L'ALGORITHME RSA : SYNTHÈSE ET IMPLANTATION

par
Sébastien Bolduc
Stéphane Drouin

présenté à
M. Marc Parizeau

Mathématiques discrètes du génie informatique
IFT-19496

Université Laval
Le 10 décembre 1997

Introduction.

À l'époque du libre accès à l'information, les mécanismes de protection de l'information privée sont devenus nécessaires. L'algorithme d'encryption développé par Ron Rivest, Adi Shamir et Leonard Adleman (RSA) est particulièrement intéressant, car il présente un avantage de taille : l'utilisation de deux clés, une clé publique et une clé privée. L'inconvénient d'avoir à transmettre l'unique clé privée à notre interlocuteur est ainsi évité. L'algorithme RSA est donc l'objet du présent rapport.

RSA fonctionne avec deux clés : une privée et une publique. Une fois que les clés sont obtenues, le processus d'encryption/décryption est très simple ! Pour encrypter le message M, il suffit d'utiliser la clé publique de faire l'opération suivante :

$$c = M^e \bmod n$$

et pour décrypter, on calcule:

$$M = c^d \bmod n$$

Les nombres (e, n) constituent la clé publique et les nombres (d, n) constituent la clé privée. L'obtention de ces nombres constitue le défi principal de l'implantation de l'algorithme RSA. C'est ce dont il sera question dans la première partie de ce rapport. Nous préciserons les relations mathématiques qui relient les clés et nous introduirons les théorèmes qui faciliteront leur compréhension. Dans un deuxième temps, nous allons préciser les détails de notre implantation, nous allons décrire nos approches de solutions. Plus particulièrement, nous décrirons nos solutions pour l'exponentielle modulo, le plus grand diviseur commun, l'inverse multiplicatif, la génération de nombres premiers et l'obtention des clés.

I. L'algorithme RSA : son fonctionnement.

Pour encrypter avec RSA, chaque participant doit d'abord créer son jeu de clés. La clé privée de chacun est gardée secrète, mais la clé publique est laissée à la disposition des autres membres du réseau. Chacun peut alors encrypter un message avec la clé publique d'un autre. Cette dernière personne sera alors la seule à pouvoir le décrypter. Un participant peut aussi signer un message avec sa clé privée. Tous les autres pourront alors vérifier la signature avec la clé publique du participant en question.

On voit donc que la fonction caractérisée par chacune des clés doit être l'inverse de l'autre afin que les messages encryptés soient décryptables ! Ces fonctions doivent donc être bijectives. Soit un message M :

$M = S(P(M))$ (1) S(M) : fonction clé secrète
$M = P(S(M))$ (2) P(M) : fonction clé publique

L'algorithme RSA est basé sur le fait qu'il est beaucoup plus aisé de générer deux grands nombres premiers que de les retrouver en factorisant leur produit. Les clés RSA se calculent de la manière suivante :

1. Générer deux grands nombres premiers : p et q.
2. Poser $n = pq$. n est la constante modulo.
3. Choisir e tel qu'il est premier par rapport au produit $(p-1)(q-1)$. C'est-à-dire que leur plus grand diviseur commun est 1. De plus, e doit être choisi tel que $e < n$. e est la composante publique.
4. Trouver d tel que $de \equiv 1 (\mathrm{mod}(p-q)(q-1))$. On dit que d est l'inverse multiplicatif de e. d est la composante privée.
5. $P = (e, n)$ est la clé publique RSA.
6. $S = (d, n)$ est la clé privée RSA.

Dès que les clés sont obtenues, les nombres p et q peuvent être détruits.

Dès lors, les fonctions d'encryption et de décryption sont les suivantes :

$$P(M) = M^e \bmod n \qquad (3)$$
$$P(C) = M^d \bmod n \qquad (4)$$

Théorème de la justesse de RSA [1] (p. 834)

Les équations (3) et (4) sont les fonctions mutuellement inverses satisfaisant aux équations (1) et (2).

Preuve :

La preuve est faite dans [1] en page 834.

II. L'implantation de l'algorithme RSA.

Cette partie du rapport présente, dans un premier temps, nos solutions aux différents problèmes posés par l'encryption avec RSA. Dans un deuxième temps, le fonctionnement de notre programme sera expliqué.

L'exponentielle modulo

L'exponentielle modulo est un calcul de la forme $a^b \bmod n$ avec $a,b \geq 0$ et $n > 0$. Cette fonction est utilisée dans l'encryption et la décryption des messages ainsi que dans la génération des nombres premiers.

La méthode de la mise au carré répétée [1] (p. 829)

MODULAR-EXPONENTIATION (a,b,n)

```
1     c ← 0
2     d ← 1
3     soit (b_k, b_k-1, ... , b_0) la représentation binaire de b
4     for i ← k downto 0
5           do c ← 2c
6           d ← (d.d) mod n
7           if b_i = 1
8                 then c ← c + 1
9                 d ← (d.a) mod n
10    return d
```

À chaque itération, une de ces identités est utilisée :

$$a^{2c} \bmod n = \left(a^c\right)^2 \bmod n$$
$$a^{2c+1} \bmod n = a\left(a^c\right)^2 \bmod n$$

Dans notre programme, la fonction exponentielle modulo a été implanté telle que décrite dans cet algorithme dans :
`ExpMod(const EntPA &M,EntPA x,const EntPA &nb).`

Le plus grand diviseur commun (Noté GCD)

Le problème du plus grand diviseur commun consiste à trouver k maximum tel que :

$$k \in N^* \text{ et } a = kx,\ b = ky \text{ avec } x, y, a, b \in N^*$$

Pour trouver le plus grand diviseur commun de deux entiers positifs, nous utilisons l'algorithme d'Euclide. L'algorithme est basé sur le théorème suivant:

Théorème sur la forme récursive du GCD [1] (p. 809)

Pour tout entier strictement positif a et pour tout entier positif b, $\gcd(a,b) = \gcd(b, a \bmod b)$

Preuve :

La preuve est faite dans [1] en page 809.

L'algorithme d'Euclide [1] (p. 809)

EUCLID(a, b)

```
1    if b = 0
2        then return a
3        else return EUCLID(b,a mod b)
```

Dans notre programme, nous utilisons la fonction plus grand diviseur commun pour trouver la clé publique. Elle a été implanté telle que décrite dans cet algorithme dans :
`gcd(EntPA a, EntPA b).`

L'inverse multiplicatif

L'inverse multiplicatif est une procédure importante de l'algorithme d'encryption RSA puisqu'elle permet d'obtenir la clé privée. Cette dernière servira à décrypter un message reçu et encrypter par la clé publique qui a été émise.

Pour expliquer l'inverse multiplicatif, nous devons d'abord définir l'ensemble suivant, {0, 1, ..., p-1), où p est un nombre premier dans GF(p) (ensemble Galois de p éléments, nommé d'après le mathématicien). Aussi, il faut assumer que p est un nombre de simple précision pour

l'ordinateur sur lequel nous prévoyons d'utiliser l'inverse multiplicatif. GF(p) va donc former un ensemble sous ces conditions. Si a, b ε GF(p), alors :

$$(a+b)\bmod p = \begin{cases} a+b & \text{si } a+b < p \\ a+b-p & \text{si } a+b \geq p \end{cases}$$

$$(a-b)\bmod p = \begin{cases} a-b & \text{si } a-b \geq 0 \\ a-b+p & \text{si } a-b < 0 \end{cases}$$

(ab) mod p = r , ab = qp + r où 0 <= r < p

(a/b) mod p = (ab^{-1}) mod p = r, qui est l'unique restant lorsque ab^{-1} est divisé par p, ab^{-1} = qp + r, 0 <= r < p

b^{-1} est l'inverse multiplicatif de b dans GF(p). Pour chaque élément b dans GF(p), excepté zéro, il existe un unique élément appelé b^{-1} de sorte que bb^{-1} mod p = 1.

Un algorithme calculant l'inverse de b dans GF(p) est obtenu en généralisant l'algorithme d'Euclide pour le calcule du plus grand commun diviseur (gcd). Il est aussi possible de calculer deux autres x, y de manière à ce que ax + by = gcd(a, b). En ayant p comme nombre premier et b ε GF(p), alors gcd(p, b) = 1 (puisque les uniques diviseurs d'un nombre premier sont lui-même et 1) et la généralisation d'Euclide est réduite à trouver x, y pour que px + by = 1. Ceci implique que y est l'inverse multiplicatif de b mod p. Comme une image vos mille mots, voici donc l'algorithme:

```
procedure EXEUCLID(b, p)
  //b ε GF(p), p est un nombre premier. EXEUCLID est une fonction//
  //dont le résultat est un integer x de sorte que bx + kp = 1//
  //l'affirmation (e, f) ← (g, h) est//
  //interprétée comme e ← g;f <-- h//
  (c, d, x, y) ← (p, b, 0, 1) //initialisation//
  while d != 1 faire
    q ← c/d   //calcule le quotient//
    e ← c - d*q       //calcule le restant//
    w ← x - y*q
    (c, d, x, y) ← (d, e, y, w)
  repeat
  si y < 0 alors y ← y + p
  retourner(y)
fin EXEUCLID
```

Ici, la boucle **while** ne sera exécutée que $O(\log_{10} p)$ fois et cela est le temps de calcul pour l'*extended Euclid* et la division modulaire. Dans

notre programme, la fonction de l'inverse multiplicatif a été implantée telle que décrite dans cet algorithme dans :
`InvMult(const EntPA &a,const EntPA &nb).`

La génération de grands nombres premiers

Selon le théorème de Fermat [1] (p. 827), si n est premier, l'équation suivante est vérifiée :

$$a^{n-1} \equiv 1 (\mathrm{mod}\, n)$$

Cependant, si n satisfait l'équation, il n'est pas nécessairement un nombre premier. L'utilisation d'un test plus élaboré est donc nécessaire pour déterminer la nature de n!

Pour générer de grands nombres premiers, nous utilisons l'algorithme de Rabin-Miller [2]. Il s'agit tout d'abord de générer un nombre aléatoire (x) puis de le tester. On peut alors écrire :

$$x = 1 + 2^b \bmod m \qquad (1)$$

Le test se fait avec un autre nombre aléatoire (a) en utilisant l'équation :

$$z = a^m \bmod x \qquad (2)$$

Si dans un premier temps z est égal à 1 ou à x-1, le nombre est possiblement premier. Si ce n'est pas le cas, on poursuit le test avec l'équation :

$$z = z^2 \bmod x \qquad (3)$$

On refait le calcul jusqu'à ce que z = 1 auquel cas x n'est pas premier ou jusqu'à ce que z = x-1 auquel cas x est possiblement premier. Ce calcul se fait un maximum de b fois (b tiré de (1)). Si x n'est pas premier, on pose x = x + 2 et on reprend le test.

Le test avec les équations (2) et (3) est faux dans 25% des cas. En répétant la procédure de test avec un nouveau nombre aléatoire (a) n fois, l'algorithme Rabin-Miller générera un faux nombre premier dans $(25\%)^n$ des cas, ce qui est négligeable pour n suffisamment grand.

Dans notre programme, la fonction génération de grands nombres premiers a été divisée comme suit:
L'implantation des phases de génération et d'incrémentation du nombre à tester est dans :

```
GenerePremier(const EntPA &lim_inf, const EntPA &lim_sup)
```

L'implantation des équations (2) et (3) est dans :

```
TryPremier(const EntPA &x,const EntPA &m,const EntPA &b)
```

L'implantation de l'équation (1) est dans :

```
TestPremier(const EntPA &prime)
```

L'obtention des clés

Les clés sont générées suivant les restrictions exposées dans la première partie de l'article. Cependant, quelques précisions s'avèrent nécessaires. Tout d'abord, nous allons éclaircir les notions entourant la longueur des clés. La façon d'obtenir une constante modulo de n_c chiffre est de multiplier entre eux deux nombres premiers de n_c/2 chiffres. Pour éviter les ennuis, on se limite donc à des longueurs de clés paires. La clé publique est choisie telle qu'elle a un ou deux chiffres de moins que la constante modulo. Une deuxième contrainte sur la longueur des clés s'ajoute, car dans ces conditions, générer une constante modulo de 2 chiffres n'aurait pas de sens. n_c doit donc être supérieur ou égal à 4.

Dans notre programme, le code pour l'obtention des clés se trouve dans : `GenereClefs(int n_c)`.

Mode d'emploi

L'utilisation de notre programme est très simple via le menu qui vous est offert. Dans ce qui suit, nous vous donnerons un exemple d'utilisation de toutes les fonctions en plus d'expliquer leur fonctionnement en détail. Ceci vous permettra par la suite d'expérimenter à votre guise à l'aide du programme.

Lorsqu'il est démarré, le programme génère par défaut un jeu de clés de 20 chiffres.

La première option du menu est :

1 – Generer un nouveau jeu de cles

Comme elle le dit, cette option permet de générer un nouveau jeu de clés. Vous devez par la suite choisir la longueur des clés :

Creation d'un jeu de clefs...
Des clefs de combien de chiffres? 20
Generation d'une cle de 20 chiffres ……..

Pour l'exemple, nous avons créé des clés de 20 chiffres puisqu'elles ne sont pas longue à générer. Des clés de 100 chiffres prennent près de 15 minutes pour être générées. Vous pouvez voir vos clés grâce à l'option 6 :

6 – Afficher cles

Et les clés seront affichées :

Cle publique e : 8285745798804428653
Cle privee d : 8457407103067942417
Constante modulo : 16010153380401917521

Vous pouvez sauvegarder vos clés grâce à l'option 4 et les récupérer grâce à l'option 5 :

4 – Sauvegarder le jeu de cles
Sauvegarde d'un jeu de cles
5 – Recuperer un jeu de cles
Recuperation d'un jeu de cles

L'option 2 sert à encrypter un message:

2 – Encrypter un message
Encryption d'un message…
Message à encrypter : 67
Message encrypte : 8554374100993523751

Par la suite, lorsque vous faites appel à l'option 3, vous n'avez qu'à entrer le message encrypté:

3 – Decrypter un message
Decryption d'un message
Message a decrypter : 8554374100993523751
Message decrypter : 67

Et voilà ! Lorsque vous voulez quitter le programme, vous n'avez qu'à choisir l'option 0 :

0 - Quitter

Conclusion.

Les applications de l'encryption avec RSA sont nombreuses dans le transfert de données confidentielles bien sûr, mais aussi dans pour l'authentification. Par exemple, il est possible d'apposer sa « signature » sur un « chèque électronique » en utilisant RSA.

Dans la pratique, RSA est souvent utilisé dans un système d'encryption hybride. En effet, encrypter un message entier en utilisant RSA est beaucoup plus long que de l'encrypter avec une méthode utilisant une clé unique d'encryption/décryption. La confidentialité lors de la transmission de cette clé unique entre les participants est assurée en l'encryptant en utilisant la méthode RSA. Cette façon de faire allie la commodité des clés publiques et la rapidité des clés uniques.

Références.

[1] CORMEN Thomas, LEISERSON Charles E., RIVEST Ronald L. *Introduction to Algorithms*, The MIT Press, McGraw-Hill Book Company, 1990.

[2] HARTMAN M.J., RICHARDS K.W. (Page consultée le 25 novembre 1997). *Secure Network Communications*, [En ligne]. Adresse URL: http://xfactor.wpi.edu/Works/MQP/securenet/root/root.html

[3] HOROWITZ Ellis, SAHNI Sartaj. Fundamentals of computer algorithms, COMPUTER SCIENCE PRESS, INC, Maryland, 1978.

[4] RSA Data Security, Inc. (Page consultée le 25 novembre 1997). *RSA – Cryptography FAQ*, [En ligne]. Adresse URL: http://www.rsa.com/rsalabs/newfaq/