ENSA
ECOLE NATIONALE DES SCIENCES APPLIQUEES
KHOURIBGA

10/12/2021

RAPPORT PROJET

CRYPTOGRAPHIE

SUJET : RSA avec la multiplication de Montgomery

Superviseur Prof. IFZANE

Auteurs

AMIDOU Abdoul Rahmane
BARRY Amadou Djoulde
KAPULA David David
GUIDIBI Teddy Sidoine Sèdjro
MONKOUN Aris Merlix
NAJID Mohammed
YASSINE Youssef

Tables des matières

Introduction

Face à l'hégémonie du numérique, la protection des données et systèmes numériques devient un enjeu crucial à notre ère. Pour ce faire, la cryptographie et la cryptanalyse deux disciplines de la cryptologie (science du secret), tiennent à assurer confidentialité, authenticité et intégrité des données, à travers des crypto systèmes tout en vérifiant leur résistance.

Le but de ce projet est de décrire le fonctionnement d'un cryptosystème moderne ainsi que la programmation de quelques cryptosystèmes (Code de Vigénère, Chiffrement de Hill, Masque jetable).

1 Présentation Cryptosystème RSA

Souvent utilisé pour la sécurisation des données confidentielles, en particulier lorsqu'elles sont transmises sur un réseau peu sûr comme internet, le cryptosystème RSA, nommé par les initiales de ses trois inventeurs (Ronald Rivest, Adi Shamir et Leonard Adleman) est un système cryptographique, pour le chiffrement à clé publique. Il a été décrit pour la première fois en 1977 par ses inventeurs du MIT (Massachusetts Institue of Technology).

Le chiffrement à clé publique, également appelé chiffrement asymétrique, utilise deux clés différentes, mais mathématiquement liées, une publique et l'autre privée. La clé publique peut être partagée avec quiconque, tandis que la clé privée doit rester secrète. Dans le chiffrement RSA, tant la clé publique que la clé privée peuvent servir à chiffrer un message. Dans ce cas c'est la clé opposée à celle ayant servir pour le chiffrement qui est utilisée pour le déchiffrement. C'est notamment grâce à cette caractéristique que le RSA est devenu l'algorithme asymétrique le plus répandu.

2 Principe et Fonctionnement

2.1 RSA Simple

Le chiffrement RSA utilise une paire de clés (des nombre entiers) composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des données confidentielles. Les deux clés sont créées par une personne (destinataire), qui souhaite que lui soient envoyées des données confidentielles. Cette dernière rend la clé publique accessible.

Cette clé est utilisée par ses correspondants (émetteur) pour chiffrer les données qui lui sont envoyées. La clé privée est quant à elle réservée au destinataire, et lui permet de déchiffrer ces données. La clé privée peut aussi être utilisée par le destinataire pour signer une donnée qu'elle envoie, la clé publique permettant à n'importe lequel de ses correspondants de vérifier la signature

Une condition indispensable est qu'il soit « calculatoirement impossible » de déchiffrer à l'aide de la seule clé publique, en particulier de reconstituer la clé privée à partir de la clé publique, c'est-à-dire que les moyens de calcul disponibles et les méthodes connues au moment de l'échange (et le temps que le secret doit être conservé) ne le permettent pas.

Etapes de création des clés

1. Choisir p et q, deux nombres premiers distincts ;
2. Calculer leur produit $n = p.q$, appelé *module de chiffrement* ;
3. Calculer $\varphi(n) = (p - 1)(q - 1)$ (c'est la valeur de l'indicatrice d'Euler en n) ;
4. Choisir un entier naturel e premier avec $\varphi(n)$ et strictement inférieur à $\varphi(n)$, appelé *exposant de chiffrement* ;
5. Calculer l'entier naturel d, inverse de e modulo $\varphi(n)$, et strictement inférieur à $\varphi(n)$, appelé *exposant de déchiffrement* ; d peut se calculer efficacement par l'algorithme d'Euclide étendu.

Comme e est premier avec $\varphi(n)$, d'après le théorème de Bachet-Bézout il existe deux entiers d et k tels que $e.d = 1 + k.\varphi(n)$, c'est-à-dire que $e.d \equiv 1 \pmod{\varphi(n)}$: e est bien inversible modulo $\varphi(n)$.

Le couple (n, e) est la *clé publique* du chiffrement, alors que sa *clé privée* est le nombre d

Chiffrement du message

Si ***M*** est un entier naturel strictement inférieur à ***n*** représentant un message, alors le message chiffré sera représenté par :

$$C \equiv M^e \pmod{n}$$

L'entier naturel ***C*** étant choisi strictement inférieur à ***n***

Déchiffrement du message

Pour déchiffrer C, on utilise d, l'inverse de e modulo $(p – 1)(q – 1)$, et l'on retrouve le message clair M par :

$$M \equiv C^d \pmod{n}$$

2.2 RSA avec Multiplication Montgomery

La réduction de Montgomery est un algorithme efficace pour la multiplication en arithmétique modulaire introduit en 1985 par Peter L. Montgomery.

Plus concrètement c'est une méthode pour calculer plus rapidement $a \times b \pmod n$.

Définition de la transformation – lien avec le produit

Soit n le modulo intervenant dans l'opération. On supposera désormais que n est un nombre impair. Le nombre de bits de n est l'entier k tel que :

$$2^{k-1} \leq n < 2^k$$

On appellera r le nombre 2^k. Comme n est impair r est premier avec n et donc inversible modulo n. On notera r^{-1} l'inverse de r modulo n.

Soit Φ (transformation de Montgomery) l'application de $I_n = \{0,1\ldots,n-1\}$ dans lui-même définie par : $\Phi(a) = a.r \bmod n$.

Cette application Φ(multiplication par r modulo n) est une bijection de I_n dans lui-même puisque r est inversible modulo n et on peut écrire : $a = \Phi(a).r^{-1} \bmod n$.

On définit le produit de Montgomery comme suit :

$A \times B = \Phi(a).\Phi(b).r^{-1} \bmod(n)$

Avec r^{-1} l'inverse de r modulo n.

On a aussi besoin de n' tel que : $r.r^{-1} - n.n' = 1$

r^{-1} et n' s'obtiennent avec l'algorithme d'Euclide étendu.

Voici l'algorithme pour calculer $A \times B$

Fonction *MonProd(Φ(a),Φ(b))*

1- $t = \Phi(a).\Phi(b)$

2- $m = t.n' \bmod(r)$

3- $u = t.n' \bmod(r)/r$

4- si $u \geq n$ **alors** return u-n **sinon** return u

Le théorème qui suit nous indique comment s'exprime le transformé d'un produit de deux termes en fonction des transformés de chacun de ces deux termes.

Théorème1 : si c = a.b(mod n) alors Φ(c) = MonProd(Φ(a), Φ(b))

Preuve : $\Phi(c) = c.r(mod\ n)$

$= a.b.r(mod\ n) = a.r.b.r.r^{-1}\ (mod\ n) = (a.r).\ (b.r).\ r^{-1}(mod\ n)$

$= \Phi(a).\ \Phi(b).\ r^{-1}mod(n) = A\ x\ B = MonProd(\Phi(a), \Phi(b))$

Théorème 2 : c = MonProd(Φ(c),1)

Preuve : $c = c.r.r^{-1}(mod\ n)$

$= \Phi(c).\ r^{-1}(mod\ n) = \Phi(c).1.\ r^{-1}(mod\ n) = MonProd(\Phi(c),1)$

Donc l'algorithme complet pour calculer c : = a.b (mod n)

Fonction Multi (a, b, n) avec n premier

Calculer *n'* en utilisant l'algorithme d'Euclid
Φ(b) = b.r (mod n)
Φ(c) = MonProd(Φ(a), Φ(b))
c = MonProd(Φ(c), 1) return c

Donc pour utiliser Montgomery pour calculer la puissance modulaire, on utilise un algorithme de calcul de puissance comme l'algorithme de *« Square and Multiply »*

L'algorithme s'écrit comme suit :

Fonction *Square and multiply*

Entrées : un tableau *N* de *K + 1* bits représentant n en binaire

Sortie : le nombre $P = a^n$

Début

P =1

i = K ;

tant que *i >= 0* faire

*P =P *P* ;

si N[i] == 1

P =P a* ;

i=i-1

retourner *P* ;

Donc on voit que en injectant la multiplication dans cet algorithme, on peut calculer

$a^k \pmod n$

Bien sûr en connaissant l'élément neutre de la multiplication de Montgomery qui est *r*

Fonction *PuissanceModulaire(M,e,n) :*

m =M.r (mod n)

c = r (mod n)

pour *i* =0 à *k-1* faire

c = *MonProd (c, c, n)*

si *k[i] ==1* alors

c = MonProd (m, c, n);

finPour

retourner *MonProd (c, 1, n)*

2.3 RSA Simple VS RSA avec Multiplication Montgomery

Ici nous avons implémenté deux programmes avec Python. Le premier *normal.py* qui utilise l'exponentiation a^b *mod(n)* représentant le RSA simple. Et le second *montgomery.py* qui utilise multiplication de Montgomery pour le traitement. L'idée est de comparer le temps de résolution. Ceci n'est remarquable que quand les variables d'entrées sont de grandes tailles.

Pour la mesure du temps de résolution, nous avons utilisés la command *Measure-Command* dans le PowerShell.

Entrée :

```
m = 14159265358979323846264338327950288419971
x1 = 26807182993661569704416700728655285
x2 = 51569210766546368030581937859356970
```

Sortie :

```
947403516972947778066531256671171402321
```

RSA Simple (exponentiation modulaire)

#Code : *normal.py*

Mesure du temps d'exécution avec le PowerShell

```
PS C:\Users\DELL> cd .\Desktop\
PS C:\Users\DELL\Desktop> Measure-Command { python normal.py }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 108
Ticks             : 1089411
TotalDays         : 1.26089236111111E-06
TotalHours        : 3.02614166666667E-05
TotalMinutes      : 0.001815685
TotalSeconds      : 0.1089411
TotalMilliseconds : 108.9411
```

Temps d'exécution : 108 Milliseconds

RSA avec la multiplication de Montgomery

#Code : montgomery.py

Mesure du temps d'exécution avec le PowerShell

```
PS C:\Users\DELL\Desktop> Measure-Command { python montgomery.py }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 101
Ticks             : 1014721
TotalDays         : 1.17444560185185E-06
TotalHours        : 2.81866944444444E-05
TotalMinutes      : 0.00169120166666667
TotalSeconds      : 0.1014721
TotalMilliseconds : 101.4721
```

Temps d'exécution : 101 Milliseconds

Interprétation

On peut dire que ce petit gap dans l'exécution des deux programmes va devenir de plus en plus grand relativement à la taille des calculs. L'algorithme de Montgomery est donc plus efficace quand les nombres évoluent en taille qui notamment nécessiteront des ressources plus puissantes et plus performantes.

3 Exemples illustratifs

Exemple 1

Multiplication de deux nombres a et b en utilisant l'algorithme de Montgomery

Soit a = 68 ; b = 57 ; n = 109

On a $2^{k-1} \leq n < 2^k$ donc r= 2^k= 2^7= 128 avec l'entier *k* le nombre de bits de *n*

On calcule l'inverse de *n* tel que *r.v* – *n.v* = 1 donc u = 27 car -107 = 27(mod 120)

On calcule *Φ(a)* et *Φ(b)*

*Φ(a) = a.r(mod n) = 68*128(mod 109) = 93*

*Φ(b) = b.r(mod n) = 57*128(mod 109) = 102*

On calcul *Φ(c)* qui est égal à *MonProd(Φ(a), Φ(b)) = MonProd(93,102) = 69*

Donc *c = MonProd(Φ(c),1) = MonProd(69,1) = 61 donc 68*57 = 61(mod 109)*

Exemple 2

Calcul de 7^{10} (mod 13)

On pose a = 7 et b = 1

On a r = 2^k = 16 et 16*9 – 13*11 =1 . alors r^{-1} = 9 et n' = 11

*Φ(a) = a.r(mod n) = 7*16(mod 13) = 8*

*Φ(b) = b.r(mod n) =1 *16(mod 13) = 3*

Alors Φ(a) = 8 et Φ(c) = 3

Calcul de *MonProd(Φ(a), Φ(b)) = MonProd(8,3)*

t = 8*3 = 24; m = 24*11 (mod 16) = 8 ; u = (24 + 8.13)/16 = 8

Alors *MonProd(8,3) = 8*

Calcul de *MonProd(Φ(c),1) = MonProd(8,1)*

t = 8*1 = 8 ; m = 8*11 (mod 16) = 8 ; u = (8 + 8*13)/16 = 7 →*MonProd(Φ(c),1) =7*

4 Cryptanalyse RSA

La cryptanalyse est l'art pour une personne non habileté de déchiffrer, décoder un message. C'est donc l'ensemble des procédés d'attaques d'un système cryptographique qui peuvent être *matérielles* (exploitent les faiblesses de l'implantation matérielle des algorithmes de cryptographie embarqués sur les composants de sécurité contenant des informations confidentielles) ou *cryptanalytiques* **(**attaques purement mathématiques**)**.

Le cryptosystème RSA est connu pour sa sécurité qui est basée sur la difficulté de factorisation de N tel que $N = p.q$. Ainsi, il s'avère fastidieux à cause de l'impuissance de nos automates qui peuvent atteindre des résultats après des millions d'années avec les différents algorithmes mis en place pour palier à cette difficulté. Ci-dessous nous avons quelques attaques mathématiques relevées.

4.1 Attaque 1

Cryptanalyse de RSA connaissant $\varphi(N)$

Proposition

Soit N un module de RSA, Si on connait $\varphi(N)$, alors on peut factoriser N.

Preuve **:**

Supposons que $\varphi(N)$ est connu. Ainsi, on dispose d'un système de deux équations en p et q :

$$\begin{cases} p.q = N \\ p + q = N + 1 - \varphi(N) \end{cases}$$

Les nombres p et q sont racines du polynôme

$P(X) = X^2 - (p + q)\,X + p.q$

Ainsi, on peut déduire p et q tels que : $b = (p + q)$, $a=1$, $c=p.q$

$\Delta = b^2 - (4.a.c)$ et $p = (-b-\sqrt{\Delta})/2.a$ et $q = (-b+\sqrt{\Delta})/2.a$

4.2 Attaque 2

La manière la plus intuitive d'essayer de décrypter la RSA est de chercher par élimination les nombres premiers *P* et *Q* tels que *N=P.Q*. Hélas, cela ne marche que pour des petits nombres. Pour y parvenir, un mathématicien américain du nom de *PETER SHORR* a proposé un algorithme en 7 étapes conduisant à la détermination des nombres P et Q quelle que soit la taille de N appelé *ALGORITHME DE SHOR* en son hommage. Elle se présente comme suit :

Nous prenons *N=391*

Etape 1 : Prendre un nombre aléatoire *a* tel que *a<N*

Proposition **:** *a=24*

Etape 2 **:** Calculer le PGCD (a, N) **(**algorithme d'Euclide)

Etape 3 **:** Analyse du résultat du PGCD

Cas1 **:** *PGCD (a, N)≠1* donc *a* est un multiple de *N* et ainsi on peut déduire *q* et *p*

Cas2 **:** *PGCD (a, N)=1* pour notre exemple, *PGCD (24 ,391)=1*

Etape 4 **:** Trouver la période

Note : Si deux nombres (au hasard *a* et *N*) sont premiers entre eux, alors la fonction $f(x) = a^X \ mod \ N$ à une période *r* telle que *f(x+r) = f(x)* et cette période se terminera toujours par *f(x) =1***.**

Exemple : 1--> 4 mod15=4 2--> 4^2 mod15=1

3--> 4^3 mod15=4 4--> 4^4mod15=1

Période=2

Attention : pour *a=11* et *N=39*, on se retrouve rapidement à une période *r=12*

Pour notre exemple, *r=16*

Etape 5 : vérifier r

Cas1 : r impair, retourner à l'étape 1

Cas2 : r pair, continuer

r =16 donc ok

Etape 6 : vérifier $a^{r/2}+1$

Cas1 : $a^{r/2}+1 \equiv 0\ (N)$, retourner à l'étape 1

Cas2 : $a^{r/2}+1 \equiv 0\ (N)$, continuer

$24^{16/2}+1 \equiv 255\ (391)$

Etape 7 : Déterminer P et Q

On a $P=PGCD\ (a^{r/2}+1, N)$ et $Q= PGCD\ (a^{r/2}-1, N)$

$P=17$ et $q=23$ et on vérifie $391=17*23$

Pourquoi ?

On sait que $a^r \equiv 1\ (N) \rightarrow a^r - 1 \equiv 0(N) \rightarrow a^r - 1 = N.k$

Or $N=P.Q$ et $a^r - 1 = (ar^{/2})^2 - 1^2 = (ar^{/2}-1)\ (a^{r/2}+1)$

Ainsi, $(a^{r/2}-1)\ (a^{r/2}+1) = P.Q.k$. On en déduit que $= (ar^{/2}-1)\ (a^{r/2}+1)$ est divisible par P et Q et partage ces deux facteurs avec N.

De plus, P et Q ne peuvent pas diviser la même partie. Si c'était le cas, cette partie serait divisible par N.

Or on s'est assuré que $a^{r/2}+1 \equiv 0(N)$ ne se réalise pas et si $a^{r/2}-1 \equiv 0(N) \rightarrow a^{r/2} \equiv 1(N)$

Or r est la plus petite période possible ou cela puisse arriver donc P et Q divisent une seule partie à la fois.

4.3 Attaque à module N identique

Soit deux clés publiques (N, e_1) et (N, e_2) possédant le même module N.

Si un message M est chiffré avec ces deux clés on peut alors retrouver ce message tel que :

Prenons : $C_1 \equiv M^{e1} mod(N)$, $C_1 \equiv M^{e2} mod(N)$

Si on a $pgcd\ (e_1, e_2) = 1$ alors on a $e_1.u + e_2.v = 1$

Il nous reste juste à calculer $C_1^u * C_2^v$ car on retrouve le message clair

$C_1^u . C_2^v = M^{u.e_1} . M^{v.e_2} = M^{u.e_1 + v.e_2 = 1} = M$

4.4 Attaque de Fermat

L'attaque de fermat consiste à casser le chiffrement grace à la factorisation de n tel que $n = p * q$ et $|p - q|$

$< c.n^{\frac{1}{4}}$ qui signifie que l'écart entre p et q ne doit pas être très grand.

1. $n = p * q \rightarrow (t - s) * (t + s)$ avec $t = \frac{p+q}{2}$ et $s = \frac{p-q}{2}$

On pose $t = \lceil \sqrt{n} \rceil$: partie entiere superieure de n

$c = t^2 - n$

Tant que c n'est pas un carré parfait on incrémente $t \leftarrow t + 1$ et on calcule c

$$c = t^2 - n$$

On calcule ensuite $s = \sqrt{c}$, $p = t + s$ et $q = t - s$

Exemple: $n = 2183$

$t = \lceil \sqrt{2183} \rceil = 47$; $c = 47^2 - 2183 = 26$

26 n'estpas un carré parfait: $t = 47 + 1 = 48$ et $c = 48^2 - 2183 = 121$

121 est un carré parfait: $s = 11$ et $p = 48 + 11 = 59$ et $q = 48 - 11 = 37$

5 Algorithmes de quelques Cryptosystèmes

Ci-joint l'aperçu explicite des programmes de quelques cryptosystèmes.

Note : les fichiers python des dits programmes sont rattachés au rapport dans un fichier zip.

5.1 Code de Vigénère

Fonction de chiffrement

```python
alphabet = "abcdefghijklmnopqrstuvwxyz "

lettre_en_chiffre = dict(zip(alphabet, range(len(alphabet))))
chiffre_en_lettre = dict(zip(range(len(alphabet)), alphabet))


def chiffrement(message, key):
    code = ""
    ##separation du message en bloc correspondant a la taille de la clé
    bloc = [
        message[i : i + len(key)] for i in range(0, len(message), len(key))
    ]
    ##parcourir chaque bloc et additionner chaque lettre mis en chiffre à la clé correspondante
    for bl in bloc:
        i = 0
        for lettre in bl:
            nombre = (lettre_en_chiffre[lettre.lower()] + lettre_en_chiffre[key[i].lower()]) % len(alphabet)
            code += chiffre_en_lettre[nombre]
            i += 1

    return code
```

Fonction de déchiffrement

```
def dechiffrement(bloc_ch, key):
    dechiffr = ""
    ##bloc
    bloc_code = [
        bloc_ch[i : i + len(key)] for i in range(0, len(bloc_ch), len(key))
    ]
    ##parcourir bloc comme pour le chiffrement mais soutraire cette fois ci
    for bl in bloc_code:
        i = 0
        for lettre in bl:
            nombre = (lettre_en_chiffre[lettre] - lettre_en_chiffre[key[i].lower()]) % len(alphabet)
            dechiffr += chiffre_en_lettre[nombre]
            i += 1

    return dechiffr
```

Test du code

```
def main():
    message = "nous avons projet de crypto"
    key = "ifzarne"
    code_message = chiffrement(message, key)
    dechiffr_message = dechiffrement(code_message, key)

    print("message original: " + message)
    print("chiffrement: " + code_message)
    print("Dechiffrement message: " + dechiffr_message)

main()
```

Sortie :

```
message original: nous avons projet de crypto
chiffrement: vtssqnzwsq fdsrjr urdkwwpja
Dechiffrement message: nous avons projet de crypto
```

5.2 Chiffrement de Hill

Définition fonction de Chiffrement

```python
#Le chiffrement de hill
import numpy as np ## on utilise numpy pour le calcul de determinant et l'inverse de la matrice
from egcd import egcd   # une fonction pour calculer l'euclid etendu
import math
M = [[3,5],[6,17]] #une matrice
m = "TEXTACRYPTERer"        # une chaine de caracteres pour faire un exemple
def chiffrement_hill(M,m):
    #M c la matrice et m le message a chifrrer
    # on travail dans Z/26Z
    messageVector =[[0]*len(M) for i in range(len(M))]
    if len(m)%len(M):
        raise Exception(f"La Longeur du message doit etre divisble par {len(M)}")
    arrVect = []
    if len(M) != len(M[0]):
        raise Exception("M doit etre une matrice carrer")
    det = int(np.linalg.det(M))
    #verifier si la matrice est inversible
    if math.gcd(det,26) !=1:
        raise Exception("La matrice doit etre inversible dans Z/26Z")
    # transformer le message en vecteurs
    for i in range(len(m)):
            arrVect.append(ord(m[i].upper())%65)
    messageVector = [[0]*len(M) for _ in range(len(m)//len(M))]
    for i in range(len(m)//len(M)):
        for x in range(len(M)):
         messageVector[i][x] = arrVect.pop(0)
#Coder le message
textchiffrer = []
for vect in messageVector:
    chifrrer = np.dot(np.array(M),np.array(vect).T)
    chifrrer = [num%26 for num in chifrrer]
    textchiffrer.append(chifrrer)
```

Création d'une fonction matrix_modulo_inverse retournant l'inverse de la matrice modulo 26

```python
def matrix_modulo_inverse(mat):
    #Le determinant
    det = int(np.round(np.linalg.det(mat)))
    det_inv = egcd(det,26)[1]%26
    mat_inver = det_inv * np.round(det*np.linalg.inv(mat)).astype(int)%26
    return mat_inver
```

Fonction de Déchiffrement

```
def dechiffrement_hill(M,c):
    arrVect = []
    for i in range(len(c)):
            arrVect.append(ord(c[i])%65)
    messageVector = [[0]*len(M) for _ in range(len(c)//len(M))]
    for i in range(len(c)//len(M)):
        for x in range(len(M)):
            messageVector[i][x] = arrVect.pop(0)

    M_inverse = matrix_modulo_inverse(M)
    textchiffrer = []
    for vect in messageVector:
        chifrrer = np.dot(M_inverse,np.array(vect).T)
        chifrrer = [num%26 for num in chifrrer]
        textchiffrer.append(chifrrer)
    trans = [chr(int(num) + 65) for num in np.array(textchiffrer).flatten()]
    #faire un join epuis retourner la chaine chifrer
    return ''.join(trans)
```

Test

```
print(f'chiffrer de {m}  = {chiffrement_hill(M, m)}')
print(f'dechiffrer de {chiffrement_hill(M, m)} = {dechiffrement_hill(M,chiffrement_hill(M, m))}')
```

Sortie :

```
chiffrer de TEXTACRYPTERer  = ZAITKIPQKXTBTB
dechiffrer de ZAITKIPQKXTBTB = TEXTACRYPTERER
```

5.3 Masque Jetable

Fonction de chiffrement

```
def masque(m,key):
    if len(key) < len(m):
        raise Exception('La clee doit etre superieur ou egal au message m')
    chaine_en_ascci =[ord(m[i].upper()) for i in range(len(m))]
    cle_en_ascci = [ord(key[i].upper()) for i in range(len(key))]
    text_chiffre = [chr((chaine_en_ascci[i] + cle_en_ascci[i])%26 +65) for i in range(len(m))]
    return ''.join(text_chiffre)
```

Fonction de déchiffrement

```
def dechifrer_masque(c,key):
    if len(key) < len(c):
        raise Exception('La longueur de la cle doit etre superieur ou egal a la longueur du message m')
    chaine_en_ascci =[ord(c[i].upper()) for i in range(len(c))]
    cle_en_ascci = [ord(key[i].upper()) for i in range(len(key))]
    text_chiffre = [chr((chaine_en_ascci[i] - cle_en_ascci[i])%26 +65) for i in range(len(c))]
    return ''.join(text_chiffre)
```

Test

```
print("Chiffrement: "+masque("HELLOtut","WMCKLtut"))
print("Déchiffrement: "+ dechifrer_masque("DQNVZMOM","WMCKLtut"))
```

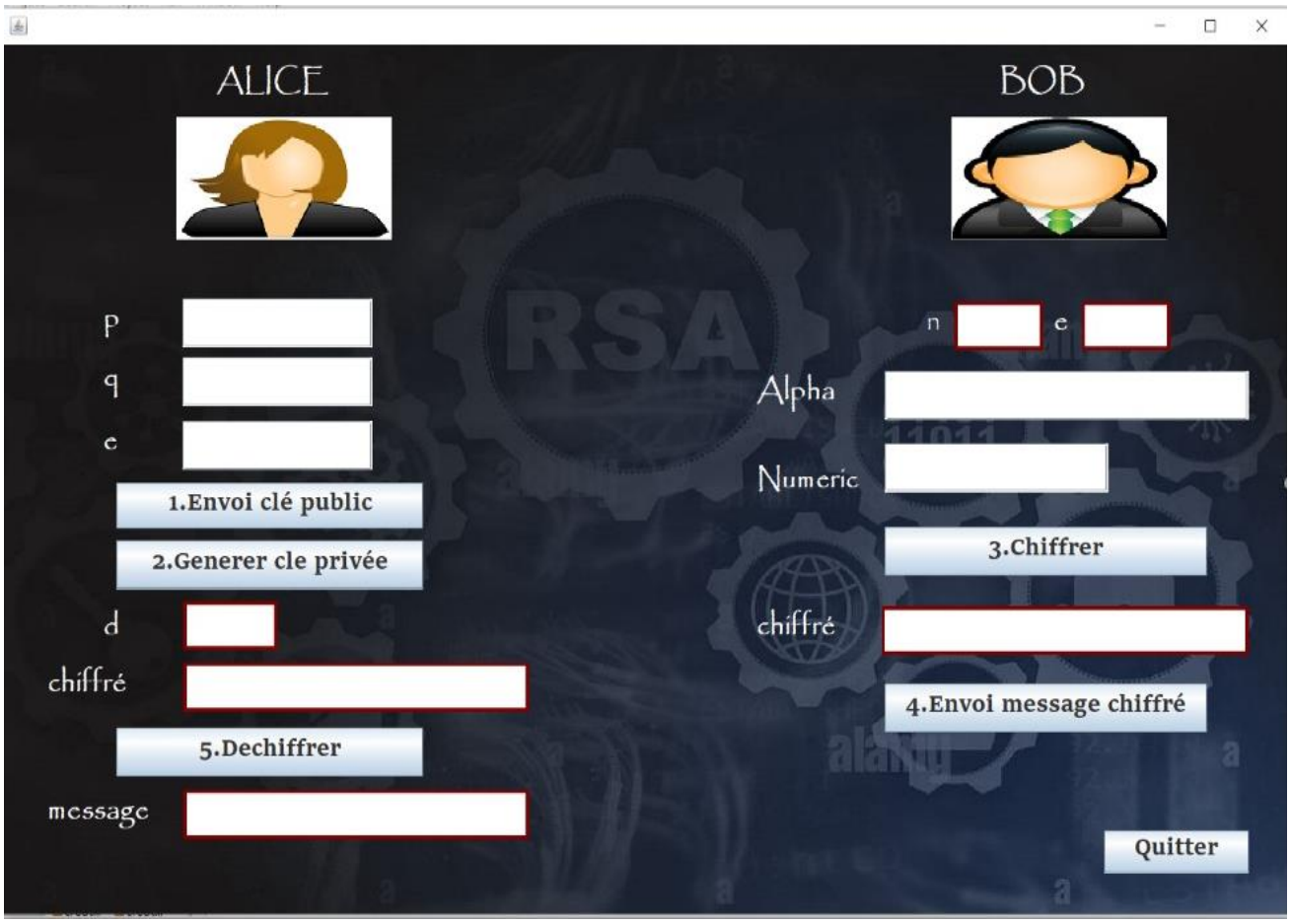
Sortie :

```
Chiffrement: DQNVZMOM
Déchiffrement: HELLOTUT
```

6 Application Simulateur RSA

Afin d'étendre notre compréhension du cryptosystème RSA, nous avons créé une application en java, permettant de simuler le RSA en chiffrant et déchiffrant un message.

Ci-dessous l'interface graphique de notre simulateur.



Conclusion

La réalisation de ce projet a été le fruit d'une synergie et d'un travail d'équipe. Cela nous a permis de partager nos différentes connaissances sur les différents cryptosystèmes abordés en classe, d'élargir notre champ de savoir et de développer notre savoir-faire. Nous tenons donc à remercier notre cher Professeur Mr Ifzane, qui a su par sa pédagogie non seulement nous partagé ses connaissances dans ce domaine mais aussi nous donné l'amour d'apprendre plus.