RSA

February 17, 2019

1 RSA

Marc Lorenzi
Juillet 2016

1.1 1 Introduction

1.1.1 1.1 Cryptographie à clés privées

Alice désire envoyer des messages à Bob. Ces messages devant rester confidentiels, ils vont être *cryptés*.

Un message est tout simplement un élément d'un certain ensemble $\mathcal{M}$. On appelle *clé de cryptage* toute application bijective (pour simplifier) de $\mathcal{M}$ vers lui-même. Ainsi, si Alice veut envoyer un message $m \in \mathcal{M}$ à Bob, ils choisissent une clé de cryptage f. Puis Alice calcule $m' = f(m)$ et expédie m', le *message crypté*, à Bob. Celui-ci applique alors la *clé de décryptage* f^{-1}, où f^{-1} est la fonction réciproque de f : il calcule simplement $f^{-1}(m') = m$, et retrouve ainsi le *message en clair*.

Bien évidemment, Bob peut aussi envoyer des messages à Alice en utilisant la clé f : Alice les décryptera grâce à la clé f^{-1}. Nous sommes en présence d'un système de cryptage *à clés symétriques* : les deux interlocuteurs cryptent (et décryptent) avec la même clé.

Le problème est évidemment pour Alice et Bob de s'entendre sur la clé f. La seule solution existant jusqu'à une époque récente (1976) était pour Alice et Bob de se communiquer secrètement la fonction f : le problème de la transmission sûre des messages était donc reporté sur le problème de la transmission sûre des clés.

1.1.2 1.2 Cryptographie à clés publiques

Le principe théorique de la cryptographie à clé publique a été découvert par Diffie en 1975. L'idée sous-jacente est simple : ce n'est pas parce que l'on connaît une fonction f que l'on sait calculer sa réciproque.

Bob choisit une fonction de cryptage f et la rend publique : tout le monde y a accès. Alice connaît donc f et peut ainsi envoyer $f(m)$ à Bob qui s'empresse de retrouver le message m en appliquant f^{-1}. On voit tout de suite où est le problème : tout le monde peut faire comme Bob ! A moins que seul Bob connaisse f^{-1} ...

Pour qu'un tel système fonctionne il faut donc disposer d'une fonction f telle que f soit facilement calculable à partir de f^{-1}, mais que f^{-1} soit impossible à calculer (en un temps raisonnable) à partir de f.

Le principe général est donc le suivant : Bob choisit une fonction g telle que g^{-1} soit calculable à partir de g, mais pas le contraire. Puis il publie la fonction $f = g^{-1}$ et garde "privée" la fonction g. Toute personne désirant envoyer un message m à Bob le crypte avec cette fonction f. Bob le décrypte avec la fonction g, qu'il est le seul à connaître.

Cette fois-ci, si Bob désire envoyer un message à Alice, il faut d'abord que Alice crée sa propre clé de cryptage g, puis publie cette clé (et garde évidemment secrète la fonction de décryptage g^{-1}) : nous sommes en présence d'un système cryptographique *à clés asymétriques*.

1.2 2 RSA

Le système RSA est le plus connu des systèmes cryptographiques à clés publiques (ce n'est pas le seul). RSA a vu le jour en 1977 et porte le nom de ses inventeurs : Ron Rivest, Adi Shamir et Leonard Adleman.

1.2.1 2.1 Détails pratiques

Les messages que nous allons manipuler seront des chaînes de caractères. Comme le langage *Python* code les caractères sur 8 bits, on définit le nombre de lettres de l'alphabet égal à 256.

```
In [1]: nb_lettres = 256
```

Voici la chaîne de caractères qui sera utilisée dans les tests. Il s'agit du plus grand secret de tous les temps.

```
In [2]: pythagore = "In omni triangulo rectangulo, quadratum lateris \
        quod recto angulo opponitur, aequale est duobus simul reliquorum laterum quadratis."

In [3]: pythagore

Out[3]: 'In omni triangulo rectangulo, quadratum lateris quod recto angulo opponitur, aequale es
```

1.2.2 2.2 Convertir une chaîne en nombres

La fonction ci-dessous convertit une chaîne s en nombre entier. S'il y a un second paramètre n, elle renvoie la liste des chiffres de cet entier en base n. Le module **rsa_utils** contient la définition de quelques fonctions de conversion. Vous pouvez bien entendu ouvrir le fichier **rsa_utils.py** dans n'importe quel éditeur pour le consulter.

```
In [4]: import rsa_utils

In [5]: def chaine_vers_chiffres(s, b):
            return rsa_utils.string_to_digits(s, b)

In [6]: x = chaine_vers_chiffres(pythagore, 1234567)

In [7]: print(x)

[18613, 309633, 934926, 188906, 105246, 1111234, 839079, 868676, 737308, 301561, 1074462, 320012
```

```
In [8]: def chiffres_vers_chaine(x, b):
            return rsa_utils.digits_to_string(x, b)

In [9]: chiffres_vers_chaine(x, 7654321)

Out[9]: "\x15Ä%ÛvkÙväÍ\x1axijý\x8dp$äè\x89tg6č\x84\x02øYhL\x13s\x80\x8fÝŕ'ô/nåæÿ\x86ò\x9a\x01\x88
```

Eh bien oui, si on convertit dans un sens en base 1234567, il vaut mieux réutiliser la même base pour la conversion inverse !

```
In [10]: chiffres_vers_chaine(x, 1234567)

Out[10]: 'In omni triangulo rectangulo, quadratum lateris quod recto angulo opponitur, aequale e
```

1.2.3 2.3 Bob crée ses clés RSA

Bob désire recevoir confidentiellement des messages d'Alice (ou de qui il veut). Il commence par choisir secrètement deux nombres premiers p et q.

```
In [11]: p = 137

In [12]: q = 151
```

Bob calcule ensuite le nombre n, produit de p et q. Si les entiers p et q sont suffisamment grands (ce n'est évidemment pas le cas dans notre exemple), la connaissance de l'entier n seul ne permettra pas de retrouver ses facteurs p et q. Puis Bob rend public l'entier n.

```
In [13]: n = p * q
         n

Out[13]: 20687
```

Bob choisit ensuite un nombre e qui est un premier avec $\varphi(n) = (p-1)(q-1)$. Par exemple, Bob peut choisit un entier à peu près au hasard entre 1 et $(p-1)(q-1)$ et regarder s'il convient. Après un très petit nombre d'essais, Bob trouve son bonheur. Puis Bob rend public le nombre e. Ce nombre servira aux correspondants de Bob pour *encrypter* leurs messages.

```
In [14]: e = 142117
```

On vérifie que cet entier convient ...

Le module **arith** contient un certain nombre de fonctions arithmétiques, comme des tests de primalité ou des fonctions de factorisation. Je n'expliquerai pas ici le fonctionnement de ce qui se trouve dans le module **arith**. Vous pouvez bien entendu ouvrir le fichier **arith.py** dans n'importe quel éditeur pour le consulter.

```
In [15]: import arith

In [16]: arith.gcd(e, (p - 1) * (q - 1))

Out[16]: 1
```

Enfin, Bob calcule le nombre d, inverse de e modulo $(p-1)(q-1)$, grâce à l'algorithme d'Euclide. Bob garde secret ce nombre d : il lui servira à *décrypter* les messages qu'il reçoit.

```
In [17]: def inverse_mod(a, b):
             u, v, d = arith.ext_gcd(a, b)
             return u % b

In [18]: d = inverse_mod(e, (p - 1) * (q - 1))
         d

Out[18]: 3853
```

Vérification ...

```
In [19]: (d * e) % ((p - 1) * (q - 1))

Out[19]: 1
```

En résumé, la clé (publique) de cryptage est le couple (n, e), et la clé (privée) de décryptage est le couple (n, d). Plus exactement, les fonctions (clés) de cryptage et de décryptage sont fabriquées de façon simple à partir de ces deux couples. Il est d'usage dans le langage courant de confondre le couple et la clé (fonction) qu'il permet de générer. Ainsi, si le nombre n est un nombre de 128 bits, on parlera de clé de 128 bits.

1.2.4 2.4 Automatisation

Voici une fonction renvoyant un triplet (n, e, d). Elle est un peu plus générale que ce qui a été décrit : on prend $e = 65537$. Puis on choisit au hasard deux nombres premiers p et q, de sorte que le nombre $n = pq$ ait environ L bits, et que e soit premier avec $p-1$ et $q-1$. Puis on fabrique une clé RSA à partir de p et q. La fonction *creerCle* prend en paramètre un entier L qui est la taille en bits de la clé désirée.

```
In [20]: import random

In [21]: def creerCle(L):
             e = 65537
             n1 = 2 ** (L // 2 - 1)
             n2 = 2 ** (L // 2) - 1
             p = random.randint(n1, n2)
             while not(arith.is_prime(p)) or arith.gcd(p - 1, e) != 1: p = p + 1
             q = random.randint(n1, n2)
             while not(arith.is_prime(q)) or arith.gcd(q - 1, e) != 1: q = q + 1
             n = p * q
             nn = (p - 1) * (q - 1)
             d = inverse_mod(e, nn)
             return (n, e, d)

In [22]: nB, eB, dB = creerCle(64)
         nB, eB, dB
```

```
Out[22]: (11424210660977350109, 65537, 2348746728610710905)
```

Bob a été trop optimiste : sa clé est facilement *cassée* par Eve ... La spécialité de Eve est de casser les clés RSA. Elle s'est écrit pour cela une fonction *Python* :

```
In [23]: def casser_RSA(n, e):
             p = arith.rho_pollard(n)
             q = n // p
             n1 = (p - 1) * (q - 1)
             d = inverse_mod(e, n1)
             return (n, e, d)
```

Eve applique sa fonction sur la clé publique de Bob :

```
In [24]: u, v, w = casser_RSA(nB, eB)
         u, v, w
```

```
Out[24]: (11424210660977350109, 65537, 2348746728610710905)
```

Et la voici en possession de la clé privée de dernier.
Avec ce qui suit, Bob devrait être plus tranquille (???)

```
In [25]: nB, eB, dB = creerCle(256)
         nB, eB, dB
```

```
Out[25]: (47454232298821134569795453593841077596145024049578750522210877726798883214201,
          65537,
          35771876012918817021706895248432652645998368518177993552725195042610540807949)
```

```
In [ ]: casser_RSA(nB, eB)
```

Inutile (?) d'insister ...

1.2.5 2.5 Crypter, décrypter

Rappelons que Bob a rendus *publics* les nombres n et e. Tout le monde connaît maintenant ces deux nombres. En revanche, p, q, d sont *privés* et seul Bob y a accès. Alice désire envoyer un message s à Bob. Elle commence par transformer s en des "chiffres" en base n. Puis elle élève ces chiffres à la puissance e modulo n. Alice envoie la liste de nombres obtenue à Bob.

```
In [26]: def rsa(s, e, n):
             xs = chaine_vers_chiffres(s, n)
             ys = [arith.power_mod(x, e, n) for x in xs]
             return chiffres_vers_chaine(ys, n)
```

```
In [27]: grotehapy = rsa(pythagore, eB, nB)
         grotehapy
```

```
Out[27]: "\x0eű#\x13Ùú\x80Ìw%ň\x1b£š+ů8â2Ý\x15ò\x8bÆ\x99ćś\x88Ã\x05n$@;ZŐźzNá\\<á\x8b?\x11\x80źÑ
```

Bob désire maintenant décrypter le message. Il fait exactement ce qu'a fait Alice, sauf qu'il utilise d à la place de e.

```
In [28]: rsa(grotehapy, dB, nB)
```

```
Out[28]: 'In omni triangulo rectangulo, quadratum lateris quod recto angulo opponitur, aequale e
```