Chapitre *VIII*

# Cryptographie à clef publique

Dans les modèles étudiés jusqu'ici, la clef $k$ définit les règles de chiffrement $e_k$ et de déchiffrement $d_k$. Ce type de cryptosystème est dit symétrique.

Une fois la clef de chiffrement $k$ connue, les fonctions de chiffrement $e_k$ et de déchiffrement $d_k$ étant publique, le système cryptographique n'est plus sûr.

Un défaut d'un cryptosystème symétrique est qu'il exige la communication de la clef $k$ par un canal sûr (potentiellement difficile à réaliser pour des communications distantes ou si l'un des protagonistes est placés en milieu "hostile").

Un cryptosystème à clef publique est tel que :

- la clef de chiffrement $a$ est publique (aussi appelée clef publique), et est utilisable par toute personne souhaitant envoyer un message chiffré au destinataire en utilisant la fonction de chiffrement $e_k$ (avec $k = a$).
- la clef de déchiffrement $b$ est impossible à retrouver à partir de $a$, et n'est connue que par la destinataire (clef privée). Ce dernier peut utiliser sa clef privée et la fonction de déchiffrement $d_k$ (avec $k = b$) pour déchiffrer le message.

**Analogie** : revient pour l'expéditeur à placer son courrier dans un coffre fort ouvert dont il ne connaît pas la combinaison, et à le fermer. Seul le destinataire peut ouvrir le coffre.

## 1 Complexité

Comme nous l'avons vu, la notion de sécurité parfaite n'existe pas pour un cryptosystème excepté pour l'instant dans le cas du chiffre de Vernam (lequel

n'est pas utilisable en pratique).

La sécurité d'un cryptosystème va donc reposer entièrement sur la complexité de sa cryptanalyse. Mais comment définir mathématiquement la complexité ?

La complexité se mesure :

- pour un programme, avec la **complexité temporelle** en trouvant la borne supérieure du temps nécessaire pour résoudre un problème avec ce programme.
- pour un objet, avec la **complexité de Kolmogorov** en trouvant la plus petite description de l'objet.

## 1.1 Complexité temporelle

Pour un programme $P$ avec une entrée $x$, on définit :

- sa longueur $n = |x|$. A noter que $x$ doit être un codage raisonnable des paramètres de $P$. Par exemple, pour stocker la valeur $v$, on utilise $\log_2 v$ bits (ou tout autre base $\geq 2$), et non par exemple $v$ bits.
- $f(n)$ est le nombre maximum d'opération que le programme $P$ exécute pour une entrée $x$ de taille $n$.

Le programme $P$ appartient à la classe de complexité TIME($g(n)$) si il existe un $n_0$ et une constante $C$ positive telles que $\forall n > n_0$, $f(n) \leq C.g(n)$. On dit que $f(n)$ est en $O(g(n))$.

**Remarques** :

- La classe de complexité ne dépend pas de la puissance de la machine (*i.e.* sur une machine 1000 fois plus puissante, multiplier $C$ par 1000).
- Un programme $P$ dans TIME($n^k$) est dit à temps polynomial. Un programme dans TIME($2^n$) est dit à temps exponentiel.
- On remarquera que $P \in$ TIME($n^k$) implique $P \in$ TIME($2^n$) puisque TIME($n^k$) $\subset$ TIME($2^n$).

Les classes de complexité servent en conséquence à borner supérieurement la complexité d'un algorithme.

**Exemples** :

- le nombre d'opérations nécessaires sur deux entiers codés respectivement sur $n$ et $m$ ($\leq n$) bits est : $O(n)$ pour la comparaison ou l'addition, $O(n.m)$ pour la multiplication ou la division. la comparaison de deux nombres de $n$ bits prend $O(n)$ opérations.
- un produit entre deux matrices de taille $n = p \times p$ effectue $p^3 = n\sqrt{n} \leq n^2$ multiplications.
- un tri d'un liste de taille $n$ effectue de l'ordre de $n \log n \leq n^2$ comparaisons et échanges.

- trouver le chemin le plus court qui passe par $n$ villes est de l'ordre de $2^n$ opérations.

En fonction de la complexité, on a les temps d'exécution suivants :

| | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| $n$ | 0,00001" | 0,00002" | 0,00003" | 0,00004" | 0,00005" | 0,00006" |
| $n^2$ | 0,0001" | 0,0004" | 0,0009" | 0,0016" | 0,0025" | 0,0036" |
| $n^3$ | 0,001" | 0,008" | 0,027" | 0,064" | 0,125" | 0,216" |
| $n^5$ | 0,1" | 3,2" | 24,3" | 1,7' | 5,2' | 13,0' |
| $2^n$ | 0,001" | 1,0" | 17,9' | 12,7 jours | 35,7 ans | 36,6 Kan |
| $3^n$ | 0,059" | 58' | 6,5 ans | 385,5 Kan | 22.7 Gan | 1,3 Tan |

où Kan=mille an, Man=million d'années, Gan=milliard d'années, Tan=mille Gan.

En résumé :

- calcul faisable = calcul en temps polynomial.
- calcul infaisable = calcul en temps exponentiel.

**Exemple**

Cherchons la complexité du test de primalité d'un nombre $m$ en utilisant la méthode naïve (à savoir tester la divisibilité de $m$ par tous les nombres de 1 à $\sqrt{m}$).

- l'entrée a une taille $n = \lfloor \log_2 m \rfloor$, donc $m \geq 2^n$.
- l'algorithme consiste à tester le résultat de $\sqrt{m}$ division (en $O(n^2)$).

Donc, l'algorithme a une complexité de $O(n^2.2^n) = O(2^n)$.

Autrement dit, il a une complexité exponentielle.

Donc, avec l'augmentation du nombre de bits, il devient exponentiellement plus difficile de tester si un nombre est premier.

Attention, il ne s'agit que de la complexité de cet algorithme, et cela ne signifie nullement qu'il n'existe aucun autre algorithme permettant de tester si un nombre est premier plus rapidement (c'est d'ailleurs le cas).

Dans le cadre de la cryptographie, on souhaiterait donc que :

- la cryptanalyse du cryptosystème s'effectue en temps exponentiel.
  dans ce cas, l'augmentation de la puissance de calcul par 1.000.000 n'améliore pas ou peu les chances de le casser. Il suffit d'augmenter un peu la taille de la clef pour augmenter dramatiquement la complexité.
- Informatique théorique = borne supérieure de complexité.
  En cryptographie, on voudrait que le complexité représente la difficulté minimale (= que ce ne soit pas moins facile que la complexité).

Malheureusement,

- avec les principes sur lesquelles sont basés les cryptosystèmes modernes, leur complexité est au mieux subexponentielle (voir ci-après).
  Donc le succès d'une attaque est toujours une affaire de temps ou de moyen.
- la mesure de la complexité est celle du temps "au pire", mais pas du temps au mieux.
  En conséquence, même avec un cryptosystème de complexité donnée, il faudra faire attention de ne pas choisir une clef qui facilite la cryptanalyse, et que la complexité avec la clef choisie atteint le niveau désiré.
- en général, la complexité d'un problème est mesurée comme celle du meilleur algorithme connu. **P** ≠ **NP** n'est toujours pas démontré.

**Définition 91** (classe de complexité $L(a,b)$)**.** Pour $b \geq 0$ et $0 \leq a \leq 1$, la classe $L(a,b)$ est la classe des algorithmes dont le temps d'exécution pour une entrée de taille $n$ est $O\left(e^{(b+o(1))n^a(\log n)^{1-a}}\right)$.

**Rappel :** $o(1)$ est une fonction que tend vers 0 quand $n \to \infty$.

On remarquera que :

- $L(0,b)$ est la classe des algorithmes à temps polynomiaux (en $O(n^b)$),
- $L(1,b)$ est la classe des algorithmes à temps exponentiel (en $O(e^{bn})$).

**Définition 92** (classe subexponentielle)**.** La classe de complexité subexponentielle est $\bigcup_{0<a<1,0<b} L(a,b)$.

**Exemple** : On conjecture que le meilleur algorithme pour factoriser un entier est en $L(1/3, 1.923)$.

## 1.2 Complexité de Kolmogorov

On avait déjà fait remarquer que la complexité temporelle permettait de mesurer la complexité la pire, et que l'on peut se retrouver dans des cas la complexité effective est beaucoup moins grande. Évidemment, ce cas de figure n'est pas acceptable dans le cadre de la cryptographie.

**Exemple** : un algorithme utilise comme masque une suite aléatoire de 64 bits tirés uniformément. Considérons maintenant les deux suites suivantes :

0101010101010101010101010101010101010101010101010101010101010101
1001101111011111101001101101011011110011011111100111100101100111

La première chaîne peut être décrite simplement comme 32 répétition de 01. La seconde chaîne n'a pas de description évidente.

Un conséquence, si cette suite est utilisée comme un masque binaire, alors le résultat du premier sera évidemment beaucoup plus prévisible que la seconde.

**Définition 93** (Complexité de Kolmogorov)**.** La complexité de Kolmogorov de $X$ est la description la plus courte qu'il est possible de faire de $X$.

On peut démontrer que pour la majorité des objets, la description la plus courte de l'objet est l'objet lui-même. Mais comment la calculer ?

Si la complexité de Kolmogorov n'est en général pas calculable, elle peut l'être pour une suite binaire finie.

En particulier, les registres à décalage à rétroaction linéaire (Linear Feedback Shift Register) constituent un modèle permettant d'estimer la complexité linéaire d'une suite binaire. Un LFSR de longueur $N$ est une suite $(s_i)$ définie par :

- ses coefficients $c = (c_0, c_1, \ldots, c_{N-1})$ où $c_i \in \mathbb{B}$,
- son état initial $s = (s_0, s_1, \ldots, s_{N-1})$ où $s_i \in \mathbb{B}$
- les termes suivants $i > N$ sont calculés par la fonction récursive : $s_{i+1} = (c_0.s_i + c_1.s_{i-1} + \ldots + c_{N-1}.s_{i-(N-1)}) \bmod 2$

On définit la complexité linéaire d'une suite binaire $(x_0, \ldots, x_{n-1})$ de longueur $n$ comme étant la longueur $N$ du plus petit LFSR telle qu'il existe des coefficients et un état initial permettant de générer la totalité de la suite.

L'algorithme de Massey-Berkelamp est un algorithme simple permettant de déterminer en temps linéaire les coefficients du plus petit LFSR (non abordé dans ce cours, cf wikipedia).

Même si un LFSR de longueur $N$ peut produire des suite de longueur $2^N - 1$ avant de se répéter, l'algorithme de Massey-Berkelamp montre que les LFSRs ont une complexité trop faible pour être utilisée en cryptographie comme fonction d'encodage.

# 2 Fonction à sens unique

**Définition 94** (Fonction à sens unique)**.**
Une fonction $f : M \to C,\ x \mapsto f(x)$ est dite à sens unique si :

1. il est possible de calculer simplement $f(x)$ à partir de n'importe quel $x$.
2. pour la plupart des $y \in f(M)$, trouver un $x$ tel que $f(x) = y$ doit être considéré comme impossible (au minimum difficile).

En notant $n = \lceil \log_2 C \rceil$ (*i.e.* le nombre de bits pour coder $x$), on entend :

- calcul simple = calcul possible en temps polynomial (en $n^k$).
- calcul "difficile" = calcul en temps subexponentiel (en $k^{n^a \log n^{1-a}}$ où $0<a<1$).
  à savoir, le calcul demande des moyens très lourds (donc coûteux), disponibles à court ou à moyen terme.
- calcul "impossible" = calcul en temps exponentiel (non polynomial, en $k^n$).
  à savoir, le temps de calcul est prohibitif ou il est déraisonnable de trouver le résultat au hasard.

**Stockage de mots de passe**

Plutôt que de stocker un mot de passe $p$ dans la mémoire ou sur un disque, on stocke $f(p)$. Pour vérifier un mot de passe $p'$, on compare $f(p')$ à $f(p)$.

Si $f(p)$ est intercepté, l'impossibilité de l'inversion ne nuit pas à la sécurité du système.

**Signature de données**

Si $f$ n'est pas inversible, elle peut être utilisée pour calculer la "signature" $y$ de données $x$. En général, $|x| >> |y|$.

La signature $y = f(x)$ peut être utilisée afin que vérifier que les données ont été correctement réceptionnées (problèmes de transmission ou de falsification).

Cette signature doit être telle que :

- $f$ doit engendrer des signatures différentes si $x$ change.
- il doit être difficile de modifier $x$ en $x'$ tel que $f(x') = f(x)$.

**Définition 95** (Fonction à sens unique à trappe)**.**
Une fonction $f : M \to C$, $x \mapsto f(x)$ est dite une fonction à sens unique à trappe si :

1. $f(x)$ est une fonction à sens unique,
2. il existe une information secrète (la trappe) qui permet de construire la fonction $g$ telle que $g \circ f = \mathrm{Id}$.

En conséquence,

- calculer $f(x)$ doit être facile, mais il doit être difficile d'inverser $f$ sans connaître $g$.
- la construction de $(f, g)$ doit être facile.
- publier $f$ ne doit pas permettre d'obtenir facilement $g$.

Il y a donc deux algorithmes : $f$ pour chiffrer et $g$ pour déchiffrer.

Le chiffrement est par conséquent **asymétrique**.

Une fonction considérée comme difficilement inversible est l'exponentiation modulo un nombre premier (appelée exponentielle discrète ou exponentielle modulaire).

**Définition 96** (Exponentielle discrète)**.**
Soit $p$ un grand nombre premier et un nombre $\alpha$ primitif modulo $p$ :

$$\begin{aligned} f : \mathbb{Z}_p^* &\to \mathbb{Z}_p^* \\ x &\mapsto \alpha^x \end{aligned}$$

**Note :** $\alpha$ est choisi primitif de façon à ce que $f$ soit bijective.

La fonction inverse du l'exponentielle discrète est le **logarithme discret**.

Tous les algorithmes connus pour calculer le logarithme discret nécessitent un temps de calcul non polynomial en $\log_2 p$. Le meilleur algorithme actuellement connu est subexponentiel (en $O(e^{(64/9 \log\log p)^{1/3}.(\log\log n)^{2/3}})$).

En pratique, un tel calcul est prohibitif si $p$ comporte plus de quelques centaines de bits.

# 3 RSA

## 3.1 Principe

Le cryptosystème RSA (acronyme des noms de ses 3 auteurs, Rivers, Shamir et Adleman) a été proposé en 1977.

On note $\mathbb{P}$ l'ensemble des nombres premiers.

**Définition 97** (cryptosystème RSA)**.**
Soit $\mathcal{K} = \{(n, p, q, a, b)$ tels que $(p, q) \in \mathbb{P}^2$, $n = p.q$ et $a.b \equiv 1 \bmod \phi(n)\ \}$.
Soit $\mathcal{P} = C = \mathbb{Z}_n$.
**Chiffrement d'un chaîne** $x \in \mathcal{P}$, $e_k(x) = x^a \bmod n$.
**Déchiffrement d'un code** $y \in C$, $d_k(y) = y^b \bmod n$.

La clef $(n, p, q, a, b)$ se décompose comme suit :

- $(n, a)$ forment la clef publique.
- $(p, q, b)$ forment la clef privée.

Montrons que le chiffrement et le déchiffrement sont des opérations inverses.

**Théorème 52** (inversibilité de RSA)**.**
*Soit* $(p, q) \in \mathbb{P}^2$ *distincts,* $n = p.q$ *et* $a.b \equiv 1 \bmod \phi(n)$*. Pour tout* $1 \leq x < n$*, on a* $x^{a.b} \bmod n = x$*.*

**Démonstration:**
Comme $a.b \equiv 1 \bmod \phi(n)$, $\exists k \geq 1$ tel que $a.b = k.\phi(n) + 1$.

On a alors deux cas :

- si PGCD$(x, n) = 1$, ce qui est normalement le cas, la probabilité que $p \mid x$ (resp. $q \mid x$) est de $1/p$ (resp. $1/q$).

$$\begin{aligned}(x^a)^b \bmod n = x^{ab} \bmod n &= x^{k.\phi(n)+1} \bmod n \\ &= x^1.\left(x^{\phi(n)}\right)^k \bmod n = x.(1)^k \bmod n = x\end{aligned}$$

- si miraculeusement $p \mid x$,
  On a vu que :
  - si PGCD$(p, q) = 1$ alors $\phi(p.q) = \phi(p).\phi(q)$.
  - si $p \in \mathbb{P}$ alors $\phi(p) = p - 1$.

  En conséquence, $\phi(n) = \phi(p.q) = \phi(p).\phi(q) = (p-1).(q-1)$.
  Donc,

$$\begin{aligned}x^{ab} \bmod q = x^{k.\phi(n)+1} \bmod q &= x^{k(p-1)(q-1)+1} \bmod q \\ &= x.\left(x^{q-1}\right)^{k(p-1)} \bmod q = x.(1)^{k(p-1)} \bmod q = x\end{aligned}$$

$$x^{ab} \bmod p = 0$$

$x^{ab} \bmod n = x$ (par le théorème des restes chinois). □

## 3.2 Utilisation

La méthode pour utiliser le cryptosystème RSA est la suivante :

1. **Construction des clefs** :
   **clef publique :** $(n, a)$
   **clef privée :** $(p, q, b)$ tels que $(p, q) \in \mathbb{P}$, $n = p.q$ et $a.b = 1 \bmod \phi(n)$.
2. **Choix d'un protocole d'encodage du message** : il s'agit de trouver une méthode de transformation d'un texte en une suite d'entiers positifs inférieurs à $n$.
3. **Chiffrement et envoi d'un message** :
   (a) conversion du message $M$ à envoyer en une suite d'entiers $(x_1, \ldots, x_p)$ tels que $1 \leq x_k < n$.

(b) pour chaque $x_k$, calculer $y_k = {x_k}^a \bmod n$ et l'envoyer.

4. **Réception et déchiffrement d'un message** :

   (a) pour chaque $y_k$, calculer $x_k = {y_k}^b \bmod n$.

   (b) reconvertir la suite d'entiers $(x_1, \ldots, x_p)$ et réassembler $M$.

Le niveau de sécurité actuel de RSA est le suivant :

- 768 bits (232 décimaux) : cassé en 2009.
- 1024 bits (309 décimaux) : pas officiellement cassé, longueur la plus utilisée actuellement.
- 1536 bits (463 décimaux) : futur standard à court terme.
- 2048 bits (617 décimaux) : suffisant jusqu'en 2030.
- 3072 bits (925 décimaux) : pour une sécurité au-delà de 2030.

Ces niveaux de sécurité sont à la merci des évènements suivants :

- la découverte d'un algorithme de factorisation/logarithme discret encore plus proche du temps polynomial,
- le développement d'un ordinateur quantique opérationnel (sur lequel on sait déjà implémenter un algorithme de factorisation) car ce problème se résout sur un tel ordinateur en temps polynomial.
  **Note :** les deux plus grands nombres jamais factorisés sur un ordinateur quantique sont 21 (correct 1 fois sur 2 sur 150.000 expériences) et 143 (2012).

Des exemples d'attaques seront donnés ultérieurement.

1. choix de deux grands nombres premiers $p$ et $q$ tel que $p.q \geq 2^k$ où $k$ est le nombre de bits RSA (actuellement 1024 bits).

   Pour contrer les algorithmes de factorisation existant, il vaut mieux choisir $p$ et $q$ de longueurs similaires, et de longueur au moins égale à 512 bits.

2. calculer le produit $n = p.q$.

3. choisir un entier $a$ tel que $1 < a < \phi(n) = (p-1).(q-1)$ et $\text{PGCD}(a, \phi(n)) = 1$.

4. calcul d'un entier $b$ tel que $1 < b < \phi(n)$ et $a.b \equiv 1 \bmod \phi(n)$.

   Comme $\text{PGCD}(a, \phi(n)) = 1$, alors par Bézout, on sait $\exists x, y$ tel que $x.a + y.\phi(n) = 1$ et qu'en conséquence $x.a = 1 \bmod \phi(n)$, donc $x$ est l'inverse multiplicatif de $a$. On utilise donc l'algorithme d'Euclide étendu pour calculer $x$, $y$.

Le choix des valeurs ci-dessus répond à certaines contraintes apparues au fur et à mesure, et dont le non respect nuit à la sécurité du crytogramme.

**Exercice 53: Base de RSA**

Soit $p = 3$ et $q = 11$.

1. Calculer $n$ et $\phi(n)$.
2. Choisir la clef privée $b$ telle que $b$ soit la plus petite possible, différente de $p$ et $q$ et tel que $1 < b < \phi(n)$ et $\text{PGCD}(a, \phi(n)) = 1$.
3. Calculer la clef publique $a$ (= l'inverse modulaire de $b$ modulo $\phi(n)$).
4. Chiffrer $x = 13$ avec la clef publique $a$.
5. Déchiffrer $y = 19$ avec la clef privée $b$.

**Remarques**

- On pourrait choisir un $a$ petit (pour chiffrer rapidement) et calculer $b$ à partir de $a$ (ou $a$ à partir de $b$).
  Mais, le choix d'un tel $a$ peut permettre des attaques [BD00]. Donc, il vaut donc mieux choisir $a$ petit (mais $> n^{0.3}$).
- De même, la différence entre $|p - q|$ ne doit pas être trop petite, sinon $p \simeq q$. Dans ce cas, $p \simeq \sqrt{n}$ et il suffit de rechercher les nombres premiers proche de $p$.
- beaucoup d'auteurs recommandent de faire en sorte que $p$ et $q$ soient des nombres premiers forts (rend l'algorithme de Pollard infaisable).
  Un nombre premier fort $P$ vérifie un sous-ensemble de conditions suivantes :

  1. $P - 1$ a un facteur premier grand (noté $R$),
  2. $R - 1$ a un facteur premier grand,
  3. $P + 1$ a un facteur premier grand,

  où l'on dit qu'un nombre $A$ a un facteur premier grand si $A$ s'écrit $A = k.B$ où $B$ est un grand nombre premier et $k$ un entier.

La raison d'être de ces conditions supplémentaires sera explicitée dans la partie sur la cryptanalyse.

Soit $\Sigma$ l'alphabet du chiffre clair. Typiquement $N = |\Sigma| = 256$, ou moins si l'alphabet contient moins de symboles.

Soit $m = c_0 \dots c_\ell$ le message à coder. Chaque symbole $c_i$ de l'alphabet représente un symbole en base $N$ (son numéro d'ordre dans l'alphabet). Le message $m$ est

donc un nombre qui s'écrit avec $\ell$ symboles en base $N$.

Le $m$ doit être représenté sous forme d'une suite de nombres $n_0 \dots n_r$ tel que chaque $0 \leq n_i < n$.

Soit $k = \lfloor \log_N n \rfloor$ = le nombre de chiffres qu'à $n$ en base $N$ = le nombre maximum de symboles entiers que je peux stocker dans un nombre $\leq n$. En conséquence, chaque nombre $n_i$ servira à stocker $k$ symboles.

En conséquence, chaque $n_i$ est calculé comme $n_i = \sum_{j=0}^{k-1} c_{i.k+j}.N^j$.

**Exemple :**

Supposons que nous voulions stocker le message $m$ = {0,1,1,2,1,2,0,2,1,2} (en base $N = 3$) dans un mot de 8 bits. $k = \lfloor \log_3 2^8 \rfloor = 5$. Donc,
$n_0 = m_0.3^0 + m_1.3^1 + m_2.3^2 + m_3.3^3 + m_4.3^4 = 0 + 1.3 + 1.9 + 2.27 + 1.81 = 147$
$n_1 = m_5.3^0 + m_6.3^1 + m_7.3^2 + m_8.3^3 + m_9.3^4 = 2 + 0.3 + 2.9 + 1.27 + 2.81 = 209$

**Exercice 54: Chiffrement d'un message RSA**

Soit l'alphabet $\Sigma = \{A, E, N, S\}$ et le chiffre RSA $(n, a, b) = (33, 7, 3)$.

1. Donner un codage de l'alphabet.
2. Calculer le nombre de chiffres en base $|\Sigma|$ que peut contenir un nombre inférieur à 33.
3. Calculer le codage du mot "ananas" en utilisant un nombre entier de symbole par entier.
4. Calculer le codage du mot "ananas" en utilisant un nombre fractionnaire de symbole par entier.
5. Chiffrer ce dernier codage avec le chiffre RSA de l'exercice précédent.
6. Déchiffrer le code $\{2, 7, 28\}$ avec le chiffre RSA de l'exercice précédent, le codage des symboles utilisés est celui de la question 3.

Pour le déchiffrement d'un modulo $n$ à $k$ bits, $b$ est aussi typiquement à $k$ bits, et a en moyenne la moitié de ses bits à 1.

Donc avec l'exponentiation rapide, on a besoin de $k$ mise au carré et de $k/2$ multiplications modulo $n$. Donc si $n$ a 1024 bits, on a 1024 mise au carré et 512 multiplications modulo $n$ pour chaque bloc déchiffré.

Tout d'abord, rappelons qu'en utilisant l'algorithme d'Euclide étendu, on peut trouver $y_p$ et $y_q$ tels que $y_p.p + y_q.q = 1$ et on en déduit que $y_p = p^{-1} \bmod q$ et $y_q = q^{-1} \bmod p$ (cf Bézout). Notons que $y_p$ et $y_q$ peuvent être précalculés.

Soit $m_e$ le message chiffré. Si $b$ est la clef privée, on peut calculer :

$m_{e,p} = {m_e}^{b \bmod (p-1)} \bmod p$ et $m_{e,q} = {m_e}^{b \bmod (q-1)} \bmod q$.
Le message décodé $m_d$ vérifie : $m_d \equiv m_{e,p} \bmod p$ et $m_d \equiv m_{e,q} \bmod q$. Or, par le théorème des restes chinois, ces équations ont une solution unique modulo $n = p.q$, et cette solution est $m = (m_{e,p}.y_q.q + m_{e,q}.y_p.p) \bmod n$.

En temps de calcul, le calcul de $m = c^d \bmod n$ prend un temps $C(k + l)k^2$ où $k$ =nombre de bits de $n$, $l$ =nombre de bits à 1 et $C$ une constante. Typiquement, $p$ et $q$ ont $k/2$ bits, ce qui conduit à un temps de calcul pour les 2 au pire de $2C(k + l)(k/2)^2$, soit deux fois moins.

**Exemple**

Soit $p = 11$, $q = 23$, donc $n = 11 \times 23 = 253$.
Donc $\phi(n) = (p - 1).(q - 1) = 10 \times 22 = 220$.
$y_p = 21$ car $p \times y_p \bmod q = 11 \times 21 \bmod 23 = 231 \bmod 23 = 1 \bmod 23$.
$y_q = 1$ car $q \times y_q \bmod p = 23 \times 1 \bmod 11 = 23 \bmod 11 = 1 \bmod 11$.

Choisissons $a$ et $b$ :

- On prend $a$ tel que PGCD$(a, \phi(n)) = 1$. On choisit $a = 3$.
- On prend $b$ tel que $a.b = 1 \bmod \phi(n)$. D'où $b = 147$.

On veut coder le message $m = 26$.
**Chiffrement :** $m_e = m^a \bmod n = 26^3 \bmod 253 = 119$.
**Déchiffrement :** $m = {m_e}^b \bmod n = 119^{147} \bmod 253 = 26$.

**Déchiffrement rapide**

$$
\begin{aligned}
m_{e,p} &= {m_e}^b \bmod p = (m_e \bmod p)^{b \bmod (p-1)} \bmod p \\
&= (119 \bmod 11)^{147 \bmod 10} \bmod 11 = 9^7 \bmod 11 = 4 \\
m_{e,q} &= {m_e}^b \bmod q = (m_e \bmod q)^{b \bmod (q-1)} \bmod q \\
&= (119 \bmod 23)^{147 \bmod 22} \bmod 23 = 4^{15} \bmod 23 = 3 \\
m &= (m_{e,p}.y_q.q + m_{e,q}.y_p.p) \bmod n \\
&= (4 \times 1 \times 23 + 3 \times 21 \times 11) \bmod 253 \\
&= 785 \bmod 253 = 26
\end{aligned}
$$

**Exercice 55: Déchiffrement rapide**

Soit le chiffre RSA $(n, p, q, a, b) = (323, 17, 19, 67, 43)$.

1. Calculer $p^{-1} \bmod q$ et $p^{-1} \bmod q$.
2. Décoder le chiffre $c = 287$ en utilisant l'algorithme d'exponentiation modulaire.
3. Même question avec l'algorithme de déchiffrement rapide.

## 3.3 Tests de primalité

Nous allons donc avoir besoin de grands nombres premiers pour appliquer ces méthodes. Mais y en a-t-il beaucoup ?

**Théorème 53** (équivalence et bornes sur le nombre de nombres premiers)**.**
*Soit $\pi(x)$ le nombre de nombre premiers inférieur ou égal à x.*

- $\pi(x) \sim x/\log x$ *(i.e.* $\lim_{x \to \infty} \pi(x)/(x/\log x) = 1$*)*
- *si* $x > 10$*,* $x/(\log x - 1) \geq \pi(x)$
- *si* $x > 598$*,* $(x/\log x)(1 + 0.992/\log x) \leq \pi(x) \leq (x/\log x)(1 + 1.2762/\log x)$

Ces différents résultats ont de nombreux auteurs (Legendre, Hadamard, de la Vallée Poussin, ...), la dernière borne étant de Pierre Dusart (1999).

Si nous avons besoin d'un nombre premier à 100 chiffres, il y en a approximativement : $\pi(10^{100}) - \pi(10^{99}) \sim \frac{10^{100}}{\log 10^{100}} - \frac{10^{99}}{\log 10^{99}} \simeq 3.9 \times 10^{97}$.
Donc, ce nombre est suffisamment dissuasif contre une attaque en force brute.

Mais comment trouver un grand nombre premier ?

Si je tire un nombre $x$ au hasard entre 1 et $n$, sa chance d'être un nombre premier est : $\Pr[x \in \mathbb{P}] = \frac{n/log(n)}{n} = \frac{1}{\log n}$
Donc, pour un nombre à $n$ bits, sa probabilité d'être premier est $\frac{1}{n \log 2}$ :

| bits | 128 | 256 | 512 | 768 | 1024 |
|---|---|---|---|---|---|
| $\Pr[x \in \mathbb{P}]$ | 1.127% | 0.564% | 0.282% | 0.188% | 0.141% |

Néanmoins, on peut écarter facilement une bonne partie des nombres non-premiers : ceux qui sont multiples de 2, 3, 5, ... :

- pour les multiples de 2, on élimine un composite sur 2, donc 50%.
- pour les multiples de nombres premiers jusqu'à 3, on en élimine 1/3, mais 1/6 étaient déjà éliminés par les multiples de 2. Donc, on en élimine : $\frac{1}{2} + \frac{1}{3} - \frac{1}{6} = \frac{2}{3} = 66\%$.
- pour les multiples de nombres premiers jusqu'à 5, on en élimine 1/5, mais 1/10 étaient déjà éliminés par les multiples de 2 et 1/15 étaient éliminé par les multiples de 3. 1/30 sont éliminés par les deux à la fois. Donc, on en élimine : $\frac{2}{3} + \frac{1}{5} - \frac{1}{10} - \frac{1}{15} + \frac{1}{30} = \frac{22}{30} = 73.3\%$
- pour les multiples de nombres premiers jusqu'à 7, on en élimine $\frac{22}{30} + \frac{1}{7} - \frac{1}{14} - \frac{1}{21} - \frac{1}{35} + \frac{1}{42} + \frac{1}{70} + \frac{1}{105} - \frac{2}{420} = \frac{27}{35} = 77.1\%$
- *etc* ...

Mais comment éliminer les nombres non premiers parmi ceux qui restent ?

### 3.3.1 Fermat

Comme il est très coûteux de prouver qu'un entier est premier (il faut soit faire un crible, soit tenter de le factoriser), il existe aussi certains algorithmes très efficaces qui peuvent assurer que cela est très probablement le cas.

On rappelle le petit théorème de Fermat : si $n$ est un nombre premier, alors $a^{n-1} \equiv 1 \bmod n$ pour tout $a \in \mathbb{Z}$ tel que $\mathrm{PGCD}(a, n) = 1$. La fonction suivante utilise ce théorème pour effectuer $k$ fois le test de Fermat sur un nombre $n$ :

*boolean FermatPrimalityTest(n,k)*
  **for** $i = 1$ *à* $k$ **do**
    choisir un entier aléatoire $a \in \{2, \ldots, n-2\}$
    si $\mathrm{PGCD}(a, n) \neq 1$ alors retourner faux // peut être omis
    calculer $r = a^{n-1} \bmod n$
    si $(r \neq 1)$ alors retourner faux
  retourner vrai

La probabilité qu'à la fin de l'exécution de la boucle, le nombre ne soit pas premier est inférieur à 1/2. En conséquence, après $k$ itérations, $\Pr[n \notin \mathbb{P}] < 2^{-k}$. Par exemple, si $k = 30$, $\Pr[n \notin \mathbb{P}] < 10^{-9}$.

Ce test peut néanmoins conclure de manière fausse que $n$ est premier, même avec $k$ très grand.

**Exercice 56: Test de Fermat**

On veut effectuer le test de primalité

1. Appliquer le test de Fermat sur 51 avec les nombres tirés au hasard suivants $(35, 23, 14, 29)$.
2. Appliquer le test de Fermat sur 53 avec les nombres tirés au hasard suivants $(35, 23, 14, 29)$.
3. Pour les nombres ci-dessus pour lesquels le test de Fermat conclut positivement, calculer la probabilité que le nombre ne soit pas premier.

En 1910, Carmichael découvre que le test de Fermat peut conclure de manière éronée qu'un nombre $n$ est premier, autrement dit, qu'il existe des nombres $n$ composites (dits nombres de Carmichael) tels qu'il existe un $a \in \{2, \ldots, n-2\}$ qui vérifie $a^{n-1} \equiv 1 \bmod n$. Ces nombres peuvent être caractérisés.

**Théorème 54** (caractérisation des nombres de Carmichael)**.**
*Un nombre impair composite n est un nombre de Carmichael si et seulement si :*

- *aucun diviseur premier n'est multiple (i.e. si n est divisible par $p \in \mathbb{P}$, alors $p^k$ ne le divise pas n pour tout $k \geq 2$), et il y en a au moins 3.*

- *pour tout diviseur premier $p$ de $n$, alors $p-1$ divise $n-1$.*

**Démonstration :** voir [Buc02], page 155.
**Exemple**

Le plus petit nombre de Carmichael est $n = 561 = 3 \times 11 \times 17$.
On a bien : $560/2 = 280$, $560/10 = 56$, $560/16 = 35$.

Les nombres de Carmichael sont des nombres assez rares : pour $n < 10^{10}$, il y a 455 052 511 nombres premiers et 14 884 nombres de Carmichael.

Il en existe une infinité de nombres de Carmichael. On a $C(n) > n^{2/7}$ pour $n$ assez grand (Alford & al, 94).

### 3.3.2 Miller-Rabin

Le test de Miller-Rabin est basé sur une modification du petit théorème de Fermat.

**Théorème 55.** *Soit $n$ premier, $s = \max\{r$ tels que $2^r$ divise $n-1\}$ et $d = (n-1)/2^s$.*
*Alors pour tout entier $a$ premier avec $n$,*

- *soit $a^d = 1 \bmod n$.*
- *soit $\exists r \in \{0, 1, \ldots, s-1\}$ tel que $a^{2^r d} = -1 \bmod n = (n-1) \bmod n$.*

**Démonstration :** voir [Buc02], page 156.

**Exemple**

Prenons le nombre de Carmichael $n = 561$, le test de Fermat échoue à prouver que $n$ est composite.
$s = 4$ car $n - 1 = 560$ est divisible par $16 = 2^4$ ($d = 560/2^s = 35$).
pour $a$=2, $2^{35} = 263[561]$, $2^{2.35} = 166[561]$, $2^{4.35} = 67[561]$, $2^{8.35} = 1[561]$.
Aucune des deux conditions n'est vérifiée, donc $n$ n'est pas premier.

Soit le nombre premier $n = 569$, $n - 1 = 568 = 2^3.71$, $s = 3$, $d = 71$.
Pour $a$=2, $2^{71} = 86[569]$, $2^{2.71} = 568[569]$, $2^{4.71} = 1[569]$, $2^{8.71} = 1[569]$. La condition est vérifiée pour $2^{2.71}$.

Afin de vérifier la primalité, il faut vérifier cette condition pour chaque $a$ premier avec $n$. Un entier $a$ permettant de vérifier que $n$ est composite est appelé un témoin.

De combien de temoins a-t-on besoin afin de vérifier la primalité ?

**Théorème 56.** *Si $n \geq 3$ est un nombre composite impair, alors l'ensemble $\{1, \ldots, n-1\}$ contient au plus $(n-1)/4$ premiers avec $n$ et qui ne sont pas témoins.*

**Démonstration :** voir [Buc02], page 157.

En conséquence, la probabilité que $n$ soit composite et que le $a$ testé ne soit pas un témoin est au plus de $1/4$.

Si on effectue $k$ fois le test et que l'on ne trouve pas de témoin, alors la probabilité qu'il soit composite est de $1/2^{2k}$. Par exemple, si $k = 10$, $Pr[n \notin \mathbb{P}] = 1/2^{20} \simeq 1/10^6$.

En pratique, la probabilité est encore plus petite.

L'algorithme du test de Miller-Rabin est par conséquent :

```
boolean MillerRabinWitnessTest(n)
    calculer n − 1 = 2^s r où r est impair.
    choisir un entier aléatoire a ∈ {2, ..., n − 2}
    if PGCD(a, n) ≠ 1 then return faux // peut être
      omis
    calculer b = a^r mod n
    if (b = 1) then return true.
    for t = 0 à s − 1 do
        if (b = −1) then return true.
        calculer b = b^2 mod n
    return false
boolean MillerRabinTest(n,k)
    for i = 1 à k do
        if (MillerRabinWitnessTest(n) = false) then
            return false
    return true
```

Pour les entiers aléatoires $a$ choisis, on peut systématiquement effectuer les premières vérifications avec 2, 3, 5, 7, ...

**Exercice 57: Test de Miller-Rabin**

On veut effectuer le test de primalité de Miller-Rabin.

1. L'appliquer sur 89 avec le témoin 2.
2. L'appliquer sur 93 avec le témoin 3.
3. L'appliquer sur 41 avec le témoin 3.
4. Parmi ces tests, quels sont ceux qui donnent une réponse certaine sur la primalité de l'entier testé ?
5. On lance un test de Miller-Rabin sur un entier $x$ avec $k = 10$ témoins qui y répond positivement. Quelle est la probabilité que $x$ soit premier ?

# 4 Cryptanalyse

Il y a plusieurs différentes méthodes d'attaques sur le RSA :

- les attaque directement de la fonction à sens unique :
  - ◇ recherche une méthode permettant à d'inverser ou de réduire la complexité supposée de cette fonction.
  - ◇ recherche de cas particuliers dans lesquels la complexité de l'inversion est moindre.
- les attaques exploitant les faiblesses du protocole :
  - ◇ attaque sur le chiffre ou la clef (à textes/chiffres choisis)
  - ◇ recherche de méthodes permettant de créer des messages falsifiés ou de pervertir les protocoles d'échange.

A noter que nous ne décrirons pas les protocoles exacts d'échange de clefs ou de structuration des blocs, mais nous indiquerons les raisons pour lesquelles ceux-ci ont été mis en place.

Ceux-ci seront abordés plus tard dans votre formation avec l'étude et l'utilisation des protocoles associés (par exemple ssh).

## 4.1 Protocole

### 4.1.1 Man-in-the-middle

Supposons que l'attaquant Manuel n'est pas seulement un observateur, mais a un contrôle complet sur les communications entre Alice et Robert.

Dans ce cas, au moment de l'échange des clefs entre Alice et Robert,

- Manuel intercepte les deux clefs publiques, $k^{\text{public}}_{\text{Alice}}$ et $k^{\text{public}}_{\text{Robert}}$, remplace dans l'échange de clefs ces clefs par la sienne $k^{\text{public}}_{\text{Manuel}}$.
- Lorsqu'Alice envoie un message à Robert, elle chiffre son message avec $k^{\text{public}}_{\text{Manuel}}$ (qu'elle pense être celle de Robert), Manuel l'intercepte, utilise sa clef privée $k^{\text{privée}}_{\text{Manuel}}$ pour le déchiffrer, puis le rechiffre avec $k^{\text{public}}_{\text{Robert}}$, et le retransmet à Robert.
- vice-versa entre Alice et Robert.

Cette attaque est particulièrement pernicieuse :

- Manuel n'a aucun problème cryptographique à résoudre. Pourtant, il est capable de déchiffrer l'ensemble des messages entre Alice et Robert,
- Alice et Robert n'ont aucun moyen de supposer que leurs communications, qu'ils pensent être sûres car chiffrées, n'ont aucun secret pour Manuel.

Elle suppose seulement que Manuel a les moyens techniques d'effectuer ces interceptions. C'est aussi la raison pour laquelle il existe des centres certifiés de

distribution de clefs.

### 4.1.2 Multiplicativité

Soit $(n, a)$ la clef RSA publique.

Si deux messages $m_1$ et $m_2$ sont chiffrés avec cette clef : $c_1 = {m_1}^a \bmod n$ et $c_2 = {m_2}^a \bmod n$. On peut calculer $c = c_1.c_2 \bmod n = (m_1 m_2)^a \bmod n$. En conséquence, à partir de deux chiffres $c_1$ et $c_2$, on peut construire un nouveau chiffre $c$ qui représente le message $m = m_1.m_2$.

Cette propriété permet une attaque à chiffre choisi : Manuel veut déchiffrer le message $z$, mais sans que Robert sache qu'il va aider à déchiffrer $z = x^b$. Manuel choisit un $r$ au hasard (inversible dans $\mathbb{Z}_n$) et calcule $z_1 = z.r^b = (xr)^b$, qu'il fait transmettre à Robert qui le décode. Le résultat $x_1 = (xr)^{ab} = xr$ revient à Manuel qui divise juste $x_1$ par $r$ pour obtenir $x$. Robert a ainsi involontairement aidé à déchiffrer $z$ sans le savoir.

Le récepteur n'a *a priori* aucune façon de déterminer si un message $m$ qu'il reçoit a été modifié ou pas. Cela nécessite de réduire l'espace des messages en clair. Une façon de faire consiste, dans chaque bloc, à répéter le même caractère au début et à la fin du bloc.

La probabilité pour qu'un chiffre altéré ait cette propriété est insignifiante.

**Exercice 58: Problème de multiplicativité**

On considère le chiffre RSA $(n, p, q, a, b) = (33, 3, 11, 7, 3)$.

1. Chiffrer $m_1 = 8$. On note $c_1$ ce chiffre.
2. Manuel veut déchiffrer $c_1$. Il choisit $r = 5$. Calculer $c_2 = c_1.r^a \bmod n$.
3. Robert accepte de déchiffrer $c_2$ pour Manuel. Calculer le message $m_2$ que Robert transmet à Manuel.
4. Utiliser la multiplicativité de RSA pour retrouver le message $c_1$.

### 4.1.3 Sécurisation

Tel que présenté, le RSA souffre encore de deux graves problèmes pouvant être utilisé en cryptanalyse :

- si un message court est envoyé souvent, alors le chiffrement du message étant déterministe, le même chiffre sera toujours envoyé.

En conséquence, si l'on arrive à associer le résultat de la transmission du chiffre au chiffre correspondant, le contenu du message lui-même n'a plus d'importance.
- comme l'on dispose de la façon de chiffrer le message, et que celui-ci est chiffré par paquet, on peut en construire autant que l'on en souhaite, et ainsi se construire un dictionnaire de messages/paquets probables. Il suffit ensuite de rechercher dans le dictionnaire construit pour reconstituer le message, ou de tester les hypothèses que l'on souhaite.

Il serait éventuellement possible d'utiliser un mode de chiffrement par bloc CBC afin de réduire la capacité à prévoir des blocs sans tenir compte des blocs précédents.

Une approche beaucoup plus efficace a été adoptée en ajoutant une composante aléatoire au RSA de la manière suivante (ce procédé est décrit dans le standard PKCS N°1), et correspond au standard EME-OAEP.

Soit $s$ la longueur du module RSA (*i.e.* $\lceil log_2 n \rceil$), $k$ un entier positif (au moins égal à 64). Soit $l = s - k - 1$ le nombre de bits utilisés pour stocker le message (donc $k < l$) et :
- un fonction d'expansion $G : \mathbb{B}^k \rightarrow \mathbb{B}^l$ (pour l'hasardisation).
- un fonction de condensation $H : \mathbb{B}^l \rightarrow \mathbb{B}^k$ (génération de la signature).

Des exemples de ces fonctions sont donnés dans la section complément plus loin.

Ces deux fonctions sont publiquement connues. On part de :
- $m \in \mathbb{B}^l$ le texte clair,
- $r \in \mathbb{B}^k$ un chiffre aléatoire.

et on utilise les deux opérateurs $\oplus$ (xor) et $\circ$ (concaténation).

Le message transmis est constitué de la concaténation de :
- $g = m \oplus G(r) \in \mathbb{B}^l$ est le message hasardisé. Comme $r$ est aléatoire, même si on transmet toujours le même message, cette portion sera toujours différente.
- $h = r \oplus H(g)$ : la valeur $r$ mélangée avec la signature du message hasardisé $C$.

Le message à coder est : $M = g \circ h = (m \oplus G(r)) \circ (r \oplus H(m \oplus G(r)))$.

Le chiffre est par conséquent : $c = M^b \bmod n$.

Mais comment décoder ?

On reçoit $c$, et après décodage on obtient : $M = c^b \bmod n = g \circ h$.

Les longueurs de $g$ et $h$ étant connues, on sait comment découper $M$ pour obtenir

$g$ et $h$. Donc, on connaît $g = m \oplus G(r)$ et $h = r \oplus H(g)$.

Or, la fonction $H$ est connue, donc on peut calculer :
$r = h \oplus H(g) = (r \oplus H(g)) \oplus H(g) = r$

Une fois $r$ connu, la fonction $G$ étant connue, on peut retrouver le message avec :
$m = g \oplus G(r) = (m \oplus G(r)) \oplus G(r) = m$

**Notes :**

- Comme le masque pseudo-aléatoire $G(r) \circ r$ est appliqué sur la totalité du message, le message transmis sera toujours différent, même si on transmet toujours le même message.
- Si le message a été altéré lors de la transmission, alors le $r$ reconstruit n'est pas correct, et comme $G(r)$ est une suite pseudo-aléatoire, le $m$ reconstruit ne l'est pas non plus.

Voir plus loin pour des exemples de fonction $G$ et $H$.

**Exercice 59: Sécurisation**

Pour donner un exemple de sécurisation d'un message RSA sur 8 bits, on donne les fonctions suivantes :

- une fonction d'expansion $G : \mathbb{B}^2 \rightarrow \mathbb{B}^6$ définie comme $G(b_0b_1) = b_0b_1b_0b_1b_0b_1$.
- une fonction de condensation $H : \mathbb{B}^6 \rightarrow \mathbb{B}^2$ définie comme $H(b_0b_1b_2b_3b_4b_5) = b_0b_1 \oplus b_2b_3 \oplus b_4b_5$.

L'expéditeur veut envoyer le message $m = 010001$ en utilisant comme clef aléatoire $r = 10$.

1. Calculer le message hasardisé : $g = m \oplus G(r)$.
2. Calculer la signature du message hasardisé : $h = r \oplus H(g)$.
3. Calculer le message $M$ à envoyer ($M = g \circ h$).
4. Après déchiffrement, le destinataire reçoit le message $M$. Vérifier qu'il peut bien reconstruire $m$ à partir de $M$ sans avoir la connaissance de $r$.

## 4.2 Factorisation

L'attaque la plus évidente sur le RSA consiste à essayer de factoriser $n$. Si cela peut être fait : l'attaquant obtient $p$ et $q$, et peut alors calculer $\phi(n) = (p-1).(q-1)$, puis ensuite calculer $b$ à partir de $a$ puisque $a.b = 1 \bmod \phi(n)$.

La sécurité de RSA repose donc au moins sur la difficulté à factoriser les grands nombres.

**Conjecture 57** (sécurité de RSA)**.**
La sécurité de RSA est polynomialement équivalente à factoriser $n$.

**Note :** On dit que deux problèmes sont polynomialement équivalent si l'existence d'un algorithme à temps polynomial pour l'un des deux problèmes équivaut à l'existence d'un algorithme à temps polynomial pour l'autre.

Les meilleurs algorithmes connus (= publiés) de factorisation d'un entier $n$ sont :

- la factorisation par crible sur les corps de nombres généralisé (GNFS). Sa complexité est subexponentielle (en $O(\exp\left((64/9\log n)^{1/3}\,(\log\log n)^{2/3}\right)$).
- en théorie des nombres, les algorithmes de factorisation sont en temps moyen $L_n\,[1/2, 1 + o(1)] = exp\left((1 + o(1)).(\log n)^{1/2}(\log\log n)^{1/2}\right)$.
- l'algorithme de Shor permet d'effectuer la factorisation en temps polynomial sur un ordinateur quantique (technologie balbutiante).

**Exercice 60: Factorisation RSA**

Soit la clef publique d'un chiffre RSA $(n, a) = (323, 67)$.

1. Trouver une factorisation facile de $n = 323$ (chercher à proximité de $\sqrt{n}$). En déduire $(p, q)$.
2. Trouver la clef de décodage $b$.

On dispose d'un nombre $n = p.q$, et on cherche à déterminer les deux entiers $p$ et $q$ premiers.

Supposons maintenant que nous arrivions à trouver $L$ tel que $(p-1) \mid L$ et $(q-1) \nmid L$. Cela signifie qu'il existe $i, j, k \neq 0$ tel que $L = i(p-1) = j(q-1)+k$.

Prenons maintenant un entier aléatoire $a$ et calculons $a^L$. Le petit théorème de Fermat nous dit que :

• modulo $p$, on a : $a^L = a^{i(p-1)} = \left(a^{p-1}\right)^i \equiv 1^i \equiv 1$.

• modulo $q$, on a : $a^L = a^{j(q-1)+k} = \left(a^{q-1}\right)^j a^k \equiv 1^i.a^k \equiv a^k$.

Comme $k \neq 0$, il est très improbable que $a^k \equiv 1 \bmod q$.
Donc, pour presque tout choix de $a$, on a : $p \mid (a^L - 1)$ et $q \nmid (a^L - 1)$.

En conséquence, on peut trouver $p$ par le calcul : $p = \text{PGCD}(a^L - 1, n)$.

Pollard fait remarquer que si $p-1$ est un produit de petits nombres premiers, alors il divisera $N!$ pour une valeur raisonnable de $N$. Donc, pour $N = 2, 3, 4, \ldots$, on choisit une valeur de $a$ (2 en pratique) et on calcule $\text{PGCD}(a^{N!} - 1, n)$.
• si PGCD = 1, alors on passe à la valeur de $N$ suivante.
• si PGCD = $n$, pas de chance, $a$ ne convient pas ($a^{N!} - 1 \mid n$), changer $a$.
• si $1 <$ PGCD $< n$, alors le PGCD est la valeur de $p$ recherchée.

**Remarque :** Il faut utiliser les règles du calcul modulaire afin d'évaluer $a^{N!}$ :

- $a^p$ est évalué par l'algorithme d'exponentiation "carré & multiplication".
- $a^{N!}$ est évalué itérativement comme $a^{(N+1)!} \equiv (a^{N!})^{(N+1)} \bmod n$.

**Algorithme**

L'algorithme prend deux paramètres : l'entier $n$ à factoriser, et $k$ est la friabilité maximale de $n$.

*entier PollardPmoins1(n,k)*
fixer $a = 2$
**for** *j=2,3,4, ...* $k$ **do**
$a = a^j \bmod n$
$d = \text{PGCD}(a-1, n)$
**if** $(1 < d < n)$ **then return** $d$
**if** $(d = 1)$ **then** la valeur de $k$ ne convient pas : échec.
**if** $(d = n)$ **then** changer $a$ et recommencer.

**Note :** afin d'augmenter l'efficacité de l'algorithme, on peut ne calculer le PGCD que toutes les $r$ itérations ($r$ à fixer).

**Complexité :** l'exponentiation $a^{k!} \bmod n$ est en temps $2k \log_2 k$. Il est donc raisonnable de calculer $a^{k!} \bmod n$ pour des grandes valeurs de $k$.

**Exemple**

Appliquons cette méthode sur $n = 13927189$ et $a = 2$ :

| $j$ | $a^{j!} - 1$ | PGCD |
|---|---|---|
| 2 | 3 | 1 |
| 3 | 63 | 1 |
| 4 | 2850026 | 1 |
| 5 | 12764116 | 1 |
| 6 | 435113 | 1 |
| 7 | 6518603 | 1 |
| 8 | 12163255 | 1 |

| $j$ | $a^{j!} - 1$ | PGCD |
|---|---|---|
| 9 | 13867883 | 1 |
| 10 | 5129508 | 1 |
| 11 | 4405233 | 1 |
| 12 | 6680550 | 1 |
| 13 | 6161077 | 1 |
| 14 | 879290 | 3823 |

Donc, $p = 3823$ est un diviseur de $n$.
Par division entière, on trouve $q = 3643$.

**Quelle est l'efficacité de cet algorithme ?**

Si $n$ est factorisable, l'algorithme garantit de trouver une réponse si $k = \sqrt{n}$.
Si $n$ est très friable (= $n - 1$ possède de petits diviseurs premiers), la réponse peut être très rapide.
Le temps d'exécution le pire est en $O(\pi(k)M(\log_2 n)\log(n))$ où $k = \sqrt{n}$ et $M(k)$ est le coût d'une multiplication entre entier de $k$ bits (classique=$O(n^2)$, Schönhage-Strassen=$O(n.\log n.\log\log n)$) et $\pi(k)$ le nombre de nombres premiers

inférieur à $k$, ce qui conduit à une complexité de $O(M(\log n).\sqrt{n})$, ce qui est identique à la méthode naïve par division.

**Exercice 61: Algorithme $p-1$ de Pollard**

Appliquer l'algorithme $p-1$ de Pollard à $n = 493$.

**Conséquences pratiques :**

- Connaissant les dangers de l'attaque $p-1$ de Pollard, il faut donc vérifier lors du choix de $p$ et $q$ que ni $p-1$, ni $q-1$ ne peuvent être factorisés entièrement avec de petits entiers premiers.
- Sachant que $p$ et $q$ sont choisis de cette manière, l'attaquant peut optimiser l'algorithme de Pollard en conséquence, et ne prendre comme valeur de $j$ que des nombres premiers.

**Conséquences théoriques**

Même si l'on construit un cryptosystème basé sur un problème qui semble difficile, il est nécessaire de tenir compte de l'ensemble des cas (subtils ou non triviaux) qui rendent le problème plus facile à résoudre que dans le cas général.

**Autres méthodes de factorisation**

- factorisation par les courbes elliptiques (ECM) qui est une amélioration de la méthode $p-1$ de Pollard, en $L(1/4, \sqrt{c})$ pour un nombre premier $L(1/2, c)$ friable.
- crible général de corps de nombres (GNFS) en $L(1/3, 64/9)$.

# 5 Compléments

## 5.1 Fonction d'expansion

En cryptographie, une fonction d'expansion $G : \mathbb{B}^k \to \mathbb{B}^l$ (avec $k < l$) a pour but de donner une longueur plus grande à l'entrée (de taille $k$).

Divers utilisations sont possibles :

- transformer un paquet de données de taille $k$ en un paquet de taille $l$ (voir par exemple le RSA), soit par copie de bits, soit par transformations déterministes.
- construire une perturbation (salt) qui va permettre d'occulter l'information qui pourrait être extraite d'un bloc ou utilisé (par exemple, l'entropie du langage pour un texte clair ou une répétition du message).

Pour les perturbations, il est courant d'utiliser à cet effet une fonction de génération de bits pseudo-aléatoire, dite crytographiquement sûre (CSPRNG) à savoir que :

- elle vérifie le test du bit suivant : il n'existe pas d'algorithme à temps polynomial permettant de prévoir le bit $k+1$ à partir du bit $k$ avec une probabilité supérieure à 50%.
- elle résiste à une compromission partielle de son état : si une partie de son état est trouvé ou deviné, il doit être impossible de reconstruire les bits précédents. Il ne doit également pas être possible de prédire l'évolution de l'état courant.

**Générateur de bits pseudo-aléatoires utilisant RSA**

Soit $p$ et $q$ les deux nombres premiers, $n = p.q$, $\phi = (p-1).(q-1)$.
La méthode de génération est alors la suivante :

$\mathbb{B}^l$ *RSA_PRBG(n,$\phi$,l)*
choisir un entier $e$ tel que $1 < e < \phi$ tel que PGCD$(e, \phi) = 1$.
choisir une graine aléatoire $x_0 \in [1, n-1]$.
**for** $i = 1$ *à* $l$ **do**
$x_i = (x_{i-1})^e \bmod n$
$z_i$ = le bit le moins significatif de $x_i$.
retourner $z = z_1 \dots z_l$

Ce générateur est cryptographiquement sûr sous l'hypothèse que la résolution du problème RSA est infaisable.

L'efficacité de ce générateur peut être amélioré (voir le générateur de bits pseudo-aléatoire de Micali-Schnorr).

## 5.2 Fonction de condensation

Une fonction de condensation (ou de hachage) $H : \mathbb{B}^l \to \mathbb{B}^k$ avec $l > k$ est une fonction à sens unique associant à un message de taille arbitraire un "condensé" de taille fixe et petite.

Ce type de fonctions est très utilisé en cryptographie comme code d'authentification de message ou comme signature de message.

$H$ est à sens unique au sens suivant :

- soit un condensé $y = H(x)$, il est difficile de trouver un $x' = x$ tel que $H(x') = y$ (sens faible).
  Autrement dit, si je prends deux messages $x$ et $x'$ au hasard, il y a peu de chance que $H(x) = H(x')$. Donc, si je reçois $x$ et $v$ calculé comme $H(x)$, alors si $H(x) = v$, je suis presque sûr que mon message n'a pas été altéré. En revanche, si $H(x) \neq v$ alors il a été corrompu.
- il est difficile de trouver une collision pour $H$, à savoir une paire $(x, x')$ de message tels que $H(x) = H(x')$ (sens fort).

> Autrement dit, si je connais $H(x)$, il est difficile de trouver un $x'$ qui est condensé comme $x$. Donc, si je veux altérer un message $x$ en un autre message $x'$, alors il est difficile de trouver un $x'$ qui vérifier $H(x) = H(x')$.

Dans ce dernier cas, on dit abusivement que la fonction est sans collision (et permet de se prémunir contre l'attaque dite "du paradoxe des anniversaires").

Le paradoxe des anniversaires se formule de la façon suivante : considérons un groupe de $n$ personnes.

1. Quelle est la probabilité que quelqu'un ait le même anniversaire que vous ?
2. Quelle est la probabilité qu'au moins deux personnes dans ce groupe aient le même anniversaire ?

Pour la première question, soit les évènements :
• $A$ = au moins un parmi $n$ a la même date d'anniversaire que vous.
• $\overline{A}$ = aucun parmi $n$ n'a la même date d'anniversaire.
• $\overline{A}_i$ = la $i^{\text{ème}}$ personne n'a pas le même anniversaire que vous ($\overline{A} = \cup_i \overline{A}_i$).
$\Pr[A] = 1 - \Pr[\overline{A}] = 1 - \prod_{i=1}^{n} \Pr[\overline{A}_i] = 1 - \left(\frac{364}{365}\right)^n$

Pour la seconde question, soit les évènements :
• $B$ = deux personnes ont le même anniversaire.
• $\overline{B}$ = les $n$ personnes ont des anniversaires différents.
• $\overline{B}_i$ = la $i^{\text{ème}}$ personne n'a pas le même anniversaire que les $i-1$ précédents.
$\Pr[B] = 1 - \Pr[\overline{B}] = 1 - \prod_{i=1}^{n} \Pr[\overline{B}_i] = 1 - \prod_{i=1}^{n} \frac{365-(i-1)}{365} = 1 - \frac{\mathrm{A}_k^{365}}{365^n}$

Ces probabilités sont-elles sensiblement différentes ?
Pour $n = 40$, l'application numérique donne $\Pr[A] \simeq 10.4\%$ et $\Pr[B] \simeq 89.1\%$

**Application**

Considérons maintenant le cas d'un ensemble à $n$ éléments. On tire un ensemble de $k$ valeurs dans cet ensemble. Quel est la probabilité $p$ qu'au moins deux éléments soient identiques ?
La probabilité $q = 1 - p$ qu'aucune des $k$ valeurs soient identiques est :

$$q = \frac{1}{n^k} \prod_{i=0}^{k-1} (n-i) = \prod_{i=0}^{k-1} (1 - \frac{i}{n})$$

Comme $1 + x \leq e^x$, on a $q \leq \prod_{i=0}^{k-1} e^{-i/n} = e^{-\sum_{i=0}^{k-1} i/n} = e^{-k(k-1)/(2n)}$.

En résolvant $q \leq 1/2$ en $k$, on trouve $k \geq (1 + \sqrt{1 + 8n \log 2})/2$.
Si $q \leq 1/2$, $p \geq 1/2$.

**Cas des collisions :** pour une fonction de hashage $h : \mathbb{B}^* \to \mathbb{B}^N$.
On suppose que le hashage des valeurs est uniforme (*i.e.* pour chaque image, il y a le même nombre d'antécédents).

En conséquence, la taille de l'ensemble $n = 2^N$. Le nombre d'éléments d'élément qu'il est nécessaire de tirer avant de trouver une collision est donc :

$$(1 + \sqrt{1 + 2^N.8 \log 2})/2 = O(2^{N/2})$$

Afin d'éviter ce type d'attaque, il est nécessaire que le calcul de $2^{N/2}$ ne soit pas faisable (actuellement $N \geq 128$).

**Exemple de fonction de condensation : `SHA-1`**

Soit $B$ le bloc de $n$ bits que l'on souhaite condenser.
Le bloc est formaté comme $B' = B10^pS$ où $|B'|$ est un multiple de 512 et $S$ est la taille de $|B|$ écrit comme un entier de 64 bits (donc, $|B| < 2^{64}$).
**Exemple :** $|B|$ =2800 bits, $6 \times 512 = 3072 = \underbrace{2800}_{B} + \underbrace{1}_{1} + \underbrace{207}_{0^p} + \underbrace{64}_{|B|}$.
Le bloc à condenser $B'$ peut s'écrire sous la forme $B' = M_1 M_2 \dots M_n$ où les $M_i$ sont des blocs de 512 bits.

On définit les fonctions suivantes :

- LSHIFT$_k(w)$ = décalage circulaire à gauche de $k$ bits de $w$.
- $f_k(B, C, D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & \text{si } 0 \leq k \leq 19 \\ B \oplus C \oplus D & \text{si } 20 \leq k \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{si } 40 \leq k \leq 59 \\ B \oplus C \oplus D & \text{si } 60 \leq k \leq 79 \end{cases}$
- $K_k = \begin{cases} 5a827999 & \text{si } 0 \leq k \leq 19 \\ 6ed9eba1 & \text{si } 20 \leq k \leq 39 \\ 8f1bbcdc & \text{si } 40 \leq k \leq 59 \\ ca62c1d6 & \text{si } 60 \leq k \leq 79 \end{cases}$

**Algorithme :**

*boolean SHA-1(B)*
// padding $B' = B10^pS = M_1 M_2 \dots M_n$
**for** $i = 1$ *à* $n$ **do**
// initialisation
$(H_0, H_1, H_2, H_3, H_4) = (67452301, efcdab89, 98badcfe, 10325476, c3d2e1f0)$
// $W_j$ est le $j^{\text{ème}}$ bloc de 32 bits de $M_i$
$M_i = W_0 W_1 \dots W_{15}$
**for** $k = 16$ *à* $79$ **do**
$W_k = \text{LSHIFT}^1(W_{k-3} \oplus W_{k-8} \oplus W_{k-14} \oplus W_{k-16})$
// on a ici 80 blocs $W_k$
$(A, B, C, D, E) = (H_0, H_1, H_2, H_3, H_4)$
**for** $k = 0$ *à* $79$ **do**
$T = \text{LSHIFT}_5(A) + f_k(B, C, D) + E + W_k + K_k$
$(A, B, C, D, E) = (T, A, \text{LSHIFT}_{36}(B), C, D)$
$(H_0, H_1, H_2, H_3, H_4)$ +=$(A, B, C, D, E)$
retourner $H_0 H_1 H_2 H_3 H_4$

On obtient des paquets de 160 bits (= $32 \times 5$ bits).

# 6 Conclusion

Les conclusions à retenir sur cette partie sur la cryptographie :

- n'utilisez jamais une méthode cryptographique que vous avez inventé, si vous n'êtes pas en mesure de prouver sa sureté.
- évitez d'implémenter vous-même les méthodes de cryptographie considérées comme sûre en ne lisant que des présentations générales. Les méthodes considérées les plus sûres peuvent avoir des faiblesses structurelles que l'implémentation ou le protocole vient corriger.

Suite de cette introduction en M2.

# Troisième partie :
# Codes correcteurs d'erreur