Algorithme de chiffrement RSA, et méthode d'attaque

Martin Wattel, 12342

Problématique

Comment fonctionne l'algorithme de cryptographie RSA ? Est-il compliqué à déchiffrer ? Peut-on améliorer cette complexité ?

Sommaire

Qu'est-ce qu'un algorithme de chiffrement

Deux types d'algorithme de chiffrement :

- symétrique (Ex : AES, etc...)
- asymétrique (Ex : RSA, etc...)

Qu'est-ce qu'un algorithme de chiffrement

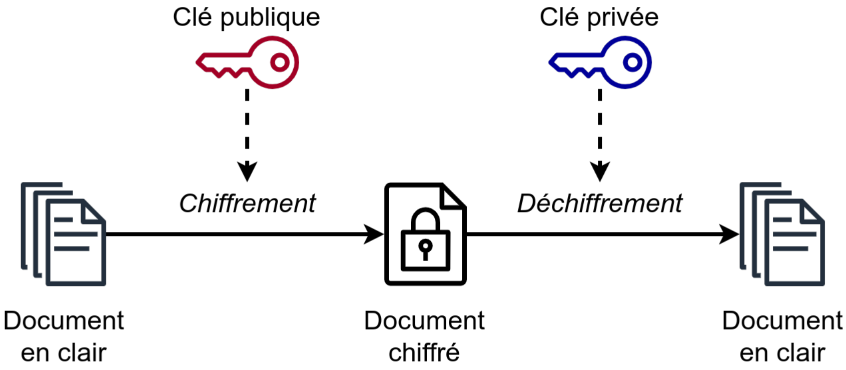

Figure 1 – Fonctionnement d'un algorithme de chiffrement asymétrique (Source : Wikipédia : Cryptographie Asymétrique)

Algorithme RSA

Definition 2.1

$< n, e >$ est une clé publique et $< n, d >$ est une clé privée, avec $n = pq$ avec $p, q \in \mathbb{P}$, et $\overline{ed} = \overline{1}$ dans $\mathbb{Z}/\varphi(n)\mathbb{Z}$, avec $\varphi(n) = (p-1)(q-1)$

Exemple 2.1

n est en générale coder sur 512 à 1024 bits pour l'utilisation actuelle de RSA.

Théorème 2.2

Pour $< n, e >$ et $< n, d >$ des clés publique et privée, on a :
$\forall m \in \mathbb{Z}, \quad |m| < n, \quad \overline{m^{ed}} = \overline{m}$ dans $\mathbb{Z}/n\mathbb{Z}$

Algorithme RSA

Fonctionnement de la génération des clés :

- On génère deux nombres premiers, et on calcule n
- On prend un $e \in [\![1, \varphi(n) - 1]\!]$ tel que $e \ \wedge \ \varphi(n) = 1$
- On calcul d

Propriété de fermat

Théorème de Fermat

Si p premier, alors $\forall a$ tel que a n'est pas un multiple de p, alors $a^{p-1} \equiv 1\ [p]$

Test de Fermat(p, k)

Génération de nombres premiers : Test de Fermat

Test de Fermat(p, k)

Pour $i = 1$, jusque k :

Génération de nombres premiers : Test de Fermat

Test de Fermat(p, k)

Pour $i = 1$, jusque k :

Prendre aléatoirement (uniformément) $a \in [\![1, p-1]\!]$

Génération de nombres premiers : Test de Fermat

Test de Fermat(p, k)

Pour $i = 1$, jusque k :

Prendre aléatoirement (uniformément) $a \in [\![1, p-1]\!]$

Si $\overline{a^{p-1}} \neq \overline{1}$, alors :

Génération de nombres premiers : Test de Fermat

Test de Fermat(p, k)

Pour $i = 1$, jusque k :

Prendre aléatoirement (uniformément) $a \in [\![1, p-1]\!]$

Si $\overline{a^{p-1}} \neq \overline{1}$, alors :

Renvoie Faux

Génération de nombres premiers : Test de Fermat

Test de Fermat(p, k)

Pour $i = 1$, jusque k :

Prendre aléatoirement (uniformément) $a \in [\![1, p-1]\!]$

Si $\overline{a^{p-1}} \neq \overline{1}$, alors :

Renvoie Faux

Renvoie Vrai

Génération de nombres premiers : Test de Fermat, correction

Définition 3.1

Un nombre de Carmichael est un entier p non premier, qui vérifie la propriété de Fermat quand $a \wedge p = 1$.

Propriété 3.2

Il existe une infinité de nombres de Carmichael.

Propriété 3.3

Si, p n'est pas un nombre de Carmichael : $P(\overline{a^{p-1}} = \overline{1} \mid p \notin \mathbb{P}) \leq \frac{1}{2}$.

Génération de nombres premiers : Test de Fermat, correction

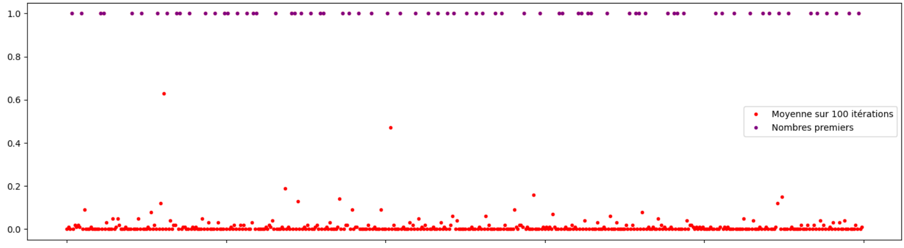

Figure 2 – Réponse moyenne de l'algorithme entre 500 et 1000 pour k = 1

Génération de nombres premiers : Test de Fermat, correction

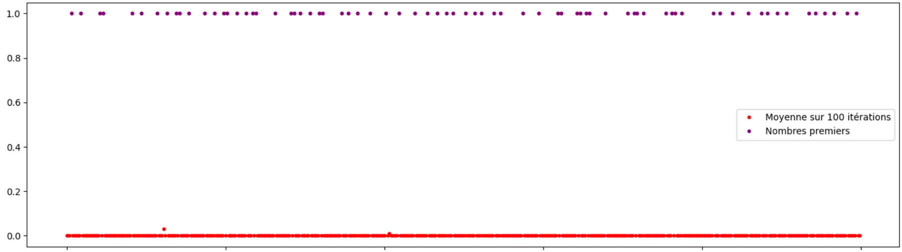

Figure 3 – Réponse moyenne de l'algorithme entre 500 et 1000 pour k = 5

Résultat sur un exemple concret en C

```
Entier de taille: 256 bits
Message initial
000000000000000000000000000000000000000000000000000000000000002f
Clé publique
e: 773d801ce9a886746d6425a1fe5c94c718faa7ec9abe24ce9a4c9dbe763edca3
n: 1a0f11b1ff7b4fd9a50f25a71752ca38f9d35f6a681bcc7a364c91312350b8b5
Clé privée
d: 0bc8fccad04400d0b90475bd2c10df9f124b99c1aab34bc3173d9d6d7fc0114b

Message chiffré
0e64e3e5e46eb7ef395dc6d233a657ea6922f77b8a0d2f4d9337410fce0b4411

Message déchiffré
000000000000000000000000000000000000000000000000000000000000002f

Temps pris: 2108.384000 ms
```

Figure 4 – Exécution du programme, qui génère des clés, et chiffre 47

L'algorithme naïf

Propriété 4.1

Pour $n = p \cdot q$, l'algorithme naïf a une complexité en $\mathcal{O}(\sqrt{n})$.

L'algorithme du Rho de Pollard, théorie

- On prend $(X_n)_{n\in\mathbb{N}^*}$ dans $[\![0, n-1]\!]$, uniformément distribué modulo p et indépendantes.

L'algorithme du Rho de Pollard, théorie

- On prend $(X_n)_{n\in\mathbb{N}^*}$ dans $[\![0, n-1]\!]$, uniformément distribué modulo p et indépendantes.
- Et L une vad dans $\mathbb{N}\backslash\{0, 1\}$, plus petit entier, tel que $\exists j \in [\![1, L-1]\!]$ tel que $X_L(\omega) = X_j(\omega)\ [p]$

L'algorithme du Rho de Pollard, théorie

- On prend $(X_n)_{n\in\mathbb{N}^*}$ dans $[\![0, n-1]\!]$, uniformément distribué modulo p et indépendantes.
- Et L une vad dans $\mathbb{N}\backslash\{0,1\}$, plus petit entier, tel que $\exists j \in [\![1, L-1]\!]$ tel que $X_L(\omega) = X_j(\omega)\ [p]$
- $p \mid X_L(\omega) - X_j(\omega)$ et $p \mid n$

L'algorithme du Rho de Pollard, théorie

- On prend $(X_n)_{n\in\mathbb{N}^*}$ dans $[\![0, n-1]\!]$, uniformément distribué modulo p et indépendantes.
- Et L une vad dans $\mathbb{N}\backslash\{0,1\}$, plus petit entier, tel que $\exists j \in [\![1, L-1]\!]$ tel que $X_L(\omega) = X_j(\omega)\ [p]$
- $p \mid X_L(\omega) - X_j(\omega)$ et $p \mid n$
- $p \mid (X_L(\omega) - X_j(\omega)) \ \wedge \ n$

L'algorithme du Rho de Pollard, théorie

- On prend $(X_n)_{n\in\mathbb{N}^*}$ dans $[\![0, n-1]\!]$, uniformément distribué modulo p et indépendantes.
- Et L une vad dans $\mathbb{N}\backslash\{0,1\}$, plus petit entier, tel que $\exists j \in [\![1, L-1]\!]$ tel que $X_L(\omega) = X_j(\omega)\ [p]$
- $p \mid X_L(\omega) - X_j(\omega)$ et $p \mid n$
- $p \mid (X_L(\omega) - X_j(\omega)) \ \wedge \ n$
- Le but est de trouver les cycles dans la suite.

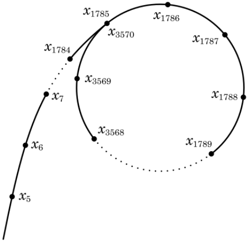

Figure 5 – Représentation de la suite

L'algorithme du Rho de Pollard, théorie

Propriété 4.2

$P(L \geq i) \leq e^{-\frac{i^2}{2p}}$, si $i \ll p$

Exemple 4.3

$P(L \geq \sqrt{p}) \leq e^{-\frac{1}{2}} \leq \frac{2}{3}$

L'algorithme du Rho de Pollard, théorie

Propriété 4.4

$E(L) \leq \sqrt{\frac{\pi p}{2}}$. Donc, l'algorithme est en moyenne en $\mathcal{O}(\sqrt{p}) = \mathcal{O}(n^{1/4})$

Implémentation

Pollard's Rho(n, f)

Implémentation

Pollard's Rho(n, f)

On pose $x_1 = y_1$ aléatoirement pris dans $[\![1, n-1]\!]$

Implémentation

Pollard's Rho(n, f)

On pose $x_1 = y_1$ aléatoirement pris dans $[\![1, n-1]\!]$

Tant que $(x_i - y_i) \wedge n = 1$ ou $(x_i - y_i) \wedge n = n$

Implémentation

Pollard's Rho(n, f)

On pose $x_1 = y_1$ aléatoirement pris dans $[\![1, n-1]\!]$

Tant que $(x_i - y_i) \wedge n = 1$ ou $(x_i - y_i) \wedge n = n$

On calcule $x_{i+1} = f(x_i)$ et $y_{i+1} = f(f(y_i))$

Implémentation

Pollard's Rho(n, f)

On pose $x_1 = y_1$ aléatoirement pris dans $[\![1, n-1]\!]$

Tant que $(x_i - y_i) \wedge n = 1$ ou $(x_i - y_i) \wedge n = n$

On calcule $x_{i+1} = f(x_i)$ et $y_{i+1} = f(f(y_i))$

Implémentation

Pollard's Rho(n, f)

On pose $x_1 = y_1$ aléatoirement pris dans $[\![1, n-1]\!]$

Tant que $(x_i - y_i) \wedge n = 1$ ou $(x_i - y_i) \wedge n = n$

On calcule $x_{i+1} = f(x_i)$ et $y_{i+1} = f(f(y_i))$

Renvoie $(x_i - y_i) \wedge n$

Implémentation

Propriété 4.5

Empiriquement $f : x \mapsto x^2 + 1 \; [n]$ donne une suite quasi-unifome modulo p.

Implémentation en Python des deux algorithmes

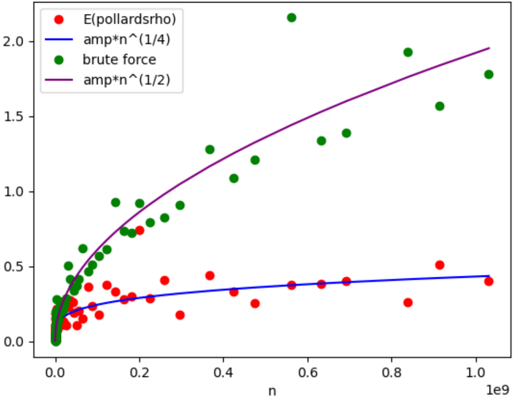

Figure 6 – Temps mis par les algorithmes en ms (n=2 à 2^{32})

Conclusion

Approfondissements sur le test de primalité : amélioration possible grâce à l'algorithme de Miller-Rabin.
Approfondissements sur la Cryptanalyse RSA : amélioration possible grâce à des algorithmes plus sophistiqués, type crible quadratique, crible algébrique, etc...

Annexe

Preuve 2.2

$\overline{ed} = \overline{1} \implies ed = 1 + k\varphi(n)$

Donc :

$\overline{m^{ed}} = \overline{m} \cdot \overline{m^{\varphi(n)}}^{k}$

Or : $\overline{m^{p-1}} = \overline{1}$, de même pour q. Puis $\varphi(n) = (p-1)(q-1)$

Finalement : $\overline{m^{ed}} = \overline{m}$

Preuve 3.3
On pose $B = \{a \mid \overline{a^{p-1}} = \overline{1}\}$
Déjà :

- Si p premier, alors $B = U(\mathbb{Z}/p\mathbb{Z})$
- Sinon, $B \subsetneq U(\mathbb{Z}/p\mathbb{Z})$

Or, B est un sous-groupe strict de $U(\mathbb{Z}/p\mathbb{Z})$, donc, d'après le théorème de Lagrange : $|B| \ / \ \varphi(p)$, donc $\varphi(p) = k|B|$.
Puis, $k \geq 2$, donc : $\varphi(p) \geq 2|B|$, et : $\varphi(p) \leq p - 1$
Finalement : $P(...) = \frac{|B|}{p-1} \leq \frac{1}{2}$.

Preuve 4.2

$$
\begin{aligned}
P(L \geq i) &= \frac{p-1}{p} \dots \frac{p-(i-1)}{p} \\
&= \left(1 - \frac{1}{p}\right) \dots \left(1 - \frac{i-1}{p}\right) \\
&\leq e^{-\frac{1}{p}} \dots e^{-\frac{i-1}{p}} \\
&\leq e^{-\frac{i(i-1)}{2p}}
\end{aligned}
$$

Or, $i \ll p$, donc $\frac{i(i-1)}{p} \approx \frac{i^2}{p}$

$$
\leq e^{-\frac{i^2}{2p}}
$$

Preuve 4.4

$$
\begin{aligned}
\mathrm{E(L)} &= \textstyle\sum_{i=3}^{\infty} P(L \geq i) \\
&\leq \textstyle\sum_{i=3}^{\infty} e^{-\frac{i^2}{2p}} \\
&\leq \textstyle\sum_{i=1}^{\infty} e^{-\frac{i^2}{2p}}
\end{aligned}
$$

Or, $e^{-\frac{i^2}{2p}} \leq \int_{i-1}^{i} e^{-\frac{t^2}{2p}}\, dt$, donc :

$$
\begin{aligned}
&\leq \textstyle\int_0^{\infty} e^{-\frac{t^2}{2p}}\, dt \\
&\leq \sqrt{2p} \textstyle\int_0^{\infty} e^{-t^2}\, dt \\
&\leq \sqrt{2p} \textstyle\int_0^{\infty} e^{-t^2}\, dt \\
&\leq \sqrt{\frac{\pi p}{2}}
\end{aligned}
$$

Annexe : Exponentiation modulaire

Exponentiation modulaire(b, p, n) :

On pose $acc = b\ [n]$

On pose $out = 1$

Tant que $p > 0$ faire :

Si p est impair, faire :

$out = out \cdot acc\ [n]$

$acc = acc \cdot acc\ [n]$

$p = p/2$

Renvoie out

L'algorithme s'exécute en $\mathcal{O}(log(p))$.

Annexe : Structure du programme Python

Liste des fonctions utilisées, et leur utilitée :

- pow_mod(b, p, n) → Calcul $b^p \ mod \ n$
- fermat_test(n, k) → Test si n et un nombre premier avec une certitude d'au moins $1 - \left(\frac{1}{2}\right)^k$
- generate_prime(size_max, size_min, k) → Génère un nombre premier entre $[\![size\_min, size\_max]\!]$.
- pgcd(a, b) → ...
- pollardsrho(n, f) → Trouve un facteur premier de n, en utilisant la fonction f (bien souvent $f = x \to x^2 + 1$.)
- pollardsrho(n) → Trouve un facteur premier de n.
- brute_force_time / pollardsrho_time → Calcul le temps que les algorithmes ont pris en ms.

Annexe : Structure du fichier header intx_t.h

Liste des fonctions (importantes) utilisées, et leur utilitée :

- type intx_t → Tableau de n entiers de type int32_t.
- intx_add(intx_t a, intx_t b, intx_t out) → Calcul $a + b$.
- intx_two_complement(intx_t in, intx_t out) → Calcul le complément à deux de a.
- intx_sub(intx_t a, intx_t b, intx_t out) → Calcul $a - b$
- intx_lshift(intx_t in, int p, intx_t out) → Fait un décalage de p bits vers la gauche.
- intx_rshift(intx_t in, int p, intx_t out) → ...
- intx_mul(intx_t a, intx_t b, intx_t out) → Multiplie a et b entre eux.
- intx_div(tntx_t n, intx_t d, intx_t q, intx_t r) → Fait la division euclidienne de n par d.
- intx_pow_mod(intx_t b, intx_t p, intx_t n, intx_t out) → Calcul $b^p \ mod \ n$.

Annexe : Structure du fichier header arithmetic.h

Liste des fonctions (importantes) utilisées, et leur utilitée :

- extended_euclidean(intx_t a, intx_t b, intx_t u, intx_t v, intx_t gcd) $\rightarrow$ Calcul la décomposition de Bézout de a et b.
- fermat_test(intx_t p, int k) $\rightarrow$ Applique le test de fermat sur p avec k itération
- generate_prime(fonction de test f, int k, intx_t out, uint64_t mask) $\rightarrow$ Renvoie un nombre premier de taille au plus $32n$ avec f une fonction de test.
- inv_mod(intx_t a, intx_t n, intx_t out) $\rightarrow$ Calcul l'inverse modulaire de a modulo n.
- select_random_coprime(intx_t n, intx_t out) $\rightarrow$ Cherche un entier out tel que $out \wedge n = 1$.

Annexe : Structure du fichier header algorithms.h

Liste des fonctions utilisées, et leur utilitée :

- type public_key (struct)
  - intx_t e
  - intx_t n
- type private_key (struct)
  - intx_t d
  - intx_t n
- generate_keypair(public_key puk, private_key prk, uint64_t mask) → Génére une clé publique, et une clé privée.
- encrypt(intx_t in, public_key puk, intx_t out) → Chiffre le message avec puk (on calcul in^e)
- decrypt(intx_t in, public_key prk, intx_t out) → Déchiffre le message avec prk (on calcul in^d)

Annexe : Principe de fonctionnement de l'addition sur $32n$ bits

On utilise le masque suivant pour déterminer la retenue :
(a & 0x80000000 && b & 0x80000000) || (! $(a + b + c$ & 0x80000000)
&& (a & 0x80000000 || b & 80000000))
L'algorithme s'exécute en $\mathcal{O}(n)$.

Annexe : Principe de fonctionnement de la multiplication sur $32n$ bits

On utilise la propriété suivante :

$$a = \sum_{i=1}^{n} a_i \cdot 2^{32i} \ et \ b = \sum_{i=1}^{n} b_i \cdot 2^{32i}$$

$$a \cdot b = \sum_{i=1}^{n} \sum_{j=1}^{n} a_i \cdot b_j \cdot 2^{32 \cdot (i+j)}$$

L'algorithme s'exécute en $\mathcal{O}(n^3)$.

Annexe : Complexité de la division euclidienne sur $32n$ bits

L'algorithme s'exécute en $\mathcal{O}(n)$.

Annexe : Approfondissement sur les nombres de Carmichael

$561 = 3 \cdot 11 \cdot 17$. Donc il y a beaucoup de a tel que $a \wedge 561 = 1$.