Vincent Simonet (http://cristal.inria.fr/~simonet/)

Cryptographie

Crypter un message consiste à lui appliquer une série de transformations (que l'on peut inverser) afin de le rendre incompréhensible pour toute personne qui n'en est pas le destinataire.
On distingue généralement deux types de systèmes cryptographiques : symétriques *et* asymétriques. *Dans les systèmes symétriques (ou à clef secrète), le codage et le décodage s'effectuent selon le même principe, avec la* même clef. *Pour garantir que leur communication reste confidentielle, deux interlocuteurs doivent donc avoir pu échanger préalablement la clef de chiffrement par une voie « sécurisée ».*
À l'inverse, dans un système asymétrique *(ou à clef publique), chaque individu possède deux clefs distinctes. La première, la clef publique, est communiquée à tous ses interlocuteurs. Elle leur permet de chiffrer les messages qu'ils souhaitent envoyer au propriétaire de la clef. Pour décoder ces messages, il est nécessaire de connaître la deuxième clef — que l'on ne peut pas « deviner » à partir de la clef publique — gardée secrète.*
Nous allons nous intéresser aujourd'hui à un tel algorithme dénommé RSA, du nom de ces trois concepteurs Rivest, Shamir et Adleman qui l'ont exposé en 1978.

1 L'algorithme d'Euclide

L'algorithme d'Euclide permet, étant donnés deux entiers a et b, de calculer leur plus grand commun diviseur (pgcd) d. Cet algorithme se base sur la propriété suivante :

$$\mathrm{pgcd}(a, b) = \begin{cases} a & \text{si } b = 0 \\ \mathrm{pgcd}(b, a \bmod b) & \text{sinon} \end{cases}$$

▶ **Question 1** *Écrivez une fonction* pgcd *qui calcule le plus grand commun diviseur de deux entiers en utilisant l'algorithme d'Euclide.*
value pgcd : $int \to int \to int$

Le théorème de Bézout nous assure également l'existence de deux entiers u et v tels que $au + bv = d$ (u et v sont des *coefficients de Bézout* de a et b). Une version étendue de l'algorithme d'Euclide permet de calculer, en plus du pgcd d des valeurs possibles pour les coefficients de Bézout u et v. Cet algorithme prend en entrée deux entiers a et b. Il procède de la manière suivante :
– Si $b = 0$ alors $d = a$, $u = 1$ et $v = 0$.
– Sinon, on applique récursivement l'algorithme sur les entiers b et $(a \bmod b)$. On obtient ainsi d', u' et v' tels que
$$\begin{cases} d' = \mathrm{pgcd}(b, a \bmod b) \\ bu' + (a \bmod b)v' = d' \end{cases}$$
On en déduit la solution pour a et b grâce aux égalités $d = d'$, $u = v'$ et $v = u' - \lfloor a/b \rfloor v'$.

▶ **Question 2** *Déduisez-en une fonction* euclide *qui étant donnés deux entiers a et b calcule le triplet (d, u, v) comme expliqué ci-dessus.*
value euclide : $int \to int \to int \times int \times int$

2 Clefs publiques et secrètes

Nous nous intéressons dans cette partie à la génération d'une paire de clefs (l'une publique et l'autre secrète) pour un individu dans le système RSA. On peut procéder de la manière suivante :
– Choisir deux "grands" entiers premiers p et q,
– Calculer leur produit $n = pq$,
– Choisir un "petit" entier impair e premier avec $\phi(n) = (p-1)(q-1)$,

© Vincent Simonet 2002
Ce document est distribué selon les termes de la GNU Free Documentation License.

– Calculer d tel que de soit congru à 1 modulo $\phi(n)$.

La clef publique est alors le couple (e, n) et la clef secrète le couple (d, n). La fiabilité du système réside dans le fait que le calcul de la partie secrète de la clef d à partir de la clef publique (e, n) nécessite *a priori* de calculer la décomposition en facteurs premiers de l'entier n, ce qui est un problème nécessitant beaucoup de ressources de calcul si les entiers p et q sont grands.

Pour simplifier, nous supposerons que les entiers premiers p et q nous sont donnés. De plus, afin de rester dans les possibilités du type *int* de *Caml*, nous nous limiterons à des entiers p et q très petits (pour une implantation plus réaliste, on pourrait utiliser la bibliothèque *num* de *Caml* qui permet de faire des calculs arithmétiques sur des nombres de taille arbitraire).

Dans les questions suivantes, nous nous intéressons au calcul des clefs publiques et secrètes à partir des entiers p et q.

▶ **Question 3** *Pour calculer e, nous allons utiliser un procédé aléatoire : il suffit en effet pour cela de « tirer au sort » des entiers jusqu'à en trouver un qui soit premier avec $(p-1)(q-1)$.*
Écrivez une fonction calcule_e *prenant pour arguments p et q et retournant une valeur possible pour e.*

value calcule_e : $int \to int \to int$

On peut montrer que, une fois l'entier e fixé, il existe un unique d tel que de soit congru à 1 modulo $\phi(n)$.

▶ **Question 4** *Programmez une fonction* calcule_d *qui calcule l'entier d correspondant à trois entiers donnés p, q et e.*

value calcule_d : $int \to int \to int \to int$

3 Codage et décodage

Le petit théorème de Fermat — que vous avez probablement énoncé en arithmétique — permet de démontrer que pour tout entier m on a $m^{de} \bmod n = m$. Cette propriété permet de définir des fonctions de codage et de décodage. Pour tous entiers m et c inférieurs à n, on pose :

$$\begin{aligned} code(m) &= m^e \bmod n \\ decode(c) &= c^d \bmod n \end{aligned}$$

On a alors bien l'égalité voulue :

$$\begin{aligned} decode(code(m)) &= decode(m^e \bmod n) \\ &= (m^e \bmod n)^d \bmod n \\ &= m^{ed} \bmod n \\ &= m \end{aligned}$$

Les fonctions *code* et *decode* nécessitent de calculer des puissances d'entiers. Pour obtenir une implémentation efficace, il est donc important d'avoir une "bonne" fonction puissance. Lors de la première séance, nous avions écrit une telle fonction qui calculait x^m en effectuant m multiplications successives. On peut en fait écrire une fonction beaucoup plus efficace en remarquant que :

$$\begin{array}{rcll} x^0 &=& 1 & \\ x^m &=& (x^{m/2})^2 & \text{si } m \text{ est pair} \\ x^m &=& x(x^{(m-1)/2})^2 & \text{si } m \text{ est impair} \end{array}$$

▶ **Question 5** *Déduisez-de ces relations une fonction* pow *plus performante que celle que nous avions écrite lors du premier TP. Combien de multiplications effectue-t-elle pour calculer x^m ?*

value pow : $int \to int \to int$

Les fonctions *code* et *decode* nécessitent en fait de calculer $x^m \bmod n$. Au lieu de calculer x^m puis d'effectuer la division euclidienne par n pour obtenir le reste, il est préférable d'effectuer une telle division à chaque étape de calcul de manière à manipuler des entiers aussi petits que possible.

▶ **Question 6** *Écrivez une fonction* powmod *telle que* powmod n x m *calcule $x^m \bmod n$.*

value powmod : $int \to int \to int \to int$

▶ **Question 7** *Déduisez-en deux fonctions* code *et* decode.

value code : $int \times int \to int \to int$
value decode : $int \times int \to int \to int$

▶ **Question 8** *Il est facile de remarquer que l'on a également pour tout entier m la relation $code(decode(m)) = m$. Voyez-vous une application de cette propriété ?*

Si vous êtes intéressés par les questions relatives à la cryptographie, vous pouvez lire un article d'introduction écrit par Anne Canteau (INRIA, Projet Codes) et François Lévy-dit-Véhel (ENSTA), disponible sur le *web* à l'adresse :
`http://www-rocq.inria.fr/codes/Anne.Canteaut/crypto_moderne.pdf`

Cryptographie

Un corrigé

L'opérateur mod de *Caml* ne renvoie le reste de la division euclidienne de ses arguments que si ceux-ci sont *positifs*. Si le premier argument est négatif, le résultat l'est aussi. Pour ce corrigé, nous définissons donc un opérateur % tel que a % b donne toujours le reste de la division euclidienne de a par b.

```
let prefix % a b =
  if a < 0
  then a mod b + b
  else a mod b
;;
```

▶ Question 1

```
let rec pgcd a b =
  if b = 0 then a
  else pgcd b (a % b)
;;
```

▶ Question 2

```
let rec euclide a b =
  if b = 0 then (a, 1, 0)
  else begin
    let (d', u', v') =
      euclide b (a % b)
    in
    (d', v', u' - (a / b) * v')
  end
;;
```

▶ Question 3

```
let rec calcule_e p q =
  let e =
    random__int ((p - 1) * (q - 1))
  in
  if pgcd e ((p - 1) * (q - 1)) = 1
  then e
  else calcule_e p q
;;
```

▶ Question 4

```
let calcule_d p q e =
  let (_, u, _) =
    euclide e ((p - 1) * (q - 1))
  in
  u % ((p - 1) * (q - 1))
;;
```

▶ Question 5

```
let square x =
  x * x
;;

let rec pow x = function
    0 -> 1
  | m when m mod 2 = 0 ->
      square (pow x (m / 2))
  | m->
      x * square (pow x ((m - 1) / 2))
;;
```

▶ Question 6

```
let rec powmod n x = function
    0 -> 1
  | m when m mod 2 = 0 ->
      (square (powmod n x (m / 2))) % n
  | m->
      (x *
        (square (powmod n x ((m - 1) / 2)) % n))
      % n
;;
```

▶ Question 7

```
let code (e, n) m =
  powmod n m e
;;

let decode (d, n) c =
  powmod n c d
;;
```