TP Caml 3: Cryptographie par RSA

lionel.rieg@ens-lyon.fr

14 & 21 octobre 2008

1 Principe de RSA

Le chiffrement[1] RSA est le plus connu des crypto-systèmes dit *asymétriques* ou *à clé publique*. Cette appellation vient de l'inégale répartition des informations entre le chiffreur et le déchiffreur : la clé publique est connue de tous tandis que la clé privée n'est connue que d'une personne. Ainsi, si tout le monde peut chiffrer des messages avec la clé publique, seul le possesseur de la clé privée est en mesure de les déchiffrer.

RSA repose en substance sur la difficulté de calculer une racine d-ème dans $(\mathbb{Z}/n\mathbb{Z})^*$. L'étape préliminaire au chiffrement est la création des clés de la façon suivante :

1. trouver deux grands entiers premiers aléatoires p et q
2. calculer $n = pq$
3. calculer $\varphi(n) = (p-1)(q-1)$
4. trouver e (***e**ncryption exponent*) premier avec $\varphi(n)$
5. calculer d (***d**ecryption exponent*) l'inverse de e modulo $\varphi(n)$

Les clés sont alors :
- (e, n) la clé publique
- (d, n) la clé privée

Comme $(\mathbb{Z}/n\mathbb{Z})^*$ est de cardinal $\varphi(n)$, si m est premier avec n, $(m^e)^d = m^{ed} = m^{k\varphi(n)+1} = m \mod n$. Le chiffrement se fait donc en élevant l'entier m qu'on souhaite envoyer à la puissance e modulo n. Le déchiffrement se fait quant à lui en élevant le cryptogramme à la puissance d toujours modulo n. Puisque (e, n) est public, n'importe qui peut chiffrer mais seul le possesseur de la clé privée (d, n) sait déchiffrer. La sûreté du système vient de l'impossibilité actuelle de retrouver m à partir de $m^d \mod n$ de façon efficace.

2 Implémentation de RSA

2.1 Outils arithmétiques

Comme on l'aura remarqué, cette technique utilise de nombreuses opérations arithmétiques, à savoir :
- un générateur aléatoire de nombres
- le calcul du pgcd
- un test de primalité
- l'algorithme d'Euclide étendu pour le calcul de l'inverse modulaire
- l'exponentiation modulaire

Et pour que ce ne soit pas trop simple, on ne peut pas utiliser le type `int` car les entiers que l'on va utiliser seront beaucoup trop grands. On va donc utiliser le type `num` défini dans le module de calcul en précision arbitraire. Je rappelle qu'il faut pour cela ouvrir le module avec un

```
#open "num";;
```

Ainsi, sauf précision contraire, les entiers seront dans tout ce TP de type `num`.

[1] Remarquez que *cryptage* est un anglicisme banni par l'Académie Française

Comme on n'utilise pas l'arithmétique usuelle de Caml Light, les opérations élémentaires +, −, ∗, /, mod, quo sont redéfinis pour le type num de manière agréable à utiliser (*i.e.* infixe) : +/, −/, ∗/, //, modN et quoN. Les autres primitives auxquelles vous aurez droit sont square_num, sqrt_num, string_of_num, num_of_string, num_of_int et int_of_num. Cette dernière fonction retourne un résultat modulo 2^{31} car on perd de la précision (selon la machine, le modulo peut varier).

On ne se préoccupera pas ici du générateur aléatoire qui est un problème hautement non trivial et on utilisera simplement la fonction random_num que je vous fournis. Pour ceux que cela intéresse, ce générateur très basique utilise la méthode du carré médian de von Neumann.

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

► **Question 1** Écrire une fonction qui calcule le pgcd de deux entiers. On peut la faire au choix de façon récursive ou itérative mais elle devra être curryfiée et aura donc le type

```
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 $1 \leqslant k \leqslant \lfloor \sqrt{n} \rfloor$ et vérifie qu'il vaut bien à chaque fois 1.

► **Question 3** Écrire une fonction qui génère un nombre premier aléatoire en prenant un nombre aléatoire 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.

► **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.

► **Question 5** Écrire l'algorithme d'Euclide étendu qui servira pour l'inversion modulaire. Il aura pour type

```
extended_pgcd : num → num → num * num
```

de sorte que extended_pgcd a b renvoie un couple (u, v) de coefficients de Bezout pour a et b, *i.e.* u et v vérifient $au + bv = \text{pgcd}(a, b)$.

► **Question 6** Écrire enfin la dernière opération arithmétique qui manque, l'exponentiation modulaire rapide. Elle sera de type

```
modular_exp : num → num → num → num
```

et on veillera à calculer les résidus modulo n après chaque multiplication, sans quoi les résultats peuvent très vite devenir énormes.

2.2 Implémentation du protocole

On va commencer par générer une paire de clés. Pour faire la distinction entre les clés privées et publiques, on utilise 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 de même signification qu'à la question 3 et fabrique une paire de clés à l'aide de la méthode décrite dans la première partie.

► **Question 8** Écrire enfin 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. Elles seront de type :

```
encode : key → int → int
decode : key → int → int
```

► **Question 9** On souhaite plutôt chiffrer des messages que des nombres, aussi il faut une fonction de conversion de l'un vers l'autre. Écrire donc une fonction qui transforme une chaîne de caractères en un entier. 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. Le type de la fonction de conversion sera :

```
encode_string : string → num
```

Pour être sûr que le résultat soit premier avec n, on pourra rajouter un caractère arbitraire à la fin du message dont on ajustera la valeur.

► **Question 10** Tester tout ce que qui a été écrit jusque là (même si cela devrait déjà avoir été fait par petits bouts au fur et à mesure) en chiffrant un message. On peut même envisager d'en échanger entre les postes par l'intermédiaire du répertoire de partage.

Les deux parties suivantes présentent deux directions dans lesquelles on peut prolonger ce TD et elles sont totalement indépendantes. Choisissez celle qui vous plaît le plus.

3 Un meilleur test de primalité

Le test de primalité qu'on utilise 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).

3.1 Le test de Fermat

La première amélioration est d'utiliser le *petit théorème de Fermat* :

$$p \text{ premier } \Longrightarrow \forall a \in [\![1,\ p-1]\!],\ a^{p-1} \equiv 1 \mod n$$

Si un entier n satisfait cette propriété pour de nombreux a, il y a de bonnes chances que n soit premier. Pour tester cela, on choisit a au hasard entre 1 et $n-1$, on l'élève à la puissance $n-1$ et on vérifie que le résultat est bien congru à 1 modulo n. On répète ce test autant de fois que nécessaire pour atteindre la précision voulue. À supposer que pour tout n composé il existe un nombre a pour lequel $a^{n-1} \not\equiv 1 \mod n$, la probabilité d'erreur est majorée par 1/2 car l'ensemble des nombres qui passent le test forme un sous-groupe de $\mathbb{Z}/n\mathbb{Z}$.

► **Question 11** Écrire un test de primalité à l'aide du test de Fermat en utilisant l'exponentiation modulaire rapide. L'algorithme prendra en paramètre le nombre maximum de tests à réaliser et l'entier à tester.

L'inconvénient de cette méthode est qu'il existe des nombres composés, dit de Carmichael, pour lesquels tous les entiers strictement plus petits passent le test de Fermat. Sur ces entiers, l'algorithme précédent se trompe nécessairement.

3.2 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 : on va chercher les racines carrées de 1. En effet, si n est premier, $\mathbb{Z}/n\mathbb{Z}$ est un corps donc il y a exactement deux solutions à l'équation $X^2 = \overline{1}$: $\overline{1}$ et $\overline{-1}$. À l'inverse, si n n'est pas premier, il peut y en avoir davantage : par exemple, dans $\mathbb{Z}/15\mathbb{Z}$, les solutions sont $\overline{-1}$, $\overline{1}$, $\overline{-4}$ et $\overline{4}$. Voici comment on va utiliser les racines carrées de $\overline{1}$ pour déterminer si n est premier :

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 $a^t \equiv \overline{1} \mod n$, 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

► **Question 12** Écrire une fonction qui décompose l'entier n donné en argument en un couple (s, t) tel que $n = 2^s \cdot t$ avec t impair.

► **Question 13** Implémenter à présent le test de primalité de Rabin-Miller.

► **Question 14** Tester le gain de temps que procure cet algorithme comparé à l'algorithme naïf.

On peut remarquer que ce test est l'un des deux utilisés en pratique, le second étant celui de Soloway et Strassen, basé sur les courbes elliptiques. Il existe aussi depuis 2002 un algorithme polynomial déterministe pour décider la primalité d'un nombre mais son exposant est trop important (supérieur à 10) pour qu'il soit utilisable en pratique.

4 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. En effet, elles se basent la plupart du temps sur des valeurs particulières des exposants e et d. Les plus performants actuellement utilisent à fond l'algorithme LLL de réduction de réseaux qui permet de trouver une base de vecteurs courts d'un réseau de $\mathbb{Z}^n$.

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.

4.1 Algorithme naïf

► **Question 15** Écrire un algorithme de factorisation naïf basé sur le même principe que le test de primalité de la question 2.

Cet algorithme trouve à coup sûr un facteur s'il en existe un mais il requiert $\lfloor\sqrt{n}\rfloor$ 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.

4.2 Algorithme rho de Pollard

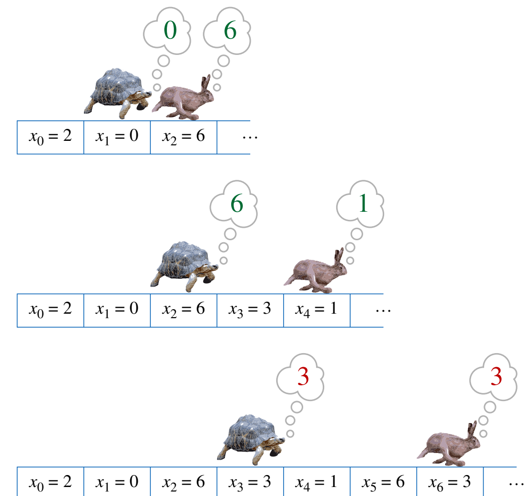
On peut faire mieux avec l'*algorithme rho* de Pollard, algorithme probabiliste basé sur deux faits : le paradoxe des anniversaires et l'algorithme du lièvre et de la tortue. Commençons par examiner l'algorithme du lièvre et de la tortue.

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.

► **Question 16** Implémenter l'algorithme du lièvre et de la tortue. Il sera de type

```
cycle_detect : (int → int) → int → int → bool
```

et, 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.



crédit image : David Eppstein, Wikimedia

Le paradoxe des anniversaires nous dit que parmi 23 personnes, la probabilité d'en avoir deux nées le même jour est supérieure à 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 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 les mêmes tests. Le nom même d'*algorithme rho* rappelle que la trajectoire du générateur va revenir sur elle-même pour former un cycle, tel un ρ.

► **Question 17** Implémenter l'algorithme rho de factorisation de Pollard.

► **Question 18** Tenter de décrypter[2] un cryptogramme. Prendre n pas trop grand !

[2] *i.e.* déchiffrer sans avoir la clé de déchiffrement