Sécurité Informatique et protection des données

TP - Protocoles cryptographiques

Required Files: tp_crypto_etu.py (Télécharger)
Nombre maximal de fichiers: 6
Type de travail: Travail individuel

Résumé de l'état du travail

Devoirs rendus: 0
Dernière soumission: Aucun

TP - Protocoles cryptographiques ; durée 5h (2 séances);

L'objectif du TP est d'implémenter de A à Z le protocole RSA en commençant par les primitives arithmétiques, puis le protocole RSA et enfin une attaque sur le protocole RSA. Attention à bien respecter les contraintes imposées dans la spécification des fonctions. Si ces contraintes ne sont pas respectées, les tests unitaires réalisés par Caséine ne passeront pas. Attention aussi à la complexité de vos fonctions. Vos algorithmes peuvent retourner le bon résultat mais les tests utilisent des grands entiers (lorsque c'est possible). Faîtes en sorte que le temps de calcul reste raisonnable.

1- Fonctions arithmétiques de base

Toutes les fonctions de cette partie existent nativement dans Python ou bien dans des bibliothèques publiques. Bien évidemment, vous n'avez pas le droit de les utiliser. En particulier, la méthode *pow* de python ne doit pas être utilisée puisque la méthode *square_and_multiply* est à implémenter.

En complétant le fichier fourni avec Caséine, implémentez les fonctions suivantes:

1. fonction ***euclid(a,b)*:** cette fonction retourne l'unique **PGCD positif** des deux entiers relatifs *a* et *b* passés en paramètres en utilisant l'algorithme d'Euclide;
2. fonction ***extended_euclid(a,b)***: les paramètres *a* et *b* sont des entiers relatifs. Cette fonction retourne un triplet *d,u,v* (dans cet ordre) avec *d* l'unique **PGCD positif** des deux entiers *a* et *b* et *(u,v)* un couple de coefficients de Bezout vérifiant *a.u+b.v=d*. L'algorithme utilisé est l'algorithme d'Euclide étendu vu en cours;
3. fonction ***modular_inverse(a,n)***: fonction qui prend en entrée deux entiers *a* et *n* dans $\mathbb{Z}$ et qui retourne l'unique inverse modulaire de *a* modulo *n* compris entre *0* et *n-1*. Si *a* n'est pas inversible modulo *n*, alors *0* est renvoyé;
4. fonction ***naive_euler_function(n)***: fonction qui prend en entrée un entier positif *n>1* et qui retourne l'indicatrice d'Euler $\varphi(n)$ en testant un par un tous les entiers compris entre *1* et *n*;
5. fonction ***square_and_multiply(a,k,n):*** fonction qui prend en entrée trois entiers *a*, *k* et *n* avec *k* positif et qui calcule $a^k \mod n$ avec l'algorithme **Square and Multiply** suivant:

   Entrées : les entiers a, k et n avec $k = \sum_{i=0}^{p} k_i 2^i$
   Sortie : $a^k \mod n$
   (a) $h \leftarrow 1$
   (b) Pour $i = p$ à 0
      i. $h \leftarrow h \times h \mod n$
      ii. Si $k_i = 1$ alors $h \leftarrow h \times a \mod n$
   (c) Retourner h

   Vous pourrez utiliser la fonction ***bin*** de python pour la décomposition binaire d'un entier. Le résultat doit être compris entre 0 et n-1.
6. fonction ***euler_function(L1,L2)***: fonction qui prend en entrée deux listes $L1 = [p_1, p_2, \ldots, p_k]$ et $L2 = [\alpha_1, \alpha_2, \ldots, \alpha_k]$ de même longueur avec $L1$ une liste de nombres premiers **distincts** et $L2$ une liste d'entiers strictement positifs. La fonction retourne l'indicatrice d'Euler $\varphi(n)$ avec $n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}$ selon la formule vue en

cours.

7. fonction ***inversibles(n):*** fonction qui prend en entrée un entier positif *n* et qui retourne la liste de tous les éléments inversibles modulo *n* (les inverses ne sont pas demandés);
8. fonction ***miller_rabin(n,d):*** fonction qui prend en entrée deux entiers positifs *n* et *d* et qui teste (avec le test de Miller-Rabin) si *n* est un nombre premier. La probabilité d'erreur de l'algorithme sera inférieure à $1/4^d$.
9. fonction ***generate_prime(k,d):*** fonction qui prend en entrée deux entiers positifs *k* et *d* et qui retourne un nombre premier ayant **exactement *k* bits**. L'algorithme pourra se tromper sur la primalité du nombre avec une probabilité inférieure à $1/4^d$.

2- Protocole RSA

Dans le même fichier, implémentez les fonctions suivantes liées au protocole RSA.

1. fonction ***generate_key(k):*** fonction qui prend en entrée un entier **positif et pair** *k* et qui retourne les éléments $[p, q, n = pq, \varphi(n), d, e]$ (dans cet ordre) avec *(n,e)* la clé publique du protocole RSA et $(p, q, \varphi(n), d)$ la clé privée associée. Les entiers *p* et *q* devront être des nombres premiers distincts avec **exactement** *k/2* bits et *n* aura **exactement** *k* bits. Pour tester la primalité, vous pourrez prendre comme paramètre *d=40* (pour Miller-Rabin).
2. fonction ***encipher(m,n,e)***: fonction de chiffrement RSA qui retourne le chiffré du message $m \in (\mathbb{Z}/n\mathbb{Z})^\times$ en utilisant la clé publique *(n,e)*.
3. fonction ***decipher(c,d,n)***: fonction de déchiffrement RSA qui retourne le message clair associé au chiffré $c \in (\mathbb{Z}/n\mathbb{Z})^\times$ en utilisant la clé privée *d*.

3- Une cryptanalyse du protocole RSA avec un petit exposant secret

L'utilisation d'un exposant secret (d) trop petit dans RSA met en danger sa sécurité. L'attaque de Wiener proposée ici est basée sur le *développement en fractions continues* d'un rationnel. On note

$$[q_0, q_1, \ldots, q_m] = q_0 + \cfrac{1}{q_1 + \cfrac{1}{\ddots \cfrac{}{q_{m-1} + \cfrac{1}{q_m}}}}.$$

Soit $f \in \mathbb{R}$, on pose

$$\begin{cases} q_0 = \lfloor f \rfloor, & r_0 = f - q_0 \\ q_i = \left\lfloor \frac{1}{r_{i-1}} \right\rfloor, & r_i = \frac{1}{r_{i-1}} - q_i, \quad i = 1, 2 \ldots \end{cases}$$

tant que r_{i-1} est non nul (remarquez que les q_i sont les quotients qui apparaissent dans l'algorithme d'Euclide).

On a donc $f = [q_0, q_1, \ldots, q_{n-1}, q_n + r_n] \; \forall \; n \in \mathbb{N}$.

La suite, finie ou infinie, des q_i s'appelle *le développement en fractions continues de* f.

Les nombres rationnels $f_i = \frac{n_i}{d_i} = [q_0, q_1, \ldots, q_{i-1}, q_i]$ sont dits *réduites ou développement partiel* de f.

Les récurrences suivantes permettent de calculer les réduites :

$$\begin{cases} n_0 = q_0 \\ d_0 = 1 \end{cases} \text{ et } \begin{cases} n_1 = q_0 q_1 + 1 \\ d_1 = q_1 \end{cases}$$

puis, pour $i \geq 2$,

$$\begin{cases} n_i = q_i n_{i-1} + n_{i-2} \\ d_i = q_i d_{i-1} + d_{i-2} \end{cases}$$

Le théorème fondamental qui va vous permettre de réaliser la cryptanalyse est le suivant :

Théorème: *Supposons que* $\text{pgcd}(a, b) = \text{pgcd}(c, d) = 1$, *et*
$\left|\frac{a}{b} - \frac{c}{d}\right| \leq \frac{1}{2d^2}$.
Alors c/d *est l'une des réduites du développement en fractions continues de* a/b.

On peut montrer, en étudiant la relation $ed = 1 \mod \varphi(N)$ que le théorème précédent permet, grâce à l'algorithme de calcul des réduites, de casser RSA lorsque l'exposant secret est trop petit.

En effet, si $ed = 1 \mod \varphi(N)$, alors il existe $k \in \mathbb{Z}$ tel que $ed = 1 + k\varphi(N)$ et on montre que
$\left|\frac{e}{N} - \frac{k}{d}\right| \leq \frac{3}{\sqrt{N}}$
sachant que *p* et *q* valent environ $\sqrt{N}$ tous les deux. Maintenant si
$(\star)$ $\frac{3}{\sqrt{N}} \leq \frac{1}{2d^2}$ ou de manière équivalente $d^2 \leq \frac{\sqrt{N}}{6}$,
alors
$\left|\frac{e}{N} - \frac{k}{d}\right| \leq \frac{1}{2d^2}$
et k/d est une réduite de la fraction (publique) e/N.

Dans le même fichier, implémentez les fonctions suivantes liées à l'attaque de Wiener:

1. fonction ***generate_key_wiener(k):*** fonction qui prend en entrée un entier **positif et pair** *k* et qui retourne les éléments $[p, q, n = pq, \varphi(n), d, e]$ (dans cet ordre) avec *(n,e)* la clé publique du protocole RSA et $(p, q, \varphi(n), d)$ la clé privée associée. Les entiers *p* et *q* devront être des nombres premiers distincts avec **exactement** *k/2* bits et *n* aura **exactement** *k* bits. Pour tester la primalité, vous pourrez prendre comme paramètre *d=40* (pour Miller-Rabin). La clé privée *d* doit satisfaire l'inégalité (*) ci-dessus pour appliquer l'attaque de Wiener.
2. fonction ***find_secret_key_wiener(n,e)***: fonction qui retourne la clé secrète d associée à la clé publique RSA (n,e) en appliquant l'attaque de Wener. Si l'attaque échoue, elle doit retourner -1. L'attaque échoue lorsque toutes les réduites ont été calculées mais qu'aucune n'a révélé la clé secrète.

Fichiers demandés

tp_crypto_etu.py