Développement de Logiciels Cryptographiques
2021 - 2022

Faculté des Sciences et Techniques
Master 2 - Maths. CRYPTIS

RSA à jeu réduit d'instructions

A l'attention de :
M. CLAVIER

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, pour *Chinese Remainder Theorem*). 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.

Rappels RSA et CRT

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 la réduction 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 Calcul du modulo de très grands nombres

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. La variable 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 `mpz_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) renvoie le résultat de a modulo b, c'est à dire le reste de la division euclidienne de a par b, et ce résultat est stocké dans res. Pour cette fonction nous ne travaillons qu'avec des `mpz_t`. Il n'y a pas de int.

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

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 trois mpz_t distincts qui sont d, e et n. Dans d sera stocké l'inverse de e modulo n. Pour ce faire, le code suivant a été écrit :

```
void computeInvert(mpz_t d , mpz_t e , mpz_t n) {
   mpz_t e0, t0, t, q, r, n0, _loc0;
   mpz_inits(e0, t0, t, q, r, n0, _loc0, NULL);

   mpz_set_ui(t, 1);
   mpz_set(n0, n);
   mpz_set(e0, e);
   mpz_tdiv_q(q, n0, e0);
   getModulus(r, n0, e0);
   do {
       mpz_mul(_loc0, q, t); // _loc0 = q * t
       mpz_sub(_loc0, t0, _loc0); // _loc0 = t0 - _loc0
       if(mpz_cmp_ui(_loc0, 0) >= 0) {
           getModulus(_loc0, _loc0, n); // _loc0 = _loc0 % n
       }
       else {
           getModulus(_loc0, _loc0, n); // _loc0 = _loc0 % n
       }
       mpz_set(t0, t);
       mpz_set(t, _loc0);
       mpz_set(n0, e0);
       mpz_set(e0, r);
       mpz_tdiv_q(q, n0, e0);
       getModulus(r, n0, e0);

   }while(mpz_cmp_ui(r, 0) > 0);
   mpz_set(d, t);

   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 :

$$\text{powering}(\mathsf{mpz\_t}\ result, \mathsf{mpz\_t}\ a, \mathsf{mpz\_t}\ b, \mathsf{mpz\_t}\ n)$$

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

```c
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); // t = b_bis % 2
        if(mpz_cmp_ui(t, 0) != 0) {
            mpz_mul(_loc, _loc, a_bis); // _loc = _loc * a_bis
            getModulus(_loc, _loc, n); // _loc = _loc % n
        }
        mpz_mul(a_bis, a_bis, a_bis); // a_bis = a_bis * a_bis
        getModulus(a_bis, a_bis, n); // 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);
}
```

Il s'agit ici de l'algorithme **Square and Multiply**.

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 les mêmes méthodes en version 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 est la suivante :

```
void genPrime(mpz_t p, mpz_t q, int n, gmp_randstate_t state) {
    mpz_t rand, _loc0, max, min;
    mpz_inits(rand, _loc0, max, min, NULL);
    mpz_ui_pow_ui(max, 2, n+1); // Borne sup 2^n+1
    mpz_ui_pow_ui(min, 2, n); // Borne inf
    do {
        mpz_urandomm(rand, state, max); // On le génère de la bonne taille
    }while(mpz_cmp(rand, min) < 0);
    bePrime(p, rand); // On cherche un premier de taille prédefinie
    do {
        mpz_urandomm(_loc0, state, max );
    }while((mpz_cmp(_loc0, min) < 0 ));
    bePrime(q, _loc0);
    if(mpz_cmp(p, q) == 0) {
        while(mpz_cmp(p, q) == 0) {
            do {
                mpz_urandomm(_loc0, state, max );
            }while((mpz_cmp(_loc0, min) < 0 ));
            bePrime(q, _loc0);
        }
    }
    mpz_clears(rand, _loc0, max, min, NULL);
}
```

Cette fonction va générer des nombres p et q de taille n et va ensuite les 'rendre' premier grâce à la fonction bePrime(mpz_t p, mpz_t t) qui est disponible dans le fichier source de notre projet. Le test de primalité utilisé est basé sur le test probabiliste de Miller-Rabin.

1.2.2 Chiffrements

Concernant le chiffrement, il n'y aucun différence entre le RSA basique et le RSA CRT. Étudions le code suivant de la fonction main() :

```
...
// RSA BASIC
// Initialisation des variables
    genNumber(msg, round(nbBits / 2), rand);
    gmp_printf("RSA à jeu réduit d'instructions pour
    n = %d, message : %Zd.", nbBits, msg);
    genPrime(p, q, round(nbBits / 2), rand);
    gmp_printf("p = %Zd\n", p);
    gmp_printf("q = %Zd\n", q);
    mpz_set_ui(e, 65537);
    gmp_printf("e = %Zd\n", e);
    mpz_mul(n, p, q); // n = p * q

    gmp_printf("n = p * q = %Zd\n", n);

    mpz_sub_ui(p_1, p, 1); // p - 1
    mpz_sub_ui(q_1, q, 1); // q - 1

    mpz_mul(phi, p_1, q_1);

    gmp_printf("phi = %Zd\n", phi);
    computeInvert(d, e, phi); // d = e ^-1 [phi]
    gmp_printf("d = %Zd\n", d);

    printf("\n\n\n");
...
```

Sont d'abord générés les éléments primordiaux pour pouvoir utiliser le cryptosystème RSA, à savoir, $p, q, n = p \cdot q$ et le message msg qui est un nombre aléatoire de taille prédéfinie. Ensuite est calculé d qui est l'inverse de e modulo $\phi(n) = (p-1) \cdot (q-1)$.

Dans le cas du RSA dit classique, la fonction pour chiffrer est la fonction

$$\text{encrypt}(\text{mpz\_t } encrypted, \text{mpz\_t } message, \text{mpz\_t } e, \text{mpz\_t } n)$$

Elle calcule le chiffré de $message$ en calculant grâce à la fonction powering

$$encrypted = (message)^e \mod n.$$

1.2.3 Déchiffrements

Pour ce qui est du déchiffrement, il y a des différences entre les deux versions. Commençons par la version basique. Aussi simpliste que pour le chiffrement, pour déchiffrer il faut calculer une puissance modulaire mais avec la clé privée d.

```
void decrypt(mpz_t original, mpz_t encrypted, mpz_t d, mpz_t n) {
    powering(original, encrypted, d, n);
}
```

Quant à la version utilisant le théorème des restes chinois. Il y a des calculs qui précédent,

```
mpz_sub_ui(e_p, p, 1); //e_p = p - 1
mpz_sub_ui(e_q, q, 1); //e_q = q - 1
computeInvert(i_p, p, q);
computeInvert(d_p, e, e_p);
computeInvert(d_q, e, e_q);
```

En effet on calcule d'abord i_p qui est l'inverse de p modulo q, d_p qui est l'inverse de e modulo e_p et d_q qui est l'inverse de e modulo e_q. Le déchiffrement est très différent, il se passe en deux étapes

```
void decrypt_CRT(mpz_t msg,  mpz_t cipher , mpz_t d_p, mpz_t p, mpz_t d_q, mpz_t q, mpz_t i_p)
    mpz_t message, m_p, m_q, n, _loc0, pq, _loc1;
    mpz_inits(message, m_p, m_q, n, _loc0, pq, _loc1, NULL);
    mpz_set_ui(message, 1);
    mpz_mul(n, p, q);
    decrypt(m_p, cipher, d_p, p); // m_p = cipher ^ d_p % p
    decrypt(m_q, cipher, d_q, q); // m_q = cipher ^ d_q % q
    mpz_sub(pq, m_q, m_p); // pq = m_q - m_p
    mpz_mul(_loc0, pq, i_p); // _loc0 = pq - ip
    getModulus(_loc1, _loc0, q); // _loc1 = _loc0 - q
    mpz_mul(message, _loc1, p); // message = _loc1 * p
    mpz_add(message, message, m_p); // message = message + m_p
    getModulus(message, message, n); // message = message + n
    mpz_set(msg, message);
    mpz_clears(message, m_p, m_q, n, _loc0, _loc1, pq, NULL);
}
```

Le théorème des restes chinois appliqués à RSA revient juste à décomposer le chiffré en deux parties et le calcul final est le suivant

$$M = \left(\left((M_p - M_q) \cdot q^{-1}\right) \mod p\right) \cdot q + M_q$$

1.2.4 Signatures

Les signatures pour le RSA basique sont très simples. Il s'agit uniquement de mettre à la puissance d le message m, le tout modulo n.

```
void sig_msg_RSA(mpz_t sig, mpz_t message, mpz_t d, mpz_t n) {
    decrypt(sig, message ,d ,n);
}
```

Concernant la version CRT, cela est légèrement plus complexe, on utilise la fonction de déchiffrement version CRT, on passe donc les éléments précalculés en paramètres

```
void sig_msg_RCT(mpz_t sig, mpz_t msg, mpz_t d_p, mpz_t p, mpz_t d_q, mpz_t q, mpz_t i_p) {
    decrypt_CRT(sig, msg , d_p,  p,  d_q,  q,  i_p);
}
```

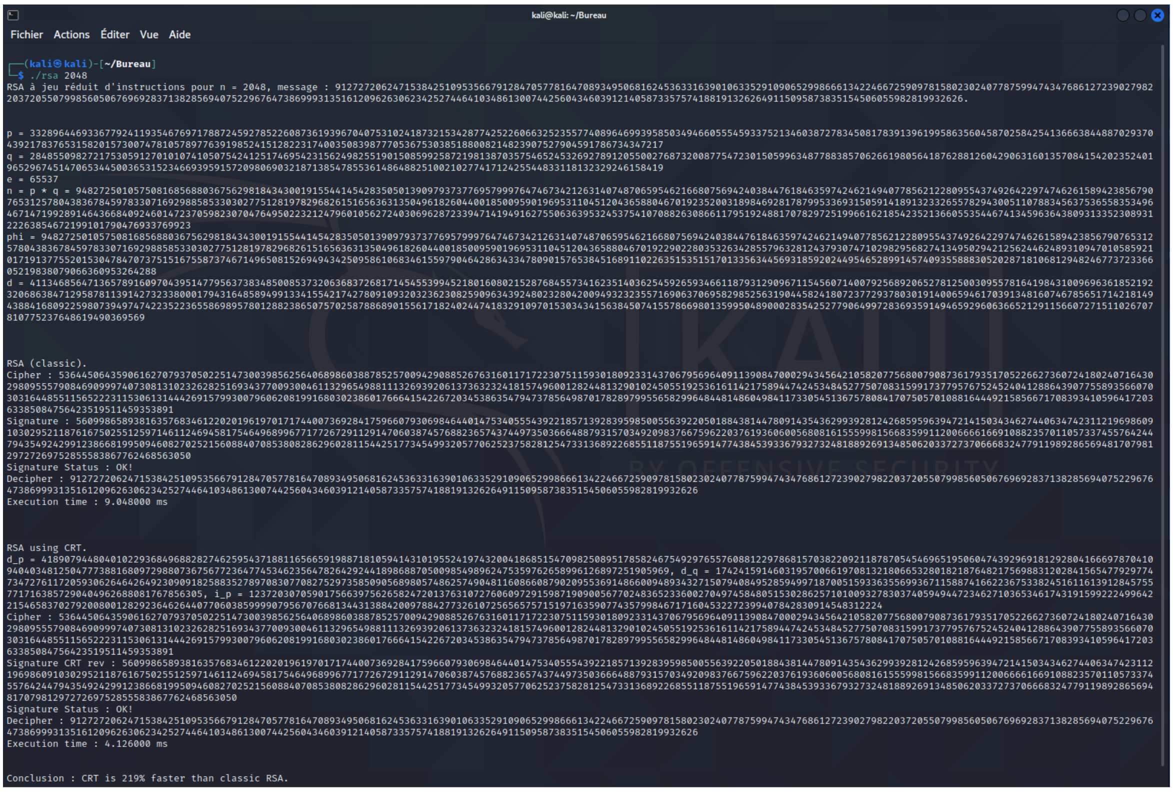
Une fonction vérifiant la signature a également été développée,

```
void verif_sig_RSA(mpz_t sig , mpz_t message, mpz_t e, mpz_t n) {
    mpz_t hash;
    mpz_init(hash);
    encrypt(hash, sig, e, n);
    if(mpz_cmp(message, hash) == 0) {
        printf("Signature Status : OK!\n");
    }
    else {
        printf("Signature Status : NOT OK ! Altered message.\n");
    }
    mpz_clear(hash);
}
```

ce qui permet de vérifier l'intégrité, d'un message.

2 Jeu d'essai

Nous allons vous montré au travers d'une impression d'écran l'exécution de notre programme avec un nombre de bits égal à 2048.



En premier lieu apparaît le message généré aléatoirement à partir d'un nombre de bits défini à l'appel du programme. Ici, 2048.
Vient ensuite l'affichage des paramètres p, q et ϕ et des parties de clés e, n et d.

Avec ces éléments, vient le RSA (classic).
Le résultat du calcul du chiffré (cipher = message^e (mod n)) est affiché. Vient ensuite la signature du chiffré, qui permettra de vérifier que le message reçu, déchiffré au préalable, n'a pas été modifié. Si tel est le cas, le Signature Status renverra "OK !". Autrement, il renverra "NOT OK ! Altered message."
Enfin, c'est le déchiffré du chiffré qui sera affiché, et on voit que le message originel et le Decipher sont les mêmes.

Enfin, le RSA avec la méthode CRT est affiché.
Pour ce RSA là, la clé privée doit être dérivée, ce qui nous donne donc d_p, d_q et également i_p. Est ensuite affiché le chiffré, qui peut être vérifié avec celui obtenu par le chiffrement RSA classique. De même que pour la signature et le déchiffrement.

Conclusion

Nous avons eu la chance de pouvoir travailler sur notre premier choix de projet. Les sujets sur RSA nous intéressaient et la contrainte de jeu réduit d'instructions nous a attiré car nous suivons l'Unité d'Enseignement Carte à Puce et Développement Java Card. Or, l'environnement des cartes à puces est contraint et le concept de jeu réduit d'instructions s'y prête tout particulièrement.

Nous avons apprécié travailler sur ce sujet pour les raisons citées ci-avant, et également car l'utilisation de la bibliothèque GMP était nécessaire et nous avions tous deux appréciés les travaux pratiques du cours de Développement de Logiciels Cryptographiques et l'utilisation de cette fameuse bibliothèque.
L'intégralité des fonctions relatives à RSA ont été implémentées, ainsi que des fonctions efficaces, satisfaisant la contrainte de jeu réduit d'instructions, pour calculer l'exponentiation modulaire ou encore l'inverse modulaire. Nous affirmons ainsi que le projet a été mené à bien. Cependant, comme dans n'importe quel projet, l'implémentation de nouvelles fonctionnalités est un plus et il est vrai que ceci nous a manqué (Faute d'imagination ?). C'est pourquoi nous nous sommes concentrés sur le respect total des contraintes qu'amène le jeu réduit d'instructions. Nous estimons donc notre programme assez bien optimisé, ce qui est essentiel dans ce contexte et avec un algorithme coûteux comme RSA, considérant la taille des clés utilisées et sa complexité.

Références