

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

<b>Introduction</b>	<b>2</b>
<b>1 Fonctions</b>	<b>3</b>
1.1 Fonctions palliatives au jeu réduit d'instructions. . . . .	3
1.1.1 Calcul du modulo de très grands nombres . . . . .	3
1.1.2 Inverse modulaire . . . . .	4
1.1.3 Puissance modulaire . . . . .	5
1.2 Fonctions relatives à RSA . . . . .	6
1.2.1 Génération des clés . . . . .	6
1.2.2 Chiffrements . . . . .	7
1.2.3 Déchiffrements . . . . .	8
1.2.4 Signatures . . . . .	9
<b>2 Jeu d'essai</b>	<b>10</b>
<b>Conclusion</b>	<b>11</b>
<b>Références</b>	<b>12</b>

## 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`.

```
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 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 :

```

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); // _loc0 = q * t
12        mpz_sub(_loc0, t0, _loc0); // _loc0 = t0 - _loc0
13        if(mpz_cmp_ui(_loc0, 0) >= 0) {
14            getModulus(_loc0, _loc0, n); // _loc0 = _loc0 % n
15        }
16        else {
17            getModulus(_loc0, _loc0, n); // _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);
30 }

```

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  $a^b \bmod n$ , qui sera stocké dans *result*.

```

1 void powering(mpz_t result, mpz_t a, mpz_t b, mpz_t n) {
2     mpz_t _loc, t, a_bis, b_bis;
3     mpz_inits(_loc, t, a_bis, b_bis, NULL);
4     mpz_set(a_bis, a);
5     mpz_set(b_bis, b);
6     mpz_set_ui(_loc, 1);
7
8     while (mpz_cmp_ui(b_bis, 0) > 0) {
9         getModulus_ui(t, b_bis, 2); // t = b_bis % 2
10        if(mpz_cmp_ui(t, 0) != 0) {
11            mpz_mul(_loc, _loc, a_bis); // _loc = _loc * a_bis
12            getModulus(_loc, _loc, n); // _loc = _loc % n
13        }
14        mpz_mul(a_bis, a_bis, a_bis); // a_bis = a_bis * a_bis
15        getModulus(a_bis, a_bis, n); // a_bis = a_bis % n
16        mpz_tdiv_q_ui(b_bis, b_bis, 2);
17    }
18
19    mpz_set(result, _loc);
20    mpz_clears(_loc, t, a_bis, b_bis, NULL);
21 }
```

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 :

```

1 void genPrime(mpz_t p, mpz_t q, int n, gmp_randstate_t state) {
2     mpz_t rand, _loc0, max, min;
3     mpz_inits(rand, _loc0, max, min, NULL);
4     mpz_ui_pow_ui(max, 2, n+1); // Borne sup 2^n+1
5     mpz_ui_pow_ui(min, 2, n); // Borne inf
6     do {
7         mpz_urandomm(rand, state, max); // On le génère de la bonne taille
8     }while(mpz_cmp(rand, min) < 0);
9     bePrime(p, rand); // On cherche un premier de taille prédefinie
10    do {
11        mpz_urandomm(_loc0, state, max );
12    }while((mpz_cmp(_loc0, min) < 0 ));
13    bePrime(q, _loc0);
14    if(mpz_cmp(p, q) == 0) {
15        while(mpz_cmp(p, q) == 0) {
16            do {
17                mpz_urandomm(_loc0, state, max );
18            }while((mpz_cmp(_loc0, min) < 0 ));
19            bePrime(q, _loc0);
20        }
21    }
22    mpz_clears(rand, _loc0, max, min, NULL);
23 }
```

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 a aucune différence entre le RSA basique et le RSA CRT. Étudions le code suivant de la fonction `main()` :

```

1  ...
2  // RSA BASIC
3  // Initialisation des variables
4  genNumber(msg, round(nbBits / 2), rand);
5  gmp_printf("RSA à jeu réduit d'instructions pour
6  n = %d, message : %Zd.", nbBits, msg);
7  genPrime(p, q, round(nbBits / 2), rand);
8  gmp_printf("p = %Zd\n", p);
9  gmp_printf("q = %Zd\n", q);
10 mpz_set_ui(e, 65537);
11 gmp_printf("e = %Zd\n", e);
12 mpz_mul(n, p, q); // n = p * q
13
14 gmp_printf("n = p * q = %Zd\n", n);
15
16 mpz_sub_ui(p_1, p, 1); // p - 1
17 mpz_sub_ui(q_1, q, 1); // q - 1
18
19 mpz_mul(phi, p_1, q_1);
20
21 gmp_printf("phi = %Zd\n", phi);
22 computeInvert(d, e, phi); // d = e ^{-1} [phi]
23 gmp_printf("d = %Zd\n", d);
24
25 printf("\n\n");
26 ...

```

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

`encrypt(mpz_t encrypted, mpz_t message, mpz_t e, mpz_t n)`

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

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



### 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  $d$ .

```

1 void decrypt(mpz_t original, mpz_t encrypted, mpz_t d, mpz_t n) {
2     powering(original, encrypted, d, n);
3 }
```

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

```

1 mpz_sub_ui(e_p, p, 1); //e_p = p - 1
2 mpz_sub_ui(e_q, q, 1); //e_q = q - 1
3 computeInvert(i_p, p, q);
4 computeInvert(d_p, e, e_p);
5 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

```

1 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)
2     mpz_t message, m_p, m_q, n, _loc0, pq, _loc1;
3     mpz_inits(message, m_p, m_q, n, _loc0, pq, _loc1, NULL);
4     mpz_set_ui(message, 1);
5     mpz_mul(n, p, q);
6     decrypt(m_p, cipher, d_p, p); // m_p = cipher ^ d_p % p
7     decrypt(m_q, cipher, d_q, q); // m_q = cipher ^ d_q % q
8     mpz_sub(pq, m_q, m_p); // pq = m_q - m_p
9     mpz_mul(_loc0, pq, i_p); // _loc0 = pq * i_p
10    getModulus(_loc1, _loc0, q); // _loc1 = _loc0 - q
11    mpz_mul(message, _loc1, p); // message = _loc1 * p
12    mpz_add(message, message, m_p); // message = message + m_p
13    getModulus(message, message, n); // message = message % n
14    mpz_set(msg, message);
15    mpz_clears(message, m_p, m_q, n, _loc0, _loc1, pq, NULL);
16 }
```

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

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

### 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  $n$ .

```
1 void sig_msg_RSA(mpz_t sig, mpz_t message, mpz_t d, mpz_t n) {  
2     decrypt(sig, message ,d ,n);  
3 }
```

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

```
1 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) {  
2     decrypt_CRT(sig, msg , d_p, p, d_q, q, i_p);  
3 }
```

Une fonction vérifiant la signature a également été développée,

```
1 void verif_sig_RSA(mpz_t sig , mpz_t message, mpz_t e, mpz_t n) {  
2     mpz_t hash;  
3     mpz_init(hash);  
4     encrypt(hash, sig, e, n);  
5     if(mpz_cmp(message, hash) == 0) {  
6         printf("Signature Status : OK!\n");  
7     }  
8     else {  
9         printf("Signature Status : NOT OK ! Altered message.\n");  
10    }  
11    mpz_clear(hash);  
12 }
```

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

## 2 Jeu d'essai

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

```

kali@kali: ~/Bureau
$ ./rsa 2048
RSA à jeu réduit d'instructions pour n = 2048, message : 91272206247153842510953566791284705778164789349506816245363133901863352910965299866134224667259097815802302407787599474347686122739027982
203720550799850806769692837138285994075229676473869993135161209626306234252744641034861300744256043460391214058733575741881913262649115095873835154506855982819932626.

p = 332894469367792411935046797178872459278526887361939670407531024187321534287742522086632535577408964699595803494660554593375213460387278345081783913961995863560458702584254136663844887029370
439217837651582051300747818578977699198524151822317408330839877783367338381880002148239075790459178672424217
q = 284855082721753859127018107410597542125174695421562498251981580592587219813870357546524532692789120500876873200877547230158599634877883857062661980564187628812604296631601337084154282352
965296745147065345086531523466939515720980690321871385478553614864882150021277417242554833118132329246158419
e = 65537
n = p * q = 94827258105758016856880367562981843430019155441454283580139097937377695799764767342126314074870659546216688756942403844761846359742462149407785621228095543749264229747462158942385679
76512788438367845978307169298858330827751281978296826151656363135049618260440018508959019695311845128436588046701923520018984692817879953369315059141891323265578294300511078834563753658353496
4671471992891464368409246014723705982307047649502232124796010562724030696287239471419451627550636395245375410788826308661179519248817078297251996616218542352136605554467413459634380931352380931
22383546721991017947693376923
phi = 948272501057580168568803675629818434300191554414542835801390979373776957997647673421263140748706595462166887569424038447618463597424621494077856212280955437492642297474621589423856790765312
5788438367845978307169298858330827751281978296826151656363135049618260440018508959019695311845128436588046701923520018984692817879953369315059141891323265578294300511078834563753658353496
017191377558153847847073751516755873746714965081526949434250958610683461559798464286343478090157653841568911022635153515701336344569318592024495465289914574093558883852028718106812948246773723366
0521982807986036895264280
d = 41134685641365789160970439514779567383485080537320636837268171454553994521801608021526784557341623514036254592659346611879312909671154560714007925689265278125003895578164198431009696361852192
320868384712958781139142732338000179431648589491341554217478891893283267388259096343924802328042809492323557169063706958298525631984458241807273797803819140065946170391341806746785651714218149
438841680922590873949747422323265869895780128023865075702587866890156171624024474183291097015303413638450741557866981359904890002833425277906499728369359149465929680366521291156607271511026707
81077237264619493095959

RSA (classic).
Cipher : 5364450643590616270793705022514730039856256406898603887852570094290885267631601171722307511593018092314730679596409113908470082943456421858207756800798736179351705226627360724180240716430
290955790846099974072081310232625169437700930046113249649881132693206137363232418157496001284481329018245055192526161142175894474245348452757087083159917379576752454041288643907755893560870
3031644855115622231153061314442626517993007964061081991680392368076664154226728345386354794737564987017828979555859648445148684984117330545136757080417078570180816444921585667170839410596417203
63385084756423519151459353891
Signature : 5609980589381635768346122020191670171744007369284175966079306984644014753405543922185713928395805563922050188438144780914354362993928124268595963947211503434627440634742311219698609
1830295218761675025512597146112469458175464969967717276729112911706838745768826574374072506646870215702420083767596220761930606056080161559891210066616691882377811057274556244
794354924299123866819590940827825156088407885380828629602811544251773454993205770625237582154733136892268551187519659147743845393367932732481889269134850620337273066683247791198928656948170781
29727697525583867762468563050
Signature Status : OK!
Decipher : 9127220624715384251095356679128470577816478934950681624536313390186335291096529986613422466725909781580230240778759947434768612273902798220372055079985080676969283713828594875229676
473869993135161209626306234252744641034861300744256043460391214058733575741881913262649115095873835154506855982819932626
Execution time : 9.048000 ms

RSA using CRT.
d_p = 41890744080410229368496882027462595437188116566591988718105941431019552419743200418685154709825089517858246754929765576088122978681570382209211878705454665195068474392969181202804166697870418
94040348125047773881680972988073675672364774534623564782642924418986887050098549896245735976258996126897251985969, d_q = 174241591460319570066197081321806653280182187642175698831202841565477929774
734727611720593062464264923909182588378978077082752973580905689805748625749048110866808790209533691486608489343271507940849528594971870851593363569936711588741662236753382451611613912845755
7717163572904049628888176785305, d_p = 12372307850815663525824720137631072768689721508719808056770243532336802784074584081530286257181009232338734059494723462180534617431915922299842
2154568370279200800128292364626407706038599909756707668134431388420097684277326107255657571519716359074735998467160453227239940784283091458431224
Cipher : 5364450643590616270793705022514730039856256406898603887852570094290885267631601171722307511593018092314730679596409113908470082943456421858207756800798736179351705226627360724180240716430
290955790846099974072081310232625169437700930046113249649881132693206137363232418157496001284481329018245055192526161142175894474245348452757087083159917379576752454041288643907755893560870
3031644855115622231153061314442626517993007964061081991680392368076664154226728345386354794737564987017828979555859648445148684984117330545136757080417078570180816444921585667170839410596417203
63385084756423519151459353891
Signature CRT rev : 5609980589381635768346122020191670171744007369284175966079306984644014753405543922185713928395805563922050188438144780914354362993928124268595963947211503434627440634742311219698609
1830295218761675025512597146112469458175464969967717276729112911706838745768826574374072506646870215702420083767596220761930606056080161559891210066616691882377811057274556244
575642479435429429912386681959094082782515608840788538082862960281154425177345499320577062523758215473313689226855118751965914774384539336793273248188926913485062033727306668324779119892865694
810798129727697525583867762468563050
Signature Status : OK!
Decipher : 9127220624715384251095356679128470577816478934950681624536313390186335291096529986613422466725909781580230240778759947434768612273902798220372055079985080676969283713828594875229676
473869993135161209626306234252744641034861300744256043460391214058733575741881913262649115095873835154506855982819932626
Execution time : 4.126000 ms

Conclusion : CRT is 219% Faster than classic RSA.

```

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  $\phi$  et des parties de clés  $e$ ,  $n$  et  $d$ .

Avec ces éléments, vient le RSA (classic).

Le résultat du calcul du chiffré (cipher = message<sup>e</sup> (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 avons 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