == 1:
                R1 = (R1 * R2) % n
                count += 1
                print("multiplication  numero ",count)
            R2 = (R2 * R2) % n
            y = y // 2
        return R1
```

En observant le nombre de fois que le nombre de multiplication conditionnelle est affichée, un attaquant peut trouver les informations sur la clé privée, parce que chaque affichage correspond à un bit égal à 1.

Lors du chiffrage (respectivement déchiffrage) de notre message (converti en numérique), et la signature du message avec notre fonction, on constate que peu importe la longueur du message, le nombre d'itération reste constant

2. Optimisation avec le CRT

Nous avons modifié l'algorithme de RSA utilisé initialement dans le programme par une fonction qui implémente le théorème du reste chinois nommé ***home_reste_chinois()***. Ceci a été fait par la traduction de pseudo-code qui nous a été fournie dans le document Exp_CRT_RSA.pdf

```
def home_reste_chinois(c,d,p,q): # Theoreme des restes chinois
#calcul secret avec le theoreme des restes chinois
    n = p*q
    invq=home_ext_euclide(q,p)
    dq = d%(q-1)
    dp = d%(p-1)
#calcul  a la reception du message
    mp=home_mod_expnoent(c,dp,p)
    mq=home_mod_expnoent(c,dq,q)
    h = ((mq-mp)*invq)%p
    m=(mq+h*q)%n
    return m
```

La fonction prend en paramètre :

- **c** : le message chiffré (sous forme d'entier)
- **d** : l'exposant privé RSA
- **p, q** : les deux grands nombres premiers utilisés pour générer la clé RSA (facteurs du module n)

Elle déchiffre séparément modulo p et q, puis combine les résultats pour retrouver le message original m.

Le théorème du reste chinois réduit la surface d'attaque parce qu'il permet de réaliser le déchiffrement RSA en travaillant modulo des petits nombres (p et q) au lieu du grand module ceci dans le but de rendre l'opération plus rapide et, surtout, rend plus difficile l'observation des fuites d'informations

.

Part 3 : Bourrage

1. Les fonctions de bourrage

Nous avons défini 2 fonctions de bourrage, l'une utilisée par Bob pour chiffrer le message et la seconde utilise par Alice pour déchiffrer le message en retirant le bourrage.

Ces fonctions ont été faites en suivant la logique du pseudo-code ci-dessous

a. Définir une taille limite de chaque bloc que nous notons k
b. Bob : chaque bloc ne doit pas contenir plus de 50% du message que nous notons $m_i$et dont la taille en octets est j.
c. Ainsi, $m=m_1\|m_2\|m_3\|\ldots\|m_i\|\ldots\|m_n$ et $j \leq k/2$.
d. Bob : Générer $k-j-3$ octets non nuls (Bob utilise alea%255+1 pour chaque octet). Le nombre issu de cette génération est noté x.
e. Bob : Constituer le bloc de la forme suivante : $00\|02\|x\|00\|mi\|$, avec 00, 02 sont des octets valant 0 et 2 en hexadécimal.
f. Bob : Chiffrer le bloc avec RSA.
g. Alice : Déchiffrer le bloc avec RSA
h. Alice : Eliminer $00\|02\|x\|00$ de chaque bloc pour obtenir mi
i. Alice : Concaténer l'ensemble de mi pour constituer le message initial.

```python
def home_bourage_chif(message, e, n):
    m_int = home_string_to_int(message) #message en entier
    m_bytes = m_int.to_bytes((m_int.bit_length() + 7) // 8, 'big')
    k = (n.bit_length() + 7) // 8  #taille du bloc de message en oc
    jmax = k // 2  # maximum message block size

    #decoupage du message en blocs
    blocks = [m_bytes[i:i+jmax] for i in range(0, len(m_bytes), jmax)]
    encrypted_blocks = []

    for block in blocks:
        # calcul de la taille du bloc
        j = len(block)
        padding_length = k - j - 3
        x = bytes([random.randint(1, 255) for _ in range(padding_length)])
        padded_block = b'\x00\x02' + x + b'\x00' + block
        # converssion en entien des octets
        padded_int = int.from_bytes(padded_block, 'big')
        # chiffrement avec RSA des différents blocs
        encrypted_block = home_mod_expnoent(padded_int, e, n)
        encrypted_blocks.append(encrypted_block)
    return encrypted_blocks
```

La fonction ***home_bourage_chif()*** utilisée par Bob prends en paramètre :

- **message** : qui est le message à chiffrer
- **e** : la clé publique
- **n** : le modulus

La fonction va executer les étapes a -f du pseudo-code et va donc retourner un bloc qui est la fusion des blocs chiffré. C'est ce bloc qui sera déchiffrée pour retrouver le message

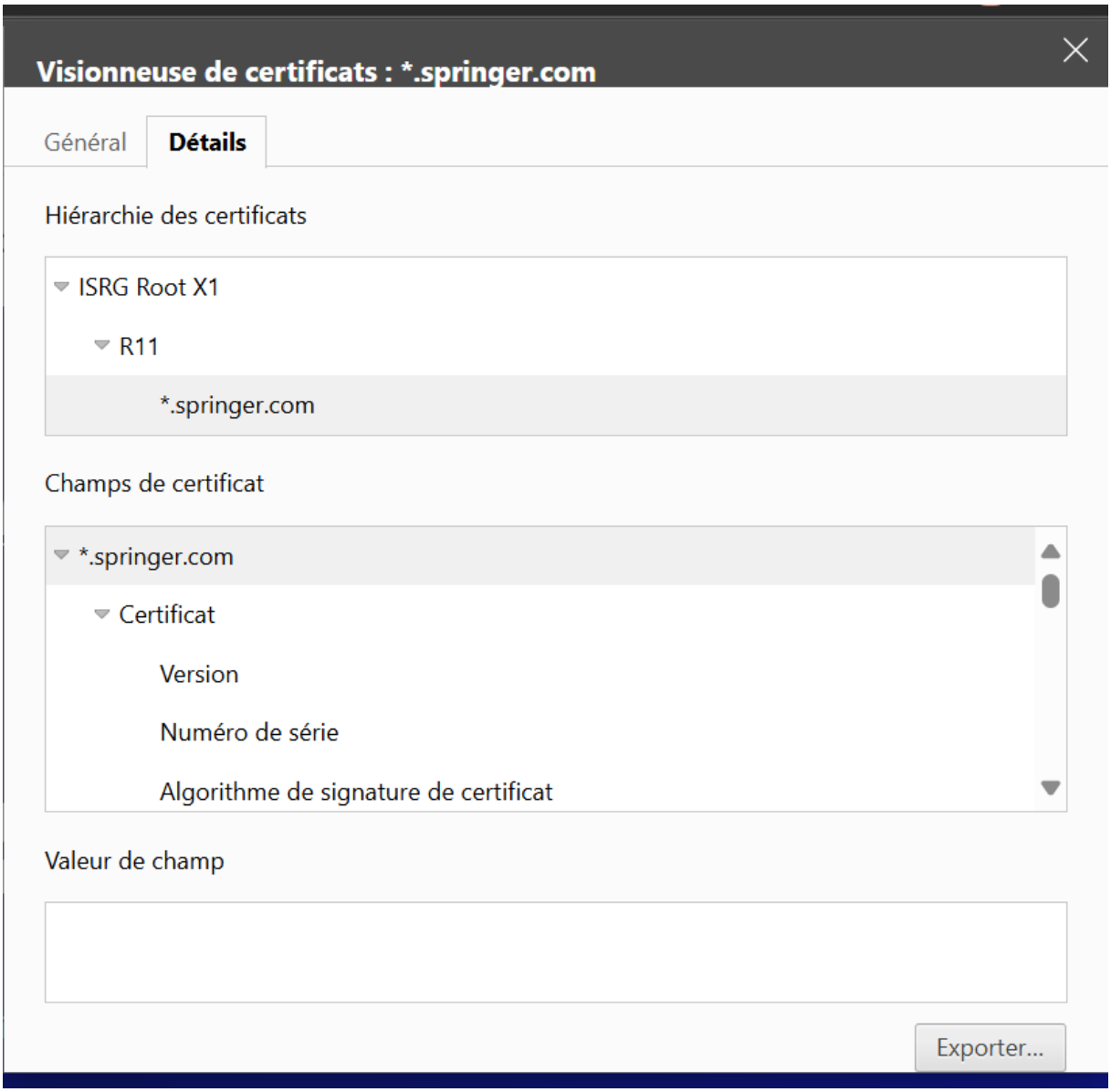
```python
def home_bourage_dechif(encrypted_blocks, d, n):
    decrypted_blocks = []
    k = (n.bit_length() + 7) // 8  # taille du bloc de message en octets

    for block in encrypted_blocks:
        # dechiffrement du bloc avec RA
        decrypted_int = home_mod_expnoent(block, d, n)
        decrypted_bytes = decrypted_int.to_bytes(k, 'big')

        #retire le bourage
        try:
            # trouve la position du separateur 00
            separator_pos = decrypted_bytes.index(b'\x00', 2)
            message_block = decrypted_bytes[separator_pos+1:]
            decrypted_blocks.append(message_block)
        except ValueError:
            raise ValueError("pas juste ceci")

    # fusionne les blocs decryptes
    full_message = b''.join(decrypted_blocks)

    # Convertit le message en entier
    full_message_int = int.from_bytes(full_message, 'big')
    return home_int_to_string(full_message_int)
```

La fonction ***home_bourage_dechif()*** utilisée par Alice prends en paramètre :

- le bloc chiffré qui est retourné à la fin du bourrage
- **d** : la clé privée
- **n** : le module

La fonction va exécuter les étapes restant du pseudo-code et retourne le message initial qui a été envoyé par Bob

2. Observation du certificat de.springer.com



a- L'algorithme de chiffrement, taille de la clé publique et valeur de l'exposant public

- Algorithme de chiffrement : Chiffrement RSA
- Taille de la clé publique : 2048 bits
- Exposant public : 01 00 01 (65537 en décimal)

b- L'algorithme de calcul de l'empreinte numérique

Il s'agit de PKCS #1 SHA-256 avec le chiffrement RSA

c- L'algorithme de bourrage

Il s'agit de **PKCS#1**