IF202 Cryptologie **TP** **2021-2022**

Que faire ?
Travail à réaliser par équipe de 2 ou 3 personnes.
Rendre : Code source + court rapport.Ecrire les noms des participants dans chacun de ces fichiers.Si vous choisissez le langage `Python`, insérer le code-source dans le fichier `rsa_start.py` (qui est un canevas du travail à réaliser)
Envoyer : par email à votre encadrant de TD/TP, avant le Jeudi 19 Mai 2022 à 23h 59.

Table des matières

Préambule

Le but de ce TP est d'écrire une implémentation simple de RSA dans le langage de votre choix. Quelques indications pour une programmation en Python sont données dans le sujet. Attention, vous allez manipuler de grands nombres entiers : tenez en compte si vous choisissez un autre langage que Python. Il est fortement recommandé de tester unitairement vos fonctions sur des entrées judicieusement choisies au fur et à mesure que vous les écrivez.

1. Un peu d'arithmétique

1.1 pgcd, inverse modulaire

Q1 Écrire une fonctions pgcd(a, b) qui renvoie le pgcd de a et b, où a et b sont deux entiers $>$ 0. Mettez en oeuvre l'algorithme d'Euclide étendu dans une fonction `euclide_ext(a, b)`. Cette fonction retourne un couple de Bezout (u, v) tel que ua + vb = pgcd(a, b).
Q2 Déduire de la question précédente une fonction `inverse_modulaire(a, N )` calculant un inverse de a modulo N . Pour mémoire, b est un inverse de a modulo N s'il existe un entier k tel que $ab = 1 + kN$, i.e. $ab \equiv 1 \pmod{N}$.

1.2 Exponentiation modulaire

L'exponentiation modulaire pour un exposant e, une base b et un module N consiste à calculer c tel que $c \in [0, N-1]$ et $c \equiv b^e \pmod{N}$.
Q3 Écrire une fonction d'exponentiation modulaire `expo_modulaire(e,b,n)` en e étapes par le produit des congruences.
Exemple : $111^3 \pmod{13} = (111 \times 111 \pmod{13}) \times 111 \pmod{13})$.
Attention à bien calculer les résidus modulaires à chaque étape pour éviter des nombre intermédiaires trop énormes.
Afficher pour chaque exécution de la fonction le nombre d'opérations "×" et modulo effectuées par votre algorithme. Il existe une méthode beaucoup plus efficace que le produit des congruences pour effectuer une exponentiation. Elle nécessite le calcul de la *représentation binaire* de l'exposant. Supposons que l'on veuille calculer b^{17} pour un b donné. Avec la version naïve de la question précédente, on effectue 17 multiplications par b. Maintenant, écrivons 17 en binaire : $17_{10} = 10001_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 2^4 + 2^0$. On a donc $b^{17} = b^{2^4+1} = b^{2^4} \times b$. Ainsi, en effectuant des élévations au carré successives ($b^2 = b \times b, b^4 = b^2 \times b^2 \ldots$), on peut calculer b^{17} avec seulement 5 multiplications.
Q4 Écrire une fonction d'exponentiation modulaire rapide utilisant la fonction précédente. Afficher le nombre d'opérations ("×" et modulo) effectuées à chaque appel de la fonction. Comparer, pour des entrées identiques, les performances avec la fonction de la question 3. Conclure.
Aide à la programmation : En Python, on pourra utiliser la fonction `bin(n)` qui renvoie la représentation binaire de n.[1]

2. Tests de primalité

Pour le cryptosystème RSA, la génération des clés nécessite de grands nombres premiers. Nous allons nous intéresser ici à comment en obtenir de façon efficace.

2.1 Crible d'Ératosthène

Une première idée pour obtenir des nombres premiers est de tous les générer jusqu'à une borne n fixée. Le principe du crible d'Eratosthène est très simple : on part de la liste d'entiers de 2 à n. À chaque étape, soit p le plus petit entier de la liste. On conserve p comme nombre premier puis on rappelle le procédé sur la liste initiale privée de tous les multiples de p. On s'arrête lorsque que le premier élément de la liste au carré est strictement supérieur à n.
Q5 Quelle est la complexité en temps et en espace du crible d'Erathostène ?
Une autre stratégie consiste à tirer un entier aléatoirement et tester ensuite s'il est premier. Pour cela, il est nécessaire de déterminer rapidement si un entier donné est premier.

2.2 Test de Fermat

Une première méthode pour tester la primalité d'un nombre est appelée test de Fermat et est fondée sur le petit théorème de Fermat :

1. poids faibles à droite, précédée de "0b"

Théorème 1 Si p est un nombre premier alors $\forall a \in [1, p-1], a^{p-1} \equiv 1 \pmod{p}$.

Si un entier n satisfait ce test pour de « nombreux » a, alors n a de fortes chances d'être premier. On choisit donc des a au hasard, on les élève à la puissance $n-1$ et on vérifie la congruence à 1 modulo n.

Q6 Écrire un test de primalité `test_fermat(n,t)` utilisant ce principe.

Reprenez votre fonction d'exponentiation rapide. Votre fonction prendra en paramètre un entier n à tester et un entier t représentant le nombre de tests à faire.

Aide à la programmation : En Python, on pourra utiliser `randint(m,M)` [2] du module `random` pour obtenir un entier aléatoire dans $[m, M]$. Toutefois, cette méthode possède une faiblesse pour les nombres composés dits « de Carmichael » qui ne sont pas premiers mais pour lesquels tous les entiers plus petits passent le test de Fermat.

2.3 Test de Miller-Rabin

Le test de Miller-Rabin (Rabin, en abrégé) permet de contourner le problème des nombres de Carmichael. Il tire parti d'une propriété de l'entier n, qui dépend d'un entier auxiliaire, le témoin. Pour un entier n impair, notons $n-1 = 2^r \times u$ où r et u sont les entiers tels que $r \geq 1$ et u impair. En d'autres termes, r est le nombre maximum de fois que l'on peut mettre 2 en facteur dans $n-1$. Soit $a \in Z_n^*$ [3]. Le nombre $a \in [1, n-1]$ est un *témoin* que n est composé si et seulement si

1. $a^u \not\equiv 1 \pmod{n}$
2. $\forall i, 0 \leq i \leq r-1 \Rightarrow (a^{2^i \cdot u}) \not\equiv -1 \pmod{n}$.

On admettra le

Théorème 2 : Soit n un entier impair. L'entier n est premier,si et seulement si, il n'existe pas de témoin de Rabin pour n. De plus, lorsque n est impair composé, il admet beaucoup de témoins :

Proposition 3 : Pour tout nombre impair composé n, au moins $\frac{3}{4}$ des entiers $a \in [1, n-1]$ sont des témoins de Rabin pour n.

Le principe du test de Rabin est de vérifier qu'un nombre suffisamment grand d'entiers $a \in Z_n^*$ ne sont pas témoins pour n. Le test se déroule comme suit :

1. Éliminer le cas où n est pair
2. Écrire $n-1$ sous la forme $n-1 = 2^r \times u$ avec u impair
3. Choisir aléatoirement $a \in [1, n-1]$
4. Si $\text{pgcd}(a, n) \neq 1$ alors n est composé
5. Si a est un témoin alors n est composé, sinon le test est concluant

La correction de ce test repose sur le Théorème 2 et la Proposition 3.

Q7 Écrire une fonction `find_ru(n)` qui pour un entier n pair, renvoie un couple (r, u) tel que $n = 2^r \times u$ avec u impair.

Q8 Écrire une fonction réalisant le test de Rabin. Il est fortement conseillé de décomposer le test en sous-fonctions afin de faciliter l'implémentation. Par exemple, écrire une fonction `temoin_rabin(a,n)` qui pour un a et un n donnés, renvoie vrai si a est un témoin de Rabin que n est composé, faux sinon. Ensuite, créer une fonction `test_rabin(n,t)` où n est l'entier à tester et t le nombre de tests à effectuer (i.e. le nombre d'appels à la fonction ci-avant avec un a tiré aléatoirement). Il n'est pas

2. Ce générateur est à proscrire dans le cadre d'applications cryptographiques sérieuses, on s'en contentera néanmoins ici.

3. On rappelle que $Z_n^* = \{a \in [1, n-1] \mid \text{pgcd}(a, n) = 1\}$

interdit de reprendre les fonctions des questions précédentes.

3. Système cryptographique RSA

Nous avons vu en cours et en TD le protocole RSA. Nous allons maintenant l'implémenter pour se convaincre que sa force ne repose pas sur sa difficulté de mise en oeuvre, mais sur les fondements mathématiques sous-jacents.

Q9 Écrire une première fonction `gen_rsa(n)` qui génère une paire de clés RSA. La fonction prendra en paramètre la taille (en nombre de bits) souhaitée pour le "module" N .

Q10 En réutilisant votre fonction d'exponentiation modulaire, écrire les fonctions `enc_rsa(pk,m)` et `dec_rsa(sk,c)` de chiffrement et de déchiffrement. On souhaiterait chiffrer des messages et non des nombres. Pour cela, on va considérer un caractère comme un chiffre en base 256 (taille de la table ASCII) qui peut être obtenu par une simple conversion Python.

Q11 1-Créer une fonction qui, à une chaîne de caractères, associe un codage de cette chaîne en entier. Créer de même la fonction réciproque.

Aide à la programmation : En Python, `ord('a')` renvoie la valeur ASCII du caractère 'a' (qui vaut 97). Inversement, `chr(97)` renverra 'a'.

2-Ecrire maintenant des fonctions `Enc_rsa(pk,m)` (resp. `Dec_rsa(pk,c)`) permettant de chiffrer (resp. déchiffrer) avec RSA des messages m (resp. c) *de type chaines de caractères* (resp. *de type entier*).

Rappel : IL EST GRANDEMENT CONSEILLE DE FAIRE DES TESTS UNITAIRES DE TOUTES VOS FONCTIONS.

Q12 À l'aide de votre implémentation, générer des clés dont le module est de taille 512 bits et chiffrer le message « ceci est le message de la question 12 ». Pour permettre son déchiffrement, vous fournirez le message chiffré ainsi que la clé secrète.

4. Quelques attaques élémentaires

Dans les questions qui suivent, le chiffrement RSA utilisé est celui que vous avez programmé dans les sections précédentes.

Q13 Déchiffrer $c = 203167233604391$ sachant que $pk = (e, N) = (105153818938879, 204274274459681)$

Les données des questions suivantes (messages, crytogrammes et clés) sont un peu longues et se trouvent à la fin du fichier `rsa_start.py`

Q14 Retrouver le message $m1$ qui a été chiffré en $c1 = \texttt{Enc\_rsa}(pk1, m1)$. ainsi que le message $m2$ qui a été chiffré en $c2 = \texttt{Enc\_rsa}(pk2, m2)$. Vous connaissez $pk1, c1, pk2, c2$ et vous remarquez que les modules $N1, N2$ ont un facteur commun.

Q15 Le même message m a été chiffré trois fois avec trois clés différentes $pk3, pk4, pk5$. Retrouver m à partir de $c3, c4, c5$.

Q16 Retrouver le message $m6$, qui a été chiffré avec la clé $pk6$. Vous connaissez aussi le couple $(pk7, sk7)$. Aide : chercher une racine carré non triviale de 1 (mod $N6$).

On pourra utiliser le

Théorème 4 :Soit N un entier composé impair. Soit a un élément aléatoire de Z_N^*, et soit $w = 2^r \cdot u$,

u impair tel que $a^w \equiv 1 \pmod{N}$. La séquence

$$a^u, a^{2 \cdot u}, a^{2^2 \cdot u}, ..., a^{2^i \cdot u}, ..., a^{2^r \cdot u}$$

contient avec probabilité $\geq 1/2$ une racine carrée non triviale (c'est-à-dire non congrue à 1 ou -1 (mod N)) de 1 (mod N).