

PROJET RÉSEAUX ET SYSTÈMES, RSA . 2020-2021

.....



FACULTÉ DES SCIENCES ET TECHNIQUES
MASTER 1 - MATHS. CRYPTIS

Protocole de communication basé TCP sécurisé par RSA

A l'attention de :
M. BROS Maxime

Rédigé par :
PIARD A.
JACQUET R.

Table des matières

| | | |
|----------|--|----------|
| 1 | Fonctions et problèmes rencontrés | 3 |
| 1.1 | La fonction creationNombreTaille(n) | 3 |
| 1.2 | La fonction premier(n) | 3 |
| 1.3 | Les fonctions egcd(a,b) et modinv(a,m) | 4 |
| 1.4 | La fonction lpowmod(x,y,n) | 4 |
| 1.5 | La fonction chiffrementRSA(plaintext) | 4 |
| 1.6 | La fonction dechiffrementRSA(ciphertext) | 4 |
| 2 | Client et serveur | 5 |
| 2.1 | Le serveur | 5 |
| 2.2 | Le client | 5 |
| 3 | Jeu d'essais | 6 |

Introduction

Le cryptosystème RSA doit son nom aux initiales de ses 3 créateurs, à savoir Rivest (Ron), Shamir (Adi) et Adleman (Leonard). C'est un algorithme de cryptographie asymétrique inventé en 1977. Ce cryptosystème date par conséquent de plus de 40 ans mais est encore utilisé dans certains cas.

A l'époque il était très efficace car les ordinateurs étaient bien moins puissants qu'aujourd'hui, donc des tailles de clés de 256 bits étaient suffisantes pour chiffrer. Aujourd'hui, des tailles de clés de 1024 bits sont nécessaires de manière à ce que le cassage des clefs pour assurer un déchiffrement soit très coûteux, principalement en temps. D'autres méthodes de chiffrement comme l'AES, Advanced Encryption Standard, sont bien plus efficaces aujourd'hui. Mais l'AES est un chiffrement symétrique.

Il est donc fréquent d'utiliser RSA pour échanger des clefs de chiffrement symétriques (pour AES par exemple) de manière sécurisée, puis de s'échanger des informations à travers ce chiffrement symétrique.

1 Fonctions et problèmes rencontrés

1.1 La fonction `creationNombreTaille(n)`

Le paramètre n est un entier en base 10 qui indiquera que le nombre créé doit être composé de n chiffres.

Etant donné que nous cherchons des nombres premiers, dans cette fonction seuls des nombres impairs sont créés. C'est la fonction suivante qui déterminera si un nombre est premier ou non.

1.2 La fonction `premier(n)`

Cette fonction, grâce à la commande "openssl prime" va nous dire si le paramètre n , qui est un nombre, est premier ou non. Si ce nombre n'est pas premier, le but va être de le modifier tout en utilisant le plus possible les chiffres qui le composent. Comme expliqué dans le sujet.

C'est une expression régulière qui va récupérer l'information de si le nombre est ou non premier. Cette expression régulière va rechercher "is not prime" dans la sortie de la commande `openssl prime n`. Ici, rechercher "not" est suffisant. Si l'expression régulière ne match pas, cela veut dire que le nombre est premier donc la fonction va renvoyer ce nombre. Si l'expression régulière match, cela veut dire que le nombre n'est pas premier et qu'il faut le changer puis tester sa primalité.

A ce moment là on le modifie en gardant le plus possible sa forme précédente. On avait ici soulevé un problème.

Problème : Si le nombre testé, qui est non premier, est par exemple de cette forme : 50421, c'est à dire que son deuxième chiffre en partant de la gauche, est un 0, alors suite à la modification de ce nombre, on obtiendra 0421 X , où $X \in \{1, 3, 7, 9\}$. C'est à dire 421 X . Or, ce nombre est composé de 4 chiffres. Dans le cas général, de $n - 1$ chiffres. Donc il n'est plus de la taille souhaitée.

Solution : On a rajouté une condition lors de la modification de n . Si le deuxième chiffre est un 0, on en choisit au hasard un nouveau, compris entre 1 et 9. Ainsi, le nombre voulu sera composé de n chiffres. On test alors la primalité de ce nouveau nombre, de la même manière que la première fois et on répète cette opération tant qu'on ne tombe pas sur un nombre premier.

De plus, lors de la modification du nombre, s'il n'est pas premier, il faut changer l'avant dernier chiffre car celui-ci sera égal au dernier chiffre du précédent nombre. Il sera donc égal soit à 1, soit à 3, soit à 7 ou soit à 9. Or, Les chiffres qui composent le nombre, hormis le premier et le dernier chiffre, doivent être compris entre 0 et 9. Donc on remplacera celui-ci par un chiffre compris entre 0 et 9.

1.3 Les fonctions `egcd(a,b)` et `modinv(a,m)`

Ces deux fonctions sont liées. La fonction `egcd` renvoie le pgcd des nombres a et b ainsi que les coefficients de Bézout de a et b . Cette fonction correspond donc à l'algorithme d'Euclide étendu.

Quant à elle, la fonction `modinv(a,m)`, renommée en `inverseModulo(a,m)` dans notre programme, fait appel à la fonction `egcd` et renvoie l'inverse modulaire d'un nombre. On utilisera donc ces fonctions pour calculer $d = e^{-1}(\text{mod}(\phi(n)))$ avec $\phi(n) = (p-1) \times (q-1)$ et $n = p \times q$.

1.4 La fonction `lpowmod(x,y,n)`

Cette fonction a été renommée `powmod(x,y,n)` dans notre programme.

Comme indiqué dans le sujet, cette fonction permet de calculer $x^y \text{ mod}(n)$ bien plus rapidement qu'avec l'opération d'exponentiation `**` de python, suivi de la réduction modulo n du résultat obtenu.

Cette fonction sera utilisée pour chiffrer et déchiffrer par RSA. En effet, l'opération de chiffrement (respectivement de déchiffrement) pour RSA est $m^e \text{ mod}(n)$ (respectivement $c^d \text{ mod}(n)$).

Etant sujets à pratiquer RSA avec de grands nombres premiers, l'implémentation de cette fonction est primordiale pour optimiser la vitesse des calculs.

1.5 La fonction `chiffrementRSA(plaintext)`

1.6 La fonction `dechiffrementRSA(ciphertext)`

2 Client et serveur

2.1 Le serveur

Le serveur, ouvert dans un terminal, attend la connexion d'un client. Le serveur étant le fichier **serveur.py** à exécuter dans le terminal avec la commande `python3 serveur.py`. C'est le serveur qui chiffrera le message que lui aura envoyé le client. Le chiffrement se fera par blocs de 3 caractères utf-8. Pour commencer le chiffrement, on traduit chaque caractère utf-8 du message reçu en son ordre dans la table utf-8 avec la commande `ord(X)` si X est le caractère dont on souhaite l'ordre. Chaque ordre sera stocké sous forme d'une cellule de liste. Comme on voudra coder par blocs de 3, il faut s'assurer que la liste est de la taille un multiple de 3 (en nombre de cellules). Si la liste n'est pas de taille 3, il suffit d'y ajouter l'ordre d'un élément, qu'on aura au préalable choisi, ici 'Z', jusqu'à ce que la taille de la liste soit un multiple de 3. Maintenant, tous les caractères utf-8 n'ayant pas la même taille en base 10 (et peu importe la base), nous avons choisi de faire du padding à chaque caractère pour que chacun d'eux soient composés de 6 chiffres. On concatènera le caractère "4" à gauche de la chaîne de caractères représentant l'entier, jusqu'à ce que cette chaîne soit de taille 6. Il ne reste plus qu'à concaténer les 3 premiers éléments de la liste ensemble, puis les trois suivants, *et cetera*, et de stocker ces différentes chaînes dans une nouvelle liste. Ensuite, on chiffre les éléments de cette nouvelle liste un par un et on crée une nouvelle chaîne de caractère à partir de la concaténation des blocs chiffrés suivi du caractère "—" pour que les blocs soient visibles à l'œil nu et que ça aère légèrement l'affichage dans le terminal. Il ne reste plus qu'à envoyer cette dernière chaîne de caractère au client sous forme de bytes.

2.2 Le client

Le client, ouvert dans un terminal, se connecte au serveur qui est ouvert au préalable et attend une connexion. Le client étant le fichier **client.py** à exécuter dans le terminal avec la commande `python3 client.py`. La partie déchiffrement se fait chez le client. Celui-ci intercepte le message chiffré envoyé par le serveur. On crée une liste remplie cellule par cellule des blocs du chiffré, séparés par le caractère "—" grâce à la fonction `split`. Il suffit alors de déchiffrer les éléments de la liste un par un et d'ajouter chacun des blocs déchiffrés à une liste initialement vide. Il faut ensuite diviser chaque chaîne de caractère de la liste en 3 chaînes de caractères de taille 6 et de supprimer les caractères "4" que l'on avait concaténé pour obtenir des nombres de taille similaire. On retrouve alors une liste de chaîne de caractères correspond aux ordres, en UTF-8, des caractères du message clair. Il suffit donc de les récupérer par la fonction `chr()` et de les concaténer pour retrouver le message initial, sans oublier d'enlever les caractères 'Z' du padding. (Il est possible de perdre de l'information avec un tel padding en supprimant un 'Z' qui faisait parti du clair, et non du padding.

3 Jeu d'essais