# MI51
## Cryptographie RSA

Compte-rendu des TP 1 et 2

Thomas Petazzoni – Julien Rosener

5 mai 2004

# Introduction

Les TP1 et TP2 de MI51 ont permis de réaliser un chiffrement et un déchiffrement à l'aide de l'algorithme cryptographique *RSA*. Le TP1 a consisté à l'écriture de diverses fonctions utiles au déroulement de l'algorithme, tandis que le TP2 a permis d'implanter l'algorithme en lui même.

L'implémentation des différents algorithmes est réalisée en C standard.

# 1 Fonctions utiles

## 1.1 Test de primalité

La première fonction utile est le test de primalité. Cette première fonction, appellée `is_prime` prend en paramètre un entier positif, et retourne 0 si cet entier n'est pas premier ou 1 si il est premier.

Listing 1 – Test de primalité

```
1  int is_prime (int n)
2  {
3    if(n <= 2)
4      return 1;
5
6    if (n%2 == 0)
7      return 0;
8    else
9      {
10       int d;
11
12       for (d = 3 ; (d*d) <n ; d = d+2)
13         {
14           if (n%d == 0)
15             return 0;
16         }
```

```
17
18          return 1;
19      }
20  }
```

**Lignes 3 et 4** Les nombres 1 et 2 sont considérés comme premiers, on élimine directement ces cas.

**Lignes 6 et 7** Les nombres pairs ne sont pas premiers. Ce test permet d'éliminer 50% des cas.

**Lignes 12 à 17** On teste tous les nombres impairs dans l'intervalle $[3; \sqrt{n}]$ pour savoir s'ils divisent ou non $n$. Si c'est le cas, alors $n$ n'est pas premier et on retourne 0.

**Ligne 18** Aucun diviseur n'a été trouvé pour $n$, alors on retourne 1 puisqu'il est premier.

## 1.2 Décomposition en facteurs premiers

La seconde fonction utile avait pour objet de décomposer un nombre en ses facteurs premiers. La méthode utilisée est la recherche brutale, qui sera suffisante pour des petits nombres, mais clairement inefficace pour des grands nombres.

La fonction prend en paramètre l'entier positif à décomposer en facteurs premiers ainsi qu'un pointeur vers un entier dans lequel le nombre de facteurs premiers est retourné. La fonction retourne le tableau des facteurs premiers. Ce tableau étant alloué dans le corps de la fonction, il conviendra de le librérer ultérieurement par un appel à la fonction `free()`.

Listing 2 – Décomposition en facteurs premiers

```
1   int* decompose(int n, int* taille)
2   {
3     int *tab = NULL;
4     int t = 0;
5
6     while (n > 1)
7       {
8         if (is_prime(n) == 1)
9           {
10            t = t + 1;
11            tab = (int*) realloc (tab, t * sizeof(int));
12            tab[t-1]=n;
13            n=1;
14          }
15
16        if (n%2 == 0)
17          {
18            t = t + 1;
19            tab = (int*) realloc (tab, t * sizeof(int));
20            tab[t-1]=2;
21            n=n/2;
22          }
23        else
24          {
25            int d;
26
27            for (d = 3; (d*d) < n ; d = d+2)
28              {
```

```
29                  if(n%d == 0)
30                    {
31                      t = t + 1;
32                      tab = (int*) realloc (tab, t * sizeof(int));
33                      tab[t-1]=d;
34                      n=n/d;
35                    }
36                }
37            }
38        }
39
40    *taille = t;
41
42    return tab;
43  }
```

**Ligne 6** Tant que la décomposition n'est pas terminée, on cherche les facteurs premiers.

**Ligne 8 à 14** Si le nombre est premier, alors il n'a pas de facteurs premiers : il est lui même son facteur premier. On augmente donc la taille du tableau (ligne 10), on réalloue le tableau (ligne 11), on enregistre la valeur à la fin du tableau (ligne 12), et on indique que l'entier vaut maintenant 1 (ligne 13), ce qui aura pour effet d'arrêter les itérations.

**Lignes 16 à 22** Si le nombre est pair, alors 2 est un facteur premier. On augmente donc la taille du tableau (ligne 18), on réalloue le tableau (ligne 19), on enregistre 2 à la fin du tableau (ligne 20), puis on recalcule l'entier (ligne 21).

**Lignes 23 à 36** Si le nombre n'est pas pair, alors on teste tous les nombres dans $[3; \sqrt{n}]$ à la recherche d'un diviseur. Si l'on en trouve un, alors on augmente la taille du tableau (ligne 31), on réallouée le tableau (ligne 32), on enregistre la valeur à la fin du tableau (ligne 33) puis on remet à jour l'entier à décomposer (ligne 34).

**Ligne 40** On retourne la taille du tableau dans l'entier pointé par `taille`.

**Ligne 42** On retourne l'adresse du tableau.

## 1.3 Calcul du PGCD

La troisième fonction utile doit calculer le PGCD de deux nombres entiers positifs. Elle prend ses deux nombres en paramètres et retourne leur PGCD.

Listing 3 – Calcul du PGCD

```
1   int pgcd (int a, int b)
2   {
3     while (b != 0)
4       {
5         int c = a % b;
6         a = b;
7         b = c;
8       }
9
10    return a;
11  }
```

## 1.4 Algorithme d'Euclide étendu

L'algorithme d'Euclide étendu permet de trouver deux nombres entiers $x$ et $y$ tels que $ax + by = pgcd(a, b)$. Dans le cas où $pgcd(a, b) = 1$, cela est particulièrement intéressant, car cela permet de trouver l'inverse d'un nombre modulo un autre nombre. Si on lui passe $a$ est le nombre à inverser, $b$ le modulo, alors le $x$ trouvé sera l'inverse de $a$. Ceci est utilisé pour déterminer la clé de décryptage à partir de la clé de cryptage RSA et de $\phi(n)$.

Listing 4 – Calcul du PGCD

```
1  void euclide (int a, int b, int *x, int *y)
2  {
3    int p = 1, q = 0;
4    int r = 0, s = 1;
5
6    int c, quotient, nouveau_r, nouveau_s;
7
8    while (b != 0)
9      {
10       c = a % b;
11       quotient = a / b;
12       a = b;
13       b = c;
14
15       nouveau_r = p - quotient * r;
16       nouveau_s = q - quotient * s;
17
18       p = r;
19       q = s;
20
21       r = nouveau_r;
22       s = nouveau_s;
23     }
24
25   *x = p;
26   *y = q;
27 }
```

TODO : ajouter explications algo d'euclide généralisé

## 1.5 Exponentiation rapide

L'algorithme de cryptage *RSA* nécessite une autre fonctionnalité utile : l'exponentiation rapide modulo un nombre. En effet, *RSA* repose sur des calculs du type $a^x modm$, avec b pouvant être grand. Au lieu de brutalement calculer $a^x$ puis de calculer le modulo, l'algorithme va permettre de faire le modulo à chaque étape du calcul, afin de travailler sur des nombres de tailles raisonnables comparés à $a$ et $x$.

On utilise pour cela un algorithme d'exponentiation qui part de la notation en binaire de l'exposant pour calculer $a^x modm$. En partant du bit de poids fort, l'algorithme va pour chaque bit prendre $r = r^2 modm$, et si le bit est à 1 va prendre $r = r * a \ mod \ m$.

Listing 5 – Exponentiation rapide

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

```
2
3  int expo(int a, int x, int m)
4  {
5    int nbits;
6    int tmp;
7    int i;
8    int r;
9
10   nbits = 0;
11   tmp   = 1;
12   while (x > tmp)
13     {
14       nbits++;
15       tmp <<= 1;
16     }
17
18   if(bit(x, nbits-1) == 1)
19     r = a;
20   else
21     r = 1;
22
23   i = nbits-2;
24   while(i >= 0)
25     {
26       r = (r * r) % m;
27
28       if(bit(x, i) == 1)
29         {
30           r = (r * a) % m;
31         }
32
33       i--;
34     }
35
36   return r;
37 }
```

**Ligne 1** La macro `bit` permet de savoir si le bit $i$ dans $x$ est à 1 ou à 0.

**Lignes 12 à 16** Calcul du nombre de bits de l'exposant, c'est à dire de la position du bit de poids fort.

**Lignes 18 à 21** Si le bit de poids fort est à 1, alors le $r$ initial vaut $a$, sinon il vaut 1.

**Lignes 23 à 34** Pour tous les autres bits, du plus fort au plus faible, on prend $r = r^2 modm$ et si le bit est à 1, on prend $r = r * amodm$.

**Ligne 36** On retourne le résultat.

# 2 Décryptage *RSA*

La deuxième séance de TP consistait à réaliser le décryptage d'un message codé selon *RSA*. Nous disposions de $n$ et de $b$, la clé de cryptage.

## 2.1 Calcul de $p$, $q$, $\phi(n)$ et de $a$

Afin de trouver $p$ et $q$ tels que $n = p * q$, nous avons utilisé la fonction `decompose`, vue en section 1.2. La méthode utilisée est donc une recherche brute de la décomposition en facteurs premiers, envisageable sur le $n$ choisi, mais pas sur les $n$ utilisés en pratique dans la cryptographie *RSA*.

Ensuite, nous calcuons $\phi(n)$, tel que $\phi(n) = (p-1)(q-1)$.

Enfin, nous calculons $a$, la clé de décryptage, qui est égale à $b^{-1}\ mod\ \phi(n)$. Pour cela, nous utilisons l'algorithme d'Euclide généralisé (voir 1.4). En effet, l'algorithme d'Euclide permet de trouver un couple d'entiers $x$ et $y$ tels que :

$$r_0 * x + r_1 * y = pgcd(r_0, r_1)$$

Ici, on posera $r_0 = b$ et $r_1 = \phi(n)$, on a donc

$$pgcd(r_0, r_1) = pgcd(b, \phi(n)) = 1$$

L'équation devient donc :

$$b * x + \phi(n) * y = 1$$

Si on prend les deux cotés de l'équation *modulo* $\phi(n)$, on obtient :

$$(b * x)\ mod\ \phi(n) + (\phi(n) * y)\ mod\ \phi(n) = 1\ mod\ \phi(n)$$

$\phi(n) * y\ mod\ \phi(n)$ étant égal à 0, on arrive à

$$(b * x) = 1\ mod\ \phi(n)$$

L'algorithme d'Euclide généralisé permet donc de calculer l'inverse d'un nombre, modulo un autre.

Le code source permettant de réaliser ces différentes opérations est le suivant :

Listing 6 – Calcul des données

```
1  int n = 18923;
2  int b = 1261;
3  int taille, phi, p, q, a, tmp;
4  int *tab;
5
6  tab = decompose(n, & taille);
7
8  p = tab[0];
9  q = tab[1];
10
11 phi = (p - 1) * (q - 1);
12
13 euclide(b, phi, & a, & tmp);
```

**Lignes 1 et 2** Définition de $n$ et $b$, qui sont des données publiques et connues de tous.

**Ligne 6** Décomposition en facteurs premiers de $n$, via la fonction `decompose`.

**Lignes 8 et 9** $n$ s'est décomposé en deux facteurs premiers, $p$ et $q$, qu'on peut trouver à l'indice 0 et à l'indice 1 du tableau `tab`.

**Ligne 11** Calcul de $\phi(n)$

**Ligne 13** Utilisation de l'algorithme d'Euclide généralisé pour calculer la clé de décryptage $a$.

## 2.2 Codage et décodage du texte

Le texte du message a été codé d'une manière spécifique avant d'être crypté avec l'algorithme *RSA*. A chaque lettre a été associée un nombre dans $[0; 25]$. Ensuite, le texte était découpé en groupes de 3 lettres, transformés en nombres de la manière suivante :

$$DOG \longrightarrow 3 * 26^2 + 14 * 26^1 + 6 * 26^0 = 2398$$

Nous avons tout d'abord une fonction pour réaliser le décodage d'un nombre en ses trois lettres correspondantes. L'utilisateur doit passer le nombre à décoder et un pointeur vers une zone de texte de 4 caractères (3 lettres + le zéro terminal). Ensuite, on procède par divisions successives par les puissances de 26 pour récupérer les indices de chaque lettre.

Listing 7 – Décodage

```
1  void integer2text(int n, char *txt)
2  {
3    int tmp;
4
5    tmp = n / (26*26);
6    txt[0] = tmp + 'A';
7
8    n %= (26 * 26);
9    tmp = n / 26;
10   txt[1] = tmp + 'A';
11
12   n %= 26;
13   txt[2] = n + 'A';
14
15   txt[3] = 0;
16 }
```

**Ligne 5** Calcul de l'indice de la lettre de rang 2, en divisant $n$ par $26^2$.

**Ligne 6** Calcul de la première lettre du texte en ajoutant à l'indice trouvé ligne 5 le code ASCII de la première majuscule (`A`).

**Ligne 8** Mise à jour de $n$ pour la prochaine étape.

**Ligne 9** Calcul de l'indice de la lettre de rang 1, en divisant le $n$ restant par 26.

**Ligne 10** Calcul de la deuxième lettre.

**Ligne 12** Mise à jour de $n$

**Ligne 13** Calcul de la troisième lettre

**Ligne 15** Ajout du zéro terminal

Pour le codage du texte, l'opération est plus simple. L'utilisateur passe un pointeur vers le texte de 3 lettres à coder, et la fonction `text2integer` retourne l'entier correspondant. Pour chaque lettre, on soustrait le code ASCII de `A` pour récupérer un nombre dans l'intervalle $[0; 25]$. Ensuite on multiplie chaque nombre par la puissance de 26 correspondante.

Listing 8 – Décodage

```
1  int text2integer(char *txt)
2  {
3    int tmp = 0;
4
5    tmp =
6      (txt[0] - 'A') * (26*26) +
7      (txt[1] - 'A') * 26 +
8      (txt[2] - 'A');
9
10   return tmp;
11 }
```

## 2.3 Fonction de décryptage

L'énoncé du TP donne un texte chiffré qu'il faut déchiffrer à l'aide des informations publiques. Le texte chiffré est le suivant :

```
12423 11524 7243 7459 14303 6127 10964 16399
```

que nous avons défini dans notre code source C de la façon suivante :

Listing 9 – Message crypté

```
1  int message_crypted[] = { 12423, 11524, 7243, 7459,
2                            14303, 6127, 10964, 16399 };
```

Nous avons ensuite écrit la fonction de décryptage, qui utilise la formule de décryptage *RSA* :

$$M = C^a \ mod \ n$$

Avec

- $M$ message décrypté
- $C$ message crypté
- $a$ clé de décryptage
- $n$ Le grand entier *RSA*

Pour effectuer cette opération sur l'ensemble d'un message codé de la façon exposée précédemment, nous avons écrit la fonction `decrypt`. Celle-ci prend en argument un tableau d'entiers contenant le message crypté, le nombre d'entiers du message, la clé de décryptage et l'entier *RSA* $n$.

Listing 10 – Décryptage *RSA*

```
1  char *decrypt(int *crypted, int size, int a, int n)
2  {
```

```
3   char *txt;
4   char *p;
5   int m;
6   int i;
7
8   txt = malloc((size * 4) * sizeof(char));
9
10  for (i = 0, p = txt; i < size ; i++, p += 4)
11    {
12      m = expo(crypted[i], a, n);
13      integer2text(m, p);
14      p[3] = ' ';
15    }
16
17  txt[size*4-1] = 0;
18
19  return txt;
20 }
```

**Ligne 8** Allocation de l'espace mémoire nécessaire au stockage du message décrypté. Pour chaque élément du tableau d'entiers, on aura besoin de 4 caractères : 3 pour les lettres, un pour l'espace. On alloue donc `size * 4` octets pour le stockage du message décrypté.

**Ligne 10** On boucle pour chaque entier du message crypté. Dans le même temps, on fait avancer un pointeur `p` qui indiquera où l'on doit stocker le prochain texte décrypté.

**Ligne 12** On applique la formule de décryptage *RSA* en calculant $C^a \ mod \ n$, avec $C$ message crypté, $a$ clé de décryptage et $n$ entier *RSA*.

**Ligne 13** Maintenant que nous avons obtenu l'entier décrypté, il faut décoder cet entier pour obtenir le texte correspondant.

**Ligne 14** Ajout de l'espace après le bloc de 3 lettres qui vient d'être décodé.

**Ligne 17** Une fois l'ensemble du texte décrypté, on ajoute le zéro terminal à la fin de la chaîne.

Nous appelons cette fonction `decrypt` de la manière suivante :

Listing 11 – Appel à `decrypt`

```
1   message = decrypt(message_crypted,
2                     sizeof(message_crypted)/sizeof(int),
3                     a, n);
```

Grâce à cette fonction, nous déterminons le message décrypté :

```
IBE CAM EIN VOL VED INA NAR GUM
```