Cryptographie et Sécurité
Série 9 : TP Python - RSA

17 Novembre 2022

Délai de rendu spécial (jeudi soir au lieu de mercredi soir) : A rendre sur **Moodle**, sous forme d'un fichier python **.py**, au plus tard le Jeudi **24 Novembre 2022 à 23h59**.

Votre code doit être **suffisamment commenté**.

Implémentation de RSA

Le but de cette série est d'implémenter RSA. Pour cela, on aura besoin d'implémenter divers outils et algorithmes :

1. **Un générateur de nombres premiers**, incluant un test de primalité (**test de Fermat**)
2. **L'algorithme d'exponentiation rapide**
3. **L'algorithme étendu d'Euclide**
4. **Un générateur de Clés**
5. **L'Encryption des messages**
6. **La Decryption des messages**.

On rappelle tous ces algorithmes en détail, puis on donnera une marche à suivre un peu plus détaillée pour l'implémentation.

Générateur de nombres premiers

Pour la génération des clés, on a besoin de générer p et q, deux entiers, premiers, de grande taille.

La méthode qu'on utilise consiste à générer un entier aléatoire, puis à vérifier s'il est premier.

Pour cela, on utilise le test de primalité de Fermat. Si n est le nombre qu'on veut tester, on choisit un nombre aléatoire a entre 2 et $n-1$ (inclus), et on calcule $a^{n-1} \mod n$. Si le résultat est différent de 1, alors n n'est pas premier (c.f. petit théorème de Fermat). Sinon, n est "probablement premier". On répète ce test avec différentes valeurs de a pour augmenter la confiance qu'on a en le fait que n soit premier.

Pour ce TP, considérez que 20 répétitions sont suffisantes.

Exponentiation rapide

Pour calculer plus facilement toutes les exponentiations, que ce soit pour le test de primalité ou l'encryption, on utilise l'exponentiation rapide (Fast Exponentiation) :

Exemple : On veut calculer $3^{42} \mod 25$. Le principe est de calculer les exposants dont la valeur est une puissance de 2 :

$3^1 \mod 25 = 3$
$3^2 \mod 25 = 9$
$3^4 \equiv 3^2 \cdot 3^2 \equiv 9 \cdot 9 \equiv 81 \mod 25 = 6$
$3^8 \equiv 6 \cdot 6 \equiv 36 \mod 25 = 11$
$3^{16} \equiv 121 \mod 25 = 21$
$3^{32} \equiv 21 \cdot 21 \equiv 441 \mod 25 = 16$

En décomposant $42 = 32 + 8 + 2$ en puissances de 2, on peut calculer aisément :

$$3^{42} \equiv 3^{32} \cdot 3^8 \cdot 3^2 \equiv 16 \cdot 11 \cdot 9 \equiv 1584 \mod 25 = 9$$

Algorithme étendu d'Euclide

On utilise l'algorithme étendu d'Euclide lorsqu'on crée les clés pour calculer l'inverse de l'exposant e modulo $\phi(n)$.

Soit deux entiers a et b, $a \geq b$, cet algorithme nous permet d'obtenir $r = pgdc(a, b)$ et s, t tels que $s \cdot a + t \cdot b = r$ (i.e. les coefficients de Bézout). Si a et b sont premiers entre eux, alors s est l'inverse de a modulo b, et t est l'inverse de b modulo a.

Ici, si $a = \phi(n)$ et $b = e$, on s'intéresse donc à r, qui nous indique si e est premier avec $\phi(n)$, et à t, l'inverse de $e \mod \phi(n)$.

L'algorithme est le suivant ($\div$ est une **division entière** !):

r_0 := a;
r_1 := b;

$s_0 := 1;$
$s_1 := 0;$
$t_0 := 0;$
$t_1 := 1;$
$q_1 := r_0 \div r_1;$

repeat until $r_{i+1} == 0$
$r_{i+1} := r_{i-1} - q_i * r_i;$
$s_{i+1} := s_{i-1} - q_i * s_i;$
$t_{i+1} := t_{i-1} - q_i * t_i;$
$q_{i+1} := r_i \div r_{i+1};$
end repeat;

return r_i, s_i, t_i;

Attention : L'algorithme retourne les coefficients de Bézout, donc des nombres positifs ou négatifs. Il faut donc penser à vérifier que s (resp. t) est bien compris entre 0 et b-1 (resp. a-1), cad que s (resp. t) est bien modulo b (resp. a).

Et lorsque r_{i+1} est nul, on ne calcule pas les autres éléments (car q_{i+1} est alors impossible car division par 0).

Détails d'implémentation

Les nombres premiers que vous générez devront être de taille "réaliste" pour un algorithme RSA, c'est à dire p et q de taille 512 bits minimum, et donc un n d'au moins 1024 bits.

Même avec des nombres de cette taille, le processus entier de votre code ne doit pas prendre plus de quelques secondes (création des clés + encryption + décryption). Si votre code prend trop longtemps (plus d'une minute), testez vos fonctions indépendamment avec de grands nombres pour voir d'ou vient la perte de temps.
Notamment, attention à bien utiliser votre exponentiation rapide dans tous les algorithmes qui font des exponentiations modulaires (y compris dans le test de Fermat ou le calcul des inverses par exemple).
N'hésitez pas à également tester vos fonctions avec de petits nombres (vérifiables à la main) ou avec des exemples déjà vus (valeurs de la série 8 par ex) pour voir si elles fonctionnent correctement.

Considérez que le message m est une simple valeur numérique, et de moins de 1024 bits, donc inférieure à n et encryptable d'un seul bloc.

Votre code doit contenir :

- L'exponentiation rapide pour calculer la puissance d'un nombre modulo un autre.
- La génération de nombres premiers aléatoires, a partir de nombre aléatoires dont on vérifie la primalité à l'aide du test de Fermat répété plusieurs fois.
- L'algorithme étendu d'Euclide pour calculer l'inverse d'un nombre modulo un autre.
- La génération de clés, en créant d'abord p et q premiers et grands avec le générateur de premiers aléatoires que vos aurez crée, puis $n = p \cdot q$, puis en générant un exposant d'encryption e petit (et premier avec $\Phi(n)$), puis en utilisant l'algorithme étendu d'Euclide pour calculer l'exposant de décryption $d = e^{-1} \mod \Phi(n)$.
- L'encryption, avec les clés publiques n et e, via l'exponentiation rapide. On considère les messages comme étant déjà sous la forme de nombres (et des nombres plus petits que n).
- La décryption.

Vous devez implémenter ces algorithmes vous-mêmes, sans utiliser une librairie ou une fonction qui le fait déjà. Entre autres, cela veut dire que vous pouvez utiliser la fonction pow() seulement avec deux arguments, pas avec trois arguments (qui fait l'exponentiation rapide modulo n). En revanche, vous pouvez tout à fait vous servir de cette fonction ou d'autres librairies à titre de comparaison pour vérifier si vos algorithmes donnent les bons résultats.