

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 getModulus_ui(mpz_t res, mpz_t a, int b) . . . . .	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). 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 `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)` qui 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  $res$ .

```
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 }
```

## 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  $q$  a l'en-tête 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 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 générés d'abord 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`

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



### 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  $d$ .

```

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 {
3     mpz_t message, m_p, m_q, n, _loc0, pq, _loc1;
4     mpz_inits(message, m_p, m_q, n, _loc0, pq, _loc1, NULL);
5     mpz_set_ui(message, 1);
6     mpz_mul(n, p, q);
7     decrypt(m_p, cipher, d_p, p); // m_p = cipher ^ d_p % p
8     decrypt(m_q, cipher, d_q, q); // m_q = cipher ^ d_q % q
9     mpz_sub(pq, m_q, m_p); // pq = m_q - m_p
10    mpz_mul(_loc0, pq, i_p); // _loc0 = pq - ip
11    getModulus(_loc1, _loc0, q); // _loc1 = _loc0 - q
12    mpz_mul(message, _loc1, p); // message = _loc1 * p
13    mpz_add(message, message, m_p); // message = message + m_p
14    getModulus(message, message, n); // message = message + n
15    mpz_set(msg, message);
16    mpz_clears(message, m_p, m_q, n, _loc0, _loc1, pq, NULL);
17 }

```

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 simple 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 été également 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, ou non, 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.

```

kali@kali: ~/Bureau
Fichier Actions Éditer Vue Aide

(kali@kali) [~/Bureau]
$ ./rsa 2048
RSA à jeu réduit d'instructions pour n = 2048, message : 912727206247153842510953566791284705778164708934950681624536316390106335291090652998666134224667259097815802302407787599474347686127239027982
20372655079958506769692831738285694075229676473869993135161289626306234252744641034861300744256043460391214058733575741881913262649115095873831515406055982819932626.

p = 3328964469336779241193546769717887245927852260873619396704075310241873215342877425226066325355774089646993958503494660555493375213460387278345081783913961995863508458702584254136663844887029370
43971783765315820157380747810578977639198504151282231740035083967770536750385188008216823907527904591786734247217
n = 28485508927217538912701810741038754241251746954231562498255384508599258721981387035745654532692789120550077687320087754723015059963487788385706261980564187628812604290631601357084154202352401
965296745147065344500365135234669395915720980690321871385478553614864882510021027741712425544833118132329246158419
e = 65537
m = p * q = 94827250185758016858688367520818434001915544145428359581390799372776957909764746734212631487487865954621668075694240384476184635974246214940778562122089554374926422974746261589423856798
76531578043836784297833071692988585330277512819782968261516563631584961826844001850095901969531104512043688846701923520031898469281787993369315891418913233655762943005110788345637536558353496
467147199289146436684092460147237059823070476495022232147960105627240386962872339471419491627550636395324537541078882630866117951924881707829725199661621854235213660553544674134596364380931335280931
22263854672199101798476933769923
phi = 948272501857580168586883673029818434001915544145428350581390799373776957999764766734212631487487865954621668075694240384476184635974246214940778562122089554374926422974746261589423856798765312
6700438378459783307169298858533027751281978296826151656363135849618268440018500959019695311045120436888467019229022803526342857963281247993074710298295682741349562942212624462489310947010585921
671191377552015304784707375167558737467149650815269443425095861068346155979046428634337880901576538416891102263515351517013356344569318592024495465289914574093558883052028718106812948246773723366
05219838079066360953264288
d = 4113468564713657891609740951477956373814880085372063837268174545539945218016080215287084557141623154836254526559246611879312009671154560714007925689306527815960309557816419843108966361852192
23886384712958711391427123000811941164888949913241554217478091092022262306250963639248022280420094922323551690637069582893256319844582418072377297803915140065961702914816874678565714218149
4388416809225980739497474223236558698957801288238650750258788689015561718240244741832910970153034341563845074155786698013399590489000283542527790649972836935914946592960366521291156067271511026707
810775237648619490369569

RSA (classic).
Cipher : 53644506435906162707937050225147300398562564068986038878525700942908852676316011712237015159301809233143706795696409113908470002943456421058207756800790873617931705226627360724180240716430
298095579084690997407208131022628216934377009300461132965490811126939208173363234181574960012624481329012450851923361114217589474245348452778070831599173795767524204128864390775893566070
30316448551156522211530611442691579930079606208199168030238601766641542267203433863547947378564987017828979956582996484814860498417330545136758084170750571088164449215856671708393410596417203
63385084756423519511459353891
Signature : 960998589381635768346122020196197017174400736928417596607930698464014753405543922185713928395985085563922050188438144780914354362993928124268595963947214150343627440634742311219698609
1030295211876167502551259714611246945815464960967717267291129147063874576882365743744073503664680793157834209937667596220376193606605608161555998156683599112006666166910882357011057337455764244
794354924299123866819950946082702515608840708538082862960281154425177345499320577062523758281254733136892268551187551965914774384539336793273248188926913485062033727370666832477911989286569481707981
29727269752855583867762468563050
Signature Status : OK!
Decipher : 9127272062471538425109535667912847057781647089349506816245363163901063352910906529986661342246672590978158023024077875994743476861272390279822037205507998560506769692831738285694075229676
473869993135161209626306234252744641034861300744256043460391214058733575741881913262649115095873831515406055982819932626
Execution time : 9.048000 ms

RSA using CRT.
d_p = 41890794480401022936849688282746259543718811656659198871810594143101955241974320041868515470825089517858246754929765576088122978681570382209211878054546965195060474392969181292804166697870410
94040148125047773861609729807367567745346235647826429241898688705009854906247535976258996126897251905969, d_q = 174241591460319570866197081321806653280182187648217598831202841565477929774
724727611720530306464264923000818283835278970830770827629738590856980574805257400461160860079029953691486600948934327150794004952859497187005159336356993671158374162230753382451611613912845755
7717163857290404962688081767856305, i_p = 1237203070590175663975626582472013763107276060972915987190900567702483652336802704974584805153028625710100932783037405949447234627103653461743191599222499642
215456583702792008001282923646264077060385999907956707668134431388420097884273261072565657515710716359074735799846717160453227399407842838914548312224
Cipher : 53644506435906162707937050225147300398562564068986038878525700942908852676316011712237015159301809233143706795696409113908470002943456421058207756800790873617931705226627360724180240716430
2980955790846909974072081310226282169294377009300461132965490811126939208173363234181574960012624481329012450851923361114217589474245348452778070831599173795767524204128864390775893566070
30316448551156522211530611442691579930079606208199168030238601766641542267203433863547947378564987017828979956582996484814860498417330545136758084170750571088164449215856671708393410596417203
63385084756423519511459353891
Signature CRT rev : 9609985893816357683461220201961970171744007369284175966079306984640147534055439221857139283959850855639220501884381447809143543629939281242685959639472141503436274406347423112
196986091030295211876167502551259714611246945815464960967717267291129147063874576882365743744073503664680793157834209937667596220376193606605608161555998156683599112006666166910882357011057337455764244
557642447943549242991238668199509460827025156088407085380828629602811544251773454993205770625237582812547331368922685511875519659147743845393367932732481889269134850620337273706668324779119892865694
8170798129727269752855583867762468563050
Signature Status : OK!
Decipher : 9127272062471538425109535667912847057781647089349506816245363163901063352910906529986661342246672590978158023024077875994743476861272390279822037205507998560506769692831738285694075229676
473869993135161209626306234252744641034861300744256043460391214058733575741881913262649115095873831515406055982819932626
Execution time : 4.126000 ms

Conclusion : CRT is 219% faster than classic RSA.

```

## Conclusion

## Références