Cryptographie sur les courbes elliptiques

Nicolas Papernot

TIPE
2010-2012

Résumé

J'ai divisé en deux grandes parties le cheminement de ce TIPE :

La première partie regroupe une introduction à la cryptographie, ou plutôt aux différents types de cryptographie, ainsi qu'une introduction à la théorie concernant les courbes elliptiques. J'ai volontairement ommis de nombreux résultats théoriques très intéressants relatifs aux courbes elliptiques pour me concentrer principalement sur ceux qui sont directement utiles à la deuxième partie de mon travail.

La deuxième partie détaille l'adaptation de la théorie à la pratique. J'ai ainsi réalisé un programme qui permet de crypter et décrypter un message à l'aide d'une courbe elliptique. Cette partie du document détaille les différentes étapes de la mise en place ainsi que les difficultés auxquelles j'ai fait face.

Table des matières

Première partie

Introduction

Chapitre 1

La cryptographie

1.1 Introduction

La cryptographie étudie les procédés qui permettent l'échange de messages sans que ceux-ci ne soient compréhensibles en cas d'interception. La transmission du message se fait en général via des voies de communications non-sécurisées. Il faut donc transformer le message afin de le rendre inintelligible à toute personne étrangère aux récipients.

La cryptographie utilise un vocabulaire spécifique. Le message à transmettre est appelé *message en clair*. Avant de le transmettre, on l'*encrypte*. Le procédé qui permet d'obtenir le message de départ à l'aide du message crypté et donc le *décryptage*. La taille du message est variable, d'une à plusieurs lettres.

On divise le procédé de transmission du message en deux étapes :
- le cryptage, auquel on associe une fonction f
- le décryptage, auquel on associe la réciproque de la fonction précédente : f^{-1}.

Le cryptosystème est alors le système que l'on représente par :

$$M \longrightarrow f(M) = C \longrightarrow f^{-1}(C) = M$$

où M est le *message en clair* et C le *message crypté*.

L'algorithme qui permet de crypter et décrypter un message est en général toujours connu publiquement. C'est pour cela que les cryptosystèmes requièrent l'utilisation d'une *clé*.

1.2 Les clés

Celles-ci vont entrer en jeu dans le cryptosystème mais elles peuvent être modifiées à tout moment afin de garantir une meilleure sécurité. En modifiant régulièrement la clé utilisée, les utilisateurs de l'algorithme rendent l'interception de leur message beaucoup plus difficile. Il existe deux types de clés.

1.2.1 Les clés privées

Il s'agit de la première mise en place des clés. Elle suppose qu'il est aussi difficile de connaître le procédé qui permet de crypter que celui qui permet de décrypter. Ces cryptosystèmes classiques, également appelés cryptosystèmes symétriques, sont très faciles à décoder une fois que l'on connait le processus de cryptage.

Un exemple de cryptosystème (très simple) à clé privée est le codage à décalage utilisé par César. Avant de transmettre son message, il décalait toutes les lettres du message de 3 positions dans l'alphabet : A devient D, B devient E, ... Z devient C. Ainsi le mot MESSAGE devient PHVVDJH. Ce cryptosystème est très facile à attaquer car il suffit, en sachant que "e" est la lettre la plus fréquente, d'analyser les fréquences d'apparition des différentes lettres et ainsi d'en déduire le nombre de décalages effectués ...

1.2.2 Les clés publiques

Les clés publiques sont introduites par W. Diffie et M. Hellman en 1976. Elles permettent à deux personnes d'échanger un message de manière sécurisée sans s'être rencontrées physiquement i.e. sans avoir échangé une *clé privée*. Un cryptosystème à clé publique a pour propriété d'être difficile à casser même si l'on connait le procédé utilisé pour le cryptage. Un tel système repose sur l'existence de fonctions mathématiques appelées *trappes* qui sont très faciles à calculer mais dont l'inverse est difficile à déterminer. L'échange de clés proposé par Diffie et Hellman exploite une fonction trappe : l'élévation à la puissance.

Échange de clés Diffie-Hellman

Soient deux personnes Aline (A) et Bob (B) désirant échanger un message. Ils choisissent un groupe multiplicatif G et un générateur g de ce groupe puis réalisent les étapes suivantes :

- A choisit $a \in G$ et envoie à B le résultat de g^a
- B choisit $b \in G$ et envoie à A le résultat de g^b
- A élève g^b à la puissance a pour trouver g^{ab}
- B élève g^a à la puissance b pour trouver également g^{ab}

Aline et Bob sont donc en possession d'une clé identique grâce à l'associativité du groupe G qui donne $(g^a)^b = (g^b)^a$. Ils sont seuls à connaître cette clé car même si une troisième personne a espionné leurs échanges, elle ne connait que les éléments g^a, g^b et g. Cette personne veut déterminer g^{ab}. Elle doit donc déduire a et b de g^a et g^b. Il s'agit du problème du logarithme discret ; que l'on ne sait pas résoudre efficacement.

1.3 Le problème du logarithme discret

1.3.1 Définition

Soit G un groupe fini, b un élément de G, et y une puissance de b appartenant à G. Le logarithme discret de y en base b est un entier x tel que $b^x = y$ i.e. $x = log_b(y)$. Le calcul d'un tel entier est beaucoup plus difficile que l'élévation à la puissance. La sécurité des cryptosystèmes à clé publique repose sur cette propriété.

1.3.2 Attaques

Nous allons maintenant présenter quelques techniques qui permettent de résoudre le logarithme discret.

Méthode de D.Shanks

Méthode de Pohlig et Hellman

Chapitre 2

Les courbes elliptiques

La cryptographie sur les courbes elliptiques, souvent abrégée par ECC (Elliptic Curve Cryptography) est introduite indépendamment par Neal Koblitz et Victor Miller en 1985. La communauté scientifique était d'abord très sceptique du fait du manque de connaissances sur les courbes elliptiques. Cependant, la cryptographie sur les courbes elliptiques a fait de nombreux progrès depuis son introduction par N. Koblitz et elle trouve maintenant de nombreuses applications. Ainsi, le site de la National Security Agency indique que plusieurs membres de l'OTAN utilise de nos jours la cryptographie sur les courbes elliptiques pour protéger les informations confidentielles lors d'échanges. De plus, la cryptographie sur les courbes elliptiques est amenée à remplacer progressivement le cryptosystème RSA dans des produits tels que les téléphones ou les cartes bancaires car la puissance de calcul disponible dans de tels appareils est faible (cf. infra).

2.1 Équation de Weierstrass

Si l'on se place dans un plan, en considérant des points de coordonnées (x, y), on peut alors définir, une courbe elliptique par l'équation générale :

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

où $a_1, ..., a_6$ sont des constantes. Il s'agit d'une équation généralisée de Weierstrass. Si le corps est de caractéristique différente de 2, alors on peut diviser par 2 et former un carré dans le membre de gauche :

$$\left(y + \frac{a_1x}{2} + \frac{a_3}{2}\right)^2 = x^3 + \left(a_2 + \frac{a_1^2}{4}\right)x^2 + \left(a_4 + \frac{a_1a_3}{2}\right)x + \left(\frac{a_3^2}{4} + a_6\right)$$

On pose $y_1 = y + \frac{a_1x}{2} + \frac{a_3}{2}$, $a' = a_2 + \frac{a_1^2}{4}$, $b' = a_4 + \frac{a_1a_3}{2}$ et $c' = \frac{a_3^2}{4} + a_6$. Si le corps n'est pas de caractéristique 3, on peut diviser par 3 et poser $x_1 = x + \frac{a'}{3}$. On obtient alors une équation de Weierstrass :

$$y_1^2 = x_1^3 + ax_1 + b$$

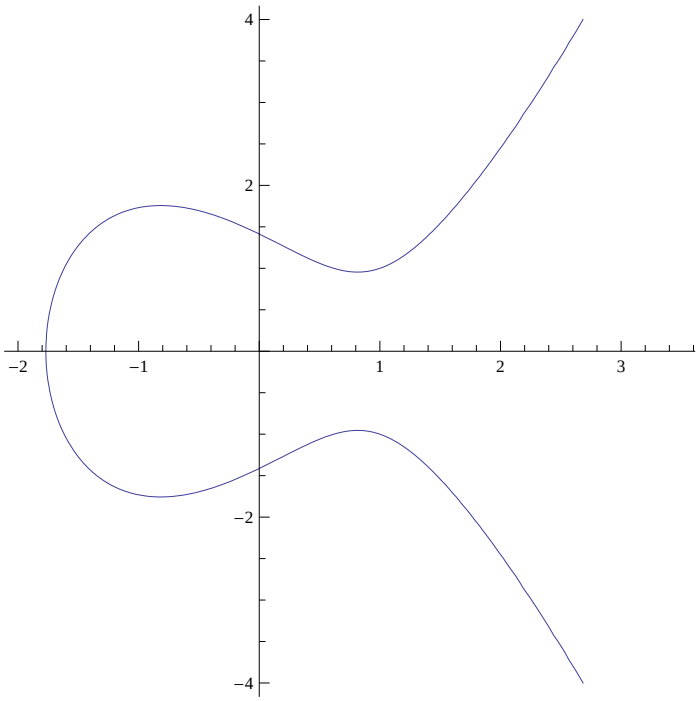


FIGURE 2.1 – Voici la courbe elliptique d'équation $y^2 = x^3 - 2x + 2$

Une courbe elliptique est l'ensemble des points :

$$E = \{O\} \cup \{(x, y) \in \mathbb{K} \times \mathbb{K}, y^2 = x^3 + ax + b\}$$

On notera l'ajout d'un point O que l'on définira plus loin lors de la mise en place de la loi de groupe (cf. infra).

2.2 Discriminant

Une courbe elliptique doit être dérivable en tout point (elle possède donc une tangente en tout point de sa courbe représentative), ne posséder aucun point double ou de rebroussement. Ces conditions sont vérifiées quand son discriminant ne s'annule pas. Il est défini par :

$$\Delta = -(4a^3 + 27b^2)$$

Si ce discriminant est négatif, alors le graphe de la courbe elliptique ne possède qu'une seule composante. Le polynôme cubique $x^3 + ax + b$ possède une unique racine qui correspond à l'abscisse du point d'intersection de la courbe avec l'axe des abscisses (cf. figure 2.1).

Si ce discriminant est positif, le graphe de la courbe elliptique possède alors deux composantes. Le polynôme cubique $x^3 + ax + b$ possède 3 racines, qui correspondent aux abscisses des trois points d'intersection de la courbe avec l'axe des abscisses (cf. figure 2.2).

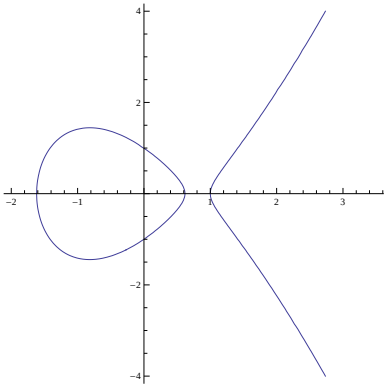


FIGURE 2.2 – Courbe de discriminant $\Delta = 5$

2.3 Loi de groupe

2.3.1 Approche géométrique

Soit $E : y^2 = x^3 + ax + b$ une courbe elliptique. Considérons deux points $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$. On construit la somme de P_1 et P_2 en 2 étapes :

- On trace d'abord la droite (D) passant par P_1 et P_2. Elle coupe la courbe en un troisième point P_3' (cf. infra).
- On considère alors le symétrique de P_3' par rapport à l'axe des abscisses : P_3. On a alors $P_1 + P_2 = P_3$

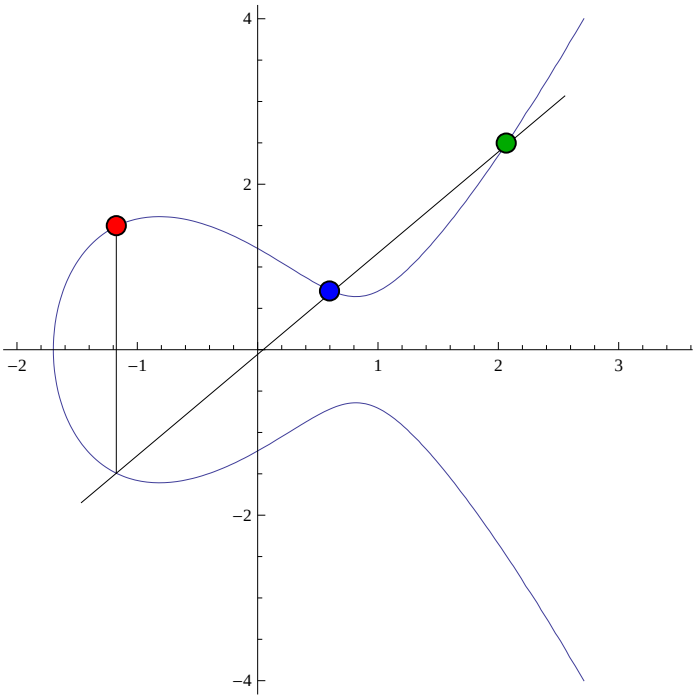


FIGURE 2.3 – P_1, P_2, P_3 correspondent respect. aux points vert, bleu, rouge

Pour certains cas particuliers, il faut apporter quelques précisions :

- Si P_1 et P_2 sont symétriques par rapport à l'axe des abscisses, la droite (D) est verticale. On considère alors que la somme des deux points est nulle. Il s'agit donc

du point à l'infini.

– Si la droite (D) est tangente en un point de la courbe, on considère qu'elle intersecte deux fois la courbe en ce point et par conséquent la somme de P_1 et P_2 est le symétrique du point de tangence (figure 2.4).

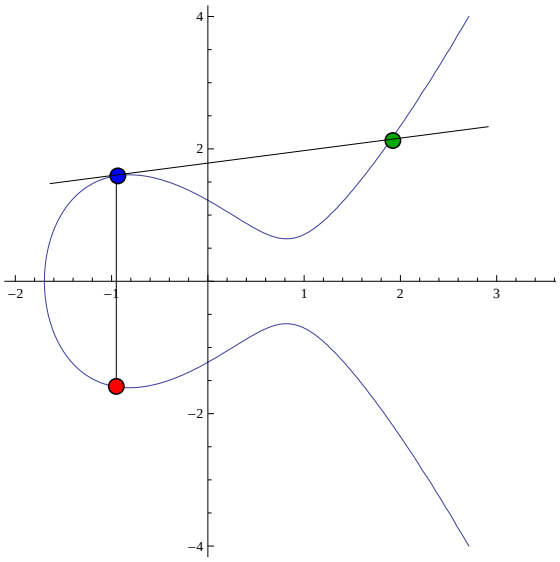


FIGURE 2.4 – (D) tangente en un point de la courbe

– Si $P_1 = P_2$ alors (D) est à nouveau la tangente à la courbe en P_1 et elle intersecte la courbe en un autre point dont le symétrique est $P_1 + P_2 = P_1 + P_1 = 2P_1$ (figure 2.5).

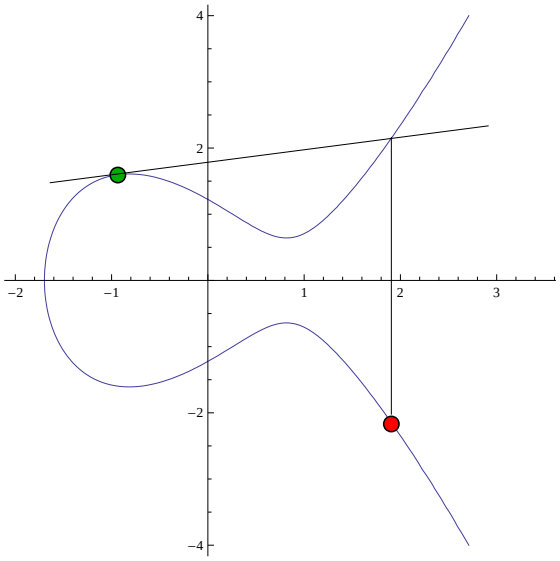


FIGURE 2.5 – Cas où $P_1 = P_2$

On a alors intuitivement défini une loi additive de groupe sur les réels.

2.3.2 Loi additive et coordonnées

Nous allons maintenant déterminer les relations entre les coordonnées de $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ et leur somme P_1+P_2. Ces relations sont indispensables afin de pouvoir appliquer la théorie à la pratique et ainsi coder une fonction addition dans un langage informatique. Il faut distinguer 4 cas.

Considérons d'abord P_1 et P_2 tels que $P_1 \neq P_2$ et qu'ils soient tous deux différents de O. La pente de la droite $(D) = (P_1P_2)$ est par définition :

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Si $x_1 = x_2$ alors $y_1^2 = x_1^3 + ax_1 + b = x_2^3 + ax_2 + b = y_2^2$. On en déduit $y_1 = -y_2$, les deux points sont donc symétriques par rapport à l'axe des abscisses. Leur somme est donc le point à l'infini i.e. $P_1 + P_2 = O$.
Supposons maintenant $x_1 \neq x_2$. L'équation de la droite (D) est alors

$$y = m(x - x_1) + y_1$$

On injecte dans l'équation de E :

$$\begin{aligned} & (m(x - x_1) + y_1)^2 = x^3 + ax + b \\ \Leftrightarrow \quad & 0 = x^3 - m^2x^2 + K \end{aligned}$$

avec $K = (2m^2x_1 - 2y_1m + a)x - m^2x_1^2 + 2y_1mx_1 + b - y_1^2$. x est donc la racine d'un polynôme du troisième degré dont nous connaissons déjà deux autres racines : x_1 et x_2. On remarque alors que pour tout polynôme de la forme $x^3 + ax^2 + bx + c$ et admettant pour racines r, s, t, alors on a : $x^3 + ax^2 + bx + c = (x - r)(x - s)(x - t) = x^3 - (r + s + t)x^2 + (rs + rt + st)x - rst$. On a alors $-a = r + s + t$ et donc en connaissant deux racines (par exemple r et s), on peut en déduire la troisième : $t = -a - r - s$. Dans notre cas, on obtient alors

$$x = -(-m^2) - x_1 - x_2$$

On en déduit :

$$\begin{cases} x_3 = m^2 - x_1 - x_2 \\ y_3 = -\left(m(x_3 - x_1) - y_1\right) = m(x_1 - x_3) - y_1 \end{cases}$$

où x_3 et y_3 sont les coordonnées de $P_1 + P_2 = P_3$.

Considérons maintenant que les deux points P_1 et P_2 soient confondus. On sait que lorsque deux points situés sur une courbe sont très proches, la droite qui passe par ces points s'approche d'une tangente. Ainsi on pose (D) tangente en P_1. On peut supposer que y_1 est non-nul car si c'est le cas, le point P_1 est une intersection de E et de l'axe des abscisses et par conséquent (D) est verticale. On pose alors $P_1 + P_2 = 2P_1 = O$ On a alors en dérivant l'équation de (E) : $2y\dfrac{dy}{dx} = 3x^2 + a$. On en déduit la valeur de la pente m de (D) :

$$m = \frac{dy}{dx}(x_1, y_1) = \frac{3x_1^2 + a}{2y_1}$$

Comme on a supposé $y_1 \neq 0$, le dénominateur est non-nul. On peut donc en déduire l'équation de (D) :

$$y = m(x - x_1) + y_1$$

On procède comme pour le cas précédent sachant que x_1 est racine double car (D) est la tangente à E en P_1. On obtient

$$\begin{cases} x_3 = m^2 - 2x_1 \\ y_3 = m(x_1 - x_3) - y_1 \end{cases}$$

Considérons enfin que $P_2 = O$. La droite (D) est d'équation $x = x_1$, elle possède donc deux intersections avec E : P_1 et son symétrique. La somme $P_1 + P_2$ est le symétrique de ce second point, il s'agit donc du point P_1 lui même.

Synthèse Prenons deux points $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$:

- Si $x_1 \neq x_2$ alors on pose $m = \dfrac{y_2 - y_1}{x_2 - x_1}$ et :

$$\begin{cases} x_3 = m^2 - x_1 - x_2 \\ y_3 = m(x_1 - x_3) - y_1 \end{cases}$$

- Si $x_1 = x_2$ avec $y_1 \neq y_2$, alors $P_1 + P_2 = O$.
- Si $P_1 = P_2$ avec $y_1 \neq 0$, alors on pose $m = \dfrac{3x_1^2 + a}{2y_1}$ et on a :

$$\begin{cases} x_3 = m^2 - 2x_1 \\ y_3 = m(x_1 - x_3) - y_1 \end{cases}$$

- Si $P_1 = P_2$ avec $y_1 = 0$, alors $P_1 + P_2 = O$.
- Si $P_2 = 0$ alors $P_1 + P_2 = P_1$.

Nous verrons plus tard que ces formules nous permettrons de coder l'addition de manière algorithmique.

2.3.3 Approche algébrique

On a définit géométriquement une loi de composition interne sur les points de E qu'on a munie de la notation additive habituelle $+$. On a également définit un élément neutre noté O. On va maintenant montrer que $(E, +)$ est un groupe abélien.

Théorème *L'ensemble des points d'une courbe elliptique E forment un groupe commutatif.*

Démonstration Il faut prouver que la loi possède un élément neutre, qu'il existe un inverse pour tout point de E, qu'elle est associative et commutative

1. L'existence de l'élément neutre provient du fait que l'ensemble des points $P \in E$ vérifient par définition $P + 0 = P$.
2. Pour tout point $P \in E$ de coordonnées (x, y), on sait que son symétrique $P'(x, -y)$ vérifie : $P + P' = 0$. On note donc $-P = P'$ qui est l'inverse. Il existe donc pour tout $P \in E$ un inverse.
3. Il nous faut maintenant prouver l'associativité i.e. que pour trois points de E, on a :

$$(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$$

Celle-ci peut se prouver à l'aide de différents résultats. La première preuve utilise le théorème de Riemann-Roch. On peut également prouver l'associativité à l'aide des formules déterminées dans le paragraphe précédent. Les calculs étant fastidieux et sans réel intérêt, nous allons ici montrer l'associativité de cette loi de groupe de manière géométrique.

Nous allons admettre le théorème de Max Noether car c'est une de ses conséquences (le théorème des neuf points) qui nous intéresse :
Soit E une courbe qui intersecte deux courbes cubiques C et C' chaque fois en 9 points. Si 8 de ces points sont identiques pour C et C' alors le 9^e l'est également.

Pour utiliser ce théorème, nous allons donc chercher à construire deux cubiques C et C' de manière à avoir pour 9^e point d'intersection avec E respectivement $(P_1+P_2)+P_3$ et $P_1 + (P_2 + P_3)$.

Commençons par définir quelques droites :
– Δ_1 la droite passant par les points P_1 et P_2
– Γ_1 la droite passant par les points $P_1 + P_2$ et O
– Δ_2 la droite passant par les points $P_1 + P_2$ et P_3
– Γ_2 la droite passant par les points P_2 et P_3
– Δ_3 la droite passant par les points $P_2 + P_3$ et 0
– Γ_3 la droite passant par les points $P_2 + P_3$ et P_1
Le produit des équations des droites Δ_1, Δ_2, et Δ_3 est l'équation d'une courbe algébrique du 3^e degré : il s'agit donc d'une cubique que l'on nomme C. De même on définit C' à l'aide du produit des équations de Γ_1, Γ_2 et Γ_3. Par construction C possède 9 points d'intersection avec la courbe E :

$$\{P_1; P_2; P_3; O; P_1 + P_2; -(P_1 + P_2); P_2 + P_3; -(P_2 + P_3); -((P_1 + P_2) + P_3)\}$$

De même C' possède 9 points d'intersection avec la courbe E :

$$\{P_1; P_2; P_3; O; P_1 + P_2; -(P_1 + P_2); P_2 + P_3; -(P_2 + P_3); -(P_1 + (P_2 + P_3))\}$$

Les deux cubiques ont donc 8 points d'intersection avec la courbe E en commun. Les 9^e points sont : $-((P_1 + P_2) + P_3)$ et $-(P_1 + (P_2 + P_3))$. D'après le théorème des 9

points, on en déduit que $-((P_1 + P_2) + P_3) = -(P_1 + (P_2 + P_3))$. Leurs symétriques respectifs sont donc également égaux. On a donc :

$$(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$$

Ce qui prouve l'associativité.

4. On remarque finalement que la commutativité de la loi + découle du fait que pour tous points $P_1, P_2 \in E$ la droite (P_1P_2) est confondue avec (P_2P_1) et que par conséquent $P_1 + P_2 = P_2 + P_1$.

On a bien montré que $(E, +)$ est un groupe abélien.

2.4 Choix d'un corps

On se place dans un corps $\mathbb{Z}/p\mathbb{Z}$. Il s'agit d'un corps de dimension finie. On va donc réaliser tous les calculs modulo p, c'est-à-dire que l'on considère uniquement les points d'abscisse entière et inférieure ou égale à p.

Ensemble de points Déterminons l'ensemble des points d'une courbe donnée par une équation mise sous la forme réduite de Weierstrass i.e. $y^2 = x^3 + ax + b$ en se plaçant dans le corps $\mathbb{Z}/p\mathbb{Z}$. Pour cela, on considère $x \in [\![1, p]\!]$. Pour chaque valeur de x, on calcule $x^3 + ax + b$ modulo p. On en déduit s'il existe un ou plusieurs entier y vérifiant $y^2 = x^3 + ax + b$. Si c'est le cas, ces valeurs de $y \in [\![1, p]\!]$ nous donnent l'ordonnée des points d'abscisse x appartenant à la courbe.

Etudions l'exemple de la courbe $(E) : y^2 = x^3 + 3x + 4[6]$:
- Si $x = 0$, alors $y^2 \equiv 4[6]$. Donc $y \in \{2, 4\}$.
- Si $x = 1$, alors $y^2 \equiv 2[6]$. Donc il n'existe pas de valeur entière pour y.
- Si $x = 2$, alors $y^2 \equiv 0[6]$. Donc $y = 0$.
- Si $x = 3$, alors $y^2 \equiv 4[6]$. Donc $y \in \{2, 4\}$.
- Si $x = 4$, alors $y^2 \equiv 2[6]$. Donc il n'existe pas de valeur entière pour y.
- Si $x = 5$, alors $y^2 \equiv 0[6]$. Donc $y = 0$.

On en déduit que l'ensemble des points (qui contient le point à l'infini) que l'on considèrera pour la loi de groupe est :

$$\{(0, 0); (0, 2); (0, 4); (2, 0); (3, 2); (3, 4); (5, 0)\}$$

Deuxième partie

Cryptosystème d'une courbe elliptique

Chapitre 3

Fonctions préliminaires

3.1 Introduction

Cherchant à élaborer un programme permettant de transmettre simplement un message de manière sécurisée en utilisant les courbes elliptiques, j'ai retenu le langage C. Ce programme exécute 3 tâches successives : il commence par réaliser un échange de clés en utilisant le protocole Diffie-Hellman, puis il code le message (un point de la courbe) et le décode.

Avant de pouvoir coder les fonctions relatives à la transmission du message, il est nécessaire d'automatiser deux opérations élémentaires : l'addition et la multiplication. Ces deux fonctions nécessitent elles-mêmes la mise en place de deux fonctions auxiliaires, la congruence et l'inversion, dont nous présenterons également le code.

3.2 Addition

Pour réaliser une fonction addition, on utilise les formules établies dans la première partie de ce document. Rappelons que nous avons distingué 5 cas pour l'addition de $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$:

- $x_1 \neq x_2$
- $x_1 = x_2$ avec $y_1 \neq y_2$
- $P_1 = P_2$ avec $y_1 \neq 0$
- $P_1 = P_2$
- $P_2 = 0$

Nous allons bien évidemment retrouver ces 5 cas possibles dans le code de notre fonction. En fait, pour simplifier le programme, nous allons coder deux fonctions : une fonction qui renvoie l'abscisse de la somme des deux points et une deuxième fonction qui renvoie l'ordonnée de ce même point.

Voici le code source des deux fonctions :

Listing 3.1 – add.c

```
1  extern a,b,p;
2  int add_x(int x1, int y1, int x2, int y2)
3  {
4      if (x1>x2)
5      {
6          int temp=x2;
7          x2=x1; x1=temp;
8          temp=y2;
9          y2=y1; y1=temp;
10     }
11     int x3, m;
12     if(x1 == 0) {x3=x2;}
13     else if(x2 ==0) {x3=x1;}
14     else if(x1!=x2)
15     {
16         m = modulo((modulo((y2-y1),p)*inverse(modulo(x2-x1,p),p)),p)
               ;
17         x3 = modulo(((m*m) -x1 -x2),p);
18     }
19     else if((x1==x2) && (y1!=y2)) {x3=0;}
20     else if(x1==x2 && (y1==y2) && y1!=0)
21     {
22         m = modulo(((3*(x1*x1)+a)*inverse((2*y1),p)),p);
23         x3 = modulo(((m*m) - x1 - x2),p);
24     }
25     else if(x1==x2 && y1==y2 && y1==0) {x3=0;}
26 return modulo(x3,p);
27 }
28
29 int add_y(int x1, int y1, int x2, int y2, int x3)
30 {
31     if (x1>x2)
32     {
33         int temp=x2;
34         x2=x1; x1=temp;
35         temp=y2;
36         y2=y1; y1=temp;
37     }
38     int m, y3;
39     if(x1 == 0) {y3=y2;}
40     else if(x2 ==0) {y3=y1;}
41     else if(x1!=x2)
42     {
```

```
43          m = modulo((modulo((y2-y1),p)*inverse(modulo(x2-x1,p),p)),p)
                ;
44          y3 = modulo(((m * modulo((x1-x3),p)) -y1),p);
45      }
46      else if((x1==x2) && (y1!=y2)) {y3=0;}
47      else if(x1==x2 && (y1==y2) && y1!=0)
48      {
49          m = modulo(((3*(x1*x1)+a)*inverse((2*y1),p)),p);
50          y3 = modulo(((m* modulo((x1-x3),p)) -y1),p);
51      }
52      else if(x1==x2 && y1==y2 && y1==0) {y3=0;}
53  return modulo(y3,p);
54  }
```

Il s'agit d'abord de remarquer que l'on attribue au point O situé à l'infini (l'élément neutre de la loi de groupe) les coordonnées $(0, 0)$ (cf. lignes 19 et 46), ceci implique que les courbes que l'on utilise ne doivent pas passer par $(0, 0)$. Ainsi, si on considère une équation de la forme de Weierstrass $y^2 = x^3 + ax + b$, il faut que $b \neq 0$.

De plus, on remarquera que l'on classe les deux points par ordre croissant (cf. lignes 4-10 et 31-37) pour éviter d'introduire un signe négatif dans le calcul de m et des coordonnées `x3` et `y3`.

Ces deux fonctions font également appel à deux fonctions auxiliaires `modulo` et `inverse`. Celles-ci sont nécessaires car on s'est placé dans le corps $\mathbb{Z}/p\mathbb{Z}$.

3.2.1 Congruence

La première, la fonction `modulo`, comme son nom l'indique, prend en argument deux entiers x et n et retourne x modulo n. La fonction est déjà incluse dans le langage C par défaut sous la forme de l'opérateur % mais je l'ai reprogrammée car % donnait des résultats non satisfaisants pour les entiers négatifs.

Listing 3.2 – modulo.c

```
1  int modulo (int x, int n)
2  {
3      if(x<0)
4      {
5          while(x<0) { x=x+n; }
6      }
7      else if(x>0)
8      {
9          while(x>n) { x=x-n; }
10     }
```

```
11      return x;
12  }
```

3.2.2 Inversion

La deuxième fonction, `inverse`, permet de réaliser l'opération inverse de la multiplication. Celle-ci est nécessaire pour le calcul de $m = \frac{y_2 - y_1}{x_2 - x_1}$ et de $x3$ et $y3$. Cette opération n'est pas tout à fait une division classique. Il s'agit de trouver, connaissant a et p, un entier b vérifiant $ab \equiv 1[p]$. En appliquant le théorème de Bézout, on remarque que cela revient à déterminer b tel que $ab + pv = 1$ où v est un entier. Pour déterminer ces coefficients « de Bézout », on utilise l'algorithme étendu d'Euclide. C'est cet algorithme qui est mis en place dans la fonction inverse. On remarquera qu'on se contente de renvoyer b car v ne nous intéresse pas ici.

Listing 3.3 – inverse.c

```
1  extern p;
2  int modulo (int x, int n);
3  int inverse (int a, int b)
4  {
5    int q, r, s, t, tmp, u, v;
6    u = 1;
7    v = 0;
8    s = 0;
9    t = 1;
10   while (b > 0) {
11     q = a / b;
12     r = a % b;
13     a = b;
14     b = r;
15     tmp = s;
16     s = u - q * s;
17     u = tmp;
18     tmp = t;
19     t = v - q * t;
20     v = tmp;
21   }
22   return modulo(u,p);
23 }
```

3.3 Multiplication

L'opération multiplication demande peu de travail. En effet, un fois que l'on dispose des fonctions permettant d'effectuer une addition, la multiplication est une simple boucle.

Ici aussi, on divise le code en 2 fonctions, l'une pour l'abscisse et l'autre pour l'ordonnée. On a alors :

Listing 3.4 – mult.c

```
1  extern a,b,p;
2  int mult_x(int x, int y, int k)
3  {
4      int Rx,Ry;
5      Rx=x;
6      Ry=y;
7      int temp, compteur;
8      for(compteur=1; compteur < k; compteur++)
9      {
10         temp=Rx;
11         Rx=add_x(x,y,Rx,Ry);
12         Ry=add_y(x,y,temp,Ry,Rx);
13         }
14     return Rx;
15 }
16 int mult_y(int x, int y, int k)
17 {
18     int Rx,Ry;
19     Rx=x;
20     Ry=y;
21     int temp, compteur;
22     for(compteur=1; compteur < k; compteur++)
23     {
24         temp=Rx;
25         Rx=add_x(x,y,Rx,Ry);
26         Ry=add_y(x,y,temp,Ry,Rx);
27         }
28     return Ry;
29 }
```

3.4 Types de variable

3.4.1 Le type `Coord`

Pour simplifier le code, on introduit deux types personnalisés. Le premier type, nommé `Coord` permet de gérer les coordonnées des points. Il permet de contenir dans la même variable l'abscisse et l'ordonnée d'un point :

```
1  typedef struct Coord Coord;
2  struct Coord
```

```
{
    int x;
    int y;
};
```

3.4.2 Le type `MessageCode`

Le deuxième type est propre à la transmission même du message. Nous verrons plus tard que lorsque l'on transmet le message sous forme cryptée, on envoie en fait un couple de points. On définit donc un nouveau type `MessageCode` qui contient deux points, qui sont donc du type `Coord` :

```
typedef struct MessageCode MessageCode;
struct MessageCode
{
    Coord lP;
    Coord M_lkBP;
};
```

3.5 Points et courbes

3.5.1 Ensemble des points d'une courbe

Pour choisir une courbe, il faut savoir si elle contient assez de points. En effet, pour pouvoir coder l'ensemble des caractères de l'alphabet, voir quelques caractères spéciaux supplémentaires, il faut disposer d'au moins une cinquantaine de points sur la courbe. Pour cela, on peut élaborer une fonction qui affiche une liste des points de la courbe :

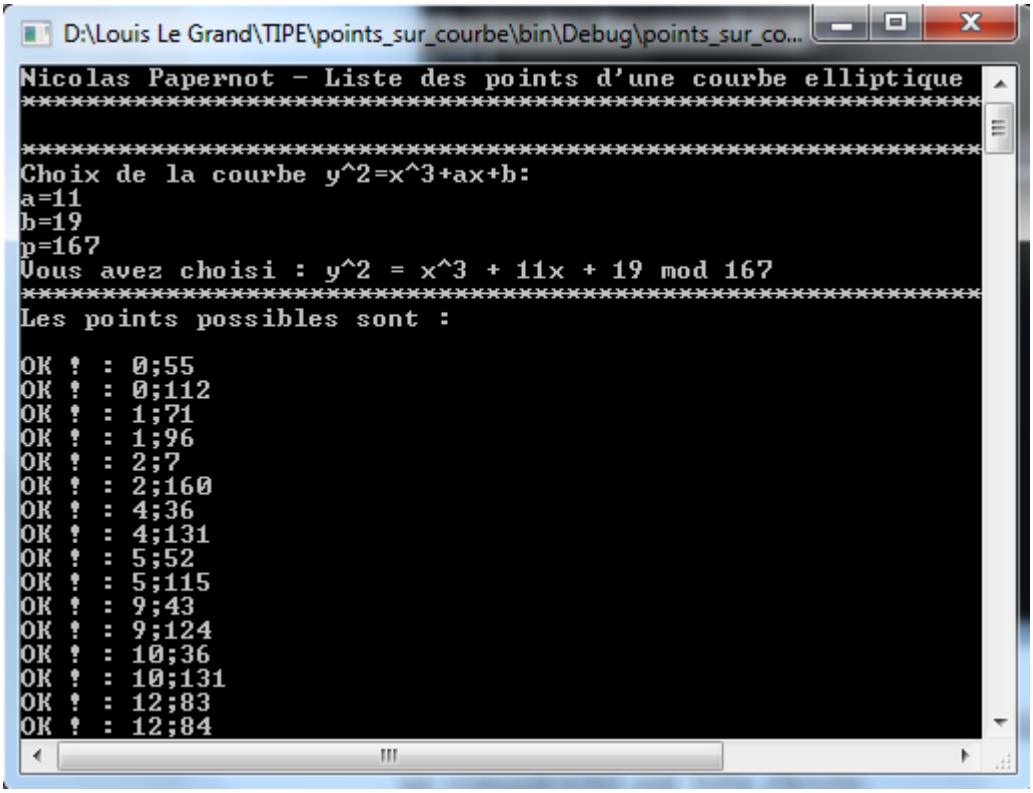
Listing 3.5 – Point aléatoire

```
int points(int p)
{
    srand(time(NULL));
    int x=0,compteur;
    double y_carre;
    do
    {
        y_carre=((x*x*x)+(a*x)+b)%p;
        int test;
        for(test=0; test!=p; test++)
        {
            if(((test*test)%p)==y_carre)
            {
                printf("\nOK ! : %d;%d",x,test);
                compteur++;
```

```
16                  }
17              }
18
19              x=x+1;
20          } while (x<p);
21          printf("\nNombre de points :%d\n",compteur);
22          return 0;
23      }
```

Cette fonction reprend le raisonnement que nous avons présenté lorsque l'on a introduit la notion de corps dans la partie introductive sur les courbes elliptiques de ce document. Ainsi, pour chaque valeur de `x` inférieure à p (cf lignes 19-20), la fonction teste toutes les valeurs de `y` possibles modulo p (cf. ligne 10) et affiche les valeurs satisfaisant l'équation réduite de Weierstrass (cf. lignes 8 et 12). Le tout est associé à une variable `compteur` (cf. lignes 4 et 15) qui permet de connaître le nombre de points en fin d'exécution.

Voici une prise d'écran du programme tournant dans la console Windows :



Le programme affiche une liste des points de la courbe $y^2 = x^3 + 11x + 19$ qui appartiennent au corps $\mathbb{Z}/167\mathbb{Z}$. La prise d'écran ne contient que les premiers mais il y en a en tout 160.

Il est bien sûr inenvisageable d'utiliser cette fonction pour de grandes valeurs de p car sa complexité est en $O(p^2)$.

3.5.2 Point aléatoire sur une courbe

On peut adapter le code précédent pour déterminer un point de la courbe aléatoirement.

Listing 3.6 – point.c

```
1  #include <math.h>
2  #include <time.h>
3  extern a,b,p;
4  int point()
5  {
6      srand(time(NULL));
7      int x,i=0;
8      double y_carre;
9      do
10     {
11         x=rand()%(p) + 1;
12         y_carre=((x*x*x)+(a*x)+b)%p;
13         int y_t=sqrt(y_carre);
14         if(y_t*y_t==y_carre)
15         {
16             i=i+1;
17         }
18     } while (i!=1);
19     return x;
20 }
```

Cette fonction diffère de la précédente car la valeur de l'abscisse, et donc de la variable `x`, est générée aléatoirement à la ligne 11. La suite de l'algorithme est similaire à la fonction précédente : on teste l'existence d'une valeur de `y` (cf. lignes 12 et 14). Si il n'y a pas de valeurs satisfaisant l'équation de Weierstrass, on change de valeur pour `x` et on recommence (cf. ligne 18).

Chapitre 4

Fonctions de transmission d'un message

4.1 Protocole Diffie-Hellman

Cet échange de clé possède la particularité de ne pas nécessiter de rencontre préalable à la transmission du message entre l'expéditeur (Aline) et le destinataire (Bob).

Aline et Bob commencent par se mettre d'accord de manière publique sur la courbe qu'ils vont utiliser. Il choisissent donc les coefficients a et b de la forme de Weierstrass $y^2 = x^3 + ax + b$. Ils sélectionnent également un point P sur cette courbe.

Aline va ensuite choisir un entier k_A secrètement et Bob de même choisir un entier k_B. Aline envoie à Bob le point k_AP et Bob lui renvoie le point k_BP. Ensuite ils multiplient chacun de leur côté le point qu'ils ont reçu par leur clé secrète : Aline multiplie k_BP par k_A et Bob k_AP par k_B. Aline obtient k_Ak_BP et Bob k_Bk_AP. Cependant, $k_Ak_BP = k_Bk_AP$, ils sont donc tous les deux en possession de la même clé privée.

Même si Ève a espionné leurs échanges, elle ne connaît que a, b, P, k_AP et k_BP. Il faudrait qu'elle détermine k_A à partir de P et k_AP pour connaître la clé privée. Il s'agit bien évidement du problème du logarithme discret. C'est sur la difficulté de résolution de ce logarithme que repose la sécurité de l'échange de clé Diffie-Hellman.

Voici les lignes de codes qui permettent d'effectuer cet échange :

Listing 4.1 – Diffie-Hellman

```
1 Coord P;
2 Coord kAP;
3 Coord kBP;
4 Coord kAkBP;
5 Coord kBkAP;
```

```
6  printf("Echange de cles Diffie-Hellman\n");
7  printf("\n(P) : Le point public est P = (");
8  P.x=point();
9  P.y=sqrt(modulo((P.x*P.x*P.x)+(a*P.x)+b,p));
10 printf("%d;%d)",P.x,P.y);
11 int kA=rand()%(10) + 1;
12 printf("\n(A) : Votre cle privee est : ");
13 printf("%d",kA);
14 kAP.x=mult_x(P.x,P.y,kA);
15 kAP.y=mult_y(P.x,P.y,kA);
16 printf("\n(A) : Envoyez (%d;%d)\n", kAP.x,kAP.y);
17 printf("\n(B) : Votre cle privee est : ");
18 int kB=rand()%(10) + 1;
19 printf("%d",kB);
20 kBP.x=mult_x(P.x,P.y,kB);
21 kBP.y=mult_y(P.x,P.y,kB);
22 printf("\n(B) : Envoyez (%d;%d)", kBP.x,kBP.y);
23 kAkBP.x=mult_x(kBP.x,kBP.y,kA);
24 kAkBP.y=mult_y(kBP.x,kBP.y,kA);
25 printf("\n(A) : kAkBP=(%d;%d)",kAkBP.x,kAkBP.y);
26 kBkAP.x=mult_x(kAP.x,kAP.y,kB);
27 kBkAP.y=mult_y(kAP.x,kAP.y,kB);
28 printf("\n(B) : kBkAP=(%d;%d)",kBkAP.x,kBkAP.y);
29 printf("\n\nEchange de cles termine !");
```

Remarquons que l'initialisation des variables `a`, `b` et `p` n'apparaît pas dans cette portion de code car elles ont été définies en tant que variables globales en dehors de la fonction `main`() afin d'être utilisables dans toutes les fonctions et fichiers du programme. Les 5 premières lignes du code font appel au type `Coord` créé précédemment pour définir les points qui seront utilisés : $\mathtt{P}, \mathtt{k_A P}, \mathtt{k_B P}, \mathtt{k_A k_B P}$ et $\mathtt{k_B k_A P}$. Ensuite, le point `P` est déterminé aléatoirement par le programme (lignes 8 et 9) à l'aide de la fonction `point`() dont on a détaillé le fonctionnement précédemment. Le reste du programme constitue l'échange de clé tel qu'il a été décrit (cf. supra). On génère d'abord une clé $\mathtt{k_A}$ privée pour (`A`) à la ligne 11. Ensuite on calcule $\mathtt{k_A P}$ aux lignes 14-15. (`B`) fait de même en générant à la ligne 18 une clé privée $\mathtt{k_B}$ puis en calculant aux lignes 20-21 le point $\mathtt{k_B P}$. Ensuite l'échange est fait et (`A`) multiplie $\mathtt{k_B P}$ par $\mathtt{k_A}$ aux lignes 23-24. (`B`) fait de même en multipliant $\mathtt{k_A P}$ par $\mathtt{k_B}$ aux lignes 26-27. L'échange de clé est alors terminé.

4.2 Cryptage et décryptage du message

Une fois le protocole de Diffie-Hellman, les deux usagers peuvent échanger des messages. Le message M à transmettre sera un point de la courbe.

Aline choisit d'abord un nombre l qu'elle garde secret. Elle calcule puis envoie le couple $(lP, M + lk_BP)$ à Bob.

Ce dernier multiplie lP par k_B puis soustrait ce point à $M + lk_BP$. Il retrouve le point constituant le message que voulait transmettre Aline. Toute personne ayant espionné les échanges doit connaître k_B pour retrouver le message. Il s'agit à nouveau du problème du logarithme discret.

Voici les lignes de code correspondant à ces échanges :

Listing 4.2 – Echange message

```
1  int l;
2  printf("Codage : (A) : Message : \n");
3  Coord message;
4  Coord lP;
5  Coord lkBP;
6  Coord M_lkBP;
7  scanf("%d",&message.x);
8  scanf("%d",&message.y);
9  printf("\n************************************************\n");
10 printf("Le message transmis sera : (%d;%d)",message.x,message.y);
11 l=rand()%(10) + 1;
12 printf("\n(A) : Nombre secret : l=%d",l);
13 lP.x = mult_x(P.x,P.y,l);
14 lP.y = mult_y(P.x,P.y,l);
15 lkBP.x = mult_x(kBP.x,kBP.y,l);
16 lkBP.y = mult_y(kBP.x,kBP.y,l);
17 M_lkBP.x = add_x(message.x,message.y,lkBP.x,lkBP.y);
18 M_lkBP.y = add_y(message.x,message.y,lkBP.x,lkBP.y,M_lkBP.x);
19 MessageCode mess_code;
20 mess_code.lP = lP;
21 mess_code.M_lkBP = M_lkBP;
22 printf("\n(A) : Envoyez le couple (lP,M+lkBP) defini par : \n  * lP
     = (%d;%d)\n  * M+lkBP = (%d;%d)", mess_code.lP.x, mess_code.lP.y,
      mess_code.M_lkBP.x, mess_code.M_lkBP.y);
23 printf("\n************************************************\
     nDecodage\n");
24 Coord kBlP;
25 Coord M;
26 kBlP.x = mult_x(mess_code.lP.x,mess_code.lP.y,kB);
27 kBlP.y = mult_y(mess_code.lP.x,mess_code.lP.y,kB);
28 printf("\n(B) : kBlP = (%d;%d)",kBlP.x,kBlP.y);
29 M.x = add_x(mess_code.M_lkBP.x,mess_code.M_lkBP.y,kBlP.x,-kBlP.y);
30 M.y = add_y(mess_code.M_lkBP.x,mess_code.M_lkBP.y,kBlP.x,-kBlP.y,M.x
```

```
    );
31  printf("\n(B) : Message = (%d;%d)", M.x,M.y);
```

On commence par déclarer les variables à l'aide du type `Coord` aux lignes 3-6. On génère ensuite l'entier `l` aléatoirement à la ligne 11. (`A`) effectue ensuite les calculs nécessaire au codage du message : on calcule d'abord `lP` comme `l`-fois `P` (cf. lignes 13-14), puis $\texttt{lk}_\texttt{B}\texttt{P}$ comme `l`-fois $\texttt{k}_\texttt{B}\texttt{P}$ (cf. lignes 15-16). On obtient alors (lignes 17-18) le point $\texttt{M} + \texttt{lk}_\texttt{B}\texttt{P}$. Ensuite on déclare la variable de type `MessageCode` à la ligne 19 avant de l'affecter avec les deux points qui constituent le message codé (cf. lignes 20-21). Le codage est alors terminé.

Pour décoder le message, (`B`) commence par déclarer les variables points aux lignes 24-25 puis on calcule (aux lignes 26-27) $\texttt{k}_\texttt{B}\texttt{lP}$ à l'aide la clé privée $\texttt{k}_\texttt{B}$ et du point `lP` reçu lors de l'échange. On retrouve alors le message en effectuant une soustraction entre le point $\texttt{M} + \texttt{lk}_\texttt{B}\texttt{P}$ reçu et $\texttt{k}_\texttt{B}\texttt{lP}$ aux lignes 29-30. Le décodage est alors terminé.

Chapitre 5

Programme principal

A partir des fonctions codées dans les deux chapitres précédents, on réalise le programme qui permet de coder et de décoder un point. Le code source est alors :

Listing 5.1 – main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <time.h>
5  #include "add.h"
6  #include "inverse.h"
7  #include "modulo.h"
8  #include "mult.h"
9  #include "point.h"
10 #include "type_coord.h"
11 #include "type_message_code.h"
12
13 int a,b,p;
14
15 int main()
16 {
17     printf("************************************************\n");
18     printf("Nicolas Papernot\nCryptographie sur les courbes
           elliptiques\nTIPE 2010-2012");
19     printf("\n************************************************\n");
20     srand(time(NULL));
21     printf("\n************************************************\n");
22     printf("Choix de la courbe y^2=x^3+ax+b: \n");
23     printf("a=");
24     scanf("%d",&a);
25     printf("b=");
26     scanf("%d",&b);
27     printf("p=");
```

```
28      scanf("%d",&p);
29      printf("Vous avez choisi : y^2 = x^3 + %dx + %d mod %d",a,b,p);
30      printf("\n***********************************************\n");
31      Coord P;
32      Coord kAP;
33      Coord kBP;
34      Coord kAkBP;
35      Coord kBkAP;
36      printf("Echange de cles Diffie-Hellman\n");
37      printf("\n(P) : Le point public est P = (");
38      P.x=point();
39      P.y=sqrt(modulo((P.x*P.x*P.x)+(a*P.x)+b,p));
40      printf("%d;%d)",P.x,P.y);
41      int kA=rand()%(10) + 1;
42      printf("\n(A) : Votre cle privee est : ");
43      printf("%d",kA);
44      kAP.x=mult_x(P.x,P.y,kA);
45      kAP.y=mult_y(P.x,P.y,kA);
46      printf("\n(A) : Envoyez (%d;%d)\n", kAP.x,kAP.y);
47      printf("\n(B) : Votre cle privee est : ");
48      int kB=rand()%(10) + 1;
49      printf("%d",kB);
50      kBP.x=mult_x(P.x,P.y,kB);
51      kBP.y=mult_y(P.x,P.y,kB);
52      printf("\n(B) : Envoyez (%d;%d)", kBP.x,kBP.y);
53      kAkBP.x=mult_x(kBP.x,kBP.y,kA);
54      kAkBP.y=mult_y(kBP.x,kBP.y,kA);
55      printf("\n(A) : kAkBP=(%d;%d)",kAkBP.x,kAkBP.y);
56      kBkAP.x=mult_x(kAP.x,kAP.y,kB);
57      kBkAP.y=mult_y(kAP.x,kAP.y,kB);
58      printf("\n(B) : kBkAP=(%d;%d)",kBkAP.x,kBkAP.y);
59      printf("\n\nEchange de cles termine !");
60      printf("\n***********************************************\n");
61      int l;
62      printf("Codage : (A) : Message : \n");
63      Coord message;
64      Coord lP;
65      Coord lkBP;
66      Coord M_lkBP;
67      scanf("%d",&message.x);
68      scanf("%d",&message.y);
69      printf("\n***********************************************\n");
70      printf("Le message transmis sera : (%d;%d)",message.x,message.y)
            ;
```

```
71      l=rand()%(10) + 1;
72      printf("\n(A) : Nombre secret : l=%d",l);
73      lP.x = mult_x(P.x,P.y,l);
74      lP.y = mult_y(P.x,P.y,l);
75      lkBP.x = mult_x(kBP.x,kBP.y,l);
76      lkBP.y = mult_y(kBP.x,kBP.y,l);
77      M_lkBP.x = add_x(message.x,message.y,lkBP.x,lkBP.y);
78      M_lkBP.y = add_y(message.x,message.y,lkBP.x,lkBP.y,M_lkBP.x);
79      MessageCode mess_code;
80      mess_code.lP = lP;
81      mess_code.M_lkBP = M_lkBP;
82      printf("\n(A) : Envoyez le couple (lP,M+lkBP) defini par : \n  *
            lP = (%d;%d)\n  * M+lkBP = (%d;%d)", mess_code.lP.x,
           mess_code.lP.y, mess_code.M_lkBP.x, mess_code.M_lkBP.y);
83      printf("\n**********************************************\
           nDecodage\n");
84      Coord kBlP;
85      Coord M;
86      kBlP.x = mult_x(mess_code.lP.x,mess_code.lP.y,kB);
87      kBlP.y = mult_y(mess_code.lP.x,mess_code.lP.y,kB);
88      printf("\n(B) : kBlP = (%d;%d)",kBlP.x,kBlP.y);
89      M.x = add_x(mess_code.M_lkBP.x,mess_code.M_lkBP.y,kBlP.x,-kBlP.y
            );
90      M.y = add_y(mess_code.M_lkBP.x,mess_code.M_lkBP.y,kBlP.x,-kBlP.y
            ,M.x);
91      printf("\n(B) : Message = (%d;%d)", M.x,M.y);
92      printf("\n**********************************************\n");
93      return 0;
94  }
```

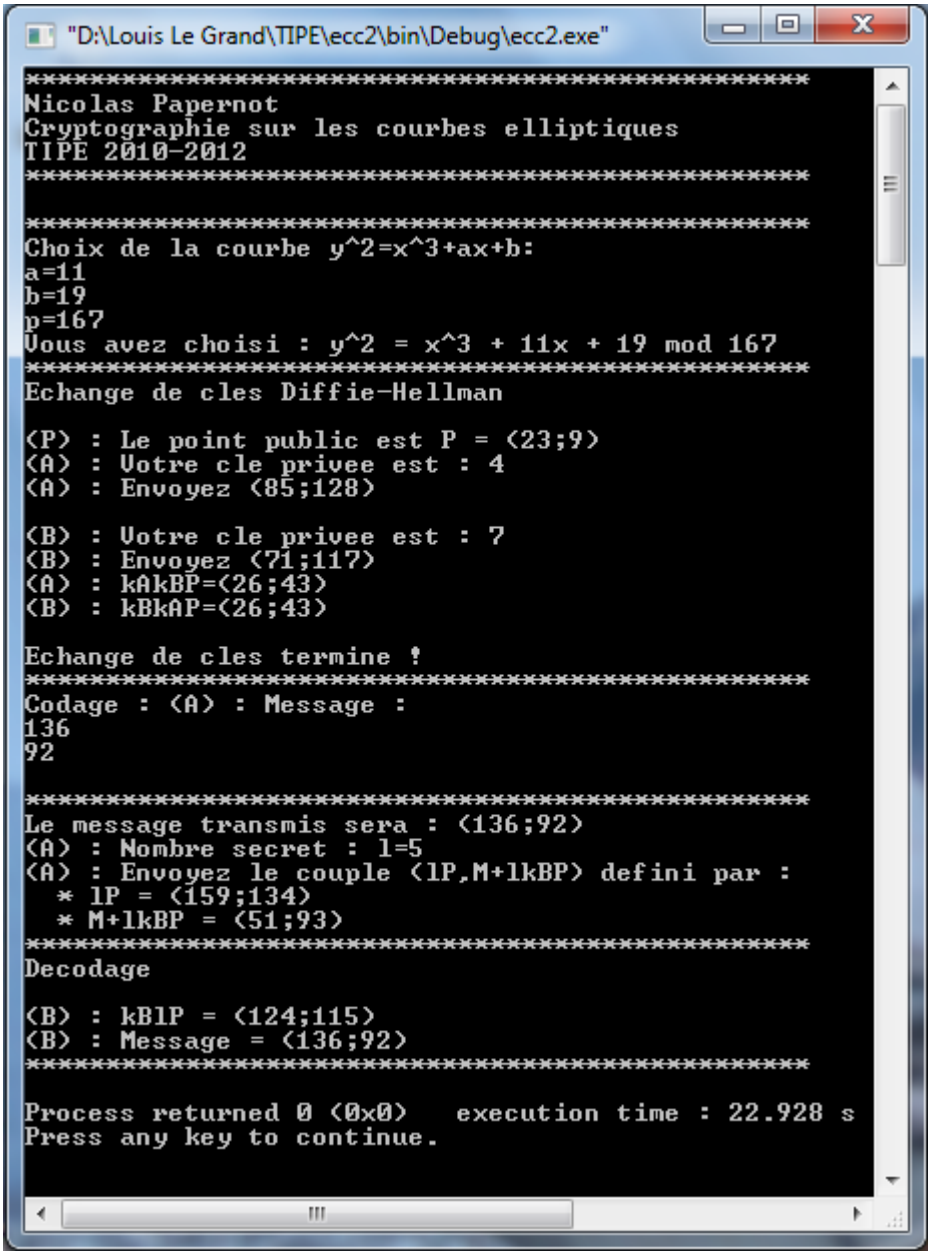
On remarque l'appel à un certain nombre de fichiers `add.h`, `inverse.h`, `modulo.h`, `mult.h`, `point.h`, `type_coord.h` et `type_message_code.h` en début de code qui permet d'accéder à toutes les fonctions codées précédemment ainsi qu'aux types personnalisés nécessaire à la gestion des coordonnées des points. L'appel aux bibliothèques `math.h` et `time.h` est nécessaire pour la génération d'entiers pseudo-aléatoire. La suite du programme est une simple juxtaposition d'un échange Diffie-Hellman avec une échange de message (cf. supra).

Exemple Considérons la courbe d'équation $y^2 = x^3 + 11x + 19$ dans le corps $\mathbb{Z}/167\mathbb{Z}$. Le discriminant de cette courbe est :

$$\Delta = -\left(4(11)^3 + 27(19)^2\right) = -15071 \neq 0$$

Il s'agit donc d'une courbe utilisable pour la cryptographie sur les courbes elliptiques (même si l'entier 167 est bien évidemment trop faible pour assurer la sécurité des échanges).

Voici une prise d'écran du programme tournant dans la console Windows :



Les caractères (A) et (B) situés en début de ligne indiquent quelle est la personne qui est entrain de réaliser une opération. On constate que les deux utilisateurs possèdent bien la même clé : $(26, 43)$. Le point P public choisi aléatoirement est ici $(23, 9)$. Le message à transmettre est $(136, 92)$. Après le cryptage, (A) transmet $lP = (159, 134)$ ainsi que $M + lkBP = (51, 93)$. A l'issue du décryptage, (B) retrouve bien le message que voulait transmettre (A), c'est-à-dire $M = (136, 92)$.

Troisième partie

Conclusion

Chapitre 6

Conclusion

6.1 Limites et défauts

On a déjà exprimé une condition sur le coefficient b qui doit être non-nul afin de s'assurer que le point de coordonnées $(0, 0)$ peut être utilisé pour le point à l'infini.

Le type de variable utilisée pour la mise en place (i.e. le type `int`) limite également la taille des coefficients et des clés que l'on utilise. Il faudrait le remplacer par le type `double` et adapter toutes les fonctions.

Nous avons également remarqué précédemment que certaines de nos fonctions sont sensibles à la valeur de p choisie pour le corps $\mathbb{Z}/p\mathbb{Z}$. Celle-ci possède une grande influence sur la complexité des calculs.

Cependant, il faut aussi rappeler que la cryptographie à clé publique possède quelques défauts. Elle est notamment plus longue en temps d'exécution que la cryptographie à clé privée. C'est pourquoi on peut envisager d'utiliser la cryptographie sur les courbes elliptiques pour échanger une clé avant chaque communication. On utilise ensuite cette clé avec un système à clé privée pour transmettre le message.

6.2 Comparaison avec le cryptosystème RSA

La cryptographie sur les courbes elliptiques a un avantage majeur : elle nécessite des clés de taille beaucoup plus petite que RSA. Ceci en fait un concurrent très sérieux pour les systèmes nécessitant un grand niveau de sécurité mais qui disposent de peu de ressources.

Le site de la National Security Agency donne un tableau comparatif des tailles de clé pour un système à clé privée, un système basé sur RSA et un système fondé sur les courbes elliptiques. Chaque ligne correspond à un niveau de sécurité identique.

Taille de la clé symétrique	Taille de la clé RSA	Taille d'une clé de courbe elliptique
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

On constate que la taille des clés RSA augmente beaucoup plus rapidement que la taille des clés sur les courbes elliptiques. C'est l'avantage principal de la cryptographie sur les courbes elliptiques. C'est ce qui amène ce cryptosystème à déloger progressivement le cryptosystème RSA.

Bibliographie

[1] National Security Agency. *The Case For Elliptic Curve Cryptography.* http ://www.nsa.gov/business/programs/elliptic_curve.shtml.

[2] D.R. Hankerson, S.A. Vanstone, and A.J. Menezes. *Guide to elliptic curve cryptography.* Springer-Verlag New York Inc, 2004.

[3] Bétréma J. *Algorithme d'Euclide.* http ://www.labri.fr/perso/betrema/deug/poly/euclide.html.

[4] N. Koblitz. *A course in number theory and cryptography*, volume 114. Springer, 1994.

[5] Husson L. *Cryptographie.* http ://pgp.apsoft.net/crypt.html.

[6] J.H. Silverman. *The arithmetic of elliptic curves*, volume 106. Springer Verlag, 2009.

[7] M. Stamp and J. Wiley. *Information security : principles and practice.* Wiley Online Library, 2006.

[8] L.C. Washington. *Elliptic curves : number theory and cryptography*, volume 50. Chapman & Hall, 2008.

[9] Wikipedia. *Courbe elliptique.* http ://fr.wikipedia.org/wiki/Courbe_elliptique.