Rapport du Projet Cryptographie

Étude des attaques contre le partage de module RSA

"Une chaîne est aussi solide que son maillon le plus faible."

Membres du groupe :

Mohamed HAJJI Marwa DIALLO Mohammed ELBARAKA

Encadrant :

Pr. Pierre Vincent KOSELEFF

17 décembre 2024

Table des matières

Remerciements

Nous tenons à exprimer nos sincères remerciements à notre encadrant et professeur de cryptographie, Pr. Pierre Vincent KOSELEFF, pour nous avoir offert cette chance exceptionnelle de découvrir les subtilités et les secrets de cette discipline fascinante, ainsi que la robustesse des fondements mathématiques sur lesquels elle repose. Votre expertise, votre pédagogie et votre soutien constant ont grandement contribué à l'enrichissement de notre compréhension.

Nous adressons également nos remerciements à notre institution, EMINES - School of Industrial Management, qui nous offre des opportunités précieuses propices à notre épanouissement académique et personnel. Grâce à l'environnement stimulant et aux ressources mises à notre disposition, nous avons pu approfondir nos connaissances et développer nos compétences dans des domaines aussi divers que variés.

Ce rapport a été rédigé avec un profond intérêt et une passion sincère, alimentée par notre émerveillement continu face à la richesse de cette matière. Nous espérons qu'il reflète à la hauteur de notre engagement l'importance et la beauté de la cryptographie, ainsi que son impact majeur dans le monde moderne.

Chapitre 1

Introduction

Le cryptosystème RSA, introduit par Rivest, Shamir et Adleman en 1977, demeure aujourd'hui l'une des méthodes de chiffrement asymétrique les plus utilisées grâce à sa robustesse mathématique. Il repose sur la difficulté de factoriser de grands nombres premiers, garantissant ainsi la sécurité des communications. Toutefois, certaines faiblesses peuvent apparaître lorsque des choix inappropriés de paramètres sont effectués ou lorsque des stratégies d'implémentation inadéquates sont employées.

Le présent rapport expose l'une des attaques éventuelles sur le chiffrement RSA qui résulte du partage du même module entre plusieurs utilisateurs. En effet, le système devient vulnérable car, en possédant la valeur du triplet (n, e, d), un utilisateur peut remonter à la valeur de p et q, en déduire la valeur de $\phi(n)$ et par suite calculer les clés de déchiffrement de tous les autres utilisateurs ayant le même module n, en inversant leurs clés publiques dans $\mathbb{Z}/\phi(n)\mathbb{Z}$.

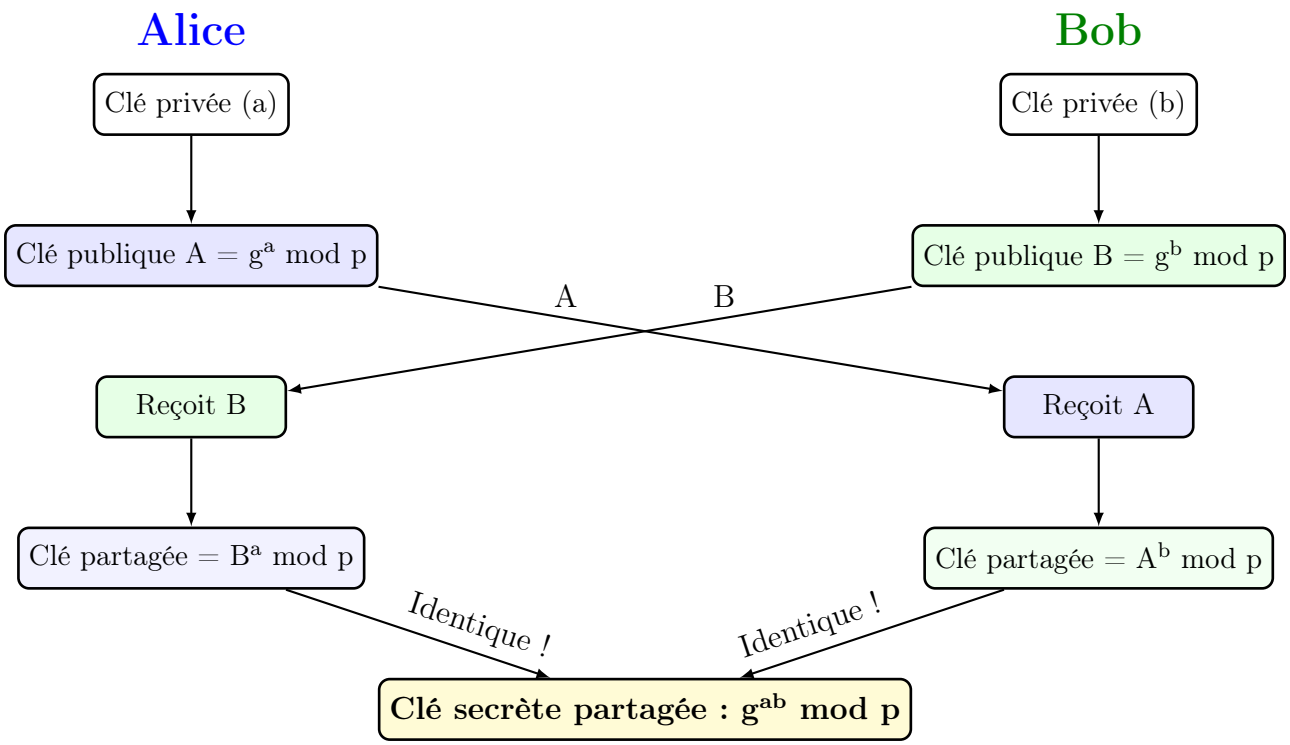
En se basant sur le texte B fourni qui avertit des risques liés à des choix trop ambitieux qui pourraient compromettre à la sécurité du système, l'objectif de cette étude est de mettre en lumière la théorie et les principes mathématiques sous-jacents aux attaques susceptibles de se produire contre RSA et qui constituent l'un des fondements de la cryptanalyse.

Chapitre 2

Principes d'échange de messages privés

2.1 Principe d'échange de messages avec partage de clés

2.1.1 Le chiffrement de Diffie-Hellman



Le chiffrement de Diffie-Hellman permet à deux personnes (**Alice** et **Bob**) de créer un code secret commun sans avoir à se rencontrer physiquement ou à échanger le secret

directement. Voici les étapes :

1. **Tout le monde connaît deux nombres publics :** un grand nombre premier (p) et un générateur (g).
2. **Chacun choisit un nombre secret :** Alice choisit a et Bob choisit b.
3. **Ils calculent chacun un nombre public :**
   — Alice calcule $A = g^a \mod p$.
   — Bob calcule $B = g^b \mod p$.
4. **Ils échangent leurs nombres publics :**
   — Alice envoie A à Bob.
   — Bob envoie B à Alice.
5. **Chacun calcule le code secret commun :**
   — Alice calcule $B^a \mod p$.
   — Bob calcule $A^b \mod p$.
   Grâce à des maths complexes, ils arrivent au même résultat, qui est leur **code secret commun**.

Même si quelqu'un intercepte les nombres publics A et B, il est **très difficile** de retrouver le code secret sans connaître les nombres secrets a et b.

2.2 Principe d'échange de messages sans partage de clés

Le schéma illustre le chiffrement asymétrique, où Alice et Bob utilisent chacun une paire de clés (publique et privée). Alice chiffre son message avec la clé publique de Bob (connue de tous), et seul Bob peut le déchiffrer avec sa clé privée (qu'il est le seul à posséder). L'échange inverse se produit pour la réponse de Bob. Ce système garantit la confidentialité car seul le destinataire possédant la clé privée correspondante peut lire le message chiffré avec la clé publique.

Il existe deux types de ce chiffrement :

2.2.1 Le chiffrement RSA

L'algorithme RSA a été inventé en 1977 par Ron Rivest, Adi Shamir et Leonard Adleman. Il repose sur un fondement mathématique discret dont la robustesse intrinsèque provient de la complexité algorithmique de la factorisation des entiers. Depuis son introduction, RSA a été largement utilisé dans les systèmes de sécurité, notamment pour les signatures numériques et les échanges confidentiels.

Formalisation mathématique : Soient $p, q \in \mathbb{N}$ deux nombres premiers distincts de cardinal suffisamment grand. On définit :

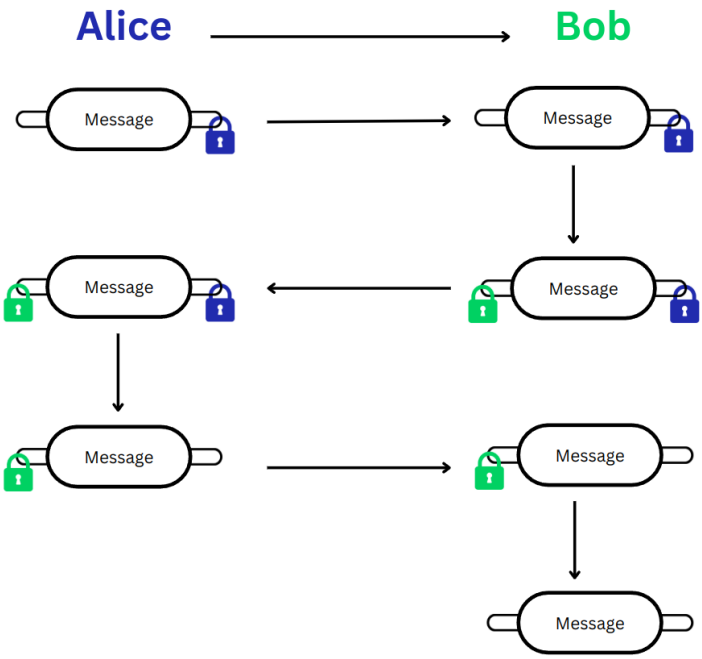
1. **Construction du module** : $n = p \times q$

2. **Calcul de la fonction d'Euler** : $\phi(n) = (p-1)(q-1)$
3. **Sélection de l'exposant public** : On choisit e inversible dans $\mathbb{Z}/\phi(n)\mathbb{Z}$, donc il faut et suffit que : $(e, \phi(n)) = 1$.
4. **Détermination de l'exposant privé** : $d \equiv e^{-1} \pmod{\phi(n)}$

(n, e) et $(d, \phi(n))$ sont intitulées respectivement clé publique et clé privée.
L'opération de chiffrement $\mathcal{E}$ et de déchiffrement $\mathcal{D}$ sont définies comme suit :

$$\mathcal{E}(m) = m^e \mod n$$
$$\mathcal{D}(c) = c^d \mod n$$

Grâce à ces principes mathématiques, Alice peut donc envoyer un message m à Bob en toute sécurité en calculant le message crypté $c \equiv m^e \pmod{n}$ à l'aide de la clé publique de Bob (n, e), qu'il peut ainsi déchiffrer en calculant $c^d \pmod{n}$ car $c^d \equiv m^{ed} \pmod{n} \equiv m \pmod{n}$.



2.2.2 Le chiffrement d'ELGAMAL

Le chiffrement d'ElGamal permet à Alice d'envoyer un message chiffré à Bob sans échange préalable de clé secrète. Voici les étapes :

1. **Bob choisit des paramètres publics :**
   — Bob choisit un grand nombre premier p et un générateur g d'un sous-groupe d'ordre élevé de $(\mathbb{Z}/p\mathbb{Z})^*$. p et g sont publics.
   — Bob choisit un nombre secret $b \in \{1, \ldots, p-1\}$, sa clé privée.
   — Bob calcule sa clé publique : $B \equiv g^b \pmod{p}$.
   — Bob publie sa clé publique B ainsi que les paramètres (p, g).

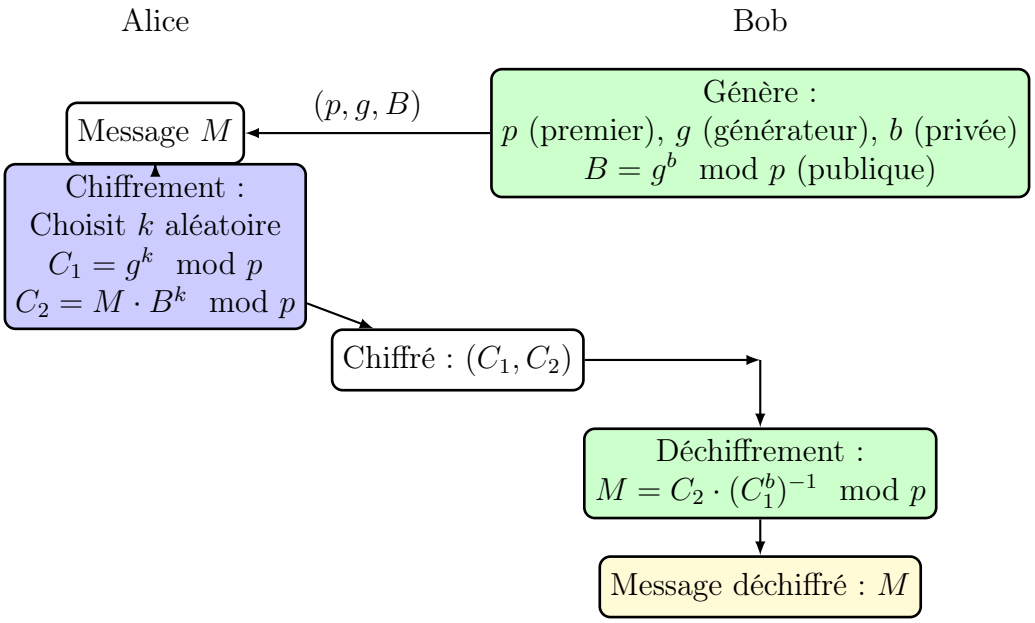
2. **Alice veut envoyer un message M à Bob :**
   — Alice récupère la clé publique de Bob (p, g, B).
   — Alice choisit un entier aléatoire secret $k \in \{1, \ldots, p-1\}$, appelé clé éphémère.
3. **Alice chiffre le message M :**
   — Alice calcule le chiffré (C_1, C_2) :

$$C_1 \equiv g^k \pmod{p}$$
$$C_2 \equiv M \cdot B^k \pmod{p}$$

4. **Alice envoie le chiffré à Bob :**
   — Alice envoie le chiffré (C_1, C_2) à Bob.
5. **Bob déchiffre le message :**
   — Bob utilise sa clé privée b pour déchiffrer :

$$M \equiv C_2 \cdot (C_1^b)^{-1} \pmod{p}$$
$$\equiv C_2 \cdot C_1^{-b} \pmod{p}$$

Sécurité : La sécurité repose sur la difficulté du problème du logarithme discret. Même en interceptant (p, g, B) et (C_1, C_2), il est calculatoirement difficile de retrouver M sans b ou k.



Chapitre 3

Partage de module RSA

3.1 Principe

Des institutions, telles que les banques et les administrations publiques, favorisent la transmission des clés à leurs utilisateurs pour optimiser leurs échanges cryptés sans avoir à mettre à jour leurs bases de données en fonction des clés choisies par chaque utilisateur. Etant donné la complexité de génération de clés, ces institutions sont souvent tentées de transmettre le même module n à plusieurs utilisateurs ce qui peut nuire à la sécurité du système. Sur ce, chaque utilisateur possède le triplet (n, e_u, d_u) où $n = pq$ est commun, alors que e_u et d_u sont spécifiques pour chacun d'entre eux. Cette option s'avère dangereuse d'après plusieurs propriétés que l'on va démontrer.

3.2 Complexité de génération de clés

La génération de clés RSA implique la recherche de grands nombres premiers puis leur multiplication. La complexité de cette tâche est liée au choix de l'algorithme de primalité. Plus le nombre est grand, plus il est difficile et coûteux en opération et en temps de vérifier qu'il est premier.

```
def find_large_prime (bits) :
    while True :
        p = ZZ(getrandbits (bits))
        if p.is_prime()==True:
            return p
%time prime = find_large_prime (2048)
print (prime)
```

Listing 3.1 – Simulation sur Sagemath pour trouver un nombre premier

CPU times : user 19.2 s, sys : 0 ns, total : 19.2 s Wall time : 19.3 s

En simulant le code pour un nombre de bits égal à 3000, le temps d'exécution devient

plus long :

CPU times : user 2min 29s, sys : 31 ms, total : 2min 29s Wall time : 2min 30s

D'autre part, la génération de clés s'avère complexe puisqu'il est difficile de factoriser un grand nombre n choisi. C'est d'ailleurs la base de la cryptographie.

```
p=next_prime(randint(10^(1000-1),10^1000-1))
q=next_prime(randint(10^(1000-1),10^1000-1))
n=p*q
%time n.factor()
```

Listing 3.2 – Simulation en Sagemath pour factoriser un module n choisi

Cette exécution prend un temps plus long. Une erreur, qui signifie que la mémoire est saturée, est générée lorsqu'on l'interrompt :

/opt/sagemath-8.9/local/lib/python2.7/site-packages/IPython/core/magics/execution.py :1189 : RuntimeWarning : cypari2 leaked 10871632 bytes on the PARI stack out = eval(code, glob, local$_n s$)

3.3 Attaques éventuelles

3.3.1 Cas où $ed - 1 = \phi(n)$

Dans ce cas particulier, il est facile de factoriser n en remarquant que p et q sont solutions de l'équation :

$$X^2 + (\phi(n) - n - 1)X + n = 0$$

```
p=next_prime(randint(10^(2-1),10^2-1))
q=next_prime(randint(10^(2-1),10^2-1))
n=p*q
phi_n=(p-1)*(q-1)
for e in range(phi_n) :
    a=ZZ(e).xgcd(phi_n)
    if a[0]==1 and a[2]==-1:
        break
d=a[1]
p,q,n,phi_n,e,d
```

Listing 3.3 – Simulation de générération d'un triplet de clés

(53, 29, 1537, 1456, 31, 47)

```
f=e*d-1
g=f-n-1
x = var('x')
equation = x^2 + g*x +n== 0
solutions = solve(equation, x)
solutions
```

Listing 3.4 – Simulation du calcul de p et q par l'utilisateur

[x == 53, x == 29]

Nous allons simuler par la suite le décryptage d'un message en récupérant p et q ou $\phi(n)$.

3.3.2 Cas général $ed - 1 = \alpha\phi(n)$ avec $\alpha \in \mathbb{Z} \setminus \{1\}$

Puisque $\phi(n) = (p-1)(q-1)$ et p et q sont différents de 2 (On choisit p et q très grands en cryptographie), alors $\phi(n)$ est divisible par 4.
On peut donc écrire $ed - 1$ sous la forme $2^s t$ avec $s \geq 2$ et t impair.

On choisit au hasard $y \in \mathbb{Z}/n\mathbb{Z} \setminus \{0\}$. Si $(y, n) \neq 1$, alors (y, n) est égal à p ou q car :

$$(y, n) \mid n \iff (y, n) \mid pq \tag{3.1}$$

En prenant $(p, (y, n)) = 1$ (sans perte de généralité puisque p et q jouent des rôles symétriques), on obtient

$$(y, n) \mid q \quad \text{et donc} \quad (y, n) = q \tag{3.2}$$

Sinon, $y \in (\mathbb{Z}/n\mathbb{Z})^*$. On a donc $y^{\phi(n)} \equiv 1 \pmod{n}$, ce qui implique

$$y^{\alpha\phi(n)} = y^{2^s t} \equiv 1 \pmod{n}.$$

Si $y^t \not\equiv 1 \pmod{n}$, alors $\exists\ j$ minimal dans $\{1, 2, \ldots, s\}$ tel que $y^{2^j} \equiv 1 \pmod{n}$

Le but est de trouver une racine carrée de 1 différente de 1 et de –1 car le lemme suivant permet de trouver un diviseur non trivial de n :

Enoncé du lemme :

Soit $x \in \mathbb{Z}/n\mathbb{Z}$ tel que $x \not\equiv \pm 1 \pmod{n}$ et $x^2 = 1 \pmod{n}$.

Alors $(x - 1, n)$ est un diviseur non trivial de n.

Démonstration :

$$x^2 \equiv 1 \pmod{n} \implies (x-1)(x+1) \equiv 0 \pmod{n}$$
$$\implies (x-1)(x+1) \equiv 0 \pmod{p} \quad \text{et} \quad (x-1)(x+1) \equiv 0 \pmod{q}$$
$$\implies ((x-1) \equiv 0 \pmod{p} \text{ et } (x+1) \equiv 0 \pmod{q}) \quad \text{ou}$$
$$((x+1) \equiv 0 \pmod{p} \text{ et } (x-1) \equiv 0 \pmod{q}) \quad \text{car } x \not\equiv \pm 1 \pmod{n}$$

D'où $(x-1, n)$ et $(x+1, n)$ sont des diviseurs non triviaux de n (différents de 1 et de n).

```
p=31
q=67
n=p*q
Zn=IntegerModRing(n)
l=[]
for i in Zn :
    if i!=ZZ(-1) and i!=ZZ(1) and i^2==ZZ(1):
        a=n.gcd(i-1)
        l.append(a)
l
```

Listing 3.5 – Simulation de la factorisation de n par le lemme précédent

[67, 31]

Remarque 1

La résolution de l'équation $x^2 \equiv 1 \pmod{n}$ avec $x \not\equiv \pm 1 \pmod{n}$ dans $\mathbb{Z}/n\mathbb{Z}$ nécessite, à priori, la connaissance de la factorisation de n en produit de nombres premiers. Si l'on savait résoudre cette équation sans utiliser cette factorisation, il serait alors facile de trouver p et q d'après le lemme précédent. Le problème de la factorisation des entiers serait ainsi résolu et la sécurité de nombreux cryptosystèmes serait complétement remise en cause.

Remarque 2

La résolution de l'équation $x^2 \equiv 1 \pmod{n}$ avec $x \not\equiv \pm 1 \pmod{n}$ dans $\mathbb{Z}/n\mathbb{Z}$ peut se faire séparément dans $\mathbb{Z}/p\mathbb{Z}$ et $\mathbb{Z}/q\mathbb{Z}$ grâce aux théorèmes des restes chinois, ce qui équivaut à résoudre :

$$(x \equiv 1 \pmod{p} \text{ et } x \equiv -1 \pmod{q}) \quad \text{ou} \quad (x \equiv -1 \pmod{p} \text{ et } x \equiv 1 \pmod{q})$$

```
Zn=IntegerModRing(n)
a=[[1,-1],[-1,1]]
h=[p,q]
solutions=[]
l=[]
for i in range (len(a)):
    x=crt(a[i],h)
    solutions.append(Zn(x))
```

```
    l.append(n.gcd(x-1))
solutions,l
```

Listing 3.6 – Simulation de la factorisation de n en utilisant le lemme et le théorème chinois

([1272, 805], [31, 67])

Donc si $y^t = 1$ ou $y^{2^{j-1}t} \equiv -1 \pmod{n}$, on doit tirer un nouveau y.

```
def generate_shared_rsa(bits):

    p = next_prime(randint(2**(bits//2 - 1), 2**(bits//2)))
    q = next_prime(randint(2**(bits//2 - 1), 2**(bits//2)))
    n = p * q
    phi_n = (p-1) * (q-1)

    # Generation de deux paires (e,d) diff rentes
    e1 = ZZ.random_element(3, phi_n)
    while gcd(e1, phi_n) != 1:
        e1 = ZZ.random_element(3, phi_n)
    d1 =e1.xgcd(phi_n)[1]

    e2 = ZZ.random_element(3, phi_n)
    while gcd(e2, phi_n) != 1 or e2 == e1:
        e2 = ZZ.random_element(3, phi_n)
    d2 = inverse_mod(e2, phi_n)

    return (n, e1, d1), (n, e2, d2), (p, q)
```

Listing 3.7 – Fonction qui génére un module RSA partagé et deux paires de clés différentes

```
def factor_from_private_key(n, e, d):

    Zn=IntegerModRing(n)
    k = e * d - 1
    # Decomposition de k en k = 2^s * t
    s = 0
    t = k
    while t % 2 == 0:
        s += 1
        t //= 2
    # Recherche d'un y approprie
    while True:
        y = ZZ.random_element(2, n)
        if gcd(y, n) != 1:
            return gcd(y, n)
        u = power_mod(y, t, n)
        if u == 1:
            continue # On tire un nouveau y
        # Recherche du plus petit j tel que u^(2^j) = 1 mod n
        for j in range(1,s):
```

```
            if u == -1:  # Si la racine carree vaut -1, on sort de
               la boucle et on tire un nouveau y
                break
            u_next = power_mod(u, 2, n)
            if u_next == 1: # On a trouvee une racine carree de 1
               non triviale, bingo!!!
                return gcd(u-1, n)
            u = u_next
```

Listing 3.8 – Fonction qui retourne l'un des facteurs premiers n en utilisant e et d

```
def demonstrate_factorisation():
     # Generation des cles
    a = generate_shared_rsa(2048)
    n, e1, d1 = a[0]
    p, q = a[2]
    print("Facteurs reels: \n p = %s, \n q = %s"%(p,q))
    p_trouve = factor_from_private_key(n, e1, d1)
    q_trouve = n / p_trouve

    print("\nFacteurs trouves:")
    print("p = %s"%(p_trouve))
    print("q = %s"%(q_trouve))

    # Verification
    success = {p_trouve, q_trouve} == {p, q}
    print("\nFactorisation reussie: %s"%(success))

demonstrate_factorisation()
```

Listing 3.9 – Fonction qui factorise un module n

Facteurs réels : p = 142185560498892448051271971045100553569864643221422047 7...,

q = 110671518944437010117530107038530313205686848204246....

Facteurs trouvés : p = 142185560498892448051271971045100553569864643221422047 7...,

q = 110671518944437010117530107038530313205686848204246....

Factorisation réussie : True

On remarque que cette attaque est fatale même pour des grands nombres de l'ordre des bits utilisés dans RSA (2048). Voici une simulation qui illustre le calcul de la clé de déchiffrement d'un utilisateur par un autre utilisateur espion qui a réussi par suite à décrypter un message qui lui est destiné.

```
a = generate_shared_rsa(2048)
n, e1, d1 = a[0] #Cles de l'utilisateur 1
n, e2, d2 = a[1] #Cles de l'utilisateur 2
def attack(c):
    #Factorisation de n par l'utlisateur 1 (espion) et calcul de
       phi_n qui lui permettera de trouver les cles de
       dechiffrement de tous les autres utilisateurs
    p_trouve = factor_from_private_key(n, e1, d1)
    q_trouve = n / p_trouve
    phi_n=(p_trouve-1)*(q_trouve-1)
    d22=e2.xgcd(phi_n)[1] # Calcul de la cle de dechiffrement de
       l'utilisateur 2
    m=c^d2
    return phi_n,d22,m
b=10^60
Zn=IntegerModRing(n)
b=Zn(b) # message qu'Alice veut envoyer a l'utlisateur 2
c2=b^e2 # message crypte
phi_n,d22,m=attack(c2)
Zphi_n=IntegerModRing(phi_n)
print(Zphi_n(d2)-Zphi_n(d22),b-m)
```

Listing 3.10 – Décryptage d'un message par un utilisateur espion

$(0, 0)$

3.3.3 Cas particulier où $(e_1, e_2) = 1$ et $m_1 = m_2$

Une attaque périlleuse peut se produire en quelques millisecondes dans le cas où deux utilisateurs ont le même module, des exposants e_1 et e_2 premiers entre eux et à qui on envoie le même message $m = m_1 = m_2$. En effet, si un espion remarque les propriétés de leurs clés publiques (même n, $(e_1, e_2) = 1$), il peut déchiffrer un message qui leur est destiné, en raison du théorème de Bézout. Soient $c_1 \equiv m^{e_1} \pmod{n}$, $c_2 \equiv m^{e_2} \pmod{n}$ et $ue_1 + ve_2 = 1$, avec u et v les coefficients de Bézout. L'espion n'a qu'à calculer $c_1^u \times c_2^v$ pour retrouver m.

Démonstration :

$c_1^u \times c_2^v \equiv ((m^{e_1})^u \pmod{n} \times (m^{e_2})^v \pmod{n}) \pmod{n} \equiv m^{ue_1+ve_2} \pmod{n} \equiv m \pmod{n}$

```
p=next_prime(randint(2^(1023),2^1024-1))
q=next_prime(randint(2^(1023),2^1024-1))
n=p*q
phi_n=(p-1)*(q-1)
l=[]
for e in range(1000) :
    e=ZZ(randint(2,phi_n))
    if e.gcd(phi_n)==1 :
        if len(l)==0 :
```

```
            l.append(e)
        else :
            if l[0].gcd(e)==1:
                l.append(e)
        if len(l)==2:
            break
e1=l[0] ; e2=l[1]
```

Listing 3.11 – Génération de p ; q ; n ; e1 et e2 premiers entre eux

```
import time
e1=l[0] ; e2=l[1]
m=3*2^(1023)
Zn=IntegerModRing(n)
m=Zn(m)
c1=m^e1
c2=m^e2
start =time.time()
u,v = e1.xgcd(e2)[1],e1.xgcd(e2)[2]
w=c1^u*c2^v
end=time.time()
m,w,m-w,end-start
m,w,m-w
```

Listing 3.12 – Simulation de l'attaque

(26965397022934738615939577861835371004269654684134...,
26965397022934738615939577861835371004269654684134...,
0,
0.0055768489837646484)

Cette fois encore, on remarque que l'attaque est efficace même pour de très grands nombres.

Evaluons à présent les chances de réussite d'un utilisateur pirate.

3.3.4 La probabilité de $y^t \equiv 1 \pmod{n}$

Dans cette section, nous allons montrer que la probabilité que $y^t \equiv 1 \pmod{n}$ est au plus $1/4$.

Supposons que $y^t \equiv 1 \pmod{n}$ et soit $\epsilon_1 \in \mathbb{Z}/n\mathbb{Z}$ tel que $\begin{cases} \epsilon_1 \equiv 1 \pmod{p} \\ \epsilon_1 \equiv -1 \pmod{q} \end{cases}$.

ϵ_1 existe grâce au théorème des restes chinois.

On a $y^t \equiv 1 \pmod{n} \implies y^t \equiv 1 \pmod{p}$ et $y^t \equiv 1 \pmod{q}$.

$$\begin{cases} (-y)^t \equiv (-1)^t \pmod{p} \equiv -1 \pmod{p} \\ (-y)^t \equiv (-1)^t \pmod{q} \equiv -1 \pmod{q} \end{cases} \implies (\text{-y})^t \not\equiv 1 \pmod{n}.$$

$$\begin{cases} (\epsilon_1 y)^t \equiv (\epsilon_1)^t \pmod{p} \equiv 1 \pmod{p} \\ (\epsilon_1 y)^t \equiv (\epsilon_1)^t \pmod{q} \equiv -1 \pmod{q} \end{cases} \implies (\epsilon_1 y)^t \not\equiv 1 \pmod{n}.$$

De même, on a :

$$\begin{cases} (-\epsilon_1 y)^t \equiv (-\epsilon_1)^t \pmod{p} \equiv -1 \pmod{p} \\ (-\epsilon_1 y)^t \equiv (-\epsilon_1)^t \pmod{q} \equiv 1 \pmod{q} \end{cases} \implies (\text{-}\epsilon_1 y)^t \not\equiv 1 \pmod{n}.$$

car par l'absurde on a :

Pour $x \in \mathbb{Z}/n\mathbb{Z}$ tel que $\begin{cases} x \equiv 1 \pmod{n} \\ x \equiv -1 \pmod{p} \quad \text{ou} \quad x \equiv -1 \pmod{q} \end{cases}$

On obtient $\quad 1 \equiv -1 \pmod{p} \quad$ ou $\quad 1 \equiv -1 \pmod{q} \implies p \mid 2 \quad$ ou $\quad q \mid 2$

Le résultat final est absurde car on prend $p \neq \text{q} \gg 2$ en cryptographie.

On a donc :

$$y^t \equiv 1 \pmod{n} \implies \begin{cases} (\epsilon_1 y)^t \not\equiv 1 \pmod{n} \\ (-y)^t \not\equiv 1 \pmod{n} \\ (-\epsilon_1 y)^t \not\equiv 1 \pmod{n} \end{cases}$$

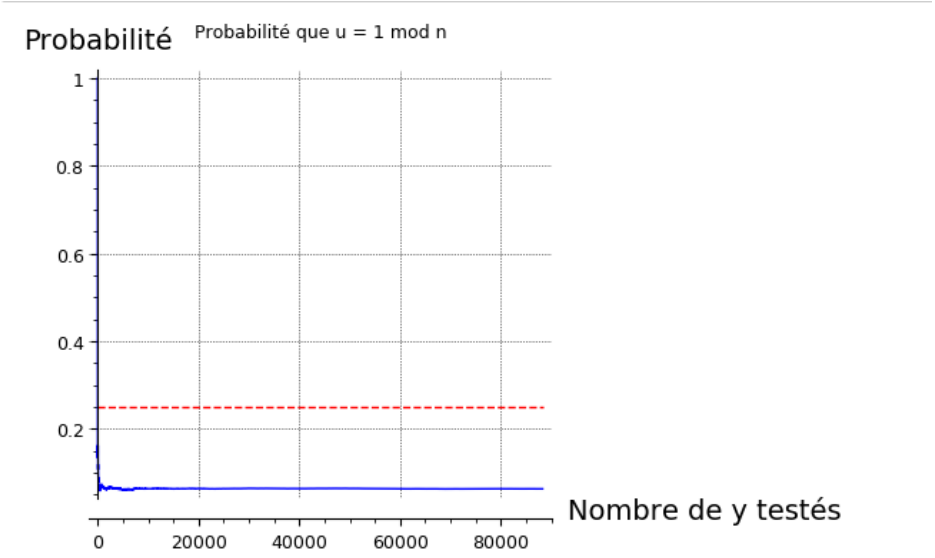
On déduit que parmi ces quatre éléments, au plus un vérifie $x^t \equiv 1 \pmod{n}$. Si l'on rassemble les éléments de $\mathbb{Z}/n\mathbb{Z}$ selon cette condition, on obtiendra des clusters disjoints de cardinal 4, sauf pour le dernier qui contient tous les nombres qui ne vérifient pas l'équation. La probabilité est donc égale à :

$$P(y^t \equiv 1 \pmod{n}) = \frac{1+1+1+\ldots}{4+4+4+\cdots+R} \leq \frac{1}{4}$$

```
def simulation_lemme_2(bits):
      # Taille des nombres premiers (petite pour la d monstration)
    # G n ration des nombres premiers et du module n
    p = random_prime(2^(bits), 2^(bits-1))
    q = random_prime(2^(bits), 2^(bits-1))
    n = p * q

    # Calcul de phi(n)
    phi_n = (p - 1) * (q - 1)
```

```
    # G n ration de e et d
    e = ZZ.random_element(3, phi_n)
    while gcd(e, phi_n) != 1:
        e = ZZ.random_element(3, phi_n)
    d = inverse_mod(e, phi_n)

    # Calcul de t (partie impaire de ed-1)
    omega = e * d - 1
    t = omega
    while t % 2 == 0:
        t = t // 2

# Initialisation des compteurs
    mauvais_y = 0
    total_y = 0
    probs = []

    # Test pour chaque y de 1    n-1
    for y in range(1, n):
        if gcd(y, n) == 1:  # On v rifie que y est inversible mod
           n
            total_y += 1
            u = power_mod(y, t, n)
            if u == 1:
                mauvais_y += 1
        if total_y > 0:
            probs.append(float(mauvais_y) / total_y)
        else:
            probs.append(0)

    # Cr ation du graphique
    plot_points = [(i + 1, probs[i]) for i in range(len(probs))]
    g = line(plot_points, color='blue')
    g += line([(0, 0.25), (len(probs), 0.25)], color='red',
       linestyle='--')

    # Ajout des axes et affichage
    g.show(gridlines=True,
    title=u"Probabilit  que u = 1 mod n",  # Use Unicode string or
        remove accented characters
    axes_labels=[u"Nombre de y test s", u"Probabilit "],
    fontsize=9)  # Same here


simulation_lemme_2(bits=10)
```

Listing 3.13 – Simulation de la probabilité de $y^t \equiv 1 \pmod{n}$



3.3.5 La probabilité de $u^{2^{j-1}} \equiv -1 \pmod{n}$

La probabilité que $u^{2^{j-1}} \equiv -1 \pmod{n}$ est au plus $\frac{1}{2}$.

Démonstration. Cette démonstration se repose sur le fait que pour n'importe quel y choisi qui vérifie $u^{2^{j-1}} \equiv -1 \pmod{n}$, on peut trouver un u_0 qui vérifie $u_0^{2^{j-1}} \not\equiv -1 \pmod{n}$.

Cela montre qu'on peut former des clusters disjoints de deux valeurs, dont uniquement une vérifie l'égalité, et un dernier cluster ne contenant que des valeurs ne vérifiant pas l'égalité, prouvant ainsi que la probabilité recherchée ne peut pas dépasser $\frac{1}{2}$ dans le pire des cas.

Première étape : Condition initiale et notations

Posons $\begin{cases} p-1 = 2^{k_1}t_1 \\ q-1 = 2^{k_2}t_2 \end{cases}$, et supposons sans perte de généralité que $k_2 \leqslant k_1$

Deuxième étape : Démontrer que $j \leqslant k_1$

On a par hypothèse, $u^{2^{j-1}} \equiv -1 \pmod{n}$, donc $u^{2^j} \equiv 1 \pmod{n}$, ce qui implique que $\begin{cases} \mathrm{ord}_p(u) \textit{ divise } 2^j \\ \mathrm{ord}_q(u) \textit{ divise } 2^j \end{cases}$, avec $\mathrm{ord}_p(u)$ et $\mathrm{ord}_q(u)$ les ordres de u dans $(\mathbb{Z}/p\mathbb{Z})^*$ et $(\mathbb{Z}/q\mathbb{Z})^*$ respectivement.

Preuve détaillée :

Par le théorème de Lagrange :

$$\mathrm{ord}_p(u) \mid (p-1) = 2^{k_1}t_1$$
$$\mathrm{ord}_q(u) \mid (q-1) = 2^{k_2}t_2$$

Notons $2^x = \text{ord}_p(u)$ et $2^y = \text{ord}_q(u)$. Comme x et y sont des puissances de 2, on a :

$$2^x \mid 2^{k_1}$$
$$2^y \mid 2^{k_2}$$

Donc $x \leq k_1$ et $y \leq k_2$. Posons $m = \max(x, y)$, alors $m \leq k_1$ et $u^{2^m} \equiv 1 \pmod{n}$.

Comme j est le plus petit exposant vérifiant $u^{2^j} \equiv 1 \pmod{n}$, on conclut que $j \leq m \leq k_1$.

Troisième étape : Construction de la stratégie de factorisation

Soit $\varepsilon \in (\mathbb{Z}/n\mathbb{Z})^*$ tel que :
- $\text{ord}_p(\varepsilon) = 1$
- $\varepsilon \equiv 1 \pmod{q}$

Définissons $y_0 = y\varepsilon$ et $u_0 = (y_0)^t = u\varepsilon^t$.

Lemme : $u_0^{2^{j-1}}$ n'est pas congru à $-1 \pmod{n}$ avec une probabilité au moins $\frac{1}{2}$.

Preuve du lemme : On montre que $u_0^{2^{j-1}} \not\equiv -1 \pmod{p}$:

$$\begin{aligned} u_0^{2^{j-1}} &\equiv u^{2^{j-1}} \varepsilon^{2^{j-1}t} \pmod{p} \\ &\equiv -\varepsilon^{2^{j-1}t} \pmod{p} \end{aligned}$$

Comme 2^j ne divise pas $2^{j-1}t$ (car t est impair), $\varepsilon^{2^{j-1}t} \not\equiv 1 \pmod{p}$.

Donc $u_0^{2^{j-1}} \not\equiv -1 \pmod{n}$.

Synthèse :

En répétant ce processus avec différents y aléatoires, la probabilité de factoriser n devient significative en un nombre raisonnable d'essais.

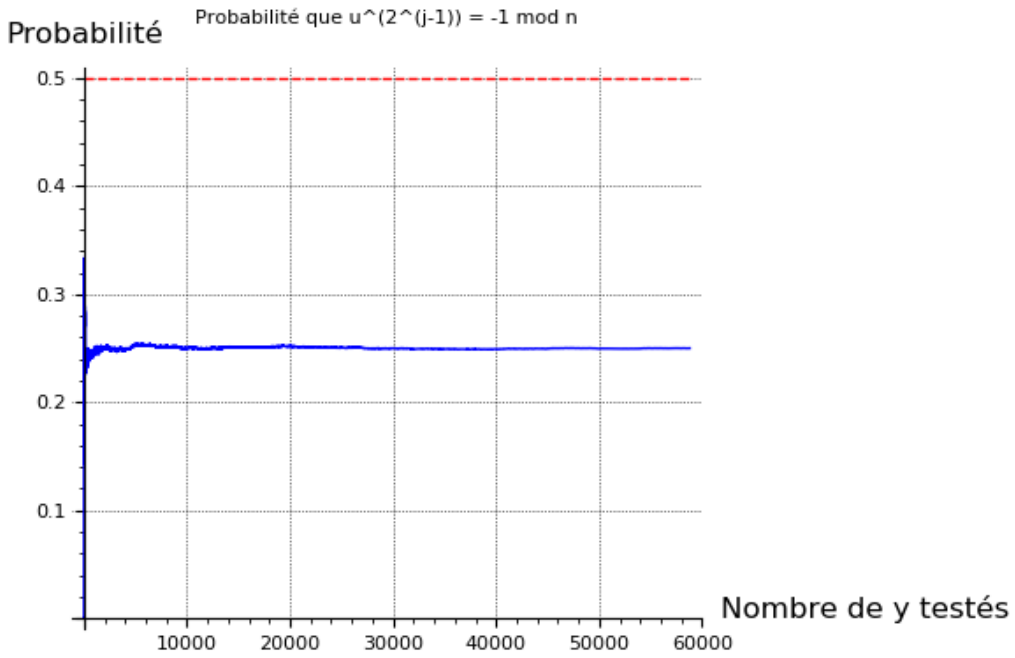
La probabilité que $u_0^{2^{j-1}}$ soit congru à $-1 \pmod{n}$ est au plus $\frac{1}{2}$, selon le principe des clusters.

En moyenne, en $\mathcal{O}(1)$ tirages aléatoires de y, cette stratégie fournit une factorisation de n, rendant le partage de module catastrophique pour la sécurité de RSA. □

```
def simulation_propo_2(bits):
    q = random_prime(2^(bits), 2^(bits-1))
    p = random_prime(2^(bits), 2^(bits-1))
    while q == p:
        q = random_prime(2^(bits//2), 2^(bits//2-1))
    n = p * q
    phin = (p-1)*(q-1)

    # G n ration de e et d
    e = ZZ.random_element(3, phin)
    while gcd(e, phin) != 1:
```

```
        e = ZZ.random_element(3, phin)
    d = inverse_mod(e, phin)

    # (ed-1 = 2^s * t)
    omega = e*d - 1
    t = omega
    s = 0
    while t % 2 == 0:
        t = t // 2
        s += 1

    # Initialisation des compteurs
    mauvais_y = 0
    total_y = 0
    probs = []

    # Test pour chaque y de 1    n-1
    for y in range(1, n):
        if gcd(y, n) == 1:  # On v rifie que y est inversible mod
           n
            total_y += 1
            u = power_mod(y, t, n)
            if u != 1:
                # Cherche le premier j tel que u^(2^j) = 1 mod n
                for j in range(1, s+1):
                    u_prev = u
                    u = power_mod(u, 2, n)
                    if u == 1 and u_prev == -1 % n:
                        mauvais_y += 1
                        break
        if total_y > 0:
            probs.append(float(mauvais_y)/total_y)
        else:
            probs.append(0)

    # Cr ation du graphique
    plot_points = [(i+1, probs[i]) for i in range(len(probs))]
    g = line(plot_points, color='blue')
    g += line([(0, 0.5), (len(probs), 0.5)], color='red', linestyle
       ='--')
    g.show(
        gridlines=True,
        title=u"Probabilit  que u^(2^(j-1)) = -1 mod n",
        axes_labels=[u"Nombre de y test s", u"Probabilit "],
        fontsize=8
    )

# Call the function
simulation_propo_2(bits=10)
```

Listing 3.14 – Simulation de la probabilité de $u^{2^{j-1}} \equiv -1 \pmod{n}$



3.3.6 Comparaison entre coût de RSA et le coût de l'attaque

L'attaque étudiée ne coûte donc pas plus cher qu'un chiffrement RSA puisque qu'en $O(1)$ tirages aléatoires de y, on peut trouver une racine carrée de 1 qui permet de factoriser n comme on l'a démontré précédemment.

```
a = generate_shared_rsa(2048)
n, e1, d1 = a[0] #Cles de l'utilisateur 1
n, e2, d2 = a[1] #Cles de l'utilisateur 2
def attack_time(c):
    start = time.time()
    #Factorisation de n par l'utlisateur 1 (espion) et calcul de
       phi_n qui lui permettera de trouver les cles de
       dechiffrement de tous les autres utilisateurs
    p_trouve = factor_from_private_key(n, e1, d1)
    q_trouve = n / p_trouve
    phi_n=(p_trouve-1)*(q_trouve-1)
    d22=e2.xgcd(phi_n)[1] #Calcul de la cle de dechiffrement de  l'
       utilisateur 2
    m=c^d2
    end = time.time()
    return phi_n,d22,m,end-start
b=10^60
Zn=IntegerModRing(n)
b=Zn(b) # message qu'Alice veut envoyer a l'utlisateur 2
c2=b^e2 # message crypte
phi_n,d22,m=attack(c2)
Zphi_n=IntegerModRing(phi_n)
c, encryption_time = rsa_encryption_time(b, e2, n)
phi_n,d22,m, attack_duration = attack_time(c)
```

```
print(Zphi_n(d2)-Zphi_n(d22),b-m)
print("Temps de chiffrement RSA : {:.2f} secondes".format(
    encryption_time))
print("Temps de l'attaque : {:.2f} secondes".format(attack_duration
    ))
```

Listing 3.15 – Simulation du temps d'exécution du chiffrement RSA et de l'attaque étudiée

(0, 0) Temps de chiffrement RSA : 0.02 secondes Temps de l'attaque : 0.02 secondes

Conclusion

D'après l'étude menée, il s'avère que le partage de module RSA entre plusieurs utilisateurs compromet la sécurité de tout un système. Certes, cette option facilite le chiffrement et l'échange entre une large communauté d'individus, mais elle est à proscrire absolument puisqu'elle offre une chance inouïe à des espions malveillants de reconstituer les clés de déchiffrement des autres utilisateurs et d'intercepter par conséquent tous leurs messages en un temps record et avec une probabilité très élevée de réussite. Dans le dernier cas, l'espion peut démasquer le message sans même avoir à factoriser le module commun, il faut donc éviter d'envoyer un même message à deux destinataires ayant le même module et des exposants publics premiers entre eux.

D'autres attaques peuvent avoir lieu si on tente de faciliter le procédé de chiffrement, puisqu'on peut remonter au message d'origine. En effet, pour e très petit, le problème du logarithme discret n'est plus assez difficile à résoudre. D'autre part, une attaque est aussi envisageable si on essaie de choisir une clé de déchiffrement trop petite en raison du principe de l'attaque de Wiener qui repose sur le calcul des convergents de la fraction continue $\frac{e}{n}$. Le défi de la cryptographie est de trouver un compromis du dilemme sécurité/complexité, car plus un système est sécurisé, plus il nécessite des grands nombres premiers difficiles à générer.

Avec l'émergence des ordinateurs quantiques,une nouvelle ère de cryptographie verra le jour, car les algorithmes basés sur la factorisation, comme RSA, deviendront obsolètes. L'exploration des alternatives résistantes aux calculs quantiques, telles que les schémas basés sur les réseaux (Lattice-based cryptography), devient essentielle.

"Le message le plus sûr est celui qui n'existe pas."
"En cryptographie, il n'y a pas de sécurité parfaite, seulement des probabilités d'échec très faibles."