Epreuve de TIPE

Martin Wattel

18 Octobre 2024

1 Introduction

1.1 Notations

On notera: On appellera le groupe $(\mathbb{Z}/n\mathbb{Z}, +, \times)$ l'anneau des classes d'équivalences modulo n. On notera $\overline{a}^n$ l'élément de $\mathbb{Z}/n\mathbb{Z}$ tel que $\overline{a}^n = n\mathbb{Z} + r$ avec $a \equiv r\ [n]$.
On notera $(U(\mathbb{Z}/n\mathbb{Z}), \times)$ le groupe des inversibles par la loi $\times$ de et anneau.

2 L'algorithme de cryptographie RSA

2.1 Fonctionnement de l'algorithme

Le but de l'algorithme RSA est de transformer un texte pour pouvoir transmettre tout en garantissant le fait qu'une tierce partie ne puisse le lire que par l'intermédiaire d'une clé que l'on explicitera ultérieurement.

La différence majeur qu'à cet algorithme parmi d'autres algorithme de chiffrement de message, c'est qu'il est asymétrique, c'est-à-dire que deux parties n'ayant pas la connaissance d'une même clé peuvent, par l'intermédiaire d'une clé publique et clé privée, transmettre un message.

Détaillons son fonctionnement de manière simpliste d'abord:

Une 1er partie (une machine comme un serveur par exemple) va envoyer à une autre partie une clé publique, tout en gardant sa clé privée. La deuxième partie va dès lors chiffrer le message avec la clé publique, et l'envoyer à la partie d'origine. Ce message chiffrer ne pourra pas être déchiffrer par une tierce partie avec la clé publique, car celle-ci ayant comme unique but de chiffrer le message. Une fois reçu le message chiffré, la partie d'origine va pouvoir le déchiffrer avec sa clé privée.

Création de la clé publique

- On commence par générer deux nombre premiers p, et q, et l'on pose $n = pq$
- On prend $e \in \mathbb{N}$ tel que $e \wedge \varphi(n) = 1$ avec φ *l'indicatrice d'Euler*.
- Finalement, on peut envoyer (e, n) qui définie notre clé publique.

Création de la clé privée

- Pour créer la clé privé il nous faut la clé publique (e, n), dès lors, on peut trouver (car c'est un élément inversible dans $\mathbb{Z}/\varphi(n)\mathbb{Z}$) $d \in \mathbb{N}$ tel que $ed \equiv 1\ [\varphi(n)]$
- Puis (d, n) représente notre clé privée

Théorème: Soit (e, n) et (d, n), clé publique et privée. Pour tout $m \in \mathbb{N}$, $m^{ed} \equiv m\ [\varphi(n)]$

Preuve: Soit $m \in \mathbb{N}$, $ed \equiv 1\ [\varphi(n)]$ donc $\exists k \in \mathbb{N}$ tel que $ed = 1 + k\varphi(n)$, on a alors $m^{ed} = m^{1+k\varphi(n)}$. Ce que l'on peut déduire du *Petit théorème de Fermat*, c'est que $m^{p-1} \equiv 1\ [p]$ et $m^{q-1} \equiv 1\ [q]$ avec pq la décomposition en nombre premier de n, puis $m^{\varphi(n)} \equiv 1\ [n]$. Donc, on peut en déduire que $m^{ed} \equiv m\ [n]$

2.2 Implémentation de l'algorithme

2.2.1 Recherche de nombre premiers

La première partie de l'algorithme RSA est de déterminer deux nombres premiers p et q. Pour cela, on va prendre un entier au hasard, et tester si celui-ci est premier ou non avec un test de primalité. Une première idée de teste de primalité, *très naïve* (comme nous le verrons) serait d'utiliser l'algorithme suivant:

Algorithme 1 Test de primalité naïf

1: *On prend N un entier à tester*
2: **Pour tout** entier n dans $[\![2, \lceil\sqrt{N}\rceil]\!]$ **faire**
3: **Si** $n \mid N$ **alors**
4: Renvoyer faux
5: **Fin Si**
6: **Fin Pour**
7: Renvoyer vrai

Cet algorithme s'exécute en $\mathcal{O}(N)$! Cela ne semble pas très contraignant si l'on ne travail pas sur de grand nombre. Mais comme nous le verrons dans la partie sur les attaques de cet algorithme, utiliser de petit nombre (nous donnerons un ordre de grandeur plus tard) résulterait en des problèmes de sécurité très importants. C'est pour cela que l'on utilisera pas cet algorithme, pour pouvoir être en concordance avec la pratique.

On se propose un nouvel algorithme, qui se base cette fois ci, non sur une approche déterministe, mais plutôt sur une approche probabiliste, ce qui va réduire considérablement le temps d'exécution.

Algorithme 2 Test de primalité de Fermat

1: *On prend N un entier à tester, et un nombre k arbitraire de répétition*
2: **Pour** i dans $[\![1, k]\!]$ **faire**
3: a = un nombre aléatoire dans $[\![1, N-1]\!]$
4: **Si** $a^{N-1} \not\equiv 1\ [N]$ **alors**
5: Renvoyer faux
6: **Fin Si**
7: **Fin Pour**
8: Renvoyer vrai

Cet algorithme s'appuie sur le *Petit théorème de Fermat*. Bien sûr, on ne teste pas ici la réciproque du théorème de Fermat, car elle n'est pas vraie en générale. Mais, on teste k fois la condition du *Petit théorème de Fermat* pour voir si l'entier ne vérifie pas la contraposée du théorème sur certains entiers, ce qui justifie le caractère aléatoire de ces entiers. On peut directement voir que notre algorithme va être en $\mathcal{O}(k \times c(puissance))$, avec $c(puissance)$ le coût de l'exponentiation d'un entier. Cette complexité est bien meilleure que celle du première algorithme.

On va se proposer un ordre de grandeur de la probabilité, après k testes sur des entiers aléatoire, que l'entier choisi est bel et bien un entier premier.

Definition: On appelle *nombre de Carmichael*, un entier $p \in \mathbb{N}$ tel que $\forall a \in \mathbb{N},\ a \wedge p = 1 \Rightarrow \overline{a^{p-1}}^p = \overline{1}^p$.
On peut également appeler ces nombres des *pseudo-premiers*.

Il se trouve que les nombres premiers sont des *nombres de Carmichael*, mais l'inclusion réciproque n'est pas forcément vraie !
On appellera un *pseudo-premier absolu* un entier qui est un *nombre de Carmichael*, mais qui n'est pas un entier premier.

Proposition: La probabilité que l'algorithme retourne vrai sur un entier alors qu'il n'est pas pseudo-premier est inférieure à $\frac{1}{2^k}$.

Preuve: Soit $N \in \mathbb{N}\backslash\{0,1\}$.
On note $B := \{a \in [\![1, N-1]\!] \mid \overline{a^{N-1}}^N = \overline{1}^N\}$
Supposons que N est un entier non premier.
$(B, \times)$ est un sous-groupe de $(U(\mathbb{Z}/N\mathbb{Z}), \times)$, puis $Card(B) \mid Card(U(\mathbb{Z}/N\mathbb{Z}))$ d'après le *théorème de Lagrange*.
Donc, $\exists p \in \mathbb{N}\ /\ Card(U(\mathbb{Z}/N\mathbb{Z})) = pCard(B)$
Puis $Card(B) \leq \frac{Card(U(\mathbb{Z}/N\mathbb{Z}))}{2} = \frac{\varphi(N)}{2}$ car $p \neq 1$, dû au fait que $B \subsetneq U(\mathbb{Z}/N\mathbb{Z})$, car N n'est pas un pseudo-premier
Donc $P(N\ n'est\ pas\ pseudo-premier) = P(Card(B) < N-1) = \frac{Card(B)}{N-1} \leq \frac{1}{2}$
Puis finalement, $P(N\ n'est\ pas\ pseudo-premier\ k\ fois) = (P(N\ n'est\ pas\ pseudo-premier))^k$ car c'est des événements indépendants.
On peut conclure que: $P(N\ n'est\ pas\ pseudo-premier\ k\ fois) \leq \frac{1}{2^k}$

On peut en conclure que pour un choix raisonnablement grand de k, la probabilité de tomber de ne pas tomber sur un pseudo-premier est faible.
Le seul problème que pose cet algorithme est le fait que l'on peut tomber sur un *nombre de Carmichael*, ce qui peut poser problème si le message que l'on veut encoder n'est pas premier avec N.
Pour régler ces problèmes, on pourrait considérer d'autres algorithmes plus puissants et efficaces, comme le *test de primalité de Rabin-Miller*, que l'on n'étudiera pas ici.

2.2.2 L'algorithme d'Euclide étendu

Afin d'obtenir l'inverse modulaire de e, d, il nous faut pouvoir obtenir la *décomposition de Bézout* de $\varphi(N)$ et e.

Pour cela, on va utiliser l'algorithme suivant:

Algorithme 3 Algorithme d'Euclide étendu

On prend a, et b, deux nombre entier strictement positifs et premiers entre eux.
$s_0 = 1$, $s_1 = 0$
$t_0 = 0$, $t_1 = 1$
$r_0 = a$, $r_1 = b$
On pose r_2 tel que $r_1 \equiv r_2\ [r_0]$
On note n le nombre d'itération de la boucle while ci-dessous, commençant à 1
Tant que $r_{n+1} \neq 0$ **faire**
    $s_{n+1} = s_{n-1} - (r_{n-1}/r_n)s_n$
    $t_{n+1} = t_{n-1} - (r_{n-1}/r_n)t_n$
    On pose r_{n+2} tel que $r_{n+1} \equiv r_{n+2}\ [r_n]$
Fin Tant que
Renvoyer (s_n, t_n)

2.2.3 Génération des clés

Les algorithmes suivants permettent de générer respectivement des nombres premiers aléatoires et les clés pour permettre le chiffrement *RSA*.

Algorithme 4 Algorithme de génération de nombre premiers

On choisit $N \in \mathbb{N}$ impair
Tant que Si le test de fermat ne retourne pas vrai pour N et k **faire**
    $N \leftarrow N + 2$
Fin Tant que
Renvoyer N

Algorithme 5 Algorithme de génération des clés RSA

On choisit p et q deux nombres premiers avec l'algorithme ci-dessus
$n \leftarrow p \times q$
On choisit e premier avec $\varphi(n)$
On applique l'algorithme d'Euclide étendu avec e et $\varphi(n)$ qui nous donne d et v tel que $de + v\varphi(n) = 1$
On a donc $\overline{ed}^{\varphi(n)} = \overline{1}^{\varphi(n)}$
On renvoie (n, e) et (n, d) respectivement la clé publique et la clé privée.

2.2.4 Implémentation des entiers

Le problème que l'on rencontre maintenant c'est la taille des entiers. En effet, si on était amené à utiliser notre algorithme sur des entiers représentés sur 64 bits, on n'aurait un problème de sécurité majeur, car il se trouve que trouver la décomposition en nombres premiers de n sur 64 bits et relativement simple, et peu couteux pour des ordinateurs modernes. Il nous faut donc un moyen de représenter les entiers sur des bases plus grandes que 64 bits.

Pour cela, on va utiliser des entiers qui ont comme base un multiple de 32 bits. C'est-à-dire qu'on va avoir un tableau de 32 bits, contenant dans chaque case de celui-ci un entier de 32 bits, et l'on va

créer des algorithmes autour de cet entier, pour pouvoir faire des opérations basiques.

On dénote dans le reste de cette partie N la taille des tableaux d'entier, que l'on nommera dès lors entier $N \times 32$.
On appellera MSB (*Most Significant Bit*) la valeur du coefficient sur le bit de point fort de la décomposition en binaire de l'entier étudié.

Algorithme 6 Algorithme d'addition de deux nombres entiers

Soit deux entiers $N \times 32$ A et B
On pose C l'entier résultant de l'addition.
On pose $retenue = 0$
Pour i dans $[\![0, N-1]\!]$ **faire**
    On pose $temp = A[i] + B[i] + retenue$ une variable temporaire
    Si MSB de $A[i]$ et $B[i]$ est 1 OU (MSB de $temp$ n'est pas 1 ET (MSB de $A[i]$ est 1 OU MSB de $B[i]$ est 1)) **alors**
        $retenue = 1$
    Sinon
        $retenue = 0$
    Fin Si
    $C[i] = temp$
Fin Pour
Renvoyer C

On justifie le choix de la condition dans l'algorithme:
On prend $i \in [\![0, N-1]\!]$
Si MSB de $A[i]$ et de $B[i]$ est 1, alors ça veut dire qu'il doit y avoir une retenue pour l'addition des deux entiers $A[i+1]$ et $B[i+1]$.
Sinon, si le MSB de $A[i] + B[i] + retenue$ est 0, alors, soit on additionner deux entiers avec un MSB de 0 dans ce cas là on a pas de retenue à l'addition en $i+1$. Mais si l'un des deux entier $A[i]$ ou $B[i]$ a un MSB de 1, alors forcément on a eu un débordement, puis dans ce cas là, on ajoute une retenue pour l'addition des entiers en $i+1$.

Cet algorithme est en $\mathcal{O}(N)$

On se propose désormais d'implémenter un algorithme de soustraction pour les entiers $N \times 32$:

Algorithme 7 Algorithme de soustraction

Soit deux entiers $N \times 32$ A et B
On pose B' l'entier $N \times 32$ obtenu après l'inversion de tous les bits de B et dont on a ajouté 1 (*Complément à deux de B*)
On pose C l'entier résultant de l'addition de A et B'.
On renvoie C

Cet algorithme est en $\mathcal{O}(N)$