Introduction à la cryptographie à clé publique, RSA et tests de primalité

christina.boura@uvsq.fr

1 Théorème des restes chinois

Une bande de 17 pirates possède un trésor constitué de pièces d'or d'égale valeur. Ils projettent de se les partager également, et de donner le reste au cuisinier chinois. Celui-ci recevrait alors 3 pièces. Mais les pirates se querellent, et six d'entre eux sont tués. Un nouveau partage donnerait au cuisinier 4 pièces. Dans un naufrage ultérieur, seuls le trésor, six pirates et le cuisinier sont sauvés, et le partage donnerait alors 5 pièces d'or à ce dernier. Quelle est la fortune minimale que peut espérer le cuisinier s'il décide d'empoisonner le reste des pirates ?

Traduisons ce problème en équations. Si x est le nombre de pièces qui constituent le trésor,

$$\begin{aligned} x &\equiv 3 \mod 17 \\ x &\equiv 4 \mod 11 \\ x &\equiv 5 \mod 6 \end{aligned}$$

et on cherche à calculer x à partir de ce système de congruences. Nous allons voir que le théorème suivant permet de résoudre ce problème.

Théorème 1.1 (Théorème des restes chinois)**.** *Soient $m_1, m_2, \ldots, m_r$ des entiers deux à deux premiers entre eux (c.-à-d.* $\text{pgcd}(m_i, m_j) = 1$ *lorsque $i \neq j$.) Alors, pour tout entiers $a_1, a_2, \ldots, a_r$, il existe un entier x, unique modulo $M = m_1 \cdot m_2 \cdots m_r$ tel que*

$$\begin{aligned} x &\equiv a_1 \mod m_1 \\ x &\equiv a_2 \mod m_2 \\ &\vdots \\ x &\equiv a_r \mod m_r. \end{aligned}$$

Démonstration. Première Partie : Existence. Pour chaque i les entiers m_i et

$$\hat{m_i} = \frac{M}{m_i} = m_1 \cdots m_{i-1} m_{i+1} \cdots m_r$$

sont premiers entre eux. En utilisant l'algorithme d'Euclide étendu on peut trouver des entiers u_i et v_i tels que $u_i m_i + v_i \hat{m}_i = 1$. Si on pose $e_i = v_i \hat{m}_i$ nous avons

$$e_i \equiv 1 \mod m_i$$

et

$$e_i \equiv 0 \mod m_j$$

pour $j \not\equiv i$. On montre maintenant que l'entier

$$x = a_1 e_1 + a_2 e_2 + \cdots + a_r e_r$$

est une solution du système. En effet, pour $1 \leq i \leq r$

$$x \equiv a_i e_i \equiv a_i(1 + u_i m_i) \equiv a_i \cdot 1 \equiv a_i \mod m_i.$$

Deuxième Partie : Unicité. On suppose que x' est également une solution du système. Nous avons alors que pour chaque i

$$x' \equiv a_i \equiv x \mod m_i,$$

par conséquent $m_i|(x - x')$. Puisque les m_i sont deux à deux premiers entre eux, leur produit divise également $x - x'$. Donc

$$x \equiv x' \mod M.$$

□

Nous allons voir maintenant comment on peut utiliser ce théorème afin de résoudre le problème des pirates.

Solution du problème des pirates. Nous pouvons vérifier que les entiers $17, 11$ et 6 sont deux à deux premiers entre eux. On pose $m_1 = 17, m_2 = 11$ et $m_3 = 6$ et $M = 17 \cdot 11 \cdot 6 = 1122$. On calcule ensuite :

$$\begin{aligned} \hat{m_1} &= \frac{M}{m_1} = 11 \cdot 6 = 66 \\ \hat{m_2} &= \frac{M}{m_2} = 17 \cdot 6 = 102 \\ \hat{m_3} &= \frac{M}{m_3} = 17 \cdot 11 = 187. \end{aligned}$$

On applique maintenant l'algorithme d'Euclide étendu aux couples $(m_1, \hat{m}_1), (m_2, \hat{m}_2)$ et $(m_3, \hat{m}_3)$ qui ont tous un pgcd égal à 1. Nous trouvons que :

$$\begin{aligned} (-31) \cdot 17 + (8) \cdot 66 &= 1 \\ (-37) \cdot 11 + (4) \cdot 102 &= 1 \\ (-31) \cdot 6 + (1) \cdot 187 &= 1 \end{aligned}$$

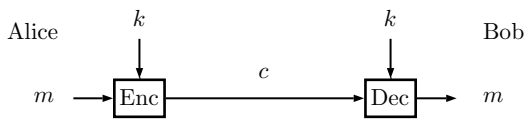
On note $e_1 = 8 \cdot 66 = 528$, $e_2 = 4 \cdot 102 = 408$, et $e_3 = 187$. Par conséquent, la solution est

$$\begin{aligned} x &= a_1 e_1 + a_2 e_2 + a_3 e_3 \\ &= 3 \cdot 528 + 4 \cdot 408 + 5 \cdot 187 \\ &= 4151 \\ &\equiv 785 \mod 1122, \end{aligned}$$

donc le trésor contient au minimum 785 pièces.

2 Cryptographie à clé publique

Jusqu'à présent nous avons exclusivement étudié les cryptosystèmes dites symétriques. Le chiffrement et le déchiffrement en utilisant un algorithme symétrique peut se résumer à l'image suivante.



Un tel système est appelé symétrique pour deux raisons :

— La même clé est utilisée pour chiffrer et déchiffrer.
— La fonction de chiffrement et celle de déchiffrement sont très similaires (dans le cas du DES elles sont même identiques).

Il existe une analogie très simple pour la cryptographie symétrique. Supposons qu'il existe un coffre-fort avec un cadenas très sécurisé. Seulement Alice et Bob possèdent la clé pour ce coffre-fort. La procédure de chiffrement peut être vue comme mettre le message dans le coffre fort. Afin de lire, c.-à-d. déchiffrer le message, Bob utilise sa clé, ouvre le coffre et lit le message.

2.1 Inconvénients de la cryptographie symétrique

1. La clé doit être établie en utilisant un canal sûr.
2. Nombre de clés : Même si on règle le problème de la distribution des clés, on doit traiter le problème du très grand nombre de clés à générer. Si chaque couple d'utilisateurs possède une paire de clés différente, pour un réseau de n utilisateurs, il existe

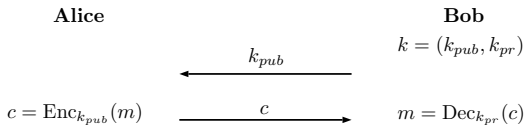
$$\frac{n(n-1)}{2}$$

clés différentes, et chaque utilisateur doit stocker $n-1$ clés de façon sécurisée. Par exemple, pour une société moyenne de 2000 personnes, plus de 4 millions de clés doivent être générées et transportées à travers un canal sûr.

2.2 La cryptographie à clé publique

Afin de combattre ces faiblesses de la cryptographie symétrique, Diffie, Hellman et Merkle ont fait une proposition révolutionnaire basée sur l'idée suivante. Il n'est pas nécessaire que la clé possédée par la personne qui chiffre le message (Alice dans notre exemple) soit secrète. L'élément crucial est que seul Bob, le récepteur du message, puisse le déchiffrer. Afin de réaliser un tel système, Bob publie sa clé de chiffrement qui peut être connue par n'importe qui. Bob possède en même temps une clé secrète associée, qui est utilisée pour le déchiffrement. Par conséquent, la clé de Bob est constituée de deux parties, une clé publique k_{pub} et une clé privée k_{pr}.

Une analogie simple de ce système est le fonctionnement d'une boîte aux lettres. N'importe qui peut mettre une lettre dans la boîte, c.-à.-d. chiffrer, mais seulement une personne avec une clé privée (secrète) peut retirer les lettres, c.-à.-d. déchiffrer. Si on suppose que nous avons des cryptosystèmes avec une telle fonctionnalité, un protocole standard pour le chiffrement à clé publique est montré dans la figure suivante :



Pour réaliser un cryptosystème à clé publique, nous avons besoin d'une fonction f à sens unique, c'est à dire

1. $c = f(m)$ est calculatoirement facile, et
2. $m = f^{-1}(c)$ est calculatoirement difficile, sauf pour les personnes ayant une information spéciale. On dit alors que la fonction f est une *fonction à trappe*.

2.3 Quelques familles d'algorithmes à clé publique

Contrairement à la cryptographie symétrique, où le nombre d'algorithmes est assez important, pour les chiffrements asymétriques, la situation est bien différente. En particulier, il n'existe que trois familles principales de cryptosystèmes à clé publique qui sont utilisées dans la pratique :

— *Cryptosystèmes basés sur le problème de la factorisation.* Le système le plus connu de cette famille est RSA.
— *Cryptosystèmes basés sur le problème du logarithme discret.*
— *Cryptosystèmes basés sur les courbes elliptiques.* Ces systèmes ont l'avantage d'avoir des clés relativement courtes, par rapport aux systèmes basés sur la factorisation et le logarithme discret.

Tous les cryptosystèmes ci-dessus sont basés sur des problèmes difficiles pour un ordinateur classique mais qui pourront être résolus par un ordinateur quantique. Pour cette raison, des cryptosystèmes basés sur d'autres problèmes difficiles (dans le sens classique et quantique) ont été développés. Ces cryptosystèmes à clé publique sont notamment basés sur des problèmes issues de la théorie des codes correcteurs d'erreurs, de réseaux euclidiens, des isogenies, des fonctions de hachage ou encore de la cryptographie multivariée. Un concours publique, lancé par le NIST en 2018 est actuellement en cours pour standardiser les candidats les plus robustes et les plus performants de la competition.

3 RSA

Le cryptosystème RSA a été inventé par Ron Rivest, Adi Shamir et Leonard Adleman en 1977. Il est aujourd'hui le cryptosystème à clé publique le plus utilisé dans le monde. Il faudrait noter que RSA était sous brevet aux États-Unis jusqu'à 2000. Ce système est basé sur le problème de la factorisation. Multiplier deux nombres premiers est calculatoirement facile, mais factoriser le résultat est très dur.

Nous allons maintenant voir comment fonctionne RSA.

3.1 Génération des clés

Une caractéristique particulière des cryptosystèmes à clé publique est qu'il existe une procédure de mise en place pendant laquelle les clés publiques et privées sont calculées. Cette phase, qui change d'un cryptosystème à l'autre, peut être assez complexe. Notons ici, qu'une telle procédure n'existe pas pour les cryptosystèmes à clé secrète, où les clés sont habituellement générées par un générateur pseudo-aléatoire.

1. Choisir deux grands nombres premiers p et q.
2. Calculer $n = p \cdot q$.
   (Il s'agit du paramètre le plus important de RSA).
3. Calculer $\phi(n) = (p-1)(q-1)$.
4. Choisir $k_{\text{pub}} = e \in \{1, 2, \ldots, \phi(n)-1\} = \mathbb{Z}^*_{\phi(n)}$ tel que $\text{pgcd}(e, \phi(n)) = 1$.
5. Calculer $k_{\text{pr}} = d$ tel que $d \cdot e \equiv 1 \mod \phi(n)$.

Formellement, les clés publiques et privées sont :
— $k_{\text{pub}} = (n, e)$
— $k_{\text{pr}} = (p, q, d)$.

3.2 Chiffrement et Déchiffrement

Chiffrement. Le chiffrement de $m \in \mathbb{Z}_n$ est

$$\text{ENC}(m) = m^e \mod n$$

Déchiffrement. Le déchiffrement de $c \in \mathbb{Z}_n$ est

$$\text{DEC}(c) = c^d \mod n.$$

3.3 Remarques

3.3.1 Sécurité de RSA

Un adversaire, disons Oscar, souhaite casser RSA. Oscar connaît n et sait appliquer l'algorithme d'Euclide étendu. L'élément essentiel qu'il empêche de faire les mêmes calculs que Bob pour obtenir d est qu'il ne connaît pas $\phi(n)$. Pour connaître $\phi(n)$, Oscar doit connaître la factorisation de n en nombres premiers.

3.3.2 Taille des clés

L'entier n qui constitue le modulo RSA, doit être de très grande taille, afin que sa factorisation ne soit pas possible avec la puissance de calcul disponible aujourd'hui. En 2020, une équipe de chercheurs (dont plusieurs chercheurs français) ont factorisé un module RSA-795, c'est-à-dire un chiffrement RSA, avec n de taille 795 bits. Aujourd'hui, l'utilisation de RSA-1024 est déconseillée. Depuis 2015, un module de 2048 bits est conseillé, voir même 4096 pour des secrets trop importants.

Remarque : Pour un module RSA n les nombres premiers sont de taille $n/2$ bits.

3.3.3 Comment trouver e et d tels que $ed \equiv 1 \mod \phi(n)$

Nous avons besoin de trouver un entier $e \in \mathbb{Z}^*_{\phi(n)}$, avec $ed \equiv 1 \mod \phi(n)$.

Notons d'abord que e doit posséder un inverse multiplicatif modulo $\phi(n)$, et c'est pour cela qu'on exige $e \in \mathbb{Z}^*_{\phi(n)}$. Puisque p et q sont des grands nombres premiers, ils sont forcement impairs, impliquant que $\phi(n) = (p-1)(q-1)$ doit être pair. Pour trouver e, on choisit un entier impair dans $\mathbb{Z}_{\phi(n)}$ et on vérifie si cet entier a des facteurs communs avec $\phi(n)$. S'il a des facteurs communs, on répète ce test pour un autre choix de e. Après quelques tests on trouve un entier e qui est relativement premier avec $\phi(n)$. Dans la pratique, il n'y a qu'un petit ensemble de valeurs qui sont utilisées pour e, et toutes ces valeurs sont petites. Une valeur de e très populaire est par exemple $e = 17$.

Pour calculer d, on applique l'algorithme d'Euclide étendu pour $\phi(n)$ et e et on trouve des entiers u et v tels que

$$\text{pgcd}(\phi(n), e) = u\phi(n) + ve.$$

L'entier v est donc l'inverse de e modulo $\phi(n)$ et on pose alors $d = v$.

3.4 Pourquoi $m^{ed} \equiv m \mod n$?

Pour répondre à cette question nous avons besoin d'un résultat important d'arithmétique. Il s'agit du Petit Théorème de Fermat.

Théorème 3.1 (Petit théorème de Fermat)**.** *Soit a un entier et p un nombre premier. Alors*

$$a^p \equiv a \mod p.$$

Si a n'est pas divisible par p, le petit théorème de Fermat s'écrit souvent sous la forme suivante (très utile pour les applications cryptographiques) :

$$a^{p-1} \equiv 1 \mod p.$$

Nous allons montrer dans cette partie que

$$m^{ed} \equiv m \mod n.$$

Il est équivalent de prouver que

$$m^{ed} \equiv m \mod p \text{ et } m^{ed} \equiv m \mod q.$$

— Comme p et q divisent $n = p \cdot q$, le premier sens est immédiat.
— Inversement, si $m^{ed} \equiv m \mod p$ et $m^{ed} \equiv m \mod q$, comme p et q sont premiers entre eux, le théorème des restes chinois implique que $m^{ed} \equiv m \mod p \cdot q = n$.

Nous commençons alors par montrer que $m^{ed} \equiv m \mod p$. Nous avons deux cas à considérer :

1. p divise m
2. p ne divise pas m.

Cas 1 : Si p divise m, alors $m \equiv 0 \mod p$ mais également $m^{ed} \equiv 0 \mod p$.

Cas 2 : Les clés e et d sont choisies de façon que

$$ed \equiv 1 \mod (p-1)(q-1).$$

Par conséquent, il existe un entier positif k tel que

$$ed = 1 + k(p-1)(q-1).$$

$$\begin{aligned} m^{ed} &\equiv m^{1+k(p-1)(q-1)} \mod p \\ &\equiv m \cdot (m^{(p-1)})^{k(q-1)} \mod p \\ &\equiv m \cdot 1^{k\cdot(q-1)} \mod p \\ &= m \mod p. \end{aligned}$$

Nous pouvons répéter la même procédure pour montrer que $m^{ed} \equiv m \mod q$, en intérchangeant simplement les rôles de p et q.

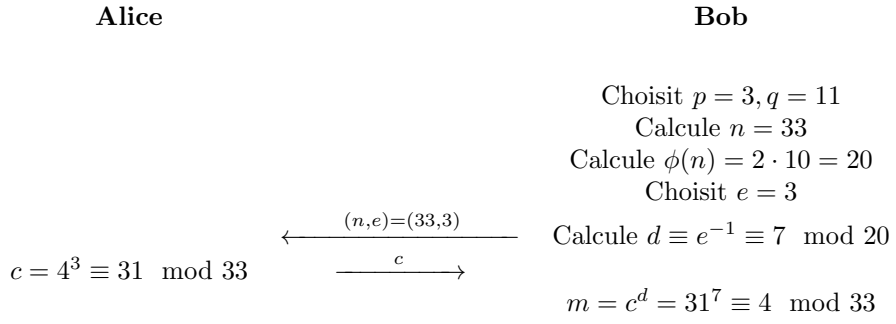


FIGURE 1 – Exemple de chiffrement avec RSA

Exemple Dans l'exemple suivant Alice souhaite envoyer le message $m = 4$ à Bob.

3.5 Exponentiation rapide : Algorithme Square and Multiply

Dans la vraie vie, la phase de génération des clés ne s'effectue qu'une seule fois. Par exemple, si notre carte bancaire utilise le cryptosystème RSA, alors la génération des clés s'effectue dans l'usine de fabrication et les clés sont ensuite gravés en dur sur notre carte bancaire. Au contraire, à chaque fois que nous souhaitons utiliser le système RSA pour envoyer et recevoir des messages, les procédures de chiffrement et de déchiffrement doivent se produire.

Par conséquent, un soin particulier doit être accordé à la manière dont ces opérations sont implémentés, sinon le système va être trop lent pour un usage pratique.

3.5.1 Introduction

Pour utiliser RSA dans la pratique nous avons besoin d'implémenter les deux opérations suivantes correspondant au chiffrement et déchiffrement.

$$\begin{aligned} c &\equiv m^e \mod n \\ m &\equiv c^d \mod n, \end{aligned}$$

avec des nombres très très grands. On commence avec un exemple sur comment effectuer l'opération d'exponentiation.

Exemple 3.2. *Nous allons voir comment calculer x^4. La façon naïve pour calculer cela est d'effectuer les opérations suivantes :*

$$\begin{aligned} x \cdot x &= x^2 \\ x^2 \cdot x &= x^3 \\ x^3 \cdot x &= x^4. \end{aligned}$$

Le coût de cette suite d'opérations est 3 multiplications, qu'on notera `3MUL`.

Question : *Est-ce qu'il existe une façon de faire le même calcul avec un coût inférieur au 3 multiplications ? La réponse est positive. Par exemple on peut calculer le même résultat en utilisant la suite suivante d'opérations :*

$$\begin{aligned} x \cdot x &= x^2 \\ x^2 \cdot x^2 &= x^4. \end{aligned}$$

Le coût de ce calcul est `2MUL`. Cependant, dans la pratique, nous n'amenons à des opérations avec des nombres beaucoup plus gros. Commençons par voir un exemple avec un nombre plus grand.

Exemple 3.3. *Calculer x^8. Commençons d'abord par la méthode naïve, de complexité `7MUL`.*

$$\begin{aligned} x \cdot x &= x^2 \\ x^2 \cdot x &= x^3 \\ &\vdots \\ x^7 \cdot x &= x^8. \end{aligned}$$

On calcule maintenant la même chose d'une façon plus intelligente :

$$\begin{aligned} x \cdot x &= x^2 \\ x^2 \cdot x^2 &= x^4 \\ x^4 \cdot x^4 &= x^8. \end{aligned}$$

La complexité maintenant est de seulement `3MUL`*.*

Cependant, dans la vraie vie, les nombres sont beaucoup plus grands. Comment faut-il faire par exemple pour calculer $x^{2^{1024}}$?

On peut comme avant appliquer la méthode naïve. Cette méthode a une complexité de $(2^{1024} - 1)$`MUL`. En faisant cela de façon intelligente nous pouvons réduire le calcul à 1024`MUL`. On voit bien que la méthode naïve a une complexité linéaire (en l'exposant) quant à la deuxième méthode elle a une complexité logarithmique.

Cependant, dans tous ces exemples, l'exposant était une puissance de 2. Nous allons maintenant voir comment généraliser cet algorithme à des exposants d'une forme quelconque.

3.5.2 Algorithme Square-and-Multiply

Commençons par un exemple.

Exemple 3.4. *Calculer x^{26}.*

$$\begin{aligned} \texttt{SQ} \quad & x \cdot x = x^2 \\ \texttt{MUL} \quad & x \cdot x^2 = x^3 \\ \texttt{SQ} \quad & x^3 \cdot x^3 = x^6 \\ \texttt{SQ} \quad & x^6 \cdot x^6 = x^{12} \\ \texttt{MUL} \quad & x \cdot x^{12} = x^{13} \\ \texttt{SQ} \quad & x^{13} \cdot x^{13} = x^{26} \end{aligned}$$

Pour déterminer à quel étape on doit élever au carré et à quelle étape on doit effectuer une multiplication, on écrit la décomposition binaire de l'exposant.

$$x^{26} = x^{11010_2}.$$

On remarque alors bien qu'en commençant par les bits du poids fort de l'exposant (on ignore cependant le bit le plus fort), on applique toujours une operation `square` mais qui est suivie d'une opération `multiply` seulement dans le cas où le bit est égal à 1.

Algorithme 1 : Square-and-Multiply

Données : Un entier x, le module n et un exposant $H = \sum_{i=0}^{t} h_i 2^i$, avec $h_i \in \{0, 1\}$ et $h_t = 1$
Résultat : $x^H \mod n$
$r \leftarrow x$;
pour $i = t - 1, \ldots, 0$ **faire**
  $r \leftarrow r^2 \mod n$;
  si $h_i = 1$ **alors**
    $r \leftarrow r \cdot x \mod n$;
Renvoyer r ;

Complexité Soit un exposant H de $t+1$ bits. Le nombre d'élévations au carré est indépendant de la valeur actuelle de H, mais le nombre de multiplications est égal à son poids de Hamming, c.-à-d. le nombre de 1 dans sa représentation binaire. Par conséquent nous avons

$$\begin{aligned} |\mathtt{SQ}| &= t \\ |\overline{\mathtt{MUL}}| &= 0.5t, \end{aligned}$$

où $\overline{\mathtt{MUL}}$ est le nombre de multiplications en moyenne. Puisque les exposants utilisés pour les applications cryptographiques ont des bonnes propriétés aléatoires, la supposition que la moitie des bits sont à 1 est une supposition tout à fait légitime.

3.6 Chiffrement rapide avec des petits exposants publics

Une astuce étonnamment simple et en même temps très efficace peut être utilisée quand il s'agit de faire des opérations avec la clé publique e. Dans ce cas, un entier très petit peut être choisi pour la clé publique e. En pratique, les trois valeurs, $e = 3, 17$ et $e = 2^{16} + 1$ sont d'une importance particulière.

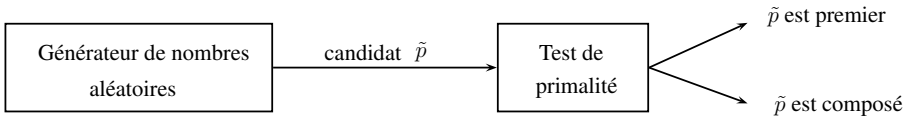
Clé publique e	représentation binaire de e
3	11_2
17	10001_2
$2^{16}+1$	10000000000000001_2

Tous les exposants du tableau ont un poids de Hamming particulièrement faible et ceci conduit à un nombre de multiplications très faible. Étonnement, la sécurité de RSA n'est pas affaiblie en utilisant des exposants publiques de petite taille. Dans ces situations, la clé privé d a en général une longueur complète même si e est petit. Cette situation produit une asymétrie dans le chiffrement et le déchiffrement. Le chiffrement est beaucoup plus rapide que le déchiffrement. Ceci est particulier à RSA et ne se vérifie pas avec d'autres cryptosystèmes à clé publique. Il existe une astuce pour légèrement accélérer le déchiffrement avec RSA, en utilisant le théorème des restes chinois. Cependant, même avec cette astuce, le ratio entre chiffrement et déchiffrement est autour de 1/3.

4 Générer des grands nombres premiers

La première étape pour la mise en place d'un cryptosystème RSA est la génération de deux très grands nombres premiers p et q. Leur produit $n = p \cdot q$ forme le module RSA. Pour cette raison, la taille de p et q en bits, doit être égale à la moitie de la taille en bits du module n. Dans le cadre par exemple de RSA-2048, les deux nombres premiers doivent avoir une longueur de 1024 bits.

L'approche générale consiste à utiliser un générateur des nombres aléatoires pour générer un entier, dont on testera ensuite la primalité. Cette situation est illustrée dans la figure ci-dessous.



Il est très important d'utiliser dans cette démarche un bon générateur aléatoire, qui ne doit dans aucun cas être prévisible. Si un attaquant réussit à deviner les nombres premiers qui composent le module RSA, alors le système est immédiatement cassé.

La première question à laquelle on doit répondre est :

"*Combien de nombres doit-on tester avant de trouver un nombre premier ?*"

La réponse à cette question est donnée par le *Théorème des Nombres Premiers.*

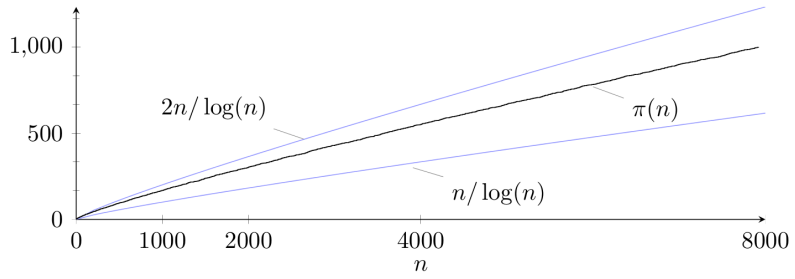


FIGURE 2 – La fonction $\pi(n)$ pour les 1000 premiers nombres premiers. ("*A Course in Cryptography*", Rafael Pass et Abhi Shelat, 2010.)

4.1 Les grands nombres premiers sont-ils fréquents ?

Euclide a prouvé dans ses *Éléments* l'infinité des nombres premiers. Cependant, la question intéressante pour nous est combien de nombres premiers d'une taille donnée existe-il ? Pour mettre par exemple en place RSA-2048, on s'intéresse au nombre de premiers d'une taille de 1024 bits. Le *Théorème des Nombres Premiers* nous aide à répondre à cette question.

Théorème 4.1 (Théorème des Nombres Premiers)**.** *Soit $\pi(n)$ le nombre de premiers qui sont inférieurs à n. Alors*

$$\pi(n) \approx \frac{n}{\ln(n)} \quad (n \to +\infty).$$

Un graphique de la fonction $\pi(n)$ pour les 1000 premiers nombres premiers est donné dans la figure 1. C'est en inspectant cette courbe que Gauss à l'âge de 15 ans a eu l'idée de cette approximation.

Le tableau suivant contient l'approximation ainsi que le nombre exact de nombres premiers pour différentes valeurs de n. Nous y voyons que l'approximation est effectivement assez bonne.

n	$n/\ln(n)$	$\pi(n)$
10^3	145	168
10^4	1 086	1 229
10^5	8 686	9 592
10^6	72 382	78 498
10^7	620 420	664 579

Supposons maintenant qu'on veut générer un nombre premier p de k bits. Alors, p se trouve entre les entiers 2^{k-1} et $2^k - 1$. Le nombre de nombre premiers dans cet intervalle peut être approximé par

$$\pi(2^k) - \pi(2^{k-1}) \approx \frac{2^k}{\ln(2^k)} - \frac{2^{k-1}}{\ln(2^{k-1})} \approx \frac{2^{k-1}}{\ln(2^{k-1})},$$

puisque $\ln(2^k) = \ln(2 \cdot 2^{k-1}) = \ln(2) + \ln(2^{k-1})$, donc $\ln(2^k) \approx \ln(2^{k-1})$ pour k grand. En résumant, il y a 2^{k-1} entiers entre 2^{k-1} et $2^k - 1$ et approximativement

$$\frac{2^{k-1}}{\ln(2^{k-1})}$$

parmi eux sont des nombres premiers. Par conséquent, si on choisit un entier n dans cet intervalle, il sera premier avec probabilité

$$\frac{1}{\ln(2^{k-1})}.$$

Le tableau suivant contient les probabilités correspondantes pour différentes valeurs de k.

k	$1/\ln(2^{k-1})$
100	1/69
200	1/138
300	1/207
400	1/277
500	1/346
1000	1/694

Nous pouvons voir en particulier, que dans le cas de RSA-2048 nous avons une chance sur $\ln(2^{1023})$ donc une probabilité de $\approx 1/709$ de générer un nombre premier de 1024 bits. Cette chance double si on se restreint sur les entiers impairs. Par conséquent, on doit générer à peu près 355 nombres avant de tomber sur un nombre premier.

5 Tests de primalité

La deuxième étape de la procédure est de tester si l'entier généré est un nombre premier ou pas. Un algorithme qui nous permet de décider cela est appelé *test de primalité*. L'idée la plus intuitive pour faire cela consiste à choisir aléatoirement un entier impair de k bits et tester tous les entiers plus petits que k pour voir si un parmi eux divise k. Ceci revient à factoriser le nombre en question.

5.1 Test naïf

Le test naïf pour décider si un nombre n est premier ou composé consiste à tester si les entiers $2, 3, \ldots, n-1$ divisent n. Si un parmi ces entiers divise n alors on déduit que le nombre est composé, sinon on conclue que le nombre est premier.

Exemple 5.1. *Tester si 13 est un nombre premier.*

On teste pour tous les entiers $k \in \{2, \ldots, 12\}$, si k divise 13. Dans cet exemple aucun de ces entiers ne divise pas 13, par conséquent on déduit que 13 est un nombre premier.

En réalité, on n'est pas obligé de tester tous les entiers jusqu'à $n-1$. Par exemple, pour $n = 50$ tous les diviseurs sont

$$1, 2, 5, 10, 25.$$

On voit ici que le plus grand diviseur est 25. Ceci est vrai dans le cas général. Un diviseur d'un entier n ne peut pas dépasser $n/2$. Encore mieux, on peut montrer qu'il suffit de tester seulement les entiers qui sont inférieurs à $\sqrt{n}$. En effet, si n possède un facteur plus grand que $\sqrt{n}$, alors il a forcement au moins un facteur plus petit que $\sqrt{n}$. Nous pouvons finalement accélérer la recherche en ne prenant en compte que des nombres premiers inférieurs à $\sqrt{n}$. Pour cela il suffit de pré-calculer et de stoker dans une table tous les nombres premiers inférieurs ou égaux à $\sqrt{n}$. Le *crible d'Eratosthène* peut par exemple être utilisé dans ce but.

Algorithme 2 : Méthode naïve

Données : Un entier n
pour *tous les nombres premiers* $p \leq \sqrt{n}$ **faire**
  si p *divise* n **alors**
    renvoyer `composé` ;
renvoyer `premier` ;

Complexité. La complexité en temps de l'algorithme 2 dans le pire cas est $\pi(\sqrt{n}) \approx \dfrac{2\sqrt{n}}{\ln(n)}$ divisions, c'est-à-dire $\mathcal{O}(\sqrt{n})$ opérations.

Exemple 5.2. *Pour un module RSA de 2048 bits, la complexité de cette méthode est non loin de*

$$\pi(\sqrt{2^{2048}}) \approx \frac{2\sqrt{2^{2048}}}{\ln(2^{2048})} = \frac{2 \cdot 2^{1024}}{2048 \cdot \ln 2} \approx 2^{1013} \text{ divisions.}$$

La méthode naïve est un exemple d'un algorithme *déterministe*, c'est-à-dire la sortie de l'algorithme est exacte avec probabilité 1. Il existe d'autres algorithmes déterministes pour tester la primalité. Un exemple est le *test de Wilson*, basé sur le théorème suivant :

Théorème 5.3 (Théorème de Wilson)**.** *Un entier $n > 1$ est un nombre premier si et seulement si*

$$(n-1)! \equiv -1 \mod n.$$

Ce test qui est basé sur une propriété très simple a cependant une complexité trop élevée. En effet, nous avons besoin d'effectuer à peu près n multiplications modulaires, par conséquent la complexité est $\mathcal{O}(n)$.

Dans la pratique, nous utilisons des tests beaucoup plus efficaces pour tester la primalité d'un nombre. Il s'agit des méthodes *probabilistes* ; ces algorithmes décident si un entier est premier ou pas avec une probabilité p. Plus précisément, en utilisant un test probabiliste pour tester un candidat $\tilde{p}$, les sorties possibles de l'algorithme sont :

1. "$\tilde{p}$ est composé" et ceci est toujours vrai.
2. $\tilde{p}$ est premier" qui est seulement vrai avec une probabilité p.

Si la sortie de l'algorithme est "composé", alors il n'y a pas de doute. Le nombre est obligatoirement composé. Si la sortie est "premier", p est probablement premier. Mais, dans des rares cas, l'algorithme énonce à tort que le nombre candidat est premier. Pour traiter ce comportement, nous répétons le test un nombre de fois suffisant pour que la probabilité que le test "se trompe" devienne très faible.

5.2 Test de Fermat

Le premier test probabiliste qu'on examinera est basé sur le Petit Théorème de Fermat :

Théorème 5.4 (Petit Théorème de Fermat)**.** *Soit p un nombre premier et a un nombre entier qui n'est pas divisible par p. Alors*

$$a^{p-1} \equiv 1 \mod p.$$

L'idée de ce test est que l'égalité $a^{p-1} \equiv 1 \mod p$ est vérifiée pour tout nombre premier p. En conséquence, si pour l'entier candidat n

$$a^{n-1} \not\equiv 1 \mod n,$$

alors n n'est sûrement pas un nombre premier. Le test de primalité basé sur cette propriété est le suivant :

Algorithme 3 : Test de Fermat

Données : Un entier n et un nombre de répétitions t
pour $i = 1$ *jusqu'à* t **faire**
  Choisir aléatoirement a tel que $1 < a < n-1$;
  si $a^{n-1} \not\equiv 1 \mod n$ **alors**
    renvoyer `composé` ;
renvoyer `premier` ;

Cependant, l'inverse du théorème de Fermat n'est pas vrai. Il peut avoir des nombres composés n qui vérifient l'égalité $a^{n-1} \equiv 1 \mod n$, pour un entier a.

Parmi les entiers a, $1 < a < n$, qui ne vérifient pas l'égalité de Fermat, il y a évidement ceux qui ne sont pas premiers avec n. Si l'on trouve un tel entier a, on dit que a est un témoin de non primalité de n qui est issu de la divisibilité. Par exemple,

$$4^5 \equiv 1024 \equiv 4 \not\equiv 1 \mod 6,$$

donc 4 est un témoin de non primalité de 6 issu de la divisibilité. Pour les entiers a premiers avec n ne vérifiant pas l'égalité de Fermat, autrement dit, ceux constituant un véritable contre-exemple au petit théorème de Fermat, on utilise la terminologie suivante :

Définition 5.5. *Soit un entier $n \geq 2$. On appelle* témoin de Fermat pour n, *tout entier a avec* $\text{pgcd}(a, n) = 1$, *tel que $1 < a < n$ et $a^{n-1} \not\equiv 1 \mod n$.*

Si n est composé, il y a très souvent de nombreux témoins de Fermat pour n, mais pas toujours. En effet, il existe des entiers composés qui ne possèdent pas de témoins de Fermat. On appelle ces entiers *nombres de Carmichael*.

Définition 5.6 (Nombre de Carmichael)**.** *Un entier positif composé n est appelé nombre de Carmichael si pour tout entier a avec* $\text{pgcd}(a, n) = 1$

$$a^{n-1} \equiv 1 \mod n.$$

Exemple 5.7 (Le plus petit nombre de Carmichael)**.** *L'entier $n = 561 = 3 \cdot 11 \cdot 17$ est un nombre de Carmichael puisque*

$$a^{560} \equiv 1 \mod 561$$

pour tout a tel que $\text{pgcd}(a, 561) = 1$.

Les nombres de Carmichael sont très rares. Il existe par exemple seulement 246.683 nombres de Carmichael inférieurs à 10^{16}. Le nombre de premiers inférieurs à 10^{16} est quant à lui égal à 279.238.341.033.925. La probabilité alors qu'un nombre premier inférieur à 10^{16} soit un nombre de Carmichael est plus petite que $1/10^9$.

Complexité. Si un algorithme rapide est utilisé pour l'exponentiation modulaire (par exemple Square-and-Multiply) la complexité en temps de l'algorithme de Fermat est

$$\mathcal{O}(t \cdot \log^2 n \cdot \log(\log(n)) \cdot \log(\log(\log(n)))),$$

où t est le nombre de fois que le test est effectué.

PGP. Le logiciel de chiffrement PGP utilise le test de Fermat comme test de primalité pour les nombres générés. En particulier, seulement quatre témoins y sont utilisés : $2, 3, 5$ et 7. Utiliser des témoins supplémentaires diminue la chance qu'un nombre composé n soit considéré premier, bien que très peu de grands nombres puissent tromper ces 4 témoins.

Si les facteurs premiers d'un nombre de Carmichael sont grands, il existe qu'un très petit nombre d'entiers a pour lesquels le test de Fermat détecte que le nombre est réellement composé.

Malgré cela, l'existence des nombres de Carmichael est la raison pour laquelle un autre test plus puissant est plus souvent utilisé dans la pratique pour tester la primalité des nombres générés pour RSA. Il s'agit du test de Miller-Rabin.