Développement de Logiciels
Cryptographiques.
2020-2021

Faculté des Sciences et Techniques
Master 2 - Maths. CRYPTIS

RSA à jeu réduit d'instructions

A l'attention de :
CLAVIER C.

Rédigé par :
PIARD A.
JACQUET R.

Table des matières

Introduction

Le but de ce travail est d'implémenter les différentes fonctionnalités de RSA (génération des clés, chiffrement, déchiffrement, signature, et méthode CRT). Ces implémentations devront satisfaire à une contrainte bien précise : ne pas disposer de fonctions mathématiques évoluées, et se limiter aux seules quatre opérations de base sur grands entiers, que sont l'addition, la soustraction, la multiplication et la division. On appelle cette contrainte *jeu réduit d'instructions*.

Pour la manipulation de grands entiers en **C**, l'utilisation de la bibliothèque GMP, étudiée ce semestre, est nécessaire.

1 Fonctions

En raison de l'utilisation des seules quatre opérations élémentaires pour ce projet, des fonctions "supplémentaires" à RSA ont dû être ajoutées. En effet, pour chiffrer comme pour déchiffrer avec RSA, certaines opérations comme l'exponentiation modulaire, l'inverse modulaire, ou plus simplement le calcul modulaire sont nécessaires. Nous allons donc présenter ces fonctions dans la première partie. Pour ce qui est de la seconde partie, vous retrouverez toutes les fonctions nécessaires à la bonne fonctionnalité de RSA, c'est à dire les fonctions de chiffrement et déchiffrement, de signature et de vérification de signature, mais également leurs analogues par la méthode CRT. De plus, vous pourrez y retrouver la fonction de génération des clés.

1.1 Fonctions palliatives au jeu réduit d'instructions.

1.1.1 getModulus_ui(mpz_t res, mpz_t a, int b)

Cette fonction permet de calculer le modulo d'un grand nombre, c'est à dire son reste par la division euclidienne. Ici, le reste de la division euclidienne de a par b sera stocké dans res. res est un mpz_t, de même que a, mais b est quant à lui un entier (int).

```
1 void getModulus_ui(mpz_t res, mpz_t a, int b) {
2     mpz_t z_b;
3     mpz_init(z_b);
4     mpz_set_ui(z_b, b);
5     mpz_fdiv_r(res, a, z_b);
6 }
```

Dans cette fonction, nous avons simplement effectué un cast de int vers mpz_t en déclarant un mpt_t z_b. On lui assigne alors la valeur de b (ligne 4). Ensuite, on récupère le reste de la division euclidienne, que l'on stocke dans res grâce à la ligne 5.

La fonction getModulus(mpz_t res, mpz_t a, mpz_t b) n'est composée que de la ligne 5 de la méthode présentée ci-dessus.

```
1 void getModulus(mpz_t res, mpz_t a, mpz_t b) {
2     mpz_fdiv_r(res, a, b);
3 }
```

1.1.2 Inverse modulaire

La fonction qui calcule l'inverse modulaire d'un très grand nombre, représenté par un mpz_t, a l'en-tête suivant :

computeInvert(mpz_t d, mpz_t e, mpz_t n)

Cette fonction prend en arguments 3 mpz_t distincts que sont d, e et n. Dans d sera stocké l'inverse de e modulo n. Pour ce faire, le code suivant a été écrit :

```
1  void computeInvert(mpz_t d , mpz_t e , mpz_t n) {
2     mpz_t e0, t0, t, q, r, n0, _loc0;
3     mpz_inits(e0, t0, t, q, r, n0, _loc0, NULL);
4
5     mpz_set_ui(t, 1);
6     mpz_set(n0, n);
7     mpz_set(e0, e);
8     mpz_tdiv_q(q, n0, e0);
9     getModulus(r, n0, e0);
10    do {
11        mpz_mul(_loc0, q, t);
12        mpz_sub(_loc0, t0, _loc0);
13        if(mpz_cmp_ui(_loc0, 0) >= 0) {
14            getModulus(_loc0, _loc0, n);
15        }
16        else {
17            getModulus(_loc0, _loc0, n);
18        }
19        mpz_set(t0, t);
20        mpz_set(t, _loc0);
21        mpz_set(n0, e0);
22        mpz_set(e0, r);
23        mpz_tdiv_q(q, n0, e0);
24        getModulus(r, n0, e0);
25
26    }while(mpz_cmp_ui(r, 0) > 0);
27    mpz_set(d, t);
28
29    mpz_clears(e0, t0, t, q, r, n0, _loc0, NULL);
```

Il s'agit ni plus ni moins de l'implémentation de l'Algorithme d'Euclide Étendu (AEE).

1.1.3 Puissance modulaire

la fonction qui calcule la puissance modulaire a l'en-tête suivant :

powering(mpz_t result, mpz_t a, mpz_t b, mpz_t n)

Cette fonction permet d'effectuer une exponentiation modulaire. Les paramètres de cette fonction permettent de calculer a^b mod n, qui sera stocké dans *result*.

```
void powering(mpz_t result, mpz_t a, mpz_t b, mpz_t n) {
    mpz_t _loc, t, a_bis, b_bis;
    mpz_inits(_loc, t, a_bis, b_bis, NULL);
    mpz_set(a_bis, a);
    mpz_set(b_bis, b);
    mpz_set_ui(_loc, 1);

    while (mpz_cmp_ui(b_bis, 0) > 0) {
        getModulus_ui(t, b_bis, 2);
        if(mpz_cmp_ui(t, 0) != 0) {
            mpz_mul(_loc, _loc, a_bis);
            getModulus(_loc, _loc, n);
        }
        mpz_mul(a_bis, a_bis, a_bis);
        getModulus (a_bis, a_bis, n);
        mpz_tdiv_q_ui(b_bis, b_bis, 2);
    }

    mpz_set(result, _loc);
    mpz_clears( _loc, t, a_bis, b_bis, NULL);
}
```

1.2 Fonctions relatives à RSA

Dans cette section, vous trouverez toutes les fonctions relatives au bon fonctionnement de RSA. Cela va de la génération des clés à la signature RSA, en passant par le chiffrement et le déchiffrement, les vérifications de signature et la méthode CRT.

1.2.1 Génération des clés

Dans un premier temps, nous avons créés des fonctions permettant de créer un nombre premier (et de vérifier qu'il est bien premier). Nous avons également implémenté une fonction qui calcule le PGCD de deux nombres. Ceci dans le but de créer efficacement les clés publiques et privées de RSA.

Vérification du caractère premier d'un nombre

La fonction qui permet la génération des grands premiers p et q a l'en-tête suivant :

genPrime(mpz_t p, mpz_t q, int n, gmp_randstate_t rnd)

1.2.2 Chiffrements

1.2.3 Déchiffrements

1.2.4 Signatures

2 Jeu d'essai

Conclusion

Références