

# Compte-rendu de TP

## Pierre Mauduit - GI04

### Printemps 2007

## Introduction

Le but de ces travaux pratiques, étalés sur 2 séances de 3 heures, est de comprendre le chiffrement RSA via le décryptage d'un texte donné, et son reencryptage.

Bien évidemment la force de ce système de cryptage prend sa source dans la taille des entiers qu'elle manipule. Ici, nous n'avons pas la prétention d'utiliser une solution sûre, nous ne travaillerons qu'avec des entiers longs non signés, mais cela reste amplement suffisant afin d'en comprendre les principes fondamentaux.

## 1 Algorithmes implémentés

Afin de mener à bien la découverte du système RSA, un certain nombre d'algorithmes ont été nécessaires. Nous allons dans cette partie donner le code C de ces fonctions, et les expliquer une à une.

### 1.1 Détermination de la primalité d'un nombre

Afin d'implémenter l'algorithme RSA, il est nécessaire d'avoir une fonction permettant de déterminer si un nombre est premier ou non. Nous allons ici utiliser une algorithmes basée sur une méthode assez naïve : Pour tout entier  $k$  inférieur ou égal à  $\sqrt{n}$ , on teste si  $n$  est divisible par  $k$ . Si oui, notre nombre  $n$  n'est pas premier. Dans le cas contraire, si tous les entiers ont été testés sans trouver de diviseur, on en conclut que notre nombre  $n$  est bien premier.

Notons que dans le chiffrement RSA utilisé dans la vie courante, le système ne peut se permettre cette recherche naïve, et des méthodes statistiques sont utilisées permettant d'affirmer qu'un nombre est premier avec un certain pourcentage d'erreurs.

Le code source C au sein de notre application est le suivant :

```
int is_prime(unsigned long num)
{
    unsigned long sqrtm = (unsigned long) sqrt (num);
    unsigned long i;
```

```
if (num <= 2)
    return TRUE;

if ((num % 2) == 0)
    return FALSE;

for (i = 3; i <= sqrtm; i++)
    if ((num % i) == 0)
        return FALSE;

return TRUE;
}
```

L'algorithme procède comme suit :

- Si le nombre donné est inférieur ou égal à 2, il est premier.
- Si il est pair, il n'est pas premier.
- sinon, on teste le reste de la division de ce nombre par tous les entiers supérieurs ou égaux à sa racine carrée. Si le reste est égal à 0, c'est que le nombre est divisible, et donc il n'est pas premier.

## 1.2 Décomposition en facteurs premiers

Pour implémenter le système RSA, nous avons aussi besoin d'une fonction de décomposition en facteurs premiers :

```
/*
 * Decomposition en facteurs premiers
 *
 *
 */
unsigned long * decomp_factor (unsigned long n , int *taille)
{
    unsigned long *tab = NULL;
    unsigned long d;
    int t = 0;

    while (n > 1)
    {
        /* on a trouvé un facteur premier */
        if (is_prime(n) == TRUE)
```

```
{
    t++;
    tab = (unsigned long *) realloc (tab , t * sizeof (unsigned long));
    tab [t -1] = n;
    n=1;
}

    /* 2 est facteur premier */
    if ((n%2) == 0)
{
    t++;
    tab = (unsigned long *) realloc (tab , t * sizeof (unsigned long));
    tab [t -1] = 2;
    n = n / 2;
}

    else
{
    /* parcours des diviseurs */
    for (d = 3; (d*d) < n; d = d+2)
        {
            /* si un diviseur est trouvé et qu'il est premier */
            if (((n%d) == 0) && is_prime(d))
{
                t++;
                tab = (unsigned long *) realloc (tab , t * sizeof (unsigned long));
                tab[t-1] = d;
                n = n/d;
            }
        }
    }
    *taille = t;
    return tab;
}
```

Cette fonction C prend en argument un nombre à décomposer en facteurs premiers et un pointeur vers la taille du tableau (composé des facteurs premiers) qui sera renvoyé.

Tant que n est supérieur à 1, il est possible de trouver des facteurs premiers. Il est donc nécessaire de continuer à boucler.

- Si n est premier, il est son propre facteur, on le rajoute à la liste.
- Si n est pair, alors 2 est un facteur premier. 2 est alors rajouté à la liste.

- Sinon, on parcourt les entiers impairs inférieurs ou égaux à *sqrtn* et on le rajoute à la liste des facteurs premiers si ce dernier est évidemment premier et s'il divise *n*.

### 1.3 détermination du PGCD de deux nombres

Nous avons encore besoin d'une autre algorithmme renvoyant le PGCD de deux nombres.

```
unsigned long pgcd (unsigned long a, unsigned long b)
{
    unsigned long c;

    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }

    return a ;
}
```

Cette fonction renvoyant un entier long reste relativement simple à comprendre. Il s'agit en réalité d'une implémentation de l'algorithme d'Euclide dite simple. Elle part toutefois du principe que *a* est supérieur à *b*. Tant que *b* est différent de 0, on enregistre le reste de la division de *a* par *b* dans une variable temporaire.

avant d'écraser *b* qui va prendre la valeur du reste sauvé dans *c*, on affecte à *a* la valeur de *b*, et ainsi de suite. Le PGCD de *a* et de *b* sera alors stocké dans *a* lors de la dernière itération.

### 1.4 Calcul du Phi

Le système RSA introduit un calcul  $\Phi(n)$ , qu'il a été nécessaire d'implémenter.

Pour un nombre *n* donné, dont la décomposition en facteurs premiers introduit deux nombres *p* et *q*, on définit  $\Phi(n)$  comme étant le produit de *p*-1 et de *q*-1.

```
unsigned long phi(unsigned long num)
{
```

```
unsigned long result = 0;

int size = 0;
unsigned long * decompo = NULL;

    decompo = decomp_factor (num, &size);

if ((decompo == NULL) || (size != 2))
{
printf("- Err in phi() !\n");
goto end;
}

/* else */

result = (decompo[0]-1) * (decompo[1]-1);

end:

if (decompo != NULL)
free (decompo);

return result;
}
```

Cette fonction vérifie dans un premier temps que la décomposition de  $n$  est bien composée uniquement de deux facteurs. Sinon, le resultat renvoyé sera 0, ce qui n'a aucun sens, et permet ainsi un contrôle d'erreur.

Dans le cas contraire, le résultat demandé est calculé, et renvoyé.

### 1.5 Algorithme d'Euclide généralisée

Cette fonction est l'implémentation de l'algorithme d'Euclide généralisée. elle permet d'obtenir le quotient et reste de la division de deux entiers  $a$  et  $b$ .

```
void euclide (unsigned long a,
             unsigned long b,
             unsigned long *x ,
```

```
        unsigned long *y)
{
    unsigned long p = 1 , q = 0;
    unsigned long r = 0 , s = 1;

    unsigned long c , quotient, nouveau_r, nouveau_s;

    while (b != 0)
    {
        c = a % b;
        quotient = a / b ;
        a = b;
        b = c;
        nouveau_r = p - quotient * r;
        nouveau_s = q - quotient * s;
        p = r;
        q = s;
        r = nouveau_r;
        s = nouveau_s;
    }
    *x = p;
    *y = q;
}
```

Cette algorithmme d'Euclide étendue est quasi similaire à l'algorithme "simple", si ce n'est qu'il est nécessaire d'ajouter deux soustractions à chaque itération.

## 1.6 Exponentiation rapide

Enfin, une dernière fonction utile dans l'implémentation du système RSA a été l'implémentation d'une algorithme d'exponentiation rapide.

```
#define bit(x,i) (((x) & (1 << (i))) ? 1 : 0)

unsigned long fast_expo (unsigned long a,
                        unsigned long x,
                        unsigned long m)
{
    unsigned long tmp;
```

```
unsigned long r;
int nbits;
int i;

nbits = 0;

tmp = 1;
while (x > tmp)
{
    nbits ++;
    tmp <<= 1;
}
if (bit(x,nbits-1) == 1)
    r = a;
else
    r = 1;

i = nbits - 2;

while ( i >= 0)
{
    r = ( r * r ) % m;
    if (bit(x,i) == 1)
{
    r = (r*a) % m;
}
    i --;
}

return r ;
}
```

Cette fonction utilise une macro `bit(x, i)` permettant de définir l'état du bit d'un nombre `x` à une position `i` donnée. Elle calcule :

$$r = a^x[m]$$

Il est en effet impensable d'envisager de calculer  $a^x$  et d'ensuite tenter de chercher le reste de la division par `m` du nombre obtenu, étant donné la valeur possiblement très élevée de `a` et `x`.

Nous avons dorénavant tous les algorithmes nécessaires à la réalisation de cette séance de travaux pratiques.

## 2 Application

### 2.1 Décomposition en facteurs premiers

Dans un premier temps, étant donné  $n = 18923$ , il est demandé de trouver deux nombres premiers  $p$  et  $q$ , tels que :

$$n = p * q$$

En appliquant simplement l'algorithme exposé précédemment permettant la décomposition en facteurs premiers, on trouve :

$$\begin{cases} p = 127 \\ q = 149 \end{cases}$$

### 2.2 Calcul du Phi

le  $n$  donné précédemment pouvant s'exprimer sous la forme d'une multiplication de deux nombres premiers, il est possible de calculer son Phi :

$$\phi(n) = (p - 1) * (q - 1) = 18648$$

### 2.3 Calcul de la clé privée $a$

Le calcul de la clé privée ( $a$ ) s'obtient dorénavant facilement via l'utilisation de l'algorithme d'Euclide étendue, en effet :

$$a = b^{-1}[\phi(n)]$$

On trouve alors

$$a = 5797$$

### 2.4 Déchiffrage d'un texte donné

Le texte suivant était donné :

12423 11524 7243 7459 14303 6127 10964 16399

Chaque groupe de nombres représentaient le chiffage RSA d'un nombre représentant le triplet de caractères ASCII, codés selon la formule suivante :

$$n = \text{char1} * 26^2 + \text{char2} * 26^1 + \text{char3} * 26^0$$

où char1, char2, char3 représentaient une position du caractère dans l'alphabet ; ainsi si la première lettre du triplet était un A, la valeur de char1 aurait été 0, si B, 1, etc ...

Une fonction de calcul des triplets de lettres en fonction du nombre donné a donc été rajouté :

```
void print_letters(unsigned long num)
{
    unsigned long first = 0;
    unsigned long secnd = 0;
    unsigned long third = 0;

    unsigned long temp = num;
    /* 26^2 = 676 */
    first = (unsigned long) num / 676;
    temp = temp % 676;

    secnd = (unsigned long) temp / 26;
    temp = temp % 26;

    third = (unsigned long) temp % 26;

    printf("%c%c%c",
    (char) first + 'A',
    (char) secnd + 'A',
    (char) third + 'A');

    return;
}
```

A partir d'un nombre num donné en argument, les triplets de lettres correspondants sont affichés sur la sortie standard.

Nous pouvons maintenant déchiffrer les différents nombres donnés et en afficher les caractères correspondants. Soit un nombre  $x$  chiffré donné. calculons son “équivalent” déchiffré  $y$ , à l’aide des fonctions exposées dans la première partie :

$$y = \text{fast\_expo}(x, b, n);$$

Le déchiffrement par notre programme principal de la séquence codée donnée donne ainsi :

```
descrambling 12423 : 5438, -> IBE
descrambling 11524 : 1364, -> CAM
descrambling 7243 : 2925, -> EIN
descrambling 7459 : 14571, -> VOL
descrambling 14303 : 14303, -> VED
descrambling 6127 : 5746, -> INA
descrambling 10964 : 8805, -> NAR
descrambling 16399 : 4588, -> GUM
```

Soit la phrase “I BECAME INVOLVED IN AN ARGUM”.

## 2.5 Re-chiffrement des données

Afin de s’assurer de la validité de nos algorithmes, il était demandé de rechiffrer le message déchiffré et de s’assurer de la cohérence avec le message chiffré donné.

Le rechiffrement s’obtient de la manière suivante : soit  $x$ , le nombre déchiffré,  $y$  le nombre chiffré via le système RSA :

$$y = \text{fast\_expo}(x, b, n)$$

Après rechiffrement, on obtient la séquence suivante :

```
rescrambling 5438 : 12423
rescrambling 1364 : 11524
rescrambling 2925 : 7243
rescrambling 14571 : 7459
rescrambling 14303 : 14303
rescrambling 5746 : 6127
rescrambling 8805 : 10964
rescrambling 4588 : 16399
```

Ce qui est cohérent avec le message chiffré donné.

## Conclusion

Ces séances de travaux pratiques ont permis de mettre en application facilement les principes fondamentaux du système RSA. Même si le système utilisé ici était simplifié - on utilisait des entiers longs, alors que la mise en application réelle de RSA de la vie courante peut mettre en jeu des nombres de 1024 bits voire plus -, ces séances ont permis d'appréhender la logique mathématique nécessaire à l'implémentation de ce système couramment utilisé dans la cryptographie moderne.

## Annexes

### Code source de util.c

```
/*
 * util.c : fonctions utilisées tout au long
 *          de la séance de travaux pratiques
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "util.h"

/*
 * Approche déterministe permettant de savoir si
 * un nombre est premier ou non.
 * Au delà de 5 ou 6 chiffres, le temps de traitement
 * devient pharaonique.
 *
 * En pratique RSA ne s'assure pas à 100 % de la primalité,
 * il s'en assure avec une probabilité très faible de se
 * tromper ...
 *
 */
int is_prime(unsigned long num)
{
    unsigned long sqrtnm = (unsigned long) sqrt (num);
    unsigned long i;

    if (num <= 2)
        return TRUE;

    if ((num % 2) == 0)
        return FALSE;

    for (i = 3; i <= sqrtnm; i++)
        if ((num % i) == 0)
            return FALSE;
}
```

```
    return TRUE;

}

/*
 * Exponentiation rapide
 *
 */
unsigned long fast_expo (unsigned long a,
                        unsigned long x,
                        unsigned long m)
{
    unsigned long tmp;
    unsigned long r;
    int nbits;
    int i;

    nbits = 0;

    tmp = 1;
    while (x > tmp)
    {
        nbits ++;
        tmp <<= 1;
    }
    if (bit(x,nbits-1) == 1)
        r = a;
    else
        r = 1;

    i = nbits - 2;

    while ( i >= 0)
    {
        r = ( r * r ) % m;
        if (bit(x,i) == 1)
    {
        r = (r*a) % m;
    }
        i --;
    }
}
```

```
    return r ;
}

/*
 * Decomposition en facteurs premiers
 *
 *
 */
unsigned long * decomp_factor (unsigned long n , int *taille)
{
    unsigned long *tab = NULL;
    unsigned long d;
    int t = 0;

    while (n > 1)
        {
            /* on a trouvé un facteur premier */
            if (is_prime(n) == TRUE)
        {
            t++;
            tab = (unsigned long *) realloc (tab , t * sizeof (unsigned long));
            tab [t -1] = n;
            n=1;
        }

            /* 2 est facteur premier */
            if ((n%2) == 0)
        {
            t++;
            tab = (unsigned long *) realloc (tab , t * sizeof (unsigned long));
            tab [t -1] = 2;
            n = n / 2;
        }

            else
        {
            /* parcours des diviseurs */
            for (d = 3; (d*d) < n; d = d+2)
                {
                    /* si un diviseur est trouvé et qu'il est premier */
                    if (((n%d) == 0) && is_prime(d))
                {
                    t++;
```

```
    tab = (unsigned long *) realloc (tab , t * sizeof (unsigned long));
    tab[t-1] = d;
    n = n/d;
}
    }
}
    }
    *taille = t;

    return tab;
}
```

```
unsigned long pgcd (unsigned long a, unsigned long b)
{
    unsigned long c;

    while (b != 0)
        {
            c = a % b;
            a = b;
            b = c;
        }

    return a ;
}
```

```
unsigned long phi(unsigned long num)
{
    unsigned long result = 0;

    int size = 0;
    unsigned long * decompo = NULL;

    decompo = decomp_factor (num, &size);

    if ((decompo == NULL) || (size != 2))
    {
        printf("- Err in phi() !\n");
        goto end;
    }
```

```
}

/* else */

result = (decompo[0]-1) * (decompo[1]-1);

end:

if (decompo != NULL)
free (decompo);

return result;
}

void euclide (unsigned long a,
             unsigned long b,
             unsigned long *x ,
             unsigned long *y)
{
  unsigned long p = 1 , q = 0;
  unsigned long r = 0 , s = 1;

  unsigned long c , quotient, nouveau_r, nouveau_s;

  while (b != 0)
  {
    c = a % b;
    quotient = a / b ;
    a = b;
    b = c;
    nouveau_r = p - quotient * r;
    nouveau_s = q - quotient * s;
    p = r;
    q = s;
    r = nouveau_r;
    s = nouveau_s;
  }
  *x = p;
  *y = q;
}
```

```
/*
 * Fonction spécifique au codage des chaînes de caractères
 * utilisées au cours du TP
 *
 * Envoie sur la sortie standard le triplet de lettres
 * associé au nombre num.
 */

void print_letters(unsigned long num)
{
    unsigned long first = 0;
    unsigned long secnd = 0;
    unsigned long third = 0;

    unsigned long temp = num;
    /* 26^2 = 676 */
    first = (unsigned long) num / 676;
    temp = temp % 676;

    secnd = (unsigned long) temp / 26;
    temp = temp % 26;

    third = (unsigned long) temp % 26;

    printf("%c%c%c",
    (char) first + 'A',
    (char) secnd + 'A',
    (char) third + 'A');

    return;
}
```

**code source de util.h**

```
#ifndef __UTIL_H
#define __UTIL_H

#define TRUE 1
#define FALSE 0

#define bit(x,i) (((x) & (1 << (i))) ? 1 : 0)

int is_prime(unsigned long i);

unsigned long fast_expo (unsigned long a,
                        unsigned long x,
                        unsigned long m);

unsigned long factor(unsigned long num);

unsigned long * decomp_factor (unsigned long n, int * taille);

unsigned long pgcd(unsigned long a, unsigned long b);

void euclide (unsigned long a,
             unsigned long b,
             unsigned long *x,
             unsigned long *y);

unsigned long phi(unsigned long num);

void print_letters (unsigned long num);

#endif
```

**Code source de main.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include <math.h>

int main (int argc, char ** argv)
{
    unsigned long n = 18923;
    unsigned long b = 1261;

    unsigned long p = 0, q = 0;

    unsigned long a;

    int size = 0;
    unsigned long * decompo = NULL;
    unsigned long y;
    unsigned long ciphredtxt[] = { 12423,
    11524,
    7243,
    7459,
    14303,
    6127,
    10964,
    16399};
    unsigned long unciphredtxt[8];
    int currc = 0;
    /** QUESTION 1 **/
    printf("Question 1 :\n");

    decompo = decomp_factor (n, &size);

    if (size == 2)
    {
        p = decompo[0];
        q = decompo[1];
        printf("\tp = %ld\n\tq = %ld\n", p, q);
    }
}
```

```
printf("\tPhi(n) = %ld\n", phi(n));
euclide(b, phi(n), &a, &y);

printf("\ta = %ld\n", a);

if (decompo != NULL)
    free (decompo);

/** QUESTION 2 **/

printf("\nQuestion 2 :\n");

for (currc = 0 ; currc < 8; currc++)
{
    printf("\tdescrambling %ld :\t", cipheredtxt[currc]);
    uncipheredtxt[currc] = fast_expo(cipheredtxt[currc], a, n);
    printf("%ld,\t->\t", uncipheredtxt[currc]);
    print_letters(uncipheredtxt[currc]);
    printf("\n");
}

/** QUESTION 3 **/
printf("\nQuestion 3 :\n");

for (currc = 0 ; currc < 8; currc++)
{
    printf("\trescrambling %ld :\t", uncipheredtxt[currc]);
    printf("%ld\n", fast_expo(uncipheredtxt[currc], b, n));

}
printf("\n");

return 0;
}
```

**Code source du Makefile**

```
tp: main.o util.o
gcc -Wall -lm -o $@ $^
util.o: util.c
gcc -Wall -c $^
main.o: main.c
gcc -Wall -c $^
clean:
rm -f *~ *.o tp
```

**Sortie de l'exécution du binaire obtenu**

```
pmauduit@pc-gi-192:~/MI51/tp1$ ./tp
```

```
Question 1 :
```

```
    p = 127
    q = 149
    Phi(n) = 18648
    a = 5797
```

```
Question 2 :
```

```
descrambling 12423 :    5438,  ->    IBE
descrambling 11524 :    1364,  ->    CAM
descrambling 7243  :    2925,  ->    EIN
descrambling 7459  :   14571,  ->    VOL
descrambling 14303 :   14303,  ->    VED
descrambling 6127  :    5746,  ->    INA
descrambling 10964 :    8805,  ->    NAR
descrambling 16399 :    4588,  ->    GUM
```

```
Question 3 :
```

```
rescrambling 5438 :    12423
rescrambling 1364 :    11524
rescrambling 2925 :    7243
rescrambling 14571 :   7459
rescrambling 14303 :   14303
rescrambling 5746 :    6127
rescrambling 8805 :   10964
rescrambling 4588 :   16399
```