Cryptographie : le système RSA

Aurélien Blais, Nicolas Iung

February 14, 2019

1 Principe du chiffrement RSA

1.1 Question 1

1.1.1 Montre que $cd = 1 + k(p-1)(q-1)$

En utilisant le théorème de Bezout $au + bv = 1$
On pose

$$\begin{aligned}
a &= c \\
u &= d \\
b &= \varphi(n) \\
v &= -k
\end{aligned}$$

$$\begin{aligned}
cd + -k\varphi(n) &= 1 \\
cd &= 1 + k\varphi(n) \\
\text{Or } \varphi(n) &= (p-1)(q-1) \\
cd &= 1 + k(p-1)(q-1)
\end{aligned}$$

1.2 Question 2

1.2.1 Déduire que $cd \equiv 1 \pmod{\varphi(n)}$

On sait que

$$\begin{aligned}
& a \equiv b \pmod{c} \\
\iff & a = b + kc
\end{aligned}$$

On peut donc en déduire que

$$\begin{aligned}
& cd = 1 + k(p-1)(q-1) \\
& cd = 1 + k\varphi(n) \\
& cd \equiv 1 \pmod{\varphi(n)}
\end{aligned}$$

1.2.2 Conclure que si c et $\varphi(n)$ premiers entre eux, il existe toujours un entier d inverse de c modulo $\varphi(n)$

On a $cd \equiv 1 \pmod{\varphi(n)}$
Or on dit que a est l'inverse de $b \equiv \pmod{n}$
Si et seulement si $ab \equiv 1 \pmod{n}$
On a donc d inverse de $c \pmod{\varphi(n)}$

1.3 Question 3

1.3.1 Montrer que M est premier avec p et avec q

M est un entier qui représente le message coder. On montrera que M est premier avec p et qu'il est aussi avec q. Sachant que p et q sont des nombres premiers.

Pour cela on admet que 1 et p divise q ou 1 et q divise p. Alors respectivement M est premier avec p et q. Car M est un message codé en un entier inférieur à n avec $n = p * q$.

1.4 Question 4

1.4.1 Déduire que $M^{p-1} \equiv 1 \mod (p)$

Notons que p est un nombre premier. D'après le théorème de Fermat, Si M n'est pas divisible par p alors $M^{p-1} \equiv 1 \mod (p)$.
Or M et p sont premier entre eux. Donc le théorème de Fermat s'applique et $M^{p-1} \equiv 1 \mod (p)$

1.4.2 Déduire que $M^{q-1} \equiv 1 \mod (q)$

Notons que pq est un nombre premier. D'après le théorème de Fermat, Si M n'est pas divisible par p alors $M^{q-1} \equiv 1 \mod (p)$.
Or M et q sont premier entre eux. Donc le théorème de Fermat s'applique et $M^{q-1} \equiv 1 \mod (p)$

1.5 Question 5

1.5.1 Déduire que $M^{cd} \equiv M \mod (p)$

Rappelons que :

$$cd = 1 + k(p-1)(q-1) \tag{1}$$

En appliquant cela à M :

$$M^{cd} = M^{1+k(p-1)(q-1)} = M * (M^{p-1})^{k(q-1)} \equiv M \mod (p) \tag{2}$$

Donc p divise $M^{cd} - M$

1.5.2 Déduire que $M^{cd} \equiv M \mod (q)$

De même que pour$M^{cd} \equiv M \mod (p)$, $M^{cd} \equiv M \mod (q)$ c'est à dire que que q divise $M^{cd} - M$

1.6 Question 6

1.6.1 Déduire que $M^{cd} - M$ est un multiple de n

p et q sont premier entre eux, $pq = n$ divise $M^{cd} - M$.
Donc $M^{cd} \equiv M \mod (n)$

2 Premier exemple

2.1 Question 1

2.1.1 Calculer $n_1 = p_1q_1$ et $\varphi(n_1) = (p_1 - 1)(q_1 - 1)$ avec $p_1 = 7307$ et $q_1 = 5923$

$$
\begin{aligned}
n_1 &= p_1q_1 \\
n_1 &= 7307 * 5923 \\
n_1 &= 43279361
\end{aligned}
$$

$$
\begin{aligned}
\varphi(n_1) &= (p_1 - 1)(q_1 - 1) \\
\varphi(n_1) &= (7307 - 1)(5923 - 1) \\
\varphi(n_1) &= 43266132
\end{aligned}
$$

2.2 Question 2

2.2.1 Choisir un entier c_1 premier avec $\varphi(n_1)$ tel que $c_1 < \varphi(n_1)$

On sait que 2 nombres sont premiers entre eux si leur $PGCD$ est égal à 1
On pose donc $c_1 = 5$

$$
\begin{aligned}
&\text{Par la Méthode d'Euclide} \\
43266132 &= 8653226 * 5 + 2 \\
5 &= 2 * 2 + 1 \\
2 &= 2 * 1 + 0
\end{aligned}
$$

Le PGCD est égal au dernier reste non nul soit 1.
Le PGCD étant égal à 1, $c_1 = 5$ et $\varphi(n_1) = 43266132$ sont premiers entre eux et $c_1 < \varphi(n_1)$.

2.3 Question 3

2.3.1 Déterminer d_1 inverse modulaire de c_1 modulo $\varphi(n_1)$

3 Fonctions de base

3.1 Question 1

3.1.1 Implémenter *exponentiationModulaire(x, k, n)*

L'algorithme implémenté suit le pseudo-code suivant :
`https://en.wikipedia.org/wiki/Modular_exponentiation#Right-to-left_binary_method`
Ayant une complexité en $O(log(k))$

```
def self.exponentiation_modulaire(x, k, n)
  result = 1
  while k > 0
    result = (result * x) % n if (k & 1) == 1
    k      = k >> 1
    x = (x**2) % n
  end
  result
end
```

3.2 Question 2

3.2.1 Implémenter *euclideEtendu(a, b)*

L'algorithme implémenté suit le pseudo-code suivant :
`https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu#L'algorithme`

```
 def self.euclide_etendu(a, b)
  r, u, v, r2, u2, v2, q = a, 1, 0, b, 0, 1, 0
  while(r2 > 0) do
    q = r/r2
    r, u, v, r2, u2, v2 = r2, u2, v2, r-q*r2, u-q*u2, v-q*v2
  end
  {pgcd: r, u: u, v: v}
end
```

La fonction retourne un $Hash$, c'est à dire ensemble clé $\Rightarrow$ valeur
Tel que $\{pgcd : valeur, u : valeur, v : valeur\}$

3.3 Question 3

3.3.1 Implémenter *inverseModulaire(a, N)*

La méthode retourne l'inverse modulaire de (a, N) et se base sur la définition fournie ici :
https://fr.wikipedia.org/wiki/Inverse_modulaire#Algorithme_d'Euclide_%C3%A9tendu
Si a et N ne sont pas premiers entre eux, la méthode lève une exception.

```
def self.inverse_modulaire(a, n)
  val = euclide_etendu a, n
  raise Exception.new("Can't find value for a: #{a} and n: #{n}") unless val[
  val[:u] % n
end
```

3.4 Question 4

3.4.1 Implémenter *generationExposants(p, q)*

On déclare $\varphi = (p-1)*(q-1)$ comme vu précédemment dans l'énoncé.
On déclare ensuite $c = 2$, afin de ne pas obtenir $c = 1$, qui est premier avec l'ensemble des entiers.
Pour obtenir c, on l'incrémente tant que $PGCD(c, \varphi)$ n'est pas égal à 1.
Enfin, on retourne un $Hash$ contenant la valeur de c et de d tel que $d = inverseModulaire(c, \varphi)$

```
def self.generation_exposants(p, q)
  phi = (p - 1) * (q - 1)
  c = 2
  while c < phi
    break if c.gcd(phi) == 1
    c += 1
  end
  {c: c, d: inverse_modulaire(c, phi)}
end
```

3.5 Question 5

3.5.1 Implémenter *chiffrement(m, n, c)*

On sait d'après l'énoncé que $m_2 \equiv m^c \mod (n)$. L'algorithme renvoie donc cette valeur, qui est l'exponentiation modulaire.

```
def self.chiffrement(m, n, c)
  exponentiation_modulaire m, c, n
end
```

3.5.2 Implémenter *dechiffrement(m, n, d)*

On sait d'après l'énoncé que $m \equiv m_2^c \mod (n)$. L'algorithme renvoie donc cette valeur, qui est l'exponentiation modulaire.

```
def self.dechiffrement(m, n, d)
  exponentiation_modulaire m, d, n
end
```

4 Chiffrement de messages textes

4.1 A vous de jouer

4.1.1 Ré-implémentation de la méthode *StringToInteger* en Ruby

```ruby
def self.string_to_integer(message)
  message = message.upcase.split(//)
  value = 0

  message.each_with_index do |char, i|
    value = value + (ALPHABET.length ** (message.length - 1 - i)) * ALPHABET.
  end
  value
end
```

La méthode est une copie de celle fournie en Java, l'alphabet est lui aussi repris
à l'identique.

```ruby
ALPHABET = %w(. A B C D E F G H I J K L M N O P Q R S T U V W X Y Z).freeze
```

4.1.2 Ré-implémentation de la méthode *IntegerToString* en Ruby

```ruby
def self.integer_to_string(number)
  quotient = number / ALPHABET.length
  remainder = number % ALPHABET.length
  message = "#{ALPHABET[remainder]}"

  while quotient > ALPHABET.length
    remainder = quotient % ALPHABET.length
    quotient = quotient / ALPHABET.length
    message += ALPHABET[remainder]
  end

  (message + ALPHABET[quotient]).reverse
end
```

La méthode est une copie de celle fournie en Java.
Exception faite que l'on construit le mot à l'envers, et qu'il est donc inversé
avant d'être renvoyé.

4.1.3 Implémentation de la méthode *Decodage(message, n, c)*

```
def self.decode(message, n, c)
  prime_div = n.prime_division
  p = prime_div[0][0]
  q = prime_div[1][0]

  d = RSA.inverse_modulaire(c, (p - 1) * (q - 1))

  message = RSA.string_to_integer message

  RSA.integer_to_string RSA.dechiffrement message, n, d
end
```

On commence par déterminer p et q, pour cela on utilise la méthode *prime_division*
qui renvoie les facteurs premiers d'un nombre donné.
On peut ainsi calculer $\varphi(n)$ et donc déterminer d en utilisant la méthode $inverse\_modulaire(c, \varphi(n))$.
Il ne reste plus qu'a déchiffrer le message, en utilisant la fonction $dechiffrement(message, n, d)$

4.1.4 Déchiffrer les valeurs données

On execute notre méthode $decode(message, n, c)$ avec les valeurs fournies.

```
puts RSA.decode "LHRZNS", 211582871, 127                       # => "RETI"
puts RSA.decode "AYMRNCI", 844991843, 349837                   # => "RUIZ"
puts RSA.decode "IVWTRM.FPL", 202899206548601, 39898535 # => "SICILIENNE"
```

Les valeurs trouvées correspondent bien a des ouvertures d'échecs.