

Cryptographie : le système RSA

Aurélien Blais, Nicolas Iung

February 6, 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 Dédurre 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

1.4 Question 4

1.4.1 D duire que $M^{p-1} \equiv 1 \pmod{p}$

1.4.2 D duire que $M^{q-1} \equiv 1 \pmod{q}$

1.5 Question 5

1.5.1 D duire que $M^{cd} - M$ est un multiple de n

2 Premier exemple

2.1 Question 1

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

$$\begin{aligned}n_1 &= p_1 q_1 \\n_1 &= 7307 * 5923 \\n_1 &= 43279361\end{aligned}$$

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

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
Pour déterminer rapidement ce chiffre, on utilise un algorithme simple, ici en Ruby.

```
(2..43266131).each do |i|  
  if p.gcd(i) == 1  
    puts i  
    break  
  end  
end
```

L'algorithme itère simplement sur les entiers entre 2 et $\varphi(n_1)$ et retourne le plus petit entier pour lequel $PGCD(\varphi(n_1), i) = 1$ soit $c_1 = 5$.

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
  base = x
  while k > 0
    if (k & 1) == 1
      result = (result * base) % n
    end
    k = k >> 1
    base = (base * base) % 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'algorithm

```
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 $m_2 \equiv m^c \pmod{n}$. 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 $m \equiv m_2^d \pmod{n}$. 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 Un mot, un nombre

4.1.1 Méthode *StringToInteger*

4.1.2 Méthode *IntegerToString*

4.2 A vous de jouer

4.2.1 Ré-implémentation de la méthode *StringToInteger* en 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.index(char)
  end
  value
end
```

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

```
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
```

4.2.2 Ré-implémentation de la méthode *IntegerToString* en 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.

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

```
def self.decode(message, n, c)
  p = n.prime_division[0][0]
  q = n.prime_division[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'à déchiffrer le message, en utilisant la fonction *dechiffrement*($message, n, d$)

4.2.4 Déchiffrer les valeurs données

On exécute 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 "IVWIRM.FPL", 202899206548601, 39898535 # => "SICILIENNE"
```

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

5 Pour aller plus loin

5.1 Utilisation de RSA en pratique

5.2 Générer et vérifier de grands nombres premiers

5.3 Algorithme de Shor et fiabilité et sécurité de RSA