Remarques sur la bibliothèque des procédures du package Kruptor_2.1

Avertissement : pour bien comprendre les différentes fonctions et leur rôle exact dans le contexte, il faut maîtriser les notions de cryptographie qui s'y rapportent. Il existe de nombreux bons livres pour cela. Je n'en citerais qu'un : le mien!

Cryptographie principes et mises en œuvre - 2e édition
Pierre Barthélemy, Robert Rolland, Pascal Véron
Hermes Lavoisier

1) Les composants de la bibliothèque kruptor_2.1

Ce sont les fichiers suivants:

- kruptor_commun_2.1.map
- kruptor_rsa_2.1.map
- kruptor_elgamal_2.1.map
- kruptor_hash_2.1.map

La bibliothèque a un but pédagogique, donc on a mis là un noyau très restreint de fonctions qui permettent d'illustrer le fonctionnement schématique de la construction et de l'utilisation de la primitive de chiffrement RSA, ainsi que de la primitive de signature RSA, la primitive de chiffrement ElGamal, la signature DSA et enfin un exemple de fonction de hachage.

2) tirage pseudo-aléatoire

Les fonctions qui permettent de programmer simplement les tirages dont on a besoin sont définies dans le composant **kruptor_commun_2.1.map.** Elles utilisent le générateur pseudo aléatoire fourni dans xcas.
Si on tire au hasard x bits, parfois le nombre entier constitué par ces bits a une taille <x. Ce cas se produit si le bit de poids fort tiré est 0. On a donc ici donné ici des fonctions qui tirent au sort tous les bits (toutalea) ou qui imposent 1 pour bit de poids fort (rralea) auquel cas la taille de l'entier tiré est exactement x. En résumé, on dispose d'au moins trois fonctions de la bibliothèque **kruptor_commun_2.1.map** pour tirer un nombre au hasard

La recherche de couples de nombres premiers p et q tels que p-1=kq est importante pour la construction de systèmes cryptographiques à base de logarithme discret sur Z/pZ. En particulier si on construit un grand nombre premier p au hasard sans se soucier d'autre chose, il est impossible après de calculer un élément primitif alpha de Z/pZ.

Comme cas spécial du cas précédent, on ne laisse aucune liberté pour k, on impose k=2, ce qui veut dire qu'on cherche p et q premiers tels que p=2q+1. Dans ce cas on dit que q est un nombre premier de Sophie Germain. On ne sait pas s'il existe une infinité de tels nombres, on le suppose, et en pratique on arrive à en construire. On remarquera toutefois, que pour des tailles égales, le programme qui cherche un nombre de Sophie Germain est bien plus long à exécuter que celui qui cherche un triplet p,q,k avec k non prédéterminé.

Dans le cas de la mise en place d'un échange de clé de Diffie-Hellman, ce qui nous importe c'est d'obtenir un groupe Z/pZ et un élément primitif de ce groupe. En conséquence on construira tout d'abord un triplet p,q,k avec p-1=kq de telle sorte que k soit suffisamment petit pour être factorisé. On dispose alors de la factorisation de p-1, et donc on peut déterminer un élément primitif alpha du groupe Z/pZ. En particulier on peut essayer de chercher un entier de Sophie Germain et donc de trouver p et q premiers tels que p-1=2q.
En revanche dans le cas du chiffrement d'Elgamal, ou de la signature DSA, il importe de construire les nombres premiers p et q vérifiant p-1=kq avec k grand, de telle sorte que par exemple p ait 1024 bits tandis que q en ait 160. Du coup on ne sait plus factoriser p-1, mais ceci n'est pas grave, car ce qu'il nous faut maintenant c'est un élément primitif de Z/qZ et non pas un élément primitif de Z/pZ.

3) La primitive RSA
Pour RSA, on donne les algorithmes classiques de constitution du système, de chiffrement de déchiffrement. Pour la signature on donne une version accélérée de la production de la signature, utilisant le théorème des restes chinois. Cette version est sensible à une attaque dite attaque par faute. On donne aussi un algorithme qui étant donné les exposants publics et privés ee et d, ainsi que le module n, permet de factoriser le module algorithme de Shank)

4) Le logarithme discret sur Z/pZ
La version que nous donnons du chiffrement d'Elgamal est une version dans laquelle on travaille dans le sous-groupe H d'ordre q du groupe multiplicatif de Z/pZ (d'ordre p-1). On peut prendre alors q bien plus petit que p, ce qui raccourcit les exposants qui entrent dans les calculs. Cependant il faut bien comprendre que les éléments de H sont écrits dans Z/pZ et qu'on a aucun moyen d'associer facilement à un message, un élément de H. On adopte donc une méthode de masquage pour chiffrer.
Plus précisément, on construit (p,q,alpha,a,b) où p et q alpha sont publics et communs à plusieurs individus, où b est public mais est spécifique d'un individu : c'est sa clé publique, où a est la clé privée de l'individu en question : on a en fait b=alpha^a.
p est un nombre premier d'au moins 1024 bits, q un nombre premier qui divise p-1 qui a 160 bits au moins, alpha est un élément primitif du sous-groupe d'ordre q du groupe multiplicatif (Z/pZ)* . Si m <q est le message à chiffrer pour le destinataire de clé publique b on tire au sort un entier k (un nouveau pour chaque chiffrement), et on calcule :

y1= alpha^k mod p

y2=h(b^k mod p) xor m

où h est une fonction de hachage publique (dont on se sert ici pour faire une compression) et transmet le couple (y1,y2) qui constitue le chiffré.
Le destinataire peut alors déchiffrer en calculant :
u1=y1^a mod p (qui est en fait b^k mod p)
u2=h(u1) xor y2.
Le fonction de compression programmée ici est une fonction de la famille NH.
On a aussi donné les algorithmes utiles pour faire de la signature DSA.
Dans tous les cas, il s'agit des primitives mathématiques, la partie encapsulation des messages (par exemple OAEP) n'est pas faite.

5) Le hachage cryptographique
Sha256 est programmé. L'utilisation prend deux formes : l'application du hachage à une chaine de caractères (d'octets), le hachage d'un entier dont on suppose que c'est un bloc de bits et donc, dont on précise la taille exacte (à cause des zéros possibles en bits de poids fort). L'exécution est très lente, et donc l'utilisation en temps raisonnable est limité à quelques milliers de bits (pas 1 mega par exemple). Rappelons que ces fonctions ont un rôle **pédagogique de démonstration ou de test**.
On peut construire à partir d'une fonction de hachage un générateur pseudo-aléatoire permettant par exemple de tirer au sort des clés. On appelle un tel générateur un générateur de masque (ou KDF pour key derivation function). On entre un germe et un nombre d'octets et le générateur calcule (de manière déterministe) une sortie pseudo-aléatoire de taille le nombre d'octets voulus. On a fourni un tel kdf construit à partir de la fonction de hachage sha256. La fonction kdf prend comme première variable un germe qu'on peut entrer à la main mais qu'on peut aller aussi récupérer dans le fichier fourni « seed » en chargeant ce fichier dans xcas par la commande read(). La variable qu'on charge s'appelle alors « germe ». Bien entendu on peut modifier à souhait la valeur donnée dans ce fichier à la variable germe.