Rapport - Travail de réalisation et d'expérimentation

Florent Pergoud et Aurélien Labate
Université de Technologie de Troyes

Printemps 2015

Remerciements

Sommaire

Chapitre 1

Introduction générale

Nous allons vous présenter un compte rendu des recherches et travaux que nous avons réalisé durant le semestre de printemps 2015 dans le cadre de notre UV de TX. Notre objectif était de développer une bibliothèque de fonctions en langage C de cryptographie. Afin de traiter cette problématique nous avons du faire des rappels arithmétique sur les entiers (anneaux et corps sur les entiers) sur les polynômes (anneaux des polynôme), sur les corps de Galois et sur les structure algébriques associées (anneau/corps/anneau cyclique et élément générateur). Nous avons pu par la suite étudié les fonctions arithmétiques « de bases » comprenant les algorithmes d'Euclide étendu, d'exponentiation rapide, les calculs dans un corps de Galois, les tests de primalité des entiers et des polynômes dans Z/Z[X] <P>. Nous avons aussi étudié les opération dans un corps de Galois et la recherche d'éléments générateur d'un corps cyclique. Et enfin, nous avons pu étudier les fonction de chiffrement et ainsi les implémenter au sein d'une bibliothèque en C++.

Mais avant de commencer, demandons nous qu'est-ce que la cryptographie. La cryptographie est une discipline s'attachant protéger des messages en assurant leur confidentialité, leur authenticité et leur intégrité. Ainsi, Kerckhoffs écrivait dans le journal des sciences militaires en 1883 un principe qui devint une des bases de la cryptographie : « La sécurité d'un cryptosystème ne doit reposer que sur le secret de la clef. Autrement dit, tous les autres paramètres doivent être supposés publiquement connus. » Ce principe indique que l'on ne doit pas garder secret la manière de chiffrer l'information. Au contraire, le seul paramètre inconnu doit être la clef en elle même. Prenons pour exemple un système sécurisé qui constituerai un cadenas à code. Le principe de Kerckhoffs impliquerai qu'en rendant le mécanisme interne du cadenas publique, des experts et autres curieux pourraient tenter de trouver une faille permettant soit de retrouver la combinaison soit de forcer le cadenas. Ainsi le fabricant du cadenas pourrait corriger ces défauts jusqu'à obtenir un cadenas dont le mécanisme est connus de tous mais qui serait pourtant inviolable. Aujourd'hui, l'algorithme le plus utilisé dans l'échange sécurisé d'informations sur internet, l'algorithme RSA, fonctionne sur ce principe. Tout le monde connait son fonctionnement mais personne n'a trouvé de faille permettant de casser le chiffrement dans une période raisonnable. Ainsi dans ce genre de chiffrement, seul la clé secrète est l'élément important.

Nous avons aborder un peu plus haut la question de clé. En cryptographie, une clé peut être un mot, une phrase, une suite d'opération ou autre, qui sert de paramètre d'entrée à un algorithme de chiffrement ou de déchiffrement. Dans la cryptographie moderne, la clé est généralement constituée d'un grand nombre dont la longueur croissante assure une croissance sur la sécurité du message chiffré. Il existe deux types de clé : les clé publiques et les clefs privées. La distinction est simple, une clé publique est une clé qui est connue de tout le monde,

son propriétaire va même surement s'occuper lui-même de la publier. La clé privée au contraire est un secret bien gardé par son propriétaire. Elle ne doit en aucun cas être retrouvée par une autre personne. Ces deux types de clé ne sont pas tout le temps employées. En effet il existe deux grandes manières de transmettre des informations chiffrées. Utiliser un système de cryptographie symétriques ou asymétrique. La cryptographie symétrique se base sur une clé secrète qui est connue par les deux personnes qui communiquent. Dans le cas de l'asymétrique, on utilise deux clés. Une clé publique va servir à chiffrer les messages et une clé privé sert à décoder ces messages. Seulement le problème avec la cryptographie symétrique c'est qu'il faut se communiquer la clé secrète sans que quelqu'un d'autre puisse l'intercepter. C'est pourquoi on utilise généralement la cryptographie asymétrique dans un premier temps pour convenir d'une clé symétrique que l'on utilisera par la suite. Pour illustrer ce principe, prenons un exemple. Imaginons que Bob et Alice souhaitent s'échanger des informations confidentielles. Alice va générer un couple clé privé, clé publique à l'aide d'un algorithme ou d'une fonction. Elle va ensuite envoyer à Bob la clé publique. Ici, il n'y a aucune crainte quant à la sécurité du message contenant la clé publique car si quelqu'un d'autre obtenait cette clé il ne pourrait qu'envoyer des messages codés à Alice et rien de plus. Par la suite Bob va envoyer une clé commune en la chiffrant grâce à la clé publique d'Alice. Le processus n'est pas inversible avec la clé publique. Ainsi Bob peut envoyer le message à Alice sans craindre que quelqu'un l'intercepte. Une fois ce message chiffré reçu, Alice peut le déchiffrer grâce à sa clé privée. Elle obtient donc la clé commune que Bob lui a envoyé. Par la suite Alice et Bob utiliseront cette clé commune pour chiffrer et déchiffrer les messages qu'ils s'enverront. Cette manière de faire est utilisée partout aujourd'hui dans le monde de l'informatique lorsque l'on veut établir une connexion sécurisée. Cependant il existe différents algorithme pour faire de l'asymétrique ou du symétrique. Ils reposent sur des fonctions mathématiques et des résultats obtenus par des mathématiciens au cours du temps. Nous allons détailler par la suite les différents algorithmes étudiés pour construire notre bibliothèque.

Chapitre 2

Travail réalisé

2.1 La bibliothèque GMP

Pour réaliser notre bibliothèque, nous avons choisis de nous baser en partie sur la bibliothèque libre de droit GMP. Cette bibliothèque permet de faire des calculs sur des entiers signés, des nombres rationnels ou encore des nombre à virgule. La précision des nombres que l'on peut utiliser ne dépend que de la mémoire disponible dans la machine qui fait tourner le programme. Ainsi nous avons pu dépasser la limite que nous impose de base le langage C sur les nombres entiers soit de 0 à $2^{64} - 1$. Ce nombre peut sembler déjà grand mais nous allons utiliser par la suite des nombres de l'ordre de 2^{1024} nous permettant d'avoir des nombres d'une taille comparable à celle utilisée en cryptographie. Nous avons cependant trouvé que le langage C était limitant et que le C++ serait plus adaptée à nos besoins.

2.2 Exponentiation rapide

2.2.1 Principe

Nous avons travaillé sur les puissances par le biais de l'exponentiation rapide. Pour calculer des puissances de très grand nombre avec de grands exposants, la méthode utilisée couramment n'est pas la plus efficace. En effet, lors du calcul de n^p, le nombre d'opération est de l'ordre de p car l'on effectue $p \times p \times p \dots n$ fois. Ainsi l'algorithme équivalent de a^p est :

```
// Le type mpz2_class permet de manipuler des grands nombres entiers (avec GMP)
mpz2_class puissance_classic(mpz2_class a, mpz2_class p)
{
    mpz2_class resultat = 2;

    // Boucler p fois
    for ( ; p > 0 ; p--)
    {
        resultat = resultat * a;
    }

    return resultat;
}
```

Il existe cependant une méthode plus efficace qui abaisse le nombre d'opération à $\log_2(p)$, il s'agit de l'exponentiation rapide. Pour expliquer le principe de cette méthode, nous prenons un exemple. On souhaite calculer 3^9 :

— Méthode classique : $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 = 19683$ soit 9 opérations.

— On remarque cependant que $3^9 = 3^{1+2\times4} = 3 \times (3 \times 3)^3$
— On peut pousser cette méthode jusque $3 \times \left(\left(\left(3^2\right)^2\right)^2\right)$ soit 4 opérations :
  — $A = 3 \times 3 = 9$
  — $A = A \times A = 81$
  — $A = A \times A = 6561$
  — $A \times 3 = 19683$
— Le nombre d'opration est de l'ordre de $\log_2(9) = 3,17 \approx 4$

2.2.2 Application

L'algorithme consiste donc à décomposer le nombre en puissance de 2. Il y a trois cas possibles pour le calcul de n^p :
— $p = 1 \Rightarrow n^p = n$
— p pair $\Rightarrow n^p = (n * n)^{p/2}$
— p impair $\Rightarrow n^p = (n * n)^{(p-1)/2} \times n$

Ainsi il est possible de réaliser un algorithme récursif utilisant cette méthode, voici celui implémenté dans notre bibliothèque :

```
mpz2_class puissance(mpz2_class a, mpz2_class p)
{
    if(p <= 1) // Quand il n'y a plus de calculs àfaire
    {
        if(p <= 0)
            return 1;
        else
            return a;
    }
    else if(p%2 == 0) // Si p est pair
    {
        return puissance(a*a, p/2);
    }
    else // Si p est impair
    {
        return puissance(a*a, (p-1)/2) * a;
    }
}
```

2.2.3 Comparaison des temps de calcul

Nous avons effectué des comparaisons de temps d'exécution de ces algorithmes. Il est possible de le faire à nouveau depuis le fichier d'exemple de la librairie. Voici un résultat obtenu, il est à prendre comme ordre d'idée de la différence de rapidité :

```
Operation :
    1481892471289742891274840219830132398 23 ^ 20000
Classic :
    Duration : 4.29346 s
    Result end : 6307f34d
Squaring :
    Duration : 0.1142 s
```

```
    Result end : 6307f34d
GMP Internal :
    Duration : 0.030614 s
    Result end : 6307f34d
```

Nous pouvons voir que la méthode « Squaring » (l'exponentiation rapide) est 39 fois plus rapide que la méthode classique. Sur plusieurs essais on arrive à une moyenne de 39,2 fois moins de temps pour l'exponentiation rapide. Il est intéressant de noter par ailleurs que la fonction interne de GMP qui effectue le même calcul est encore plus rapide, il exhiste d'autres méthodes et améliorations possibles pour l'algorithme de calcul des puissances, la preuve en est donnée ici par cette troisième méthode qui va 140 fois plus vite que la méthode dite classique et 4 fois plus vite que l'exponentiation rapide.

2.3 Test de primalité de Rabin-Miller

2.3.1 Principe

Il existe plusieurs manières de tester la primalité des nombres. Deux grandes méthodes s'opposent. Les méthodes probabilistes et les non-probabilistes. Ces dernières concsitent généralement à tester d'une manière plus ou moins complexe tous les nombres qui pourraient diviser l'entier dont on teste la primalité. Au contraire, les algorithme probabilistes ne testent pas tous les valeurs possibles de leurs tests. Ainsi nous n'avons jamais une certitude à 100% que le nombre tiré soit premier. Il est cependant possible d'avoir de très très faibles chances de se tromper. L'algorithme de Rabin-Miller est un test de primalité probabiliste. Il faut donc lui donner en entrée le nombre à tester mais aussi le nombre de test à effectuer sur ce nombre. La probabilité de se tromper sur l'affirmation « le nombre est probablement premier » alros qu'en réalité il ne l'est pas est de 4^{-k} où k est le nombre de test que l'on applique au nombre. On se rend compte que l'on a pas besoin de tester beaucoup de fois le nombre pour savoir s'il est probablement premier ou non.

Le principe de l'algorithme de Rabin-Miller est que si :

$$a^d \neq 1 \pmod{n} \quad \text{et} \quad \forall r \{0, 1, \cdots, s-1\} \, a^{2^r d} \neq -1 \pmod{n}$$

2.3.2 Application

Ainsi nous avons l'algorithme suivant qui permet de tester la primalité d'un nombre :

Entrée

$n > 2$: un entier dont on teste la primalité

k : un paramètre qui définit la précision du test

Sortie

composé : si n est un nombre composé

probablement premier : si les k tests n'ont pas montré qu'il était composédu test

Algorithme

Écrire $n-1$ sous la forme $2^s \times d$ avec d impair.

Pour cela, diviser $n-1$ par 2 (s fois) jusqu'à ce que le quotient d soit impair.

***BOUCLE* : répéter** k fois :

Tirer un nombre aléatoire a compris dans l'interval $[2, n-1]$

$x \leftarrow a^d \mod n$

si $x = 1$ **ou si** $x = n-1$: **aller à la prochaine** *BOUCLE*

pour $r = 1 \cdots s-1$

$x \leftarrow x^2 \mod n$
si $x = 1$: **retourner** *composé*
si $x = n - 1$: **aller à la prochaine** *BOUCLE*
retourner *composé*
retourner *probablement premier*

2.3.3 Comparaison des temps de calcul

Nous avons ici aussi effectué un test de rapidité selon plusieurs algorithmes, voici leurs résultats :

```
Ce programme essai de vérifier si un nombre est premier avec l'algorithme de
    Rabin-Miller. Le nombre suivant sera testé 5000 fois :

  9b83c987e7277d74c350556297556cf013f5d8d1e334b3af169303569b7b8f30
  968e19cc340d1dfed4a382d072eda1c33a46e864310dc86c175aa9069f61c38a
  d518272ced4cd7ca7d7ef27b172b0cfd1b26b4d5241154dfc146325fd056771f
  7bd002b59c67b3bbee16826faced5b7a104b5f64d3331d31f212174039075fe3

Simple Rabin-Miller :
  Duration : 10.0266 s
  Probably prime
GMP Internal :
  Duration : 9.90365 s
  Probably prime
```

Ici encore la bibliothèque GMP est plus rapide. Et pourant la fonction de GMP fait plus que notre fonction de Rabin-Miller. En effet si un nombre à tester a déjà été tiré au hasard, la fonction tire un autre nombre à la place alors que la notre tesera à nouveau cette valeur en faisant faussement augmenter la probabilité de primalité.

De plus si tous les nombres ont été testés, alors la fonction retourne 2 soit *le nombre est premier* car en testant tous les nombres de 2 à $n - 1$ on passe à un test non probabiliste et ainsi on peut affirmer avec certitude que n est premier.

Il faut aussi noter que dans notre exemple nous avons pris $k = 5000$ ce qui est un nombre démesurément grand par rapport aux besoins du test. Avec $k = 100$ on a déja une probabilité de se tromper en affirmant que le nombre est premier alors qu'il ne l'est pas de $4^{-100} = 6,2230153.10^{-61}$.

2.4 Diffie-Helmann

2.4.1 Principe

L'échange de clé Diffie-Hellman est un système de cryptographie asymétrique. Il permet à deux personnes (prenons par exemple Alice et Bob) de se transmettre d'une manière sécurisée une clé commune. Pour expliquer son fonctionnement nous allons au préalable faire quelques rappels mathématiques.

Nous travaillons dans l'anneau Z/Z_p qui est un cas particulier d'anneau commutatif, il correspond au calcul de module sur les restes de la division entière par p. Or les nombres p que nous utilisons sont uniquement des nombres premiers, ainsi Z/Z_p est dans notre cas un corps fini et donc commutatif.

Nous allons maintenant faire un exemple pour illustrer cet échange de clé. Nous allons garder nos deux protagonistes Alice et Bob.

— **Première étape** : Alice et Bob vont se mettre d'accords sur un nombre premier p et une base g de l'ensemble Z/Z_p. Pour cela ils n'ont pas besoin d'un canal de communication sécurisé. Dans un même temps Alice et Bob vont choisir chacun de leur côté un nombre secret a pour Alice et b pour Bob.

— **Deuxième étape** :
Alice calcule :

$$A = g^a \mod p$$

De son coté Bob calcule :

$$B = g^b \mod p$$

Ils s'envoient chacun ce résultat.
Alice connait donc a, A et B et Bob connait b, B et A.

— **Troisième étape** :
Alice calcule :

$$\begin{aligned} B^a &= \left(g^b\right)^a \mod p \\ &= g^{ab} \mod p \\ &= S \end{aligned}$$

Bob calcule de la même manière :

$$\begin{aligned} A^b &= (g^a)^b \mod p \\ &= g^{ba} \mod p \\ &= S \end{aligned}$$

On voit donc qu'à l'issue de ces deux calculs, Alice et Bob ont tous les deux une clé secrète en commun S. L'avantage de cet échange de clé est qu'il peut être fait sur un canal non sécurisé. En effet une personne qui écouterait les conversations pourrait obtenir :

$$A = g^a \mod p$$

et

$$B = g^b \mod p$$

Or le problème du logarithme discret rend impossible avec des nombres de départs p, a et b suffisamment grand de retrouver depuis A et B les valeurs a et b même avec une recherche exhaustive.

2.4.2 Application

Dans notre bibliothèque nous avons créer deux fonctions correspondant aux étapes 2 et 3 de l'échange de clé :

```
mpz2_class diffieHellman_etape1(mpz2_class a, mpz2_class p = 0, mpz2_class g = 0)
{
    // Si non spécifiés, les p et g utilisés
    // seront ceux inclus dans la librairie
    if (p == 0)
```

```
    {
        p = Crypto::dh_p;
    }
    if (g == 0)
    {
        g = Crypto::dh_g;
    }

    // On calcule g^a mod p
    mpz2_class A;
    A = g.powmod(a,p);

    return A;
}
```

```
mpz2_class diffieHellman_etape2(mpz2_class a, mpz2_class B, mpz2_class p)
{
    // Récupération du p par défaut
    if (p == 0)
    {
        p = Crypto::dh_p;
    }

    // On calcule B^a mod p.
    mpz2_class S;
    S = B.powmod(a,p);

    return S;
}
```

Nous avons choisi de mettre des valeurs par défaut pour p et g dans le cas où les utilisateurs ne voudraient pas chercher des couples p et g. Ce couple choisi est directement issu d'une norme sur l'échange de clé de Diffie-Hellman. En effet, les paramètres p et g n'ont pas besoin d'être tenus secret. C'est pourquoi il sont normalisés et peuvent être trouvés en ligne.

2.5 ElGamal

2.5.1 Principe

Le cryptosystème de ElGamal est un algorithme basé essentiellement sur le même concept que l'échange de clé de Diffie-Hellman dont le but ne serait pas d'obtenir une clé commune aux deux personnes qui se parlent mais plutôt pour une personne d'envoyer un message à un autre. Prenons de nouveau Alice et Bob pour notre exemple :

— **Première étape** : Alice va choisir un nombre premier p et un générateur g du groupe Z/Z_p. Par la suite Alice va choisir une clé secrète a suffisamment grande pour que le calcul du logarithme discret soit infaisable. Alice va par la suite calculer :

$$A = g^a \mod p$$

Le triplet (A, g, p) sera la clé publique d'Alice. Elle va d'ailleurs la publier pour que quiconque voudrait communiquer avec elle puisse la trouver.

— **Deuxième étape** : Bob souhaite envoyer un message chiffré à Alice. Pour cela il va récupérer la clé publique d'Alice et calculer

$$\begin{aligned} A^b &= (g^a)^b \mod p \\ &= g^{ab} \mod p \\ &= S \end{aligned}$$

Il va ensuite transformer son message en un ou plusieurs nombre. Pour chaque nombre m obtenu Bob va calculer

$$M = S \times m \mod p$$

Il va par ailleurs calculer

$$B = g^b \mod p$$

Le duet (M, B) sera le message que Bob va envoyer à Alice.

— **Troisième étape** : Alice va recevoir le duet (M, B) de Bob elle va par la suite calculer :

$$\begin{aligned} B^a &= \left(g^b\right)^a \mod p \\ &= g^{ba} \mod p \\ &= S \end{aligned}$$

Ensuite elle déchiffrera le message en calculant

$$m = \frac{M}{S} \mod p$$

2.5.2 Application

Nous avons suivi les trois étapes en créant trois fonctions dans notre bibliothèque : La première est :

```
mpz2_class elGamal_generationCles(mpz2_class a, mpz2_class p, mpz2_class g)
{
    return diffieHellman_etape1(a, p, g);
}
```

Nous pouvons voir ici que la première étape d'ElGamal est exactement la même que celle de Diffie-Hellman. Cependant pour plus de lisibilité du code nous avons créé une fonction ElGamal appelant une fonction de Diffie-Hellmann.

```
mpz2_class elGamal_chiffre(mpz2_class a, mpz2_class B, mpz2_class message,
    mpz2_class p)
{
    // Récupération du p par défaut
    if(p == 0) {
        p = Crypto::dh_p;
    }

    mpz2_class cleCommune, messageChiffre;

    // Calcule de la clé commune
    cleCommune = diffieHellman_etape2(a, B, p);
```

```
    // Chiffrement du message
    messageChiffre = message * cleCommune;

    return messageChiffre;
}
```

```
mpz2_class elGamal_dechiffre(mpz2_class a, mpz2_class B, mpz2_class
    messageChiffre, mpz2_class p)
{
    // Récupération du p par défaut
    if (p == 0) {
        p = Crypto::dh_p;
    }

    mpz2_class cleCommune, messageDechiffre;

    // Calcule de la clé commune
    cleCommune = Crypto::dh_step2(a, B, p);

    // Dechiffrement du message
    messageDechiffre = messageChiffre / cleCommune;

    return messageDechiffre;
}
```

Les deuxième et troisièmes méthodes se basent aussi en partie sur du Diffie-Hellman mais cependant des modifications sont ajoutées afin de pouvoir envoyer un message.

2.6 RSA

2.6.1 Principe

2.6.2 Application

Chapitre 3

Conclusion

Annexe A

Suivi du projet