Projet de programmation

Cryptographie, le système RSA

Stéphanie GENDREAU et Emeline POINSIGNON

2009

Introduction

Le but de ce projet est de programmer deux fonctions réciproques, l'une permettant de crypter un message suivant la méthode RSA, et l'autre de le décrypter.

En 1977, Ron Rivest, Adi Shamir et Leonard Adleman ont inventé le tout premier système à clé publique : RSA. RSA est la première véritable méthode de chiffrement à clé publique (le fonctionnement de RSA sera expliqué en détail dans la prochaine section). La sécurité de RSA repose sur un problème mathématique très difficile à résoudre : la factorisation des entiers. RSA, parmi les méthodes cryptographiques à clé publique, demeure celle qui a été la plus utilisée. Notons qu'il existe d'autres méthodes à clé publique. Nommons, parmi celles-ci, DSA ainsi que les méthodes qui utilisent des courbes elliptiques (méthodes qui exploitent aussi un autre mystère des mathématiques, comme RSA).

Nous allons dans un premier temps nous intéresser au fonctionnement de cette méthode, en regardant un cas particulier, puis nous nous concentrerons sur le programme.

1 Etude préliminaire

1.1 Quelques résultats mathématiques

Le système RSA est basé sur les propriétés du groupe $(\mathbb{Z}/n\mathbb{Z})^\times$, l'ensemble des éléments inversibles de $\mathbb{Z}/n\mathbb{Z}$. Nous allons donc donner quelques détails sur ce groupe, qui nous seront utiles dans la suite de ce projet.

Proposition 1. *k est inversible dans $\mathbb{Z}/n\mathbb{Z}$ si et seulement si k et n sont premiers entre eux.*

$$k \in (\mathbb{Z}/n\mathbb{Z})^\times \quad \Longleftrightarrow \quad pgcd(n,k) = 1$$

Démonstration.

$$\begin{aligned} & k \in (\mathbb{Z}/n\mathbb{Z})^\times \\ \Leftrightarrow \quad & \exists l \in (\mathbb{Z}/n\mathbb{Z}) \text{ tel que } k \cdot l = 1 \\ \Leftrightarrow \quad & \exists l \in \mathbb{Z}, \exists m \in \mathbb{Z} \text{ tel que } kl - 1 = nm. \\ \Leftrightarrow \quad & \exists l \in \mathbb{Z}, \exists m \in \mathbb{Z} \text{ tel que } kl - nm = 1. \\ \Leftrightarrow \quad & pgcd(k, n) = 1 \text{ d'après le théorème de Bézout.} \end{aligned}$$

□

On en déduit immédiatement, que le cardinal de $(\mathbb{Z}/n\mathbb{Z})^\times$ est égal à $\phi(n)$, où ϕ est la fonction d'Euler. En effet le cardinal de $(\mathbb{Z}/n\mathbb{Z})^\times$ correspond au nombre d'éléments premiers avec n.

Remarque. Dans le cas où n est premier, $(\mathbb{Z}/n\mathbb{Z})^\times = (\mathbb{Z}/n\mathbb{Z}) \setminus \{0\} = \{1, ..., n-1\}$, et donc $card((\mathbb{Z}/n\mathbb{Z})^\times) = n - 1$.

Il n'y a pas de résultat simple permettant de donner l'ordre des éléments de $(\mathbb{Z}/n\mathbb{Z})^\times$ et de déterminer s'il s'agit d'un groupe cyclique dans le cas général.

1.2 Fonctionnement du système RSA

Avant de se lancer dans le cryptage à proprement parler, il faut tout d'abord déterminer ce qu'on appelle les clés publique et privée, la première étant nécessaire à la personne désirant envoyer le message, et la deuxième à celle voulant le décrypter.

On commence par choisir deux entiers (p, q) premiers, sachant que plus ils sont grands, plus le cryptage est efficace. Pour comprendre le principe nous nous limiterons à

$$\begin{aligned} (p, q) &= (29, 37) \\ n = p * q &= 1073 \\ m = \phi(n) &= 1008 \end{aligned}$$

Le but est de trouver deux entiers e et d tels que pour tout α, β dans $\mathbb{N}$, on ait les deux égalités suivantes :

$$\begin{cases} \alpha^e = \beta \mod(n) \\ \beta^d = \alpha \mod(n) \end{cases}$$

Pour cela on prend un entier e tel que e et m soient premiers entre eux. On peut alors utiliser le théorème de Bézout qui nous dit qu'il existe alors u et v deux entiers tels que $eu + mv = 1$ et on pose $d = u$ (c'est donc l'inverse de u modulo m). Prenons ici

$$\begin{aligned} e &= 71 \\ d &= 1079 \end{aligned}$$

D'où les clés :

$$\begin{aligned} \text{clé publique} \quad & (e,n) = (71, 1073) \\ \text{clé privée} \quad & (d,n) = (1079, 1073) \end{aligned}$$

* Utilisons maintenant ces clés pour crypter le mot « Hello »
  - Transformation en code ascii (on affiche les zéros devant les nombres, afin de n'avoir que des codes de 3 chiffres) : H e l l o → 072 101 108 108 111
  - On sépare le code ascii en blocs ayant un chiffre de moins que n, en rajoutant des zéros à la fin si besoin (on ne fait pas obligatoirement apparaître les zéros figurant devant les nombres).

$$[72, 101, 108, 108, 111]$$

  - On encrypte chaque bloc grâce à la clé publique :

$$\begin{aligned} 72^{71} &= 943 \ \ mod(1073) \quad 101^{71} = 566 \ \ mod(1073) \\ 108^{71} &= 530 \ \ mod(1073) \quad 111^{71} = 111 \ \ mod(1073) \end{aligned}$$

Le message crypté est donc :

$$943566530530111$$

* Il s'agit maintenant de décrypter le message.
  - Il s'agit dans un premier temps de reconsituer des blocs ayant un chiffre de moins que n, afin de pouvoir utiliser la réciprocité du passage à la puissance e et d dans $(\mathbb{Z}/n\mathbb{Z})^{\times}$.

$$[943, 566, 530, 530, 111]$$

On décrypte chaque bloc grâce à la clé privée :

$$\begin{aligned} 943^{1079} &= 72 \ \ mod(1073) \quad 566^{1079} = 101 \ \ mod(1073) \\ 530^{1079} &= 108 \ \ mod(1073) \quad 111^{1079} = 111 \ \ mod(1073) \end{aligned}$$

  - Si jamais un des nombres otenus n'a pas un chiffre de moins que n, alors on rajoute autant de zéro que nécessaire devant :

$$072101108108111$$

  - On fait maintenant des blocs de longueur 3 (ne laissant pas appraitre les zéros devant les nombres), qu'on transforme en texte grâce au code ascii :

$$[72, 101, 108, 108, 111] \rightarrow [H, e, l, l, o]$$

Le message décrypté est donc :

$$\text{Hello}$$

2 Programme

Définissons dans un premier temps les fonctions générales qui nous serviront pour le cryptage et le décryptage.

2.1 Crible d'Eratosthène

Le principe du crible est très simple. Si on cherche les entiers premiers inférieurs où égaux à n, on prend l'ensemble des nombres inférieurs ou égaux à n. On commence par mettre 2 de coté, et on raye tous les multiples de 2. On recommence la même étape, en mettant de côté le premier entier non rayé, et en enlevant tous ses multiples. Chaque nombre étant mis de côté est premier. On s'arrête lorsque le nombre premier que l'on considère a son carré strictement supérieur à n, et tous les entiers restant sont premiers. En effet, si un entier non premier est strictement supérieur à $\sqrt{n}$ alors il a au moins un diviseur inférieur à $\sqrt{n}$ et aura donc déjà été rayé.

D'où le programme :

```
def crible(n) :
    L = range(2, n + 1)
    prem = [ ]
    while (L[0] * L[0]) <= n :
        prem.append(L[0])
        L = [x for x in L[1 :] if x%L[0] ! = 0]
    prem.extend(L)
    return prem
```

2.2 Puissance modulaire

Le but est de calculer a^n modulo m de la manière la plus efficace possible. Pour cela on réduit modulo m à chaque étape du calcul, et pas seulement une seule fois tout à la fin. En effet, en réduisant modulo m à chaque étape, on travaille avec des entiers beaucoup plus petits, et les calculs sont donc plus rapides. Pour cela, on introduit un entier p, et on vérifie immédiatement qu'à chaque étape de notre calcul, $p \cdot a^n$ est un invariant modulo m.

D'où le programme :

```
def puis_ mod(a, n, m) :
    p = 1
    while n > 0 :
        while (n%2 == 0) :
            a = (a * a)%m
            n = n/2
        p = (p * a)%m
        n = n − 1
    return p
```

2.3 Inverse modulaire

On cherche maintenant à calculer l'inverse d'un entier a modulo b. Le théorème de Bézout nous dit que si deux entiers a et b sont premiers entre eux, alors il existe deux entiers u et v tels que $au + bv = 1$.
Nous aurons donc $au = 1 \bmod(b)$, et donc u est l'inverse modulaire de a modulo b. L'algorithme d'Euclide nous permet de déterminer u et v en fonction de a et b. Ici on se limitera à calculer u, qui seul nous intéresse. L'algorithme d'Euclide est constitué d'une suite de divisions euclidiennes, et repose sur le fait que le pgcd de a et b est le même que celui de b et r, où r est le reste de la division euclidienne de a par b. En remontant l'algorithme d'Euclide on retrouve u et v.

Calculons l'inverse de a_0 modulo m_0. Regardons l'algorithme sous sa forme matricielle. On peut se limiter à des matrices de $M_4(\mathbb{Z})$. On initialise la première matrice à :

$$\begin{pmatrix} a_0 & 1 \\ m_0 & 0 \end{pmatrix}$$

On pose $(q, r) = (Ent(\frac{a}{m}), a\%m\ )$, et on définit la relation de récurrence suivante :

$$\begin{pmatrix} a & u \\ m & s \end{pmatrix} \Longrightarrow \begin{pmatrix} m & s \\ r & u - qs \end{pmatrix}$$

Tant que b (qui prend les valeurs des restes successifs de l'algorithme d'Euclide) n'est pas nul, on réitère le procédé, et on affirme qu'à chaque étape, $a = a_0 u + m_0 s$.

Démonstration. L'initialisation est évidente : $a_0 = a_0 \cdot 1 + m_0 \cdot 0$.
Supposons l'hypothèse de récurrence vrai à l'étape k, regardons à l'étape k+1 (on notera α_i l'élément α à l'étape i).

$$\begin{aligned} a_0 u_{k+1} + m_0 s_{k+1} &= a_0 \cdot (u_{k-1} - \frac{m_{k-2}}{m_{k-1}} \cdot s_{k-1}) + m_0 \cdot (s_{k-1} - \frac{m_{k-2}}{m_{k-1}} \cdot s_k) \\ &= a_{k-1} - \frac{m_{k-2}}{m_{k-1}} \cdot (a_0 s_{k-1} + m_0 s_k) \\ &= a_{k-1} - \frac{m_{k-2}}{m_{k-1}} \cdot (a_0 u_k + m_0 s_k) \\ &= a_{k-1} - \frac{m_{k-2}}{m_{k-1}} \cdot (a_k) \\ &= a_{k+1} \end{aligned}$$

La propriété est donc héréditaire. Elle est donc vrai quelque soit k, et à fortiori pour la dernière étape. On a donc à la dernière étape $a = a_0 u + m_0 v$. On voit aisément que a_0 est inversible si et seulement si a vaut 1, et dans ce cas l'inverse de a_0 modulo m_0 vaut u. Ceci conclut la preuve. □

On en déduit donc rapidement le programme suivant :

```
def inv_ mod(a0, m0) :
    (a, m, u, s) = (a0, m0, 1, 0)
    while m! = 0 :
        (q, r) = (a/m, a%m)
        (a, m, u, s) = (m, r, s, u − q ∗ s)
    if a == 1 :
        while u < 0 :
            u = u + m0
        return u
    else :
        return "a0 n'est pas inversible modulo m0."
```

2.4 Programme RSA

Il ne reste plus qu'à faire le programme, maintenant que les fonctions importantes ont été faites.

Nous commencerons donc par programmer le choix des clés publique et privée. Pour ceci on utilise la fonction *random.choice* qui permet de faire un choix aléatoire parmi un ensemble déterminé. Dans notre cas, étant donné qu'on veut des nombres premiers on prend donc *random.choice(crible(n))*, qui donne donc un nombre premier inférieur ou égal à n. A la vue des explications données dans le paragraphe concernant le fonctionnement du système RSA (1.2), il nous semble inutile de donner plus d'indications sur le programme suivant, qui est assez explicite.

```
def cles() :
    p = random.choice(crible(100))
    q = random.choice(crible(100))
    n = p*q
    m = (p-1)*(q-1)
    e = random.randint(10, 100)
    while inv_ mod(e, m) == "a0 n'est pas inversible modulo m0." :
        e = random.randint(10, 100)
    d = inv_ mod(e, m)
    return e, d, n
```

Pour finir, on constate que les étapes du cryptage et du décryptage sont réciproques les unes des autres. Par soucis de lisibilité, nous avons donc programmé les étapes individuellement, en définissant à chaque fois un couple de fonctions réciproques, dont l'une est utilisée dans le cryptage, et l'autre dans le décryptage. Les fonctions étant élémentaires, il nous semble juste nécessaire d'expliquer la commande *"".join(zfill(x, l) for x in L)*, le reste étant très clair dans le programme. Cette commande permet, dans une liste L, de rajouter des zéros devant le caractère x, de manière à n'obtenir que des blocs de longueur l.

On s'assure que les fonctions sont bien deux-à-deux réciproques, tant au niveau des opérations effectuées, qu'au niveau du type de caractères qu'on a en entrée et en sortie.