Programmation fonctionnelle – TP

Cryptographie RSA

Décembre 2018

L'objectif de ce TP est de permettre le chiffrement et le déchiffrement asymétriques de messages via l'algorithme RSA [1].

Les notes de bas de page donnent des indices, qu'on n'est pas obligé de lire.

On pourra se référer à Hoogle [2] pour la documentation sur l'API standard.

Pour faciliter la mise au point, on pourra enfin utiliser la fonction `trace` pour afficher certaines informations lors de l'évaluation des fonctions.

1 Quelques mots sur RSA

La cryptographie consiste à rendre une *information* inintelligible pour qui ne connaît pas un certain *secret*. Dans le cas de RSA, le secret est la connaissance des facteurs premiers d'un entier très grand.

En effet, pour tout entier n il existe une unique liste d'entiers premiers $(p_i)_{i\in[1..k]}$ inférieurs à n et une unique liste d'exposants entiers $(m_i)_{i\in[1..k]}$ tels que $\prod_{i\in[1..k]} p_i^{m_i} = n$.

Cependant, trouver la liste $(p_i)_i$ à partir de n est un problème *très difficile*, dans l'état actuel des connaissances mathématiques.

Dans la cryptographie *symétrique* on utilise le même secret pour *chiffrer* et *déchiffrer* l'information à cacher. On appelle ce secret la *clé*. C'est une méthode rapide et efficace mais il faut par ailleurs un moyen sûr d'échanger la clé.

Dans la cryptographie asymétrique, la clé est en deux parties : une partie *publique* accessible à tout le monde et une partie *privée* détenue par une seule personne. Ce qui est chiffré par la clé publique ne peut être déchiffré que par la clé privée et inversement [3].

Soient deux entiers premiers p et q distincts et soit $n = pq$. Les nombres p et q forment la base de notre secret.

On choisit un entier e n'ayant aucun facteur en commun avec la fonction indicatrice d'Euler [4] $\varphi(n) = (p-1)(q-1)$ (ils sont premiers entre eux) compris strictement entre 1 et $\varphi(n)$. Le couple (e, n) est notre clé publique à laquelle tout le monde peut avoir accès.

Notre clé privée est le couple (d, n) tel que $de = 1 \pmod{\varphi(n)}$, mod étant l'opérateur *modulo*. Le calcul de d nécessite la connaissance de $\varphi(n)$ donc de notre secret p et q.

On peut montrer que pour tout message entier $m < n$, $(m^d)^e = (m^e)^d = m \pmod{n}$.

1. `https://fr.wikipedia.org/wiki/Chiffrement_RSA`.
2. `https://www.haskell.org/hoogle/`.
3. Cet inverse constitue le principe de la signature électronique.
4. De nos jours, on utilise plutôt la fonction indicatrice de Carmichael qui vaut ici $\mathsf{ppcm}(p-1, q-1)$.

Pour chiffrer le message m, on calcule donc $c = m^e \pmod{n}$. Et pour déchiffrer le message chiffré c, on calcule $c^d \pmod{n}$ ce qui nous redonne m.

2 Messages

Q1. Pour représenter notre information à chiffrer, on utilise une liste d'entiers. Définir le nouveau type algébrique `Message` correspondant [5] ;

Q2. On veut pouvoir passer d'une chaîne de caractères à un `Message` et inversement. Définir les fonctions `stringToMessage` et `messageToString` réalisant ces conversions [6] [7].

3 Blocs

On ne va pas chiffrer chaque caractère séparément mais plutôt les regrouper sous la forme de blocs. Cela nécessite que la longueur du message soit divisible par la longueur d'un bloc. Comme ce n'est pas le cas en général, on applique un prétraitement appelé *padding* qui consiste à ajouter des éléments à notre message pour qu'il ait la bonne taille. On utilise ici la méthode PKCS7 décrite dans la RFC 5652 [8]. L'idée est de rajouter N entiers valant N au message où N est le nombre d'entiers manquant pour avoir une longeur divisible par la longueur du bloc. Si N vaut 0 on ajoute un bloc complet plutôt et donc on prend à la place N valant la longueur du bloc.

Q3. Écrire une fonction `pad` qui étant donnée une longueur de bloc [9] et un message, ajuste la longueur de ce message en utilisant la méthode PKCS7 [10] ;

Q4. Écrire une fonction `unpad` qui réalise l'opération inverse de `pad` pour retrouver le message original.

On groupe ensuite les entiers représentant les caractères par bloc de la longueur `bsize`. On suppose que ces entiers sont tous inférieurs à 256. Un bloc sera obtenu en considérant que `bsize` tels entiers représentent l'écriture en base 256 d'un grand entier : le bloc est justement ce grand entier. Par exemple, le bloc de taille 4 correspondant à la liste `[128,54,33,99]` est l'entier $2151031139 = 99 * 256^0 + 33 * 256^1 + 54 * 256^2 + 128 * 256^3$.

Q5. Écrire une fonction `groupBytes` qui étant donnée une liste d'entiers supposée de longueur égale à la longueur de bloc, calcule l'entier représentant le bloc comme expliqué ci-dessus [11] ;

Q6. Écrire une fonction `ungroupBytes` qui étant donnée la longueur de bloc, et un entier représentant un bloc, retrouve la liste des entiers constituant le bloc [12]

5. Indice : les entiers manipulés étant souvent très grands, on préférera `Integer` à `Int`.
6. Indice : on pourra utiliser les fonctions `ord` et `chr` de `Data.Char`.
7. Indice : pour transformer un `Int` en `Integer` (et inversement), on pourra utiliser la fonction polymorphique `fromIntegral`.
8. `https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7`.
9. Indice : pour la longueur de bloc, le type `Int` sera le plus pratique.
10. Indice : la fonction `replicate` pourra être utile.
11. Indice : on pourra utiliser la fonction `foldl'` de `Data.List`
12. Indice : on pourra par exemple utiliser la fonction `iterate`.

Q7. Écrire une fonction `groupN` qui étant donnés une longueur de bloc `bsize` et une liste d'entiers de taille supposée divisible par `bsize`, donne une liste de listes d'entiers, chacune des sous-listes étant de taille `bsize` [13] ;

Q8. Écrire une fonction `makeBlocks` qui, étant donnés une longueur de bloc `bsize` et un message de longueur supposée divisible par `bsize`, produit le message `bsize` fois plus petit contenant les blocs formés à partir des entiers du message d'entrée groupés par sous-listes de `bsize` éléments. On rappelle qu'un bloc est un entier calculé à partir de `bsize` entiers entre 0 et 255 ;

Q9. Écrire une fonction `splitBlocks` qui, étant donnés une longueur de bloc et un message contenant des blocs, retrouve le message dans lequel les blocs ont été divisés pour retrouver les entiers représentant directement les caractères [14].

4 Chiffrement et déchiffrement

On va maintenant procéder au chiffrement et au déchiffrement proprement dits. Pour cela nous aurons besoin de tester si un nombre est premier, pour pouvoir choisir des nombres premiers.

Q10. Écrire une fonction `prime` qui teste si un entier `n` donné est premier. On ne considérera comme diviseurs potentiels que 2, 3, et les entiers inférieurs ou égaux à la racine carrée de `n` et qui s'écrivent $6k-1$ ou $6k+1$, avec $k \geq 1$ [15] [16].

Q11. Écrire une fonction `choosePrime` qui, étant donné un entier `b` donne le plus petit entier premier supérieur ou égal à `b` [17] [18] ;

Une fois choisis p et q, il reste à choisir e et d. Pour e, on choisit 65537, ce qui constitue un choix classique. Pour déterminer d, il nous faut donc calculer l'inverse de e modulo $(p-1)(q-1)$. On calcule cette inverse en utilisant l'algorithme d'Euclide étendu [19].

Q12. Écrire une fonction `euclid` qui pour deux entiers `a` et `b` calcule le PGCD g de `a` et `b` ainsi que les coefficients u et v tels que $\texttt{a}u+\texttt{b}v = g$ [20] ;

Q13. Déduire de la Q12, une fonction `modInv` qui, pour `e` et `n` premiers entre eux cacule l'inverse de `e` modulo `n`.

Pour chiffrer et déchiffrer on utilise l'exponentiation modulaire. On adapte l'algorithme de l'exponentiation rapide [21] en utilisant le fait que $(ab) \bmod n = ((a \bmod n)(b \bmod n)) \bmod n$.

Q14. Écrire une fonction `modExp` qui réalise l'exponentation modulaire en utilisant le principe décrit ci-dessus ;

13. Indice : on pourra utiliser la fonction `splitAt`.
14. Indice : on pourra utiliser la fonction `concat`.
15. Explication : il est clair que tous les entiers naturels s'écrivent $6k+i$, avec $k \geq 0$ et $i \in [0..5]$, par une simple division euclidienne par 6. Remarquons ensuite que pour tout k, 2 divise $6k$, $6k+2$, et $6k+4$ et que 3 divise $6k+3$. Donc, 1 n'étant pas premier, seuls 5, $6k+1$ et $6k+5$ pour $k \geq 1$ peuvent être premiers, c'est-à-dire seuls $6k-1$ et $6k+1$ pour $k \geq 1$.
16. Indice : on pourra utiliser la fonction `any`.
17. Indice : on pourra utiliser la fonction `dropWhile`.
18. En pratique on effectue évidemment un choix aléatoire, plutôt que de prendre le premier. Ici, on se contentera de cette petite simplification.
19. `https://fr.wikipedia.org/wiki/Algorithme_d'Euclide_étendu`
20. Indice : on pourra écrire une fonction auxiliaire `euclid'` récursive terminale.
21. Rappel : pour $n \geq 1$, $x^n = (x^2)^{\frac{n}{2}}$ si n est pair et $x^n = x * x^{n-1}$ sinon.

Q15. Écrire une fonction `encrypt` qui à partir d'une clé (publique) `(e, n)`, d'une longueur de bloc, et d'une chaîne de caractères renvoit un message chiffré par bloc, dans lequel chaque bloc a été chiffré par RSA ;

Q16. Écrire une fonction `decrypt` qui à partir d'une clé (privée) `(d, n)`, d'une longueur de bloc, et d'un message chiffré par bloc par RSA renvoit la chaîne de caractères déchiffrée.

Q17. Écrire une fonction principale permettant de tester tout le processus. Chiffrer, déchiffrer quelques messages. Méditer un instant sur l'importance du respect de la vie privée et du secret des correspondances.