

R.S.A.

Marc Lorenzi

février 2014

Table des matières

| | | |
|------------|--|----------|
| I | Introduction | 1 |
| I.1 | Cryptographie à clés privées | 1 |
| I.2 | Cryptographie à clé publique | 2 |
| I.3 | RSA | 2 |
| I.4 | Détails pratiques | 2 |
| I.5 | Convertir une chaîne en nombres | 2 |
| I.6 | Convertir des nombres en chaîne | 3 |
| II | RSA | 3 |
| II.1 | Bob crée ses clés RSA | 3 |
| II.2 | Automatisation du procédé | 4 |
| II.3 | Crypter, décrypter | 5 |
| III | Explications mathématiques | 5 |
| III.1 | Pourquoi cela fonctionne | 5 |
| III.2 | Pourquoi le système est-il sûr (?) | 5 |
| III.3 | Méthodes non mathématiques | 6 |
| IV | Envoi de messages avec signature | 6 |
| IV.1 | Alice crée sa clé RSA | 6 |
| IV.2 | Alice envoie un message « signé numériquement » | 6 |
| IV.3 | Bob décrypte le message et vérifie sa provenance | 6 |
| V | Annexes | 6 |
| V.1 | rsa_utils.py | 6 |
| V.2 | arith.py | 7 |

I Introduction

I.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.

I.2 Cryptographie à clé publique

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*.

I.3 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.

I.4 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.

```
>>> 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.

```
>>> pythagore =
    "In omni triangulo rectangulo, quadratum
    lateris quod recto angulo opponitur, aequale
    est duobus simul reliquorum laterum quadratis."
```

I.5 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 fichier `rsa_utils.py` est fourni en annexe, il contient la définition de quelques fonctions de conversion.

```
def chaine_vers_chiffres(s, b):
    return rsa_utils.string_to_digits(s, b)

>>> x = chaine_vers_chiffres(pythagore, 1234567)
>>> x
[18613L, 309633L, 934926L, 188906L, 105246L, 1111234L, 839079L, 868676L, 737308L,
301561L, 1074462L, 320012L, 1156608L, 867303L, 82768L, 199237L, 62482L, 405029L,
258043L, 27675L, 379180L, 115820L, 359575L, 1159404L, 356226L, 1231146L, 555337L,
660882L, 665273L, 159108L, 880811L, 928878L, 995712L, 67226L, 819543L, 310732L,
405722L, 92260L, 663470L, 662149L, 667226L, 775272L, 549383L, 574327L, 594501L,
1153232L, 690245L, 652252L, 973515L, 1001811L, 808757L, 362732L]
```

I.6 Convertir des nombres en chaîne

```
def chiffres_vers_chaine(x, b):
    return rsa_utils.digits_to_string(x, b)

>>> chiffres_vers_chaine(x, 7654321)
?%?vk?v?x???p$?g6???YhLs??/n???yo???u0l>??Ur?H]z2u??$??r?B?Tt.-??1??#F
[?P???d?]Kup??j3?;??V&??n???Sb#??K?u??M??5???L
```

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

```
>>> chiffres_vers_chaine(x, 1234567)
In omni triangulo rectangulo, quadratum lateris quod recto angulo opponitur,
aequale est duobus simul reliquorum laterum quadratis.
```

II RSA

II.1 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 .

```
>>> p = 137
>>> 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 .

```
>>> n = p * q
>>> n
20687
```

Bob choisit ensuite un nombre e qui est un premier avec $\varphi(n) = (p-1)(q-1)$. Par exemple, Bob peut choisir 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.

```
>>> e = 142117
```

On vérifie que cet entier convient ...

```
>>> arith.gcd(e, (p - 1) * (q - 1))
1
```

Le fichier `arith.py` est fourni en annexe. Il contient un certain nombre de fonctions arithmétiques, comme des tests de primalité ou des fonctions de factorisation.

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.

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

>>> d = inverse_mod(e, (p - 1) * (q - 1))
>>> d
3853
```

Vérification ...

```
>>> (d * e) % ((p - 1) * (q - 1))
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.

II.2 Automatisation du procédé

Voici une fonction renvoyant un triplet (n, e, d) . Elle est un peu plus générale que ce qui a été décrit : elle prend $e = 65537$. Puis elle 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 elle 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.

```
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)

>>> nB, eB, dB = creerCle(64)
>>> nB, eB, dB
8467091703989446169, 65537, 923619307402247633
```

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* :

```
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 :

```
>>> u, v, w = casser_RSA(nB, eB)
>>> u, v, w
8467091703989446169, 65537, 923619307402247633
```

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

```
>>> nB, eB, dB = creerCle(256)
>>> nB, eB, dB
4278888591814615255808426838942239514454883357947923853578834727109444
3854709L, 65537, 20920766459788473174217529212540424609051299997318968
464548233022425184692997L

>>> casser_RSA(nB, eB)
... ^C
>>>
```

Inutile (?) d'insister ...

II.3 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.

```
def rsa(s, e, n):
    xs = chaine_vers_chiffres(s, n)
    ys = map(lambda x: arith.power_mod(x, e, n), xs)
    return chiffres_vers_chaine(ys, n)

>>> grotehapy = rsa(pythagore, eB, nB)
>>> grotehapy
'Y????!????;B?!?yu ?????????@????F_??'d??%?wE:_?!?o)?TS(???.M???D4????A?
4????0c%8Z??FTK?????r??E??????'
```

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

```
>>> rsa(grotehapy, dB, nB)
In omni triangulo rectangulo, quadratum lateris quod recto angulo opponitur,
aequale est duobus simul reliquorum laterum quadratis.
```

III Explications mathématiques

III.1 Pourquoi cela fonctionne

Soient p et q deux nombres premiers distincts. Soit $n = pq$. Posons $\varphi(n) = (p-1)(q-1)$. La fonction φ n'est autre que la fonction indicatrice d'Euler. Soient enfin d et e tels que $de \equiv 1 \pmod{\varphi(n)}$. On a donc $ed = 1 + k\varphi(n)$ où k est un nombre entier.

Soit x un entier compris entre 0 et $n-1$. Soit $y = x^e \pmod{n}$.

→ Supposons dans un premier temps que x est premier avec n . Alors $y^d = x^{ed} = x^{1+k\varphi(n)} = (x^{\varphi(n)})^k x$. Le théorème d'Euler nous apprend que $x^{\varphi(n)} \equiv 1 \pmod{n}$. Donc $y^d \equiv x \pmod{n}$.

→ Si x n'est pas premier avec n , alors x est divisible par p ou par q (mais pas par les deux). Par exemple, p divise x et x est premier avec q . On a donc $x \equiv 0 \pmod{p}$. Dans un premier temps, $x^{k\varphi(n)+1} \equiv 0 \equiv x \pmod{p}$. En fait, toutes les puissances de x (sauf x^0) sont congrues à 0 modulo p . De plus, $k\varphi(n)+1 = K(q-1)+1$ donc $x^{k\varphi(n)+1} = (x^{q-1})^K x \equiv x \pmod{q}$ puisque x est premier avec q (c'est le petit théorème de Fermat). Ainsi, $y^d - x$ est à la fois un multiple de p et un multiple de q . Comme p et q sont premiers et distincts, ils sont premiers entre-eux, et ainsi $y^d - x$ est un multiple de pq . Donc, $y^d \equiv x \pmod{n}$.

→ Enfin, si $x = 0$, on a trivialement $y = 0$ puis $y^d = 0 = x$.

III.2 Pourquoi le système est-il sûr (?)

Une analyse de RSA ne se fait certainement pas en quelques lignes. Le problème est le suivant : connaissant n et e , peut-on trouver d ? Plusieurs pistes mathématiques s'offrent à nous :

→ On peut penser factoriser n pour obtenir p et q . Ensuite, il est facile de calculer d à l'aide de l'algorithme d'Euclide. A l'heure actuelle, on ne sait pas factoriser les nombres lorsqu'ils sont trop grand.

→ On peut penser calculer $\varphi(n)$ par des méthodes détournées, puis appliquer l'algorithme d'Euclide pour trouver d . Mais il est facile de prouver que si n est un entier qui est le produit de deux nombres premiers distincts, on sait factoriser n si et seulement si on sait calculer $\varphi(n)$. On est donc ramenés à l'idée (mauvaise) précédente.

→ Autre idée : le problème est en fait de trouver x en connaissant e et x^e modulo n : ce n'est finalement qu'un problème de calcul de racine e -ième ... eh bien à l'heure où ce papier est écrit, aucun algorithme de calcul de racine e -ième n'est connu (?), sauf à connaître la factorisation de n .

→ Il existe d'autres attaques possibles (les ressources des cryptanalystes sont immenses et insoupçonnées). A chaque fois qu'une attaque réussit (et cela arrive), on corrige le tir pour RSA. Ainsi, le choix de $e = 65537$ n'est pas arbitraire. On a pu montrer que si e est un nombre de Fermat premier, alors certaines attaques connues échouaient. On a également montré que si $e = 3, 5$ ou 17 , de nouvelles attaques existaient.

III.3 Méthodes non mathématiques

Nous parlerons ici à mots couverts de méthodes de cassage de clés très efficaces, souvent utilisées, mais indignes d'un gentleman. Citons-en quelques-unes :

- virus informatique (choix de prédilection pour récupérer les clés privées, voire les messages en clair)
- torturer Bob (c'est mal)
- faire boire Bob (c'est mal aussi)
- week-end à Las Vegas avec Alice (sympa) ... mais cela n'aura aucun intérêt si on veut décrypter les messages destinés à Bob.

IV Envoi de messages avec signature

Le protocole précédent pose un problème : Bob n'a aucune certitude que le message qu'il a reçu provient bien d'Alice. Si c'est le théorème de Pythagore, ce n'est pas grave, d'autant que ce n'est plus vraiment un secret pour l'humanité. Cela pourrait être ennuyeux si Eve envoyait à Bob le message : « Bonjour, c'est Alice. Appuie sur le gros bouton rouge marqué LANCEMENT MISSILES ». Il existe une parade à cela.

IV.1 Alice crée sa clé RSA

```
>>> nA, eA, dA = creerCle(64)
```

IV.2 Alice envoie un message « signé numériquement »

Alice veut envoyer un message à Bob. Elle commence par crypter le message avec sa clé de décryptage (!!) . Puis elle recrypte le tout avec la clé de cryptage de Bob.

```
>>> toryphage = rsa(rsa(pythagore, dA, nA), eB, nB)
```

IV.3 Bob décrypte le message et vérifie sa provenance

Bob n'a plus qu'à décrypter ... en deux temps. Avec sa clé privée, il fait sauter le premier cadenas (il est le seul au monde à pouvoir faire cela). Puis, avec la clé publique d'Alice, il fait sauter le sceau d'Alice (tout le monde peut faire sauter ce sceau, mais Alice est la seule au monde à pouvoir l'apposer) et retrouve le message en clair.

```
>>> rsa(rsa(toryphage, dB, nB), eA, nA))
In omni triangulo rectangulo, quadratum lateris quod recto angulo opponitur,
aequale est duobus simul reliquorum laterum quadratis.
```

V Annexes

V.1 rsa_utils.py

```
# -*- coding:UTF-8 -*-

nb lettres = 256

# Convertir une liste de chiffres en base b en un nombre
def digits_to_number(xs, b):
    n = 0
    pw = 1
    xs.reverse()
    for x in xs:
        n += x * pw
        pw = pw * b
    return n

# Convertir un nombre en une liste de chiffres en base b
def number_to_digits(x, b):
```

```

s = []
while x != 0:
    s.append(x % b)
    x = x // b
s.reverse()
return s

# Convertir une chaîne de caractères en nombre
def string_to_number(s):
    return digits_to_number(map(ord, s), nb_lettres)

# Convertir une chaîne de caractères en liste de chiffres en base b
def string_to_digits(s, b):
    return number_to_digits(string_to_number(s), b)

def toCharacterCode(s):
    return map(ord, s)

```

V.2 arith.py

```

# -*- coding: UTF-8 -*-

#-----
# Test de Primalité de Miller-Rabin

# Exponentiation rapide modulaire. Renvoie  $x^n \bmod p$ 
def power_mod(x, n, p):
    z = 1
    m = n
    y = x
    while m != 0:
        if m % 2 == 1: z = (z * y) % p
        m = m // 2
        y = (y * y) % p
    return z

# Renvoie (s, d) où  $n - 1 = 2^s d$  et d est impair
def factor2(n):
    d = n - 1
    s = 0
    while d % 2 == 0:
        s += 1
        d = d // 2
    return (s, d)

# Témoin de non-primauté
# Renvoie True si a est un témoin de non primalité pour n
# Renvoie False sinon

def witness(a, n):
    s, d = factor2(n)
    x = power_mod(a, d, n)
    if x == 1 or x == n - 1: return False
    for r in range(1, s):
        x = (x * x) % n
        if x == 1: return True
        if x == n - 1: return False
    return True

```

```

# Test de primalité (probabiliste) de Miller-Rabin
def is_prime(n):
    if n <= 10000: return is_naive_prime(n)
    if n % 2 == 0: return False
    for a in [2, 3, 5, 7, 11, 13, 17, 19, 23]:
        if witness(a, n): return False
    return True

#-----
# Test naïf de primalité

def is_naive_prime(n):
    if n <= 1: return False
    if n == 2: return True
    return least_divisor(n) == n

def least_divisor(n):
    k = 2
    while k * k <= n and n % k != 0: k += 1
    if k * k > n: return n
    else: return k

#-----
# Recherche d'un facteur non trivial de n - Méthode de Pollard
# L'algorithme ci-dessous échoue lorsque n est un nombre premier.

def rho_pollard(n):
    def f(x): return (x * x + 1) % n
    x = 2
    y = 2
    d = 1
    while True:
        while d == 1:
            x = f(x)
            y = f(f(y))
            d = gcd(abs(x-y), n)
        if d != n: return d

#-----
# Plus petit nombre premier strictement supérieur à n
def next_prime(n):
    p = n + 1
    while not(is_prime(p)): p += 1
    return p

# pgcd
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# pgcd étendu. Renvoie un triplet (u, v, d) tel que ua + vb = d
def ext_gcd(a, b):
    u1, v1, r1 = 1, 0, a
    u2, v2, r2 = 0, 1, b
    while r2 != 0:
        q = r1 // r2
        u, v, r = u1 - q * u2, v1 - q * v2, r1 - q * r2
        u1, v1, r1 = u2, v2, r2

```



```
    u2, v2, r2 = u, v, r
    return (u1, v1, r1)
```