Cryptographie par RSA

Étienne Miquey
etienne.miquey@ens-lyon.fr

Ce sujet de TP est un éhonté repiquage de celui créé en mon temps par Lionel Rieg, à qui il me faut donc rendre hommage ici.

1 Préambule

Nous allons y étudier le chiffrement RSA qui est le plus connu des crypto-systèmes, sur lequel repose bon nombre de système de chiffrement dans la vrai vie. Ce système est dit à clé publique ou asymétrique : on donne une clé publique, avec laquelle tout le monde peut chiffrer des messages, et seul le déchiffreur possède la clé publique qui lui permet de décrypter des messages. La première étape est la création de la paire de clé par le déchiffreur, de la façon suivante :

1. Trouver aléatoirement deux grands entiers premiers p et q
2. Calculer $n = pq$
3. Calculer $\varphi(n) = (p-1)(q-1)$
4. Trouver e (pour *encryption exponent*) premier avec $\varphi(n)$
5. Calculer d (pour *decryption exponent*) l'inverse de e modulo $\varphi(n)$

Les clés sont alors :

- (e, n) la clé publique
- (d, n) la clé privée

Comme vous le savez, $(\mathbb{Z}/n\mathbb{Z})^*$ est de cardinal $\varphi(n)$, et de plus si m est premier avec n, alors $(m^e)^d \equiv m^{ed} \equiv m^{k\varphi(n)+1} \equiv m \pmod{n}$. Le chiffrement se fait donc en élevant l'entier m que l'on souhaite envoyer à la puissance e modulo n. Le déchiffrement se fait quant à lui en élevant le cryptogramme à la puissance d modulo n. N'importe qui détenant la clé publique (e, n) est en mesure de chiffrer un message, mais seul le possesseur de la clé privée (d, n) pourra déchiffrer. La sûreté du système vient de la difficulté (et même concrètement l'impossibilité actuelle) de retrouver m à partir de $m^d \pmod{n}$ de façon efficace.

2 Outils arithmétiques

Comme vous l'aurez remarqué, on va avoir besoin d'effectuer un certain nombre d'opérations arithmétiques, notamment un calcul du pgcd, un test de primalité, une inversion modulaire et une exponentiation modulaire. Je vous recommande une fois de plus vivement de les tester une par une au fur et à mesure que vous les écrirez. Pour que notre implémentation soit quelque peu réaliste (et aussi parce que cela vous fera du bien), nous n'utiliserons pas le type `int` qui ne permet pas de travailler avec de grands entiers. Nous nous servirons de la bibliothèque `num` de calcul en précision arbitraire. Il faut pour cela ouvrir le module avec

```
#open ''num'';;
```

Les opérations élémentaires `+`, `-`, `*`, `/`, `mod`, `quo` sont redéfinis pour le type `num` de manière agréable à utiliser (i.e. infixe) : `+/`, `-/`, `*/`, `//`, `modN`,`quoN`. Vous pourrez avoir besoin de quelques autres primitives, à savoir `square_num`, `sqrt_num`, `string_of_num`, `num_of_string`, `num_of_int` et `int_of_num`. Comme vous pouvez l'imaginer, cette dernière fonction retourne un résultat modulo 2^{31} car on perd de la précision (selon la machine, le modulo peut varier).

Question 0. Définir les entiers 0, 1 et 2 de type `num`.

Question 1. Écrire une fonction qui calcule le pgcd de deux entiers de type `num`.

```
pgcd : num -> num -> num
```

Question 2. Écrire un test de primalité naïf qui, sur l'entrée n, calcule le pgcd de n et k pour $1 < k \leq \sqrt{n}$ et vérifie qu'il vaut bien à chaque fois 1.

```
is_prime : num -> bool
```

Question 3. Écrire une fonction qui génère un nombre premier aléatoire en prenant un nombre aléatoire (on se servira pour cela de la fonction `random_num` qui vous est fournie) et en testant sa primalité. Afin de permettre un contrôle plus fin, la fonction prendra en arguments deux entiers `inf` et `range` qui seront respectivement une borne inférieure sur la valeur du nombre et la longueur de l'intervalle dans lequel on autorise la recherche.

```
generate_prime : num -> num -> num
```

Question 4. Écrire une fonction `prime_with` telle que `prime_with n` rendra un nombre e choisi aléatoirement entre 2 et n et qui soit premier avec n.

```
prime_with : num -> num
```

Question 5. Écrire l'algorithme d'Euclide étendu pour déterminer un couple de Bézout qui servira pour l'inversion modulaire.

```
extended_pgcd : num -> num -> num * num
```

Question 6. Écrire enfin la dernière opération arithmétique qui manque, l'exponentiation modulaire rapide. On veillera à calculer les résidus modulo n après chaque multiplication, sans quoi les résultats risquent de devenir très vite énormes.

```
modular_exp : num -> num -> num -> num
```

3 Implémentation du protocole

On va commencer par générer une paire de clés. Pour faire la distinction entre les clés privées et publiques, nos clés aurons le type suivant :

```
type key = Private of num * num | Public of num * num;;
```

Question 7. Écrire une fonction qui prend en paramètre deux entiers `inf` et `range` comme pour la question 3 et fabrique une paire de clés à l'aide de la méthode décrite en préambule.

```
num -> num -> key * key
```

Question 8. À l'aide de votre fonction d'exponentiation, écrire les fonctions de chiffrement et de déchiffrement d'un entier. On veillera à retourner une erreur dans le cas où le mauvais type de clé est donné.

```
encode : key -> num -> num
```

```
decode : key -> num -> num
```

On souhaite plutôt chiffrer des messages que des nombres, de fait nous allons avoir besoin d'une fonction de conversion de l'un vers l'autre. De plus, pour être sûr que notre résultat est premier avec n, on pourra ajouter un caractère arbitraire à la fin de la chaine dont on ajustera la valeur.

Question 9. Écrire donc une fonction qui transforme une chaîne de caractères en un entieri, ainsi que son inverse à gauche. Pour cela, on verra un caractère comme un chiffre dans une base de numération à 256 chiffres dont le numéro peut être obtenu à l'aide de la fonction `int_of_char`.

```
string_to_num : key -> string -> num
```

```
num_to_string : num -> string
```

Vous disposez désormais de tout ce qu'il faut pour crypter/décrypter des messages. Testez donc tout ce que qui a été fait jusque là en chiffrant un message. Vous pouvez même vous amuser à en échanger via le dossier de partage.

Les deux parties suivantes présentent deux directions dans lesquelles on peut prolonger ce TP et elles sont totalement indépendantes. Vous pouvez donc choisir celle qui vous plaît le plus.

4 Amélioration du test de primalité

Le test de primalité qu'on utilise jusqu'ici est évidemment extrêmement coûteux (sa complexité est exponentielle en $log(n)$) et donc inutilisable en pratique. On lui préfère donc des tests probabilistes dont le taux d'erreur peut être rendu si faible qu'il est illusoire de vouloir faire mieux, l'ordinateur ayant davantage de chances d'imploser durant le calcul (ou plus probablement de faire une erreur matérielle).

4.1 Test de Fermat

Le premier test possible (qui est utilisé en pratique par le logiciel PGP par exemple) est basé sur le petit théorème de Fermat que vous connaissez bien :

$$p \text{ premier } \Rightarrow \forall a \in [\![1, p-1]\!], a^{p-1} \equiv 1 \bmod p$$

L'idée est de dire que si un entier n satisfait cette propriété pour de nombreux a, il a de fortes chances d'être premier. Pour tester cela, on choisit donc a au hasard, on l'élève à la puissance $n-1$ et on vérifie que ce résultat

est bien congru à 1 modulo n. Il est facile [1] de montrer que le taux d'erreur, en supposant l'existence d'un témoin, est majoré par $\frac{1}{2}$, et l'on répète alors ce test autant de fois que nécessaire pour atteindre la précision voulue.

Question 10. Écrire un test de primalité à l'aide du test de Fermat en utilisant l'exponentiation modulaire rapide. Votre fonction prendra en arguments le nombre [2] maximum de tests à réaliser et l'entier à tester.

```
fermat : int -> num -> bool
```

Cette méthode présente toutefois un inconvénient, puisqu'il existe des nombres composés, dit de Carmichael, pour lesquels tous les entiers strictement plus petits passent le test de Fermat, *i.e.* l'algorithme se trompe nécéssairement.

4.2 Test de Rabin-Miller

On veut modifier le test de Fermat pour ne pas se laisser abuser par les nombres de Carmichael. Pour cela, on rajoute une technique de détermination de composition. L'idée clé est que si n est premier impair, pour tout $a \in (\mathbb{Z}/n\mathbb{Z})*$ premier avec n, on a nécessairement [3]

$$a^d \equiv 1 \mod n \text{ ou } a^{2^r \cdot d} \equiv -1 \pmod{n} \text{ pour un certain } 0 \leq r \leq s-1$$

L'algorithme est donc le suivant :

1. On élimine le cas où n est pair
2. On écrit $n - 1 = 2^S \cdot t$ avec t impair
3. On choisit aléatoirement a dans $[\![2, n-2]\!]$
4. Si $a^t \equiv 1 \pmod{n}$, on s'arrête
5. On calcule $(a^t)^{2^k}$ pour $k \in [\![1, s-1]\!]$
6. Si aucun des $(a^t)^{2^k}$ ne vaut -1, alors n est composé

Le taux d'erreur d'une itération est ici majorée par $\frac{3}{4}$, ce qui nous garantit encore une fois rapidement une bonne précision.

Question 11. Écrire une fonction qui décompose un entier n en un couple (s, t) tel que $n = 2^s \cdot t$ avec t impair.

```
decompose : num -> num * num
```

Question 12. Implémenter le test de primalité de Rabin-Miller. Je vous recommande vivement de décomposer votre code en petites sous-fonctions.

```
rabin_miller : int -> num -> bool
```

Question 13. Tester le gain de temps que procure cet algorithme comparé à l'algorithme naïf.

5 Attaques de RSA

Casser un crypto-système revient à être capable de retrouver n'importe quel message à partir de son crytogramme. Il existe de nombreuses techniques d'attaque de RSA mais aucune ne représente une menace sérieuse lorsqu'il est bien utilisé. Je laisse le soin au curieux de chercher sur la toile les dernières techniques à la pointe.

On ne va ici étudier que deux algorithmes simples pour la plus élémentaire des attaques, la force brute. Elle est très coûteuse en terme de ressources mais ne repose sur aucun cas particulier : l'idée est simplement de factoriser n. En effet, une fois que l'on possède p et q, on peut calculer $\varphi(n)$ et déduire d de e par inversion modulaire.

5.1 Algorithme naïf

Question 14. Écrire un algorithme de factorisation naïf basé sur le même principe que le test de primalité de la question 2.

```
factorize : num -> num list
```

Cet algorithme trouve à coup sûr un facteur s'il en existe un mais il requiert n calculs de pgcd. Il a donc un coût exponentiel en la taille de n et ne peut pas être utilisé en pratique car n est bien trop grand.

1. *En effet, si n est composé, en supposant qu'il existe un témoin a tel que* $a^{n-1} \not\equiv 1 \pmod{n}$, *si* $a_1, \ldots, a_s$ *sont tels que* $\forall i, a_i^{n-1} \equiv 1 \pmod{n}$, *alors* $\forall i, (a \cdot a_i)^{n-1} \equiv a^{n-1} \not\equiv 1 \pmod{n}$, *donc* $a \cdot a_i$ *est aussi un témoin.*

2. *Un entier suffira, puisque* $2^{-2^{31}}$ *devrait être un taux d'erreur tout à fait acceptable même pour les plus pointilleux d'entre vous*

3. *Je vous laisse démontrer ça, on part du théorème de Fermat, et on prends les racines carrées, en utilisant ici le fait que* $\mathbb{Z}/n\mathbb{Z}$ *est un corps où les seules racines carrées de 1 sont 1 et -1*

5.2 Algorithme ρ de Pollard

On peut faire mieux avec l'algorithme ρ de Pollard, algorithme probabiliste basé sur deux faits : le paradoxe des anniversaires et l'algorithme du lièvre et de la tortue. Cet algorithme a été conçu en 1975 par John Pollard, dans le but spécifique d'obtenir des petits facteurs de nombre non premiers.

Commençons par examiner l'algorithme du lièvre et de la tortue. Il est dû à Floyd et permet de détecter les cycles dans une suite du type $u_{n+1} = f(u_n)$. Cette restriction permet d'assurer que si une même valeur apparaît deux fois dans la suite, alors il y a un cycle. Le fonctionnement de l'algorithme est très simple comme nous allons le voir. Le lièvre et la tortue partent initialement d'un même point. À chaque étape, la tortue avance de 1 dans la suite et le lièvre de 2. Si le lièvre croise la tortue (*i.e.* ils ont la même valeur), alors il y a un cycle. Réciproquement, s'il y a un cycle dans la suite, alors la tortue va y entrer au bout d'un certain temps. Une fois tous les deux dans le cycle, le lièvre rattrape la tortue de 1 pas à chaque étape donc il vont finir par se rencontrer. Sur l'exemple ci-contre avec la suite $2, 0, 6, 3, 1, 6, 3, 1, 6, \ldots$, on voit que le lièvre rattrape la tortue au bout de 3 étapes.

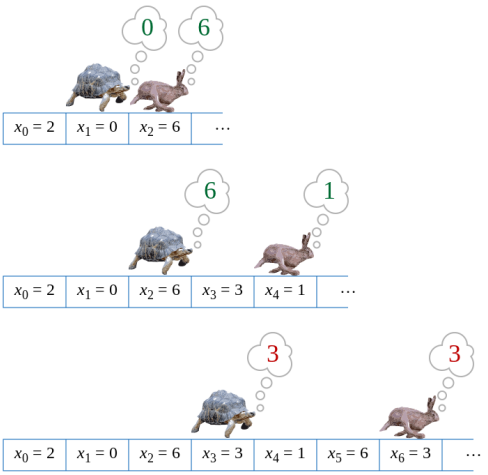

FIGURE 1 – Crédit image : David Eppstein, Wikimedia

Question 15. Implémenter l'algorithme du lièvre et de la tortue. En plus de la fonction génératrice de la suite, il prendra en argument une graine pour amorcer la suite et le nombre maximum d'étapes à réaliser, il sera donc de type `cycle_detect : ( int -> int ) -> int -> int -> bool`

Le paradoxe des anniversaires nous dit [4] que dès lors qu'un groupe contient plus de 23 personnes, la probabilité d'en avoir deux nées le même jour est supérieure à $\frac{1}{2}$. En considérant les dates de naissance (supposées aléatoires) en jours, cela revient à trouver deux entiers congruents modulo 365. De même, trouver deux entiers aléatoires dont la différence soit multiple d'un entier p quelconque avec une probabilité supérieure à $\frac{1}{2}$ ne nécessite qu'environ $\sqrt{2p \ln 2} \approx 1,177\sqrt{p}$ tentatives. L'idée de l'algorithme rho est alors d'utiliser une fonction de $[\![1, n]\!]$ dans $[\![1, n]\!]$ comme générateur d'une suite (u_n) pseudo-aléatoire de nombres dont on espère que les différences seront multiples d'un facteur premier de n. On calcule ensuite le pgcd de n et de la différence des deux entiers u_i et u_{2i}. L'algorithme du lièvre et de la tortue permet de détecter les cycles dans la suite des nombres générés afin de ne pas répéter [5] les mêmes tests. On n'obtiendra cette fois-ci (en cas de réussite) qu'un facteur premier, mais comme n est produit de deux facteurs premiers, cela suffit.

Question 16. En modifiant votre code pour la détection de cycle, implémenter l'algorithme rho de factorisation de Pollard, qui prendra en argument une fonction `f` et l'entier `n` dont on cherche des facteurs.

```
rho : (num-> num) -> num -> num list
```

Question 17. Tenter de décrypter un cryptogramme avec les deux méthodes. Ne prenez pas un n trop grand, et pour l'algorithme de Pollard, utilisez des fonctions de la forme $f(x) = x^2 + c$ en changeant éventuellement la valeur de c en cas d'échec.

4. *Mais vous pouvez aussi faire le calcul pour vérifier, c'est rapide*
5. *Le nom même d'algorithme rho rappelle que la trajectoire du générateur va revenir sur elle-même pour former un cycle, tel un ρ.*