Numéro d'identification :

Académie de Montpellier

U n i v e r s i t é M o n t p e l l i e r I I

— Sciences et Technique du Languedoc —

T h è s e

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

Spécialité	: Informatique
Formation Doctorale	: Informatique
École Doctorale	: Sciences pour l'Ingénieur

Arithmétique pour la Cryptographie basée sur les Courbes Elliptiques

par

Nicolas Méloni

Soutenue le 24 septembre 2007 devant le Jury composé de :

M. Thierry Berger, Professeur, Univ. de Limoges, ... Rapporteur
M. Gilles Villard, Directeur de Recherche, École Normale Supérieure de Lyon, ... Rapporteur
M. Valérie Berthé, Directrice de Recherche, LIRMM, ... Examinatrice
M. Marc Girault, Expert France Telecom, Caen, ... Examinateur
M. Jean-Claude Bajard, Professeur, Univ. Montpellier II, LIRMM, ... Directeur de Thèse
M. Sylvain Duquesne, Maître de Conférence, Univ. Montpellier II, LIRMM, ... Co-directeur de Thèse

Sommaire

Introduction

Longtemps cantonnée aux sphères militaires et politiques, la cryptographie est devenue, depuis quelques décennies, un véritable enjeu de société. Le développement rapide des réseaux de communication numériques, ainsi que l'augmentation du nombre de ses utilisateurs, ont posé de façon de plus en plus critique le problème de l'échange des clés de chiffrement. Très schématiquement, le principe de la cryptographie classique (ou cryptographie à clé privée) consiste à échanger des données chiffrées à partir d'un secret (appelé clé) connu uniquement par les parties concernées, le chiffrement comme le déchiffrement nécessitant la clé. Le paradoxe étant que pour pouvoir échanger des informations secrètement, les deux parties doivent déjà partager un secret. S'il parait simple à résoudre dans le cadre de communications entre deux personnes, le problème d'échange de clés prend une tout autre dimension à l'heure d'internet et de son milliard d'usagers. C'est pour répondre à ce problème qu'a été inventée la cryptographie à clé publique (ou cryptographie asymétrique). Le principe de la cryptographie à clé publique est d'utiliser non plus une clé pour chiffrer et déchiffrer mais un couple de clés : une clé publique, donc connue de tous, servant au chiffrement, et une clé privée, connue uniquement du destinataire, permettant le déchiffrement. Ainsi si une personne a besoin de se faire envoyer des informations de façon confidentielle, il lui suffit de générer elle même un couple clé publique-clé privée et plus aucun échange de clés n'est alors nécessaire (mais se pose alors le problème de la diffusion de cette clé publique).

Si le principe de la cryptographie à clé publique fut proposé en 1975, ce n'est qu'en 1977 que fut présenté le premier protocole effectif : RSA (du nom de ses auteurs Rivest, Shamir et Adleman). Principalement basé sur le problème de la factorisation des grands entiers, RSA est encore aujourd'hui la primitive la plus utilisée en cryptographie. Cependant les nombreux progrès effectués dans le domaine de la factorisation font que la taille des clés RSA augmente plus vite que ne le requiert l'augmentation de la puissance des ordinateurs. C'est l'une des raisons pour lesquelles la cryptographie basée sur les courbes elliptiques (ECC) connaît un tel intérêt depuis son introduction par Miller et Koblitz [Mil86, Kob87] en 1987. Reposant sur le problème du logarithme discret, ECC requiert, à niveau de sécurité équivalent, des clés bien plus petites que RSA (une clé ECC de 160 bits est aussi robuste qu'une clé RSA de 1024 bits), celui-là étant donc plus adapté à des environnements à puissance réduite (tels que les cartes à puce).

Jusqu'au milieu des années 90 les principaux travaux concernant ECC ont porté sur l'amélioration de son efficacité en termes de temps de calcul et de ressources, ainsi que sur la robustesse du problème du logarithme discret. Cependant, en 1996, Paul Kocher

présenta une nouvelle forme d'attaque dite par canaux cachés [Koc96]. Le principe n'est pas de résoudre le problème théorique sur lequel repose de la sécurité du protocole cryptographique, mais de tirer partie des informations que relâche immanquablement le système physique sur lequel est implanté le protocole (chaleur, consommation d'énergie, temps de calcul etc)afin de retrouver la clé secrète du système.

Les protocoles cryptographiques doivent s'exécuter le plus rapidement possible afin de ne pas ralentir les échanges de données. Toutefois, il faut veiller à ce que l'implantation, aussi bien logicielle que matérielle, soit la moins sujette possible aux attaques par canaux cachés. Ce travail d'implantation peut s'effectuer à deux niveau : au niveau de l'arithmétique de la courbe et au niveau de l'arithmétique du corps sous-jacent.

La principale opération à effectuer lors d'un protocole utilisant les courbes elliptiques est la multiplication de point par un scalaire. Étant donné un entier k et un point P sur une courbe, l'opération consiste à calculer le point $k \times P$. Ce calcul s'effectue à partir d'une chaîne de calculs faisant intervenir les opérations élémentaires de la courbe (addition de points, doublement etc). L'algorithme le plus classique dans ce domaine et l'algorithme dit de doublement et addition. Celui-ci est basé sur un schéma de Horner et sur l'écriture du scalaire en base deux. La plupart des autres méthodes de multiplication par une scalaire suivent ce schéma. Les améliorations consistant, en général, à précalculer certains points, afin de diminuer le nombre d'additions, et à remplacer le doublement par une opérations plus efficace (le morphisme de Frobenius en caractéristique 2 par exemple).

Concernant l'arithmétique sur le corps de définition de la courbe, la représentation des éléments et les algorithmes de compositions jouent un rôle majeur quand à l'efficacité, mais aussi à la sécurité, du protocole général. De nombreux système de représentation existent et parmi ceux-ci le système de représentation RNS. celui-ci est basée sur le théorème des restes chinois :

Théorème 1 Soit $\mathcal{B} = (m_1, m_2, \ldots, m_n)$ un ensemble d'entiers premiers deux à deux. Notons $M = \prod_{i=1}^{n} m_i$ alors,

$$\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_n\mathbb{Z} \simeq \mathbb{Z}/M\mathbb{Z}$$

L'isomorphisme précédent étant un isomorphisme d'anneaux, il est donc possible de diviser une addition ou un multiplication modulo M, en n additions ou multiplications modulo les m_i. Ce système est très intéressant dès lors que l'on veut effectuer des calculs modulo un entiers friables, cependant, ce n'est en général pas le cas en cryptographie. Un travail d'adaptation a donc été nécessaire afin de proposer des algorithmes de multiplication modulaire modulo des entiers peu ou pas friables.

Le présent document se compose de trois parties.

La première partie est consacré à l'état de l'art concernant l'arithmétique des courbes elliptiques. Le chapitre 1 est un tour d'horizon des principales propriétés des courbes

elliptiques et des différentes formules d'addition de points selon le système de coordonnées choisi. Dans le chapitre 2, nous présentons les principales méthodes de multiplication de points par un scalaire, aussi bien pour les courbes définies sur des corps premiers que pour les courbes définies sur des corps binaires.

La deuxième partie a pour objet l'étude de nouvelles formules d'addition de points sur les courbes et les nouveaux algorithmes de multiplication de points que l'on peut en déduire. Le chapitre 3 détaille les nouvelles formules d'addition de points, ainsi que l'algorithme dit de « Fibonacci » et addition. Dans le chapitre 4 nous présentons un type de chaînes d'additions, les chaînes d'additions différentielles, naturellement adaptées aux formules introduites dans le chapitre précédent, puis, nous proposons une construction de chaînes particulières, afin d'en déduire un algorithme de multiplication de point le plus efficace possible.

La troisième partie traite de la représentation RNS et de son adaptation à l'arithmétique des courbes elliptiques. Dans le chapitre 5 nous faisons un rappel des propriétés principale de la représentation RNS. Nous proposons, dans le chapitre 6, des bases RNS particulières permettant d'améliorer l'efficacité des calculs. Ensuite, dans le chapitre 7, nous proposons un algorithme d'inversion modulaire en RNS. Enfin, le chapitre 8 est consacré à l'étude de la complexité des sommes de produits modulaires en fonction du système de représentation choisi, puis à l'aménagement des formules d'additions de points sur les courbes afin de tirer avantages des spécificité du RNS.

Première partie

Etat de l'Art

Chapitre 1

Arithmétique des Courbes Elliptiques

Ce chapitre est une présentation synthétique des éléments de base nécessaires à la compréhension de l'arithmétique des courbes elliptiques. Nous ne donnerons donc pas les démonstrations et nous adopterons une approche ad'hoc des courbes elliptiques dans le sens où l'on passera sous silence la théorie des courbes algébriques sous-jacente. Pour une approche générale on pourra se référer à [Sil86] (un bon niveau en mathématique est préférable), à [HMV04] pour une vision plus spécifique au cas de la cryptographie et enfin à [MWZ96] pour une introduction aux courbes hyperelliptiques.

Dans tout le chapitre $\mathbb{K}$ désigne un corps fini.

1.1 Généralités

Dans cette partie, $\mathbb{A}^2(\mathbb{K})$ désigne l'ensemble des points (x, y) à coordonnées dans $\mathbb{K}$ et l'ensemble $\mathbb{P}^2(\mathbb{K})$ désigne l'ensemble des triplets $(X : Y : Z) \neq (0 : 0 : 0)$ à valeurs dans $\mathbb{K}$ muni de la relation d'équivalence

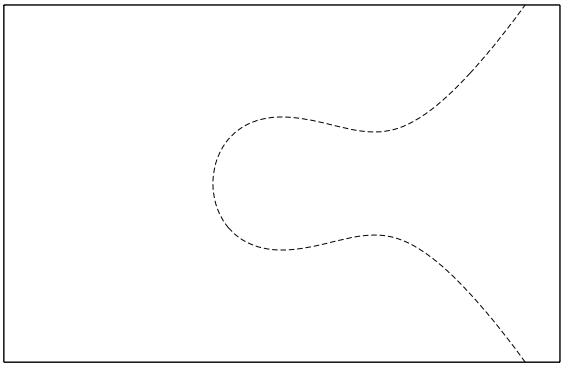
$$\forall \lambda \in \mathbb{K}^*,\ (X : Y : Z) \sim (\lambda X : \lambda Y : \lambda Z). \tag{1.1}$$

Définition 1 Une courbe elliptique sur $\mathbb{K}$, notée $E(\mathbb{K})$, est l'ensemble des solutions dans $\mathbb{P}^2(\mathbb{K})$ d'une équation homogène de la forme :

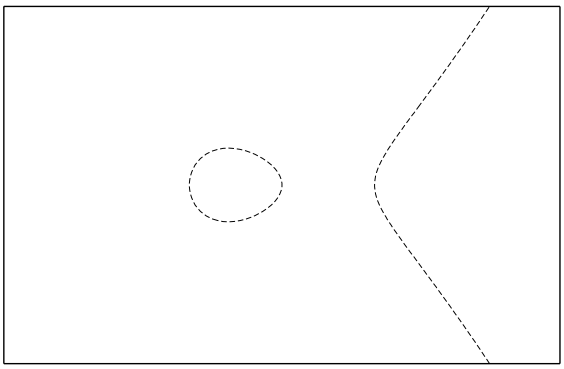
$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{1.2}$$

où $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$.

Par convention on imposera qu'une courbe elliptique soit une courbe lisse, c'est-à-dire qu'aucun point de la courbe n'annule toutes les équations aux dérivées partielles, ceci pour assurer que les formules de compositions de points que nous allons voir sont toujours valides (en particulier les formules de doublement).



(a) $E_1 : y^2 = x^3 - \frac{3}{2}x^2 - \frac{5}{4}x$



(b) $E_2 : y^2 = x^3 - x$

Fig. 1.1 – Courbes elliptiques sur $\mathbb{R}$

Remarquons qu'il n'y a qu'un seul point vérifiant à la fois l'équation de la courbe et l'équation $Z = 0$: le point $(0 : 1 : 0)$. Nous appellerons ce point le point à l'infini et nous le noterons $\mathcal{O}$.

Pour simplifier l'écriture, on considérera également la version affine de la définition d'une courbe elliptique :

Définition 2 Une courbe elliptique sur $\mathbb{K}$ (toujours notée $E(\mathbb{K})$) est l'ensemble des solutions dans $\mathbb{A}^2(\mathbb{K})$ d'une équation de la forme :

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1.3}$$

auquel on ajoute un point à l'infini noté $\mathcal{O}$.

La figure 1.1 montre deux exemples de courbes elliptiques définie sur $\mathbb{R}$.

Selon la caractéristique du corps de base, l'équation d'une courbe elliptique peut être simplifiée. On peut ainsi distinguer les trois cas suivants :

– si $\mathrm{Car}(\mathbb{K}) \geq 5$ (en pratique $\mathbb{K} = \mathbb{F}_p$ avec p un grand nombre premier), alors $E(\mathbb{K})$ est isomorphe à une courbe $E^{(1)}(\mathbb{K})$ donnée par :

$$E^{(1)} : y^2 = x^3 + ax + b \tag{1.4}$$

avec $4a^3 + 27b^2 \neq 0$,

- si $\mathrm{Car}(\mathbb{K}) = 3$ et $a_1^2 \neq -a_2$, alors $E(\mathbb{K})$ est isomorphe à une courbe $E^{(2)}(\mathbb{K})$ donnée par :

$$E^{(2)} : y^2 = x^3 + ax^2 + b \tag{1.5}$$

avec $a^3 b \neq 0$,

- si $\mathrm{Car}(\mathbb{K}) = 2$ et $a_1 \neq 0$, alors $E(\mathbb{K})$ est isomorphe à une courbe $E^{(3)}(\mathbb{K})$ donnée par :

$$E^{(3)} : y^2 + xy = x^3 + ax^2 + b \tag{1.6}$$

avec $b \neq 0$.

Remarque 1 Concernant les caractéristiques 2 et 3, les conditions $a_1^2 \neq -a_2$ et $a_1 \neq 0$ sont là pour assurer que les courbes ne sont pas supersingulières. Nous ne nous étendrons pas sur ce type de courbes ; sachons néanmoins qu'elles sont cryptographiquement peu sûres car vulnérables aux attaques par couplage. Cependant c'est cette même faiblesse qui les rend intéressantes dans le cadre de la cryptographie basée sur l'utilisation des couplages ; en particulier en caractéristique 3.

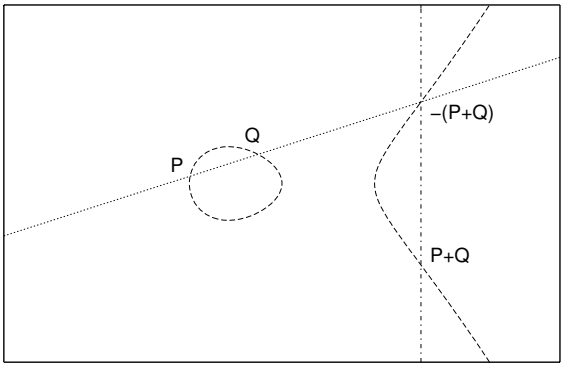
L'ensemble des points d'une courbe elliptique peut être muni d'une structure de groupe abélien dont l'élément neutre est le point à l'infini. La loi de groupe peut être interprétée géométriquement grâce à la fameuse méthode dite corde et tangente (voir figure 1.2). Pour la somme de deux points, on commence par tracer la droite passant par ces deux points. Cette droite coupe la courbe en un troisième point ; la somme des deux premiers points est alors le symétrique du troisième point par rapport à l'axe de symétrie de la courbe.

Proposition 1 Soit $E(\mathbb{K}) : y^2 = x^3 + ax + b$ une courbe elliptique avec $\mathrm{Car}(\mathbb{K}) \geq 5$, alors $E(\mathbb{K})$ est un groupe pour la loi de composition suivante :

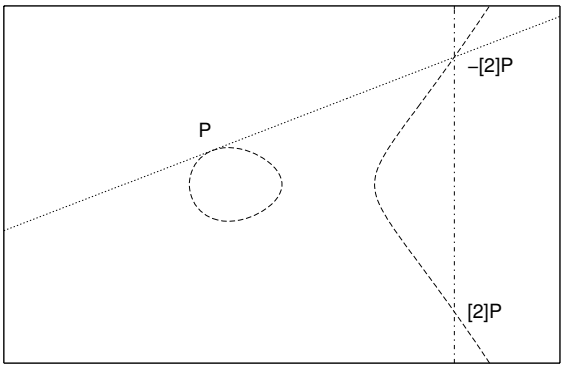
1. $P + \mathcal{O} = \mathcal{O} + P = P$ pour tout $P \in E(\mathbb{K})$.
2. Soit $P = (x, y) \in E(\mathbb{K})$, on définit $-P$ (ou $\overline{P}$) par : $-P = (x, -y)$.
3. Soient $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$ deux points sur la courbe tels que $P_1 \neq -P_2$, alors $P_1 + P_2 = (x_3, y_3)$ avec :

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

où $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ si $P_1 \neq P_2$, et $\lambda = \frac{3x_1^2 + a}{2y_1}$ sinon.



(a) Addition de deux points



(b) Doublement de point

Fig. 1.2 – Loi de groupe sur $E_2(\mathbb{R})$

Le coût d'une addition de deux points est donc d'une inversion (I), de deux multiplications (M), et d'un carré (C) sur le corps $\mathbb{K}$ (I+2M+C), le doublement revenant à I+2M+2C.

Proposition 2 Soit $E(\mathbb{K}) : y^2 = x^3 + ax^2 + b$ une courbe elliptique avec $\mathrm{Car}(\mathbb{K}) = 5$, alors $E(\mathbb{K})$ est un groupe pour la loi de composition suivante :

1. $P + \mathcal{O} = \mathcal{O} + P = P$ pour tout $P \in E(\mathbb{K})$.
2. Soit $P = (x, y) \in E(\mathbb{K})$, on définit $-P$ (ou $\overline{P}$) par : $-P = (x, -y)$.
3. Soient $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$ deux points sur la courbe tels que $P_1 \neq -P_2$, alors $P_1 + P_2 = (x_3, y_3)$ avec :

$$\begin{aligned} x_3 &= \lambda^2 - a - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

où $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ si $P_1 \neq P_2$, et $\lambda = \frac{ax_1}{y_1}$ sinon.

Le coût de l'addition et du doublement sont ainsi respectivement de I+M+C et I+3M.

Proposition 3 Soit $E(\mathbb{K}) : y^2 + xy = x^3 + ax^2 + b$ une courbe elliptique avec $\mathrm{Car}(\mathbb{K}) = 2$, alors $E(\mathbb{K})$ est un groupe pour la loi de composition suivante :

1. $P + \mathcal{O} = \mathcal{O} + P = P$ pour tout $P \in E(\mathbb{K})$.
2. Soit $P = (x, y) \in E(\mathbb{K})$, on définit $-P$ (ou $\overline{P}$) par : $-P = (x, x + y)$.
3. Soient $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$ deux points sur la courbe tels que $P_1 \neq -P_2$, alors $P_1 + P_2 = (x_3, y_3)$ avec :

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 - x_3) + x_3 + y_1 \end{aligned}$$

où $\lambda = \frac{y_2+y_1}{x_2+x_1}$ si $P_1 \neq P_2$, et $\lambda = x_1 + \frac{y_1}{x_1}$ sinon.

Le coût de l'addition et du doublement sont ainsi de I+2M+C chacun.

Une fois la loi d'addition définie sur les courbes, nous pouvons définir, pour tout entier $k \in \mathbb{N}$, le morphisme de multiplication scalaire par k :

$$\begin{aligned} [k] \ : \ & E \longrightarrow E \\ & P \longrightarrow [k]P = \underbrace{P + P + \cdots + P}_{k \ fois} \end{aligned}$$

Une bonne partie de ce document est consacrée à l'étude de différentes méthodes permettant d'effectuer cette multiplication par un scalaire de la manière la plus efficace possible. Elle est en effet l'opération principale de la plupart des protocoles cryptographiques utilisant les courbes elliptiques. Quelles que soient ces méthodes, elles font toujours appel aux formules de composition vues juste au dessus. Ces formules ont l'inconvénient de faire intervenir, à chaque appel, une inversion sur le corps, qui est en pratique une opération bien plus coûteuse qu'une multiplication.

Afin de pallier cette inconvénient, l'on utilise en pratique des systèmes de coordonnées projectives permettant de factoriser l'ensemble des inversions intervenant au cours des calculs en une seule inversion finale. Les deux sections suivantes présentes les différents choix de coordonnées possibles concernant les courbes elliptiques définies sur des corps premiers puis binaires.

1.2 Choix des coordonnées dans $\mathbb{F}_p$

Nous détaillons ici les différentes coordonnées utilisées sur des courbes définies sur des corps premiers. Le tableau 1.1 page 10 synthétise les différents coûts des additions et doublement de points en fonction des représentations décrites ci-dessous.

Coordonnées projectives

En coordonnées projectives, un point P de la courbe est représenté par un triplet $(X : Y : Z)$ correspondant au point affine $(X/Z, Y/Z)$.

L'équation de E devient $Y^2Z = X^3 + aXZ^2 + bZ^3$, l'inverse de $(X : Y : Z)$ est $(X : -Y : Z)$ et le point à l'infini est représenté par $(0 : 1 : 0)$. On pourra se référer à [CF06, HMV04] pour le détail des formules de composition de points.

On retiendra simplement qu'avec ce système de coordonnées, plus aucune inversion n'est requise pour composer les éléments d'une courbe. L'addition de deux points nécessite 12M+2C (respectivement 9M + 2C si l'un des deux points est donné en coordonnées affines) et un doublement 7M+5C. Les algorithmes de multiplication par un scalaire ne nécessitent alors plus qu'une seule inversion finale (et deux multiplications) pour obtenir un résultat en coordonnées affines.

Coordonnées jacobiennes

En coordonnées jacobiennes un point P de la courbe est représenté par un triplet $(X : Y : Z)$ correspondant au point affine $(X/Z^2, Y/Z^3)$.

L'équation de E devient $Y^2 = X^3 + aXZ^4 + bZ^6$, l'inverse de $(X : Y : Z)$ est $(X : -Y : Z)$ et le point a l'infini est représenté par $(1 : 1 : 0)$.

Addition

Soient $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (X_2 : Y_2 : Z_2)$ tel que $P_1 \neq \pm P_2$ et $P_1 + P_2 = (X_3 : Y_3 : Z_3)$, alors en posant :

$$A = X_1Z_2^2, \quad B = X_2Z_1^2, \quad C = Y_1Z_2^3, \quad D = Y_2Z_1^3, \quad E = B - A, \quad F = D - C,$$

on a

$$X_3 = -E^3 - 2AE^3 + F^2, \quad Y_3 = -CE^3 + F(AE^2 - X_3), \quad Z_3 = Z_1Z_2E.$$

Doublement

Soient $P_1 = (X_1 : Y_1 : Z_1)$ et $P_3 = [2]P_1 = (X_3 : Y_3 : Z_3)$, alors en posant :

$$A = 4X_1Y_1^2, \quad B = 3X_1^2 + aZ_1^4,$$

on a

$$X_3 = -2A + B^2, \quad Y_3 = -8Y_1^4 + B(A - X_3), \quad Z_3 = 2Y_1Z_1.$$

Ici le coût d'une addition est de 12M+4C (8M + 3C si l'un des deux points est donné en coordonnées affines) et de 4M+6C pour un doublement. On peut remarquer que l'addition est plus coûteuse qu'en coordonnées projectives mais que le doublement demande lui moins d'opérations. Afin d'améliorer l'efficacité de l'addition, on peut utiliser les coordonnées dites de Chudnovsky. Un point est alors représenté par ses coordonnées jacobiennes auxquelles on ajoute les quantités Z_1^2 et Z_1^3. Les coûts de l'addition et du doublement deviennent alors respectivement 11M + 3C et 5M+6C.

Si l'on veut privilégier le doublement, on peut utiliser les coordonnées jacobiennes modifiées, introduites par Cohen et al. [CMO98]. Dans cette représentation, on adjoint aux trois coordonnées jacobiennes la quantité aZ_1^4. Les formules d'additions ne changent pas (si ce n'est qu'il faut rajouter le calcul de aZ_1^4), par contre pour le calcul du doublement on introduit la quantité $C = 8Y_1^4$, et les formules deviennent alors

$$X_3 = -2A + B^2, \quad Y_3 = B(A - X_3) - C, \quad Z_3 = 2Y_1Z_1, \quad aZ_3^4 = 2C(aZ_1^4).$$

Le nombre d'opérations sur le corps passe alors à 13M+6C (9M+5C si l'un des deux points est en coordonnées affines) concernant l'addition et à 4M+4C concernant le doublement.

Coordonnées mixtes

La notion de coordonnées mixtes a été introduite par Cohen et al. [CMO98], et est basée sur la remarque suivante : il n'existe pas de système de coordonnées fournissant à la fois un doublement et une addition rapide. Par exemple, les coordonnées jacobiennes modifiées assurent le meilleur doublement (dès que I > 3.6M) mais l'addition est relativement coûteuse par rapport aux coordonnées de Chudnovsky. L'idée est alors de changer de système de coordonnées au cours de l'algorithme afin d'utiliser systématiquement le système de coordonnées le plus approprié à l'opération en cours. Le choix se fait alors selon deux critères : l'efficacité vis-à-vis de l'opération concernée et le coût du passage d'un système de coordonnées au suivant. Afin d'exprimer simplement ces deux notions nous adopterons la notation suivante : soient $\mathcal{C}^1, \mathcal{C}^2$, et $\mathcal{C}^3$ trois systèmes de coordonnées, on notera $\mathcal{C}^1 + \mathcal{C}^2 \to \mathcal{C}^3$ l'opération consistant à calculer la somme de deux points donnés respectivement en coordonnées $\mathcal{C}^1$ et $\mathcal{C}^2$ et à retourner le résultat en coordonnées $\mathcal{C}^3$. Par exemple, $\mathcal{J}^m + \mathcal{A} \to \mathcal{J}$ signifie que l'on additionne un point en coordonnées jacobiennes modifiées avec un point en coordonnées affines et que le résultat est donnée en coordonnées jacobiennes.

Pour le doublement, on utilisera la notation similaire suivante : $2\mathcal{C}^1 \to \mathcal{C}^2$. Enfin les différents types de coordonnées seront désignés par les symboles $\mathcal{A}, \mathcal{P}, \mathcal{J}, \mathcal{J}^c$ et $\mathcal{J}^m$. On se reportera au tableau 1.2 pour le coût des différentes opérations de ce type.

Comme nous le verrons dans le chapitre suivant, les algorithmes de multiplication par un scalaire consistent, pour la plupart, en une série de doublements entrecoupée d'additions. Afin d'optimiser l'utilisation de coordonnées mixtes on effectue cela de la façon suivante :

1. on effectue la série de doublements avec un seul système de coordonnées ; d'après le tableau 1.1 $\mathcal{J}^m$ offre les meilleures performances,
2. le résultat du dernier doublement est donné dans un certain système de coordonnées $\mathcal{C}$ afin de préparer l'addition ;
3. on calcule la somme du point précédemment calculé et d'un autre point, qui lui, est une constante de l'algorithme, que l'on prend en général en coordonnées $\mathcal{A}$ (ou $\mathcal{J}^c$ selon le coût de l'inversion par rapport à la multiplication et aux nombres de points à pré-calculer) ;
4. le résultat de cette somme est donné en coordonnées $\mathcal{J}^m$ afin de préparer une nouvelle série de doublements.

Finalement, seul manque le choix du système de coordonnées $\mathcal{C}$. Celui-ci doit permettre de minimiser le coût total des opérations $2\mathcal{J}^m \to \mathcal{C}$ et $\mathcal{C}+\mathcal{A}$ (ou $\mathcal{J}^c$) $\to \mathcal{J}^m$. La lecture du tableau 1.2 nous montre que le système de coordonnées $\mathcal{J}$ est une solution à ce problème. C'est donc ce système que nous utiliserons lors de l'évaluation des coûts des différents algorithmes.

En résumé cela signifie que, durant notre algorithme de multiplication de point, les doublements successifs s'effectueront en coordonnées $\mathcal{J}^m$, le doublement précédent une addition se fera à l'aide de l'opération $2\mathcal{J}^m \to \mathcal{J}$ et les additions se feront en utilisant $\mathcal{J} + \mathcal{A} \to \mathcal{J}^m$.

Courbes de Montgomery

Montgomery a introduit en 1987 [Mon87] un type particulier de courbes elliptiques sur lesquelles il est possible d'obtenir des formules d'addition et de doublement très efficaces.

Définition 3 Courbe de Montgomery
Soit E une courbe elliptique définie sur $\mathbb{F}_p$. On dit que E est une courbe de Montgomery si elle est isomorphe à une courbe de la forme :

$$E_M : By^2 = x^3 + Ax^2 + x. \tag{1.7}$$

On parle dans ce cas de courbe écrite sous forme de Montgomery.

Il est facile de voir que toute courbe de Montgomery peut être réécrite sous forme canonique. Pour cela il suffit de poser $a = 1/B^2 - A^2/3B^2$ et $b = -A^3/27B^3 - aA/3B$. Par contre la réciproque n'est pas vraie puisque conditionnée par l'existence du morphisme de la définition 3. Le théorème suivant précise les conditions d'existence des courbes de Montgomery :

Théorème 2 Soit $E : y^2 = x^3 + ax + b$ une courbe elliptique définie sur $\mathbb{F}_p$, E est une courbe de Montgomery si et seulement si :

1. $x^3 + ax + b$ a au moins une racine α dans $\mathbb{F}_p$,

2. $3\alpha^2 + a$ est un carré dans $\mathbb{F}_p$.

Cela vient du fait qu'un isomorphisme m permettant d'écrire la courbe sous forme de Montgomery est défini de la manière suivante :

1. soit α une racine de $x^3 + ax + b$,
2. soit s une racine carré de $(3\alpha^2 + a)^{-1}$,
3. $m : (x, y) \to (x/s + \alpha, y/s)$.

En plus des formules de compositions classiques, nous disposons de formules spécifiques, en coordonnées projectives, permettant de calculer les coordonnées en x et en z en s'affranchissant du calcul de la coordonnée en y. Plus précisément elles permettent de calculer les coordonnées en x et en z du point $[m+n]P$ à partir des coordonnées en x et en z des points $[m]P$, $[n]P$ et $[m-n]P$.

Addition ($m \neq n$)

$$\begin{aligned} X_{m+n} &= Z_{m-n}((X_m - Z_m)(X_n + Z_n) + (X_m + Z_m)(X_n - Z_n))^2, \\ Z_{m+n} &= X_{m-n}((X_m - Z_m)(X_n + Z_n) - (X_m + Z_m)(X_n - Z_n))^2. \end{aligned}$$

Doublement ($m = n$)

$$\begin{aligned} 4X_nZ_n &= (X_n + Z_n)^2 - (X_n - Z_n)^2, \\ X_{2n} &= (X_n + Z_n)^2(X_n - Z_n)^2, \\ Z_{2n} &= 4X_nZ_n((X_n - Z_n)^2 + ((A+2)/4)(4X_nZ_n)). \end{aligned}$$

Ici, pour tout i, le triplet (X_i, Y_i, Z_i) représente le point $[i]P$.

Le coût d'une addition est dans ce cas de 4M+2C (3M+2C si le point $[m-n]P$ est en coordonnées affines) et celui d'un doublement est de 3M+2C.

On citera également le papier de Brier et Joye [BJ02], dans lequel on trouve une généralisation de ce type de formules à n'importe quelle courbe elliptique. Les formules sont les suivantes :

Addition ($m \neq n$)

$$\begin{aligned} X_{m+n} &= Z_{m-n}(-bZ_mZ_n(X_mZ_n + X_nZ_m) + (X_mX_n - aZ_nZ_m)^2), \\ Z_{m+n} &= X_{m-n}(X_mZ_n - X_nZ_m)^2. \end{aligned}$$

Doublement ($m = n$)

$$\begin{aligned} X_{2n} &= (X_n^2 - aZ_n^2)^2 - 8bX_nZ_n^3, \\ Z_{2n} &= 4Z_n(X_n(X_n^2 + aZ_n^2) + bZ_n^3). \end{aligned}$$

Le nombre d'opérations passe alors à 9M+2C pour l'addition et à 6M+3C pour le doublement.

Coordonnées	Addition	Addition mixte	Doublement
Affines	I+2M+C	-	I+2M+2C
Projectives	12M+2C	9M+2C	7M + 5C
Jacobiennes	12M+4C	8M+3C	4M+6C
Chudnovsky	11M+3C	7M+2C	5M+6C
Modifiées	13M+6C	9M+5C	4M+4C
Brier & Joye	9M+2C	9M+2C	6M+3C
Montgomery	4M+2C	3M+2C	3M+2C

Tab. 1.1 – Coût des différents types de coordonnées sur $\mathbb{F}_p$

Doublement		Addition	
Opération	Coût	Opération	Coût
$2\mathcal{J}^m \to \mathcal{J}^c$	4M+5C	$\mathcal{J}^m + \mathcal{J}^c \to \mathcal{J}^m$	12M+5C
$2\mathcal{A} \to \mathcal{J}^c$	3M+5C	$\mathcal{J} + \mathcal{J}^c \to \mathcal{J}^m$	12M+5C
$2\mathcal{J}^m \to \mathcal{J}$	3M+4C	$\mathcal{J}^c + \mathcal{J}^c \to \mathcal{J}^m$	11M+4C
$2\mathcal{A} \to \mathcal{J}^m$	3M+4C	$\mathcal{J} + \mathcal{A} \to \mathcal{J}^m$	9M+5C
$2\mathcal{A} \to \mathcal{J}$	2M+4C	$\mathcal{J}^m + \mathcal{A} \to \mathcal{J}^m$	9M+5C
-	-	$\mathcal{J}^c + \mathcal{A} \to \mathcal{J}^m$	8M+4C
-	-	$\mathcal{A} + \mathcal{A} \to \mathcal{J}^m$	5M+4C

Tab. 1.2 – Coût des doublements et additions mixtes sur $\mathbb{F}_p$

1.3 Choix des coordonnées dans $\mathbb{F}_{2^n}$

Nous reprenons ici les différentes coordonnées utilisées sur des courbes définies sur des corps binaires. Le tableau 1.3 synthétise les coûts des différentes opérations en fonction des différents types de coordonnées.

Coordonnées projectives et jacobiennes

Sur $\mathbb{F}_{2^n}$ la notion de coordonnées projectives et jacobiennes reste identique à celle sur $\mathbb{F}_p$, c'est pour cela que nous ne présenterons pas les formules explicites. Nous rappelons simplement qu'en coordonnées projectives une addition de points requiert 16M+2C et un doublement 12M+2C, alors qu'en coordonnées jacobiennes l'addition requiert 16M+3C et le doublement 5M+5C. Comme précédemment on pourra ce reporter à [CF06, HMV04] pour plus de détails.

Coordonnées de López et Dahab

López et Dahab [LD98] ont introduit en 1998 un autre système de coordonnées dans lequel le triplet $(X : Y : Z)$ représente le point $(X/Z, Y/Z^2)$. L'équation de la courbe devient alors $Y^2+XYZ = X^3Z+aX^2Z^2+bZ^4$, l'inverse de $(X : Y : Z)$ est $(X : XZ+Y : Z)$ et enfin $\mathcal{O} = (1 : 0 : 0)$.

Le coût en nombre d'opérations de ce système de coordonnées est de 13M+4C pour l'addition (10M+3C si l'un des points est en coordonnées affines) et de 6M + 4C pour le doublement. Plusieurs améliorations ont été apportées, on citera [HT00, ADMRK02] en ce qui concerne les formules d'addition et [Lan04] pour ce qui est du doublement.

Coordonnées mixtes

Tout comme sur $\mathbb{F}_p$, il est possible d'utiliser un système de coordonnées mixtes. Cela dit, l'approche n'est ici pas très pertinente dans la mesure où, en pratique, le meilleur choix s'avère être d'utiliser exclusivement les coordonnées de López et Dahab (avec des additions mixtes López-Dahab+affine) ou les coordonnées affines (ce choix dépendant du coût de l'inversion).

Formules de Montgomery

Comme l'ont montré López et Dahab [LD99], des formules similaires à celles développées dans le cas des courbes de Montgomery existent dans le cadre des courbes définies sur $\mathbb{F}_{2^n}$. La principale différence étant que toute courbe elliptique définie sur $\mathbb{F}_{2^n}$ peut être mise sous forme de Montgomery. Les formules de compositions de points reposent toujours sur le fait de calculer seulement les coordonnées en x et en z de la somme de deux points en tirant partie de la connaissance des coordonnées en x de en z de la différence de ces deux points. On obtient ainsi une addition de points en 4M+1C (voire 3M+1C) et un doublement en 2M+3C.

Coordonnées	Addition	Addition mixte	Doublement
Affines	I+2M+C	-	I+2M+C
Projectives	16M+2C	12M+2C	8M + 4C
Jacobiennes	16M+3C	11M+3C	5M+5C
Lòpez-Dahab	13M+4C	9M+5C	4M+5C
Montgomery	4M+1C	4M+1C	2M+3C

Tab. 1.3 – Coût des différents types de coordonnées sur $\mathbb{F}_{2^n}$

1.4 Courbes elliptiques et cryptographie

Nous avons dit que le principe de la cryptographie à clé publique repose sur un couple de clés, l'une publique, l'autre privée. Retrouver la clé privée à partir de la clé publique doit revenir à résoudre un problème considéré comme difficile (en termes de temps de calcul). Dans le cas des courbes elliptiques, le problème en question est celui du logarithme discret (ou PLD). Nous allons donc le définir, analyser son niveau de sécurité dans le cadre les courbes elliptiques, puis faire une rapide comparaison avec RSA.

Définition 4 Soient G un groupe (noté additivement) et P un élément de G d'ordre fini. Soit $H =< P >$ le sous-groupe engendré par P, alors :

$$\forall Q \in H,\ \exists n \in \mathbb{N} : Q = nP;$$

n est appelé le logarithme discret de Q par rapport à P.

Le problème du logarithme discret dans un groupe consiste donc à retrouver l'entier n à partir de la donnée publique (H, P, Q). La sécurité des protocoles basés sur les courbes elliptiques repose sur la résolution de ce problème.

Exemple 1 Échange de clé de Diffie-Hellman

Supposons que deux personnes A et B veuillent partager un secret commun mais qu'elles ne puissent pas communiquer par un canal sécurisé. L'échange de clé de Diffie-Hellman se déroule de la manière suivante :

1. A et B choisissent publiquement un groupe G et un point $P \in G$,
2. A choisit secrètement un entier a et calcule $(a \times P)$,
3. B choisit secrètement un entier b et calcule $(b \times P)$,
4. A et B échangent publiquement les données $(a \times P)$ et $(b \times P)$,
5. A peut calculer $[a \times (b \times P)]$,
6. B peut calculer $[b \times (a \times P)]$,
7. A et B possèdent tous deux la quantité $(a \times b \times P)$ qui sera leur secret commun.

Un attaquant éventuel n'aura quant à lui en sa possession que les données G, P, $a \times P$, $b \times P$ pour retrouver $a \times b \times P$. C'est ce que l'on appelle le problème de Diffie Hellman calculatoire (PDHC). Il est évident que le PDHC est réductible au PLD, dans le sens où savoir résoudre le PLD permet la résolution immédiate du PDHC. La réduction réciproque a été prouvée dans plusieurs cas et on considère en général ces deux problèmes comme équivalents. De ce point de vue la sécurité de l'échange de Diffie-Hellman repose bien sur la difficulté du PLD.

Un cryptosystème reposant sur ce problème peut être alors définie par la donnée d'un quadruplet (G, P, ab, abP). La donnée publique est le triplet (G, P, abP) et la clé privé est l'entier ab. Remarquons d'ailleurs que G étant d'ordre fini la clé privée peut se restreindre à $ab \mod \#G$, ce qui permet dans limiter la taille.

Il existe très peu de résultats théoriques concernant la difficulté supposée du PLD sur les groupes usuellement utilisés en cryptographie. En pratique, un groupe est considéré comme plus ou moins sûr selon que l'on connaisse ou non un algorithme permettant d'y résoudre le PLD en temps raisonnable. Cependant aucun résultat ne permet de dire si de meilleurs algorithmes existent ou non.

Il existe toutefois, dans le cadre des groupes génériques, des résultats théoriques justifiant l'utilisation du PLD comme primitive cryptographique. Rappelons qu'un groupe générique est un groupe pour lequel on ne dispose que de l'opération de composition de deux éléments et du test d'égalité, sans pour autant connaître explicitement la loi de groupe. Dans ce cadre précis, on a le résultat suivant :

Théorème 3 Shoup, 1997 Soient G un groupe d'ordre l et p le plus grand nombre premier divisant l, alors la résolution du PLD par un algorithme générique nécessite $O(p^{\frac{1}{2}})$ opérations.

Pour la démonstration, on pourra se reporter directement à l'article de Shoup [Sho97].

Ce résultat est important ; il dit en substance qu'on ne peut, a priori, pas résoudre le PLD en temps polynômial (ni même sous-exponentiel) dès lors que l'ordre du groupe concerné est premier ou divisible par un grand nombre premier. Bien entendu, un groupe peut être muni de structures supplémentaires (anneau, corps, extensions, etc) pouvant conduire à des algorithmes de résolution bien plus efficaces. L'exemple typique est celui du groupe additif $\mathbb{Z}/n\mathbb{Z}$ dans lequel, une fois muni de sa structure d'anneau, la résolution du PLD se réduit au calcul d'un inverse modulo n.

Cette remarque reste vraie dans le cas des courbes elliptiques, cependant toutes les attaques connues tirant partie d'un surplus de structure ne s'appliquent qu'à des courbes particulières. Elles peuvent donc être contrées par un choix de paramètres judicieux. On se reportera au chapitre 5 de [CF06] pour un aperçu de ces attaques et des contre-mesures associées.

Nous ne disposons donc pas de meilleurs algorithmes que les algorithmes génériques pour résoudre le PLD sur une courbe bien choisie. Ainsi, si l'on considère que 2^{80} opérations sont impossibles à effectuer dans un laps de temps raisonnable, le PLD sur une courbe elliptique dont le cardinal est divisible par un nombre premier de 160 bits est impossible à résoudre ; mais encore faut il que de telles courbes existent. Le théorème de Hasse permet de répondre à cette question par l'affirmative en donnant une estimation du cardinal d'une courbe elliptique définie sur un corps fini. Ce théorème est très important dans le cadre cryptographique : il assure que le nombre d'éléments du groupe des points d'une courbe elliptique est du même ordre que le cardinal du corps de définition de la courbe.

Théorème 4 H. Hasse, 1933

Soit $\mathbb{K} = \mathbb{F}_q$ un corps fini. Alors $E(\mathbb{F}_q)$ est de cardinal fini et

$$\#E(\mathbb{F}_q) = q + 1 - t, \tag{1.8}$$

où t vérifie : $|t| \leq 2\sqrt{q}$.

Cela signifie que pour avoir une sécurité de 2^80 opérations, il faut utilisé une courbe définie sur un corps d'au moins 2^160 éléments. Ainsi l'utilisation de protocoles basés sur les courbes elliptiques requiert l'utilisation de clés (privée comme publiques) codées sur seulement 160 bits : d'une part la clé privée est un entier inférieur au cardinal de la courbe et d'autre part la clé publique est un élément de la courbe, c'est-à-dire un élément du corps de base (il est en effet possible de ne conserver que la coordonnée en x du point plus un bit dépendant de la coordonnée en y comme donnée sans perte d'information ; voir [CF06] pages 289 et 302) lui aussi codé sur 160 bits. C'est là le gros avantage des courbes elliptiques par rapport à RSA puisque pour un niveau de sécurité équivalent, RSA nécessite l'utilisation de clé de plus de 1024 bits. De plus, le rapport de taille devient de plus en plus important à mesure que le niveau de sécurité augmente. Ainsi une clé ECC de 256 bits est aussi robuste qu'une clé RSA de 3072 bits. Pis encore, avoir le niveau de sécurité d'une clé ECC de 512 bits requiert l'utilisation de clés RSA de 15360 bits ! Cela vient tout simplement du fait qu'il existe des algorithmes pour déchiffrer RSA de complexité sous-exponentielle. Dans le tableau 1.4 nous donnons différentes tailles de clé ayant des niveaux de sécurité équivalent sur différents algorithmes.

Sécurité (en bits)	RSA	ECC
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Tab. 1.4 – Comparaison de tailles de clé à niveau de sécurité équivalent

1.5 Attaques par canaux cachés

Si la robustesse théorique d'un problème est un pré-requis à l'élaboration d'un protocole cryptographique, cela n'est en rien une condition suffisante à sa sécurité en pratique. Ainsi, de même que le PLD, prouvé difficile sur un groupe générique, peut être facilement résolu dès lors qu'une instance de groupe est choisie, un protocole cryptographiquement sûr sur le papier peut être cassé lors de son application pratique.

L'idée n'est pas nouvelle, par exemple, une très ancienne méthode de chiffrement consiste à utiliser de l'encre sympathique pour écrire un message. Si le procédé peut paraître relativement sûr, il est possible de récupérer le message original sans même avoir le message chiffré en sa possession. En effet, pour peu que la feuille de papier sur laquelle a été écrit le message soit issue d'un cahier ou d'un bloc de feuille quelconque, il est possible de retrouver le message d'origine en examinant les sillons laissés par la plume sur les pages inférieures. Ces pages peuvent être vues alors comme un canal d'information caché.

Le même type d'attaque peut être mis en oeuvre dans le cadre des procédés modernes de chiffrement (et donc en particulier dans celui des courbes elliptiques). En 1995 Kocher proposa une première attaque basée sur l'analyse du temps de calcul d'une carte à puce [Koc96] effectuant une exponentiation modulaire. Plus généralement l'exécution d'un algorithme de chiffrement par un processeur peut laisser échapper de nombreuses informations en rapport avec la clé de chiffrement. Ces informations peuvent être le temps de calcul [Sch00], la consommation d'énergie [KJJ99], le rayonnement électromagnétique [QS01], voire même le son émis par le processeur. L'évolution dans le temps de ces canaux cachés dépend en grande partie des données manipulées par le processeur ainsi que des opérations effectuées par celui-ci. Une étude précises des données peut alors permettre de distinguer, par exemple, une multiplication d'un carré modulaire, ou une addition d'un doublement de point. Nous verrons dans le chapitre suivant que bon nombre de méthodes classiques de multiplication de point sur une courbe elliptique sont basées sur une succession de doublements et d'addition de points dont l'enchaînement dépend directement de la clé. Être capable de faire la distinction entre ces doublements et ces additions permet donc de récupérer la clé de chiffrement sans même à avoir à résoudre un quelconque problème mathématique.

Pour illustrer ceci, prenons l'exemple du protocole RSA. Soient p, q deux nombres premiers, $N = pq$, $\phi(N) = (p-1)(q-1)$ l'indicateur d'Euler, et enfin d et e deux entiers tels que $ed \equiv 1 \mod \phi(N)$. Dans ce contexte, la clé publique est le couple (e, N) et la clé privée le triplet (d, p, q). Pour chiffrer un message m (vu comme un élément de $\mathbb{Z}/N\mathbb{Z}$), on calcule $c = m^e \mod N$. Pour déchiffrer il suffit alors de calculer $c^d \mod N \equiv m^{ed} \mod N$. Or $ed \equiv 1 \mod \phi(N)$, et donc d'après le théorème d'Euler ($\forall x \in \mathbb{N}$ tel que $\text{pgcd}(x, N) = 1, x^{\phi(N)} \equiv 1 \mod N$), on a bien $c^d \mod N = m$.

La phase déchiffrement consiste donc en une exponentiation modulaire. Celle-ci peut s'effectuer grâce à l'algorithme 1 dont le comportement dépend directement des bits de la clé privée.

Algorithme 1 : Carré et multiplication

```
Données : x ∈ ℤ/Nℤ et d = (d_{l-1} ... d_0)_2 ∈ ℕ.
Résultat : x^d mod N.
début
    y ← x
    pour i = l − 2 à 0 faire
        y ← y^2 mod N
        si d_i = 1 alors
            y ← y × x mod N
        fin
    fin
fin
retourner y
```

Si un attaquant est capable de mesurer, par exemple, la consommation d'énergie du processeur effectuant l'exponentiation, il peut alors être capable de déduire les bits de la

clé privé. En effet, à chaque étape de l'algorithme une mise au carré est effectuée, suivie d'une multiplication si le bit de clé concerné est un 1. Très souvent, la multiplication est une opération beaucoup plus coûteuse que l'élévation au carré, ainsi son exécution fait apparaître un pic de consommation (puisqu'elle demande plus de ressources) et peut être distinguée d'un carré très facilement, comme s'illustre la figure 1.5. Il est facile de déduire les bits de la clé à partir de la lecture du graphique de consommation du processeur.

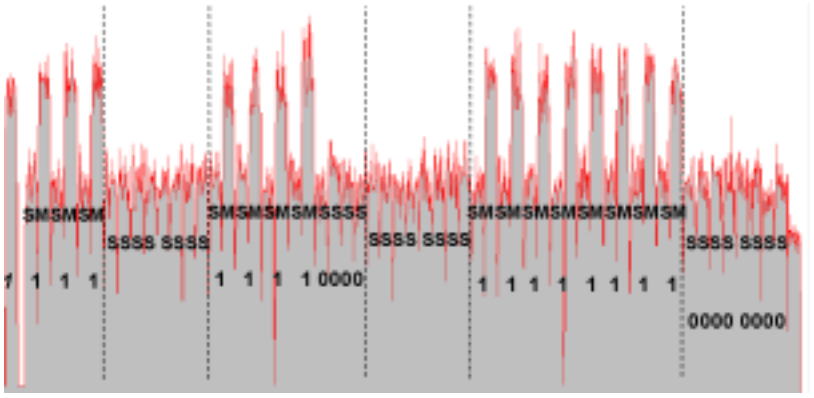


Fig. 1.3 – Analyse de consommation d'une exponentiation modulaire

Chapitre 2

Multiplication par un Scalaire

Nous avons vu dans le chapitre précédent (par exemple dans le cas de l'échange de clé de Diffie-Hellman) que la principale opération lors d'un protocole cryptographique à base de courbes elliptiques est le calcul de l'image du point P par le morphisme de multiplication par un scalaire. Le but de ce chapitre est de faire un état des lieux des différentes techniques actuelles permettant de réaliser cette opération sur les courbes et d'en étudier la complexité.

2.1 Méthodes génériques

Cette section est dédiée aux algorithmes génériques de multiplication par un scalaire les plus communément utilisés avec les courbes elliptiques. L'adjectif générique signifiant ici que ces algorithmes peuvent être utilisés sur n'importe quel groupe. Notons que l'on trouve souvent ces algorithmes sous l'appellation d'algorithmes d'exponentiation. Ce sont bien entendu exactement les même algorithmes à ceci prés que le groupe est vu comme un groupe multiplicatif est non comme un groupe additif. C'est donc juste une question de notations. Et cela justifie l'abus de langage consistant à parler d'algorithme d'exponentiation sur les courbes elliptiques.

Doublement et addition

Il existe de nombreux algorithmes permettant d'effectuer une multiplication de point par un scalaire. Ils découlent en général directement de la façon dont on représente ledit scalaire. Commençons par un algorithme classique basé sur la représentation binaire du scalaire. C'est principalement une série de doublements entrecoupée d'additions (dépendantes du nombre de 1 dans la représentation du scalaire). Nous verrons par la suite que la plupart des autres algorithmes en sont des raffinements.

Algorithme 2 : Doublement et addition

Données : $P \in E(\mathbb{K})$ et $k = (k_{l-1} \dots k_0)_2 \in \mathbb{N}$.
Résultat : $[k]P \in E(\mathbb{K})$.
début
  $Q \leftarrow P$
  pour $i = l - 2$ à 0 faire
    $Q \leftarrow [2]Q$
    si $k_i = 1$ alors
      $Q \leftarrow Q + P$
    fin
  fin
fin
retourner Q

Comme dit plus haut, cet algorithme n'est que la version additive de l'algorithme 1, ainsi toutes les améliorations décrites dans la suite de cette section ne sont que des traductions de celles développées dans le cadre l'exponentiation. Une étude rapide de la complexité de cet algorithme permet de voir qu'il est nécessaire d'effectuer $l - 1$ doublements plus un certain nombre d'additions égal poids de Hamming de k (c'est-à-dire du nombre de bits non nuls dans la représentation binaire de k exception faite du bit de poids fort), que l'on notera $w(k)$ (en moyenne $w(k) = l/2$).

Ramené en nombre d'opérations sur le corps de base on obtient, par exemple :

1. $(l + w(k) - 1)\mathrm{I} + 2(l + w(k) - 1)\mathrm{M} + (2l + w(k) - 2)\mathrm{C}$ en coordonnées affines sur $\mathbb{F}_p$
2. $(l + w(k) - 1)\mathrm{I} + 2(l + w(k) - 1)\mathrm{M} + (l + w(k) - 1)\mathrm{C}$ en coordonnées affines sur $\mathbb{F}_{2^n}$,
3. $4(l + 2w(k) - 1)\mathrm{M} + (4l + 5w(k) - 4)\mathrm{C}$ en coordonnées mixtes sur $\mathbb{F}_p$
4. $(4l + 9w(k) - 4)\mathrm{M} + 5(l + w(k) - 1)\mathrm{C}$ en coordonnées mixtes sur $\mathbb{F}_{2^n}$.

Pour des questions de comparabilité nous retiendrons surtout le comportement en moyenne des différents algorithmes. Dans le cas présent, nous considérons que le scalaire est pris au hasard, et que donc la densité de 1 dans sa représentation binaire est de $1/2$. Ainsi, en coordonnées mixtes sur $\mathbb{F}_p$, la complexité en moyenne de l'algorithme 2 est $4(2l - 1)\mathrm{M} + (13l/2 - 4)\mathrm{C}$.

Le tableau 2.1 page 28 offre un récapitulatif de la complexité des différents algorithmes traités dans cette section. Notons qu'afin de simplifier la lecture de ces tableaux nous faisons l'hypothèse que le ratio entre le coût d'un carré et celui d'une multiplication sur un corps est de 0.8. C'est une hypothèse classique de la littérature sur le sujet (notamment dans [CMO98]). Les meilleures implantations logicielles permettent d'obtenir un ratio plutot de l'ordre de 0.66 [?]. Cependant, obtenir un tel résultat nécessite d'utiliser un corps particulier. L'hypothèse C=0.8M n'est pas complétement déraisonnable et permet surtout de facilement comparer les différents résultats entre eux.

Forme non-adjacente

Nous avons vu dans les deux sections précédentes que le calcul de l'opposé d'un point sur une courbe elliptique est une opération très peu coûteuse. Par exemple, comme nous l'avons vu dans le chapitre précédent, si $P = (x, y)$ est un point d'une courbe définie sur un corps de caractéristique supérieure ou égale à 5, alors $-P = (x, -y)$. Il peut donc être intéressant d'utiliser une représentation signée du scalaire afin d'en diminuer le nombre de chiffres non nuls. L'idée consistant à recoder des éléments d'un calcul pour en améliorer l'efficacité n'est évidemment pas nouvelle, déjà en 1951 Booth avait utilisé cette approche dans le cadre de la multiplication de deux entiers [Boo51]. De nombreux recodage ont ainsi été proposés depuis. Citons par exemple [JY00] ou encore [PB04]. Parmi toutes ces représentations la forme non-adjacente (NAF en anglais) est particulièrement appropriée.

Définition 5 Soit k un entier positif. La forme non-adjacente de k est son écriture sous la forme $k = \sum_{i=0}^{l'-1} k_i 2^i$ où $k_i \in \{0, \pm 1\}, k_{l'-1} \neq 0$ et $\forall\ i \geq 0,\ k_i k_{i+1} = 0$.

Proposition 4 La forme non-adjacente est intéressante grâce aux propriétés suivantes :

1. Tout entier k possède une unique forme non-adjacente (que l'on notera NAF(k)).
2. NAF(k) est une représentation signé de poids de Hamming minimal (c'est-à-dire qu'elle contient le moins de chiffres non nuls possible).
3. La forme non-adjacente d'un entier possède au plus un chiffre de plus que sa représentation binaire (c'est-à-dire que $l' = l$ ou $l' = l + 1$).
4. La densité moyenne de chiffres non nuls est approximativement de $1/3$.

Calculer la forme non-adjacente d'un entier est relativement facile, il suffit d'appliquer l'algorithme suivant :

Algorithme 3 : Calcul de la forme non-adjacente d'un entier

```
Données : Un entier k ≥ 1.
Résultat : NAF(k).
début
    l' ← 0
    tant que k ≥ 1 faire
        si k est impair alors
            k_{l'} ← 2 − (k mod 4), k ← k − k_{l'}
        sinon
            k_{l'} ← 0
        fin
        k ← k/2, l' ← l' + 1
    fin
fin
retourner (k_{l'−1} k_{l'−2} ... k_1 k_0)
```

On peut alors modifier l'algorithme 2 afin d'obtenir un algorithme de multiplication par un scalaire tirant parti de la forme non-adjacente du scalaire.

Algorithme 4 : Doublement et addition signée

Données : $P \in E(\mathbb{K})$ et $k \in \mathbb{N}$.
Résultat : $[k]P \in E(\mathbb{K})$.
début
  Calculer NAF$(k) = (k_{l'-1}k_{l'-2}\dots k_1k_0)$;
  $Q \leftarrow P$
  pour $i = l' - 2$ à 0 faire
    $Q \leftarrow [2]Q$
    si $k_i = 1$ alors
      $Q \leftarrow Q + P$
    fin
    si $k_i = -1$ alors
      $Q \leftarrow Q - P$
    fin
  fin
fin
retourner Q

Méthodes à fenêtres

Il est possible de creuser encore plus la représentation de k en élargissant l'intervalle des valeurs que peuvent prendre les chiffres. Cela donne lieu à une version généralisée de la forme non-adjacente appelée forme non-adjacente de largeur w (traduction peu inspirée de width-w NAF).

Définition 6 Soient k un entier positif et w un entier supérieur ou égal à deux, la forme non-adjacente de largeur w de k est son écriture sous la forme $k = \sum_{i=0}^{l'-1} k_i 2^i$ où $|k_i| < 2^{w-1}, k_{l-1} \neq 0$ et toute séquence de w bits consécutifs comporte au plus un bit non nul.

Proposition 5

1. Tout entier k possède une unique forme non-adjacente de largeur w (que l'on notera NAF$_w(k)$).
2. NAF$_2(k) =$ NAF(k).
3. Pour tout $w \geq 2$ NAF$_w(k)$ possède au plus un chiffre de plus que la représentation binaire de k.
4. La densité de chiffres non nuls est approximativement de $1/(w + 1)$.

Calculer la forme non-adjacente de largeur w d'un entier se fait à partir de l'algorithme suivant :

Algorithme 5 : Calcul de la forme non-adjacente de largeur w d'un entier

Données : Deux entiers $k \geq 0$ et $w \geq 2$.
Résultat : $\mathrm{NAF}_w(k)$.
début
  $l' \leftarrow 0$
  tant que $k \geq 1$ faire
    si k est impair alors
      $k_{l'} \leftarrow k \mod 2^w$
      si $k_{l'} \geq 2^{w-1}$ alors
        $k_{l'} \leftarrow k_{l'} - 2^w$
      fin
      $k \leftarrow k - k_i$
    sinon
      $k_l \leftarrow 0$
    fin
    $k \leftarrow k/2, l' \leftarrow l' + 1$
  fin
fin
retourner $(k_{l'-1}k_{l'-2}\ldots k_1k_0)$

Exemple 2 Considérons l'entier $k = 11235813$. Pour des questions de lisibilité, nous écrirons $\overline{k_i}$ au lieu de $-k_i$. L'exemple qui suit montre la représentation binaire de k ainsi que ses différentes formes adjacentes de longueur 2 à 6 :

$k_2 =$	$1\,0\,1\,0$	$1\,0\,1\,1$	$0\,1\,1\,1$	$0\,0\,0\,1$	$1\,1\,1\,0$	$0\,1\,0\,1$
$NAF_2(k) =$	$1\;0\,\overline{1}\,0\,\overline{1}$	$0\,\overline{1}\,0\,0$	$\overline{1}\,0\,0\,\overline{1}$	$0\,0\,1\,0$	$0\,0\,\overline{1}\,0$	$0\,1\,0\,1$
$NAF_3(k) =$	$1\;0\,0\,\overline{3}\,0$	$0\,3\,0\,0$	$\overline{1}\,0\,0\,\overline{1}$	$0\,0\,1\,0$	$0\,0\,0\,0$	$\overline{3}\,0\,0\,\overline{3}$
$NAF_4(k) =$	$1\,0\,0\,0$	$5\,0\,0\,0$	$7\,0\,0\,0$	$0\,0\,\overline{7}\,0$	$0\,0\,\overline{1}\,0$	$0\,0\,0\,5$
$NAF_5(k) =$	$1\,0\,0\,0$	$0\,11\,0\,0$	$0\,0\,0\,\overline{9}$	$0\,0\,0\,0$	$0\,0\,15\,0$	$0\,0\,0\,5$
$NAF_6(k) =$	$5\,0$	$0\,0\,0\,0$	$23\,0\,0\,0$	$0\,0\,\overline{7}\,0$	$0\,0\,0\,0$	$0\,0\,0\,\overline{27}$

Nous pouvons remarquer que, comme souhaité, la densité de chiffres non nuls diminue en même temps que la taille de la fenêtre augmente.

On peut donc maintenant donner un algorithme de multiplication par un scalaire semblable à l'algorithme 2, tirant cette fois parti de la forme adjacente de largeur w du scalaire :

Algorithme 6 : Doublement et addition avec fenêtres

Données : $P \in E(\mathbb{K})$, $k \in \mathbb{N}$ et $w \geq 2$.
Résultat : $[k]P \in E(\mathbb{K})$.
début
  Calculer $NAF_w(k) = (k_{l'-1}k_{l'-2}\dots k_1k_0)$;
  Calculer $P_i = [i]P$ pour $i \in \{1, 3, 5, \dots, 2^{w-1} - 1\}$
  $Q \leftarrow P_{k_{l'-1}}$
  pour $i = l' - 2$ à 0 faire
    $Q \leftarrow [2]Q$
    si $k_i \neq 0$ alors
      si $k_i > 0$ alors
        $Q \leftarrow Q + P_{k_i}$
      sinon
        $Q \leftarrow Q - P_{-k_i}$
      fin
    fin
  fin
fin
retourner Q

L'algorithme 6 ne précise pas la manière dont sont précalculés les P_i. Un manière de faire est de calculer $[2]P, [3]P, \dots, [2^{w-1} - 1]P$ en coordonnées jacobiennes de Chudnovsky. Si l'on veut utiliser des P_i en coordonnées affines, il faut alors calculer les points $P'_i = (\frac{X_{P_i}}{Z^2_{P_i}} : \frac{Y_{P_i}}{Z^3_{P_i}} : 1)$, nécessitant en gros $i/2$ inversions. On peut utiliser l'astuce de Montgomery (consistant à calculer l'inverse de $Z_{P_3}Z_{P_5}\dots Z_{P_{2^{w-1}-1}}$ et à récupérer les différents inverses par de simples multiplications) pour limiter à un le nombre d'inversions, tout en rajoutant $(3(2^{w-2} - 1) - 3)$ multiplications.

Base double

Toujours dans l'idée de diminuer le nombre d'additions, une méthode récente d'exponentiation repose sur une représentation du scalaire en base double. L'idée d'utiliser ce système de représentation a d'abord été introduite par Dimitrov et al. dans le domaine du traitement du signal ??. Rapidement ce système s'est avéré très intéressant concernant l'exponentiation modulaire [DJM98]. Puis vinrent de nombreux développements dans le cadre de la multiplication par un scalaire sur les courbes elliptiques [ACS04, ADDS06, DI06, DIM07], auxquels nous allons maintenant nous intéresser.

Pour cela nous allons voir en quoi consiste ce nouveau type de bases puis décrire un des algorithmes de multiplication scalaire qui en découlent.

Définition 7 Soit k un entier positif. On appelle représentation en base double de k toute écriture de k de la forme :

$$k = \sum_{i,j} a_{i,j} 2^i 3^j, \qquad a_{i,j} \in \{0,1\}.$$

Exemple 3

$$\begin{aligned} 127 &= 2^2 3^3 + 2^1 3^2 + 2^0 3^0 \\ &= 2^5 3^1 + 2^2 3^0 + 2^0 3^3 \\ &= 2^4 3^0 + 2^2 3^3 + 2^1 3^0 + 2^0 3^0 \\ &= 2^2 3^3 + 2^2 3^1 + 2^2 3^0 + 2^1 3^0 + 2^0 3^0 \end{aligned}$$

Il est très facile de voir qu'une telle écriture existe toujours ; en effet, la définition précédente contient par exemple les représentations en base 2 ou 3. L'intérêt de ce nouveau système est qu'il existe énormément de représentations différentes pour un même entier, dont certaines très creuses. Par exemple le nombre 10 possède exactement 5 représentations différentes, 100 en possède 402 et 1000 en possède 1 295 579. Parmi toutes ces représentations, les plus intéressantes sont celles possédant le plus petit nombre de $a_{i,j}$ non nuls (on parlera de bits non nuls par abus de langage), on parle dans ce cas de représentation canonique. Ces représentations sont extrêmement creuses, par exemple celles de tous les entiers strictement inférieurs à 431 ont au plus trois bits non nuls.

Le seul inconvénient est qu'il est très difficile de trouver une représentation canonique pour un entier (la seule méthode connue étant la recherche exhaustive), c'est pour cela qu'en pratique on utilise des représentations quasi-canoniques obtenues à partir de l'algorithme glouton 7.

Même si les représentations obtenues sont rarement canoniques, il a été prouvé qu'elles restent très creuses. Plus précisément le nombre de bits non nuls est de l'ordre de $\log(k)/\log\log(k)$, c'est à dire sous linéaire par rapport à la taille de l'entier concerné (alors que ce nombre est d'environ $\log(k)/2$ pour la représentation binaire classique).

Algorithme 7 : Algorithme Glouton

Données : Un entier k.
Résultat : k sous la forme $\sum_{i,j} a_{i,j} 2^i 3^j, \quad a_{i,j} \in \{0,1\}$.
début
  tant que $k \neq 0$ faire
    Trouver la meilleure approximation de k sous la forme $z = 2^i 3^j$
    $a_{i,j} \leftarrow 1$
    $k \leftarrow k - z$
  fin
fin
retourner $\sum_{i,j} a_{i,j} 2^i 3^j$

Concernant l'opération consistant à approcher un entier par un autre entier de la forme $2^i 3^j$ sachons simplement qu'elle peut s'effectuer très rapidement, en particulier en utilisant

la méthode décrite dans [BI04] et que le coût d'une telle réécriture est donc négligeable par rapport au coût d'une exponentiation modulaire ou à celui d'une multiplication de point par un scalaire.

Maintenant que l'on sait comment représenter un entier en base double, on peut développer un premier algorithme basique de multiplication par un scalaire ; il suffit pour cela de calculer $[2^i3^j]P$ pour tout les $a_{i,j}$ non nuls, et de faire la somme des points ainsi obtenus. Des formules de triplement ont été proposées [DJM98] afin d'optimiser le calcul des $[2^i3^j]P$. Ces formules sont les suivantes :

Triplement

Soient $P_1 = (X_1 : Y_1 : Z_1)$ et $[3]P_1 = (X_3 : Y_3 : Z_3)$. Alors en posant :

$$M = 3X_1^2 + aZ_1^4, \quad E = 12X_1Y_1^2 - M^2, \quad T = 8Y_1^4$$

on a

$$\begin{aligned} X_3 &= 8Y_1^2(T - ME) + X_1E^2, \\ Y_3 &= Y_1(4(ME - T)(2T - ME) - E^3), \\ Z_3 &= Z_1E. \end{aligned}$$

Le coût total de ces formules est de 10M+6C.

En suivant la même idée que Cohen et. al. dans [CMO98], il a été montré [DJM98] que si plusieurs triplements consécutifs doivent être effectués, il possible d'économiser un carré pour tous les triplements suivants le premier en réutilisant la quantité aZ_1^4. En effet, si l'on veut tripler le point $[3]P_1$, il faut calculer la quantité $M = 3X_3^2 + aZ_3^4$. Or $aZ_3^4 = (aZ_1^4) \times (E^4)$. La quantité E^2 ayant été elle aussi calculée lors du triplement précédent, il possible de calculer $3X_3^2 + aZ_3^4$ pour seulement 1M+2C au lieu des 1M+3C du premier triplement.

Malgré ces formules efficaces, l'indépendance relative des couples de (i, j) font qu'il est rare de pouvoir réutiliser une grande partie des calculs d'un point pour les calculs du suivant. C'est afin de pallier cet inconvénient qu'ont été introduites les chaînes de base double.

Définition 8 Soit k un entier, une chaîne de base double calculant k est une suite $(C_i)_{i\geq 1}$ vérifiant

$$C_1 = 1, \quad C_{i+1} = 2^{u_i}3^{v_i}C_i + 1,$$

avec $u_i, v_i \geq 0$ et $k = C_l$ ou $k = 2^a3^bC_l$ pour un certain l.

Une autre façon de voir les choses est de considérer une des écritures en base double de $k = \sum_{i=1}^{l} c_i 2^{a_i}3^{b_i}$ où $(a_i)_{i\geq 1}$ et $(b_i)_{i\geq 1}$ sont deux suites décroissantes, de sorte qu'il est

possible d'écrire un simili de schéma de Horner :

$$k = 2^{a_l}3^{b_l}(2^{u_{l-1}}3^{v_{l-1}}(\dots(2^{u_1}3^{u_1}+1)\cdots+1)+1)$$

où $\forall i \in \{1,\dots l-1\}$ $u_i = a_i - a_{i+1}$ et $v_i = b_i - b_{i+1}$.

Remarquons que, là encore, une telle écriture existe toujours ; l'écriture en base 2 (ou 3) en est une fois de plus un cas particulier. Trouver une écriture d'un entier sous cette forme peut se faire relativement simplement en modifiant l'algorithme 7. On trouvera dans [DIM07] une étude sur la gestion des paramètres qu'il faut y introduire et des résultats obtenus.

Exemple 4 On a vu dans l'exemple précédent que

$$\begin{aligned} 127 &= 2^2 3^3 + 2^1 3^2 + 2^0 3^0 \\ &= 2^5 3^1 + 2^2 3^0 + 2^0 3^3. \end{aligned}$$

La première écriture est bien l'expression d'une chaîne de base double. Il suffit pour s'en convaincre de poser $(a_i)_{i=1\dots3} = (2,1,0)$ et $(b_i)_{i=1\dots3} = (3,2,0)$.

Cela signifie, en outre, que l'on peut écrire k sous la forme

$$k = 2^0 3^0(2^1 3^2(2^1 3^1 + 1) + 1). \tag{2.1}$$

Par contre, la deuxième écriture ne permet pas une telle factorisation de k et ne peut donc pas être associée à une chaîne de base double.

L'avantage de cette écriture est qu'elle permet alors d'effectuer la multiplication de point à l'aide de l'algorithme 8 de doublement-triplement et addition permettant de grandement diminuer le nombre de doublements et de triplements. Il est par contre évident que le nombre d'additions de points augmente. Même si cela n'a pas encore été démontré, il semble que l'on perde même le caractère sous-linéaire concernant le nombre de bits non nuls.

Algorithme 8 : Doublement-triplement et addition

Données : $P \in E(\mathbb{K})$ et $k = \sum_{i=1}^{l} c_i 2^{a_i} 3^{b_i} \in \mathbb{N}$.
Résultat : $[k]P \in E(\mathbb{K})$.
début
  $Q \leftarrow P$
  pour $i = 1$ à $l - 1$ faire
    $u_i \leftarrow a_i - a_{i+1}$, $v_i \leftarrow b_i - b_{i+1}$
    $Q \leftarrow [3^{v_i}]Q$
    $Q \leftarrow [2^{u_i}]Q$
    $Q \leftarrow Q + P$
  fin
  $Q \leftarrow [3^{b_l}]Q$
  $Q \leftarrow [2^{a_l}]Q$
fin
retourner Q

Tout comme l'algorithme de doublement et addition, l'algorithme précédent peut très bien faire l'objet de raffinements tels que l'utilisation de représentations signées et de méthodes à fenêtres. On se reportera à [DI06] pour plus de détails. Notons simplement qu'il n'existe pas de résultats théoriques sur le nombre moyen d'opérations à effectuer, voilà pourquoi nous ne faisons pas figurer cette approche dans le tableau comparatif 2.1. Enfin, en caractéristique 2, le doublement est tellement plus efficace que le triplement que les meilleurs résultats sont obtenus avec la représentation en base 2. Ceci explique pourquoi, comme précédemment, nous n'avons pas inclus de résultats pratiques concernant l'utilisation des bases doubles sur $\mathbb{F}_{2^n}$ dans le tableau 2.2.

L'échelle de Montgomery

Un algorithme moins connu, dû à Montgomery [Mon87], permet d'effectuer la multiplication scalaire d'une manière originale.

Nous avons vu que les formules de Montgomery permettent d'additionner deux points d'une courbe de façon très efficace dès lors que la différence de ces deux points est connue.

L'idée principale de l'algorithme consiste à effectuer une double exponentiation, tout en maintenant une différence constante entre les deux points concernés. Plus précisément, à chaque étape, l'algorithme calcule de manière entremêlée deux points P_1 et P_2, dont la différence est constamment égale à P. Il en résulte l'algorithme 9. Cet algorithme est parfaitement générique, dans la mesure où celui-ci peut être utilisé dans n'importe quel groupe. Il est toutefois important de garder à l'esprit qu'à l'origine, il a été inventé afin de tirer partie des formules de Montgomery sur les courbes. C'est dans ce cadre que s'exprime toute son ingéniosité et son efficacité.

Algorithme 9 : Échelle de Montgomery

```
Données : P ∈ E(𝕂), k = (k_{l-1} ... k_0)_2.
Résultat : [k]P.
début
    (P_1, P_2) ← (𝒪, P)
    pour i = l − 1 à 0 faire
        si k_i = 0 alors
            (P_1, P_2) ← ([2]P_1, P_1 + P_2)
        fin
        si k_i = 1 alors
            (P_1, P_2) ← (P_1 + P_2, [2]P_2)
        fin
    fin
fin
retourner P_1
```

Exemple 5 Supposons que l'on veuille calculer $[k]P$ avec $k = 25 = 11001_2$. L'échelle de Montgomery se déroule de la façon suivante :

	P_1	P_2
initialisation	$\mathcal{O}$	P
$k_4 = 1$	P	$[2]P$
$k_3 = 1$	$[3]P$	$[4]P$
$k_2 = 0$	$[6]P$	$[7]P$
$k_1 = 0$	$[12]P$	$[13]P$
$k_0 = 1$	$[25]P$	$[26]P$

L'analyse de la complexité est très intéressante, premièrement comme dit plus haut la différence $P_2 - P_1$ est constamment égale à P, ce qui explique pourquoi cet algorithme est particulièrement bien adapté aux formules d'addition de points sur les courbes de Montgomery (où rappelons-le, l'addition $P_1 + P_2$ peut se faire seulement si les points P_1 et P_2 et $P_2 - P_1$ sont connues). Ensuite, quel que soit le scalaire, l'algorithme effectue l additions et l doublements. Notons que la première addition est en fait une simple affectation de variable et que la dernière opération, celle calculant $[k+1]P$, n'est pas nécessaire. Cela dit, on peut considérer que la complexité de cet algorithme est en gros de l doublements et $l-1$ additions. En prenant en compte le fait que le point P est donné en coordonnées affines, on obtient, en termes d'opérations sur le corps, $(6l-4)$M+$(4l-2)$C sur une courbe de Montgomery définie sur $\mathbb{F}_p$ ($(15l-13)$M+$(5l-3)$C sur une courbe quelconque) et $(6l-7)$M + $(4l-2)$C sur $\mathbb{F}_{2^n}$.

Le tableau suivant résume les données importantes de ce paragraphe. Pour des questions de lisibilité on considérera (comme cela est le cas en pratique) que sur $\mathbb{F}_p$, C = 0.8M.

Concernant les corps binaires la mise au carré est une opération linéaire ($(a+b)^2 = a^2+b^2$) ce qui permet d'obtenir des implantations très peu gourmande en temps de calcul. En pratique on néglige donc le nombre de carrés dans le calcul de la complexité des différents algorithmes.

Les complexités données ci-dessous sont des décomptes du nombre d'opérations à effectuer, cependant dans le cas de la méthode à fenêtres le coût des pré-calculs n'a pas été pris en compte dans la mesure cela alourdi notablement les formules de complexités sans changer fondamentalement le nombre effectif d'opérations. Notons également que toutes les formules sont données en fonction d'un choix optimal de coordonnées mixtes, excepté dans le cas de l'échelle de Montgomery où le nombre d'opérations est celui obtenu sur les courbes de Montgomery (avec les formules spécifiques à ces courbes).

Algorithme	nbr. dbl/trpl/add	nbr. op. sur $\mathbb{F}_p$	nbr. op. sur $\mathbb{F}_{2^n}$
Double-and-add	$(l-1)$D + $\frac{l}{2}$A	$(13.2l-7.2)$M	$(8.5l-4)$M
NAF Double-and-add	$(l-1)$D + $\frac{l}{3}$A	$(11.2l-7.2)$M	$(7l-4)$M
w-NAF Double-and-add	$(l-1)$D + $\frac{l}{w+1}$A	$(\frac{7.2w+19.2}{w+1}l-7.2)$M	$(\frac{4w+13}{w+1}l-4)$M
Échelle de Montgomery	lD + $(l-1)$A	$(9.2l-5.6)$M	$(6l-7)$M

Tab. 2.1 – Complexité moyenne des différents algorithmes génériques

Algorithme	nbr. dbl/trpl/add	nbr. op. sur $\mathbb{F}_p$	nbr. op. sur $\mathbb{F}_{2^n}$
Double-and-add	159D + 80A	2104.8M	1382M
NAF Double-and-add	159D + 54A	1784.8M	1137M
4-NAF Double-and-add	159D + 32A	1528.8M	941M
Chaîne double base	94D + 40T+ 51A	1815.8M	-
Échelle de Montgomery	160D + 159A	1466.4M	971M

Tab. 2.2 – Nombres d'opérations des différents algorithmes génériques sur des corps de 2^{160} (corps premier) et 2^{163} (corps binaire) éléments

2.2 Spécificité des courbes définies sur $\mathbb{F}_{2^n}$

Ce paragraphe est consacrée à différentes techniques de multiplication par un scalaire tirant partie de propriétés particulières aux courbes définies sur $\mathbb{F}_{2^n}$. Ces techniques tirent à la fois partie de certaines spécificités de ces courbes (morphisme de Frobenius) mais également des particularités de l'arithmétique du corps de base (comme l'extraction aisée de racines carrées).

Division de point

Étant donnée une courbe elliptique définie sur un corps binaire, l'idée directrice de ce qui va suivre est de déterminer l'image réciproque du morphisme

$$\begin{array}{rcl}[2] \ : E(\mathbb{F}_{2^n}) & \longrightarrow & E(\mathbb{F}_{2^n}) \\ P & \longrightarrow & [2]P.\end{array}$$

Le but final est d'exprimer l'exponentiation an termes de ce morphisme réciproque. Cette approche originale a été d'abord proposée par Knudsen [Knu99], puis reprise et améliorée dans [ACS04, FHLM04, KR04], enfin adaptée sur les courbes hyperelliptiques [KKT05, Bir06]. Nous allons en traiter ici les mécanismes généraux sans parler des généralisations et autres améliorations contenues dans les papiers précédemment cités.

Commençons par quelques notions supplémentaires sur les courbes définies sur $\mathbb{F}_{2^n}$.

Proposition 6 Soit E une courbe elliptique définie sur $\mathbb{F}_{2^n}$. Notons $\overline{E}$ l'ensemble des solutions de E dans $\overline{\mathbb{F}}_{2^n} \times \overline{\mathbb{F}}_{2^n}$. Soient maintenant t un entier et $E[2^t]$ l'ensemble des points P de $\overline{E}$ tels que $[2^t]P = \mathcal{O}$ (appelés points de 2^t torsion). Alors

$$E[2^t] \simeq \mathbb{Z}/2^t\mathbb{Z}. \tag{2.2}$$

On sait de plus que toute courbe elliptique définie sur $\mathbb{F}_{2^n}$ possède un élément d'ordre 2 (dans $\mathbb{F}_{2^n} \times \mathbb{F}_{2^n}$). Notons r l'ordre d'une telle courbe, l'existence d'un élément d'ordre 2 implique donc que 2 divise r. Soit maintenant t le plus grand entier tel que 2^t divise r. On a donc $r = s \times 2^t$ avec s impair.

On peut en déduire qu'il existe un sous-groupe S de $E(\mathbb{F}_{2^n})$ d'ordre impair tel que :

$$E(\mathbb{F}_{2^n}) \simeq S \times E[2^t]. \tag{2.3}$$

Si $t = 1$ on dira que $E(\mathbb{F}_{2^n})$ est de 2-torsion minimale.

Étudions maintenant le morphisme $[2] : E(\mathbb{F}_{2^n}) \longrightarrow E(\mathbb{F}_{2^n})$. On sait qu'il existe un point $T_2 \neq \mathcal{O}$ de $E(\mathbb{F}_{2^n})$ tel que $E[2] = \{\mathcal{O}, T_2\}$, ce qui signifie en particulier que [2] n'est pas injectif. En effet soient P et Q tels que $Q = [2]P$, alors $Q = [2](P + T_2)$.

Par contre si l'on restreint [2] au sous groupe S défini plus haut, il devient alors un isomorphisme de groupe et l'on peut alors définir son morphisme réciproque :

$$\begin{array}{rcl}\left[\frac{1}{2}\right] \ : S & \longrightarrow & S \\ P & \longrightarrow & Q \text{ tel que } [2]Q = P.\end{array}$$

On définit de même le morphisme

$$\left[\frac{1}{2^i}\right] = \underbrace{\left[\frac{1}{2}\right] o \ldots o \left[\frac{1}{2}\right]}_{i \text{ fois}}.$$

Nous pouvons maintenant définir un algorithme Halving-and-add à partir de la proposition suivante :

Proposition 7 Soient k un entier de l bits et s un entier impair, alors il existe un rationnel de la forme

$$\sum_{i=0}^{l-1} \frac{c_i}{2^i}, \qquad c_i \in \{0, 1\}$$

tel que

$$k \equiv \sum_{i=0}^{l-1} \frac{c_i}{2^i} \mod s,$$

où $\frac{1}{2^i} \mod s$ représente l'inverse de 2^i modulo s.

Calculer le rationnel dont il est question dans la proposition précédente est très simple, il s'agit de calculer le reste de la division euclidienne de $2^{l-1}k$ par s, puis de diviser le résultat par 2^{l-1}.

Exemple 6 Prenons $k = 12 = 2^3 + 2^2$ et $s = 17$:

1. $2^3 k = 96 = 5 \times 17 + 11$,
2. $11 = 2^3 + 2^1 + 2^0$,
3. $11/2^3 = 1 + 1/2^2 + 1/2^3$,
4. et on a bien $12 = 1 + (2^2)^{-1} + (2^3)^{-1} \mod 17$.

Cet exemple montre donc que si P est un point d'un sous groupe de $E(\mathbb{F}_{2^n})$ d'ordre impair s, alors $[12]P = [1 + 1/2^2 + 1/2^3]P = P + [1/2^2]P + [1/2^3]P$. Plus généralement, multiplier un point P d'une courbe par un entier k peut alors s'effectuer grâce à l'algorithme suivant :

Algorithme 10 : Division et addition

Données : $P \in S$ sous groupe d'ordre impair de $E(\mathbb{F}_{2^n})$ et $k \geq 0$ sous la forme $\sum_{i=0}^{l-1} \frac{c_i}{2^i}$.
Résultat : $\sum_{i=0}^{l-1} \left[\frac{c_i}{2^i}\right] P = [k]P$.
début
  $Q \leftarrow P$
  pour $i = l-2$ à 0 faire
    $Q \leftarrow [\frac{1}{2}]Q$
    si $c_i = 1$ alors
      $Q \leftarrow Q + P$
    fin
  fin
fin
retourner Q

Il nous faut maintenant être capable de calculer $[\frac{1}{2}]P$.

Pour ce faire, l'idée est d'inverser les formules de doublement de point. Introduisons pour cela l'écriture suivante : un point (x, y) sera noté (x, λ_P) où $\lambda_P = x + \frac{y}{x}$.

Soient maintenant $P = (x, y)$ et $Q = (u, v)$ deux point tels que $[2]Q = P$. On a (après réécriture) les formules :

$$\lambda_Q = u + \frac{v}{u} \tag{2.4}$$
$$x = \lambda_Q^2 + \lambda_Q + a \tag{2.5}$$
$$y = (x+u)\lambda_Q + x + v. \tag{2.6}$$

Là aussi, quelques lignes de calculs permettent d'obtenir les formules suivantes :

$$\lambda_Q^2 + \lambda_Q = x + 0 \tag{2.7}$$
$$u^2 = x(\lambda_Q + 1) + y = x(\lambda_Q + \lambda_P + x + 1) \tag{2.8}$$
$$v = u(u + \lambda_Q). \tag{2.9}$$

La résolution de ce système donne alors deux points

$$[\frac{1}{2}]P \in S \quad \text{et} \quad [\frac{1}{2}]P + T_2 \in E(\mathbb{F}_{2^n}) \backslash S.$$

A priori, rien ne différencie ces deux points. On se reportera à l'article de Knudsen [Knu99] pour voir comment distinguer la solution qui nous intéresse, c'est-à-dire l'image de P par le morphisme $\left[\frac{1}{2}\right] : S \longrightarrow S$. Nous donnons simplement ici l'algorithme permettant à la fois de résoudre le système et de choisir le point solution.

Algorithme 11 : Division de point

Données : $P = (x, y) = (x, x(x + \lambda_P)) \in S$ donné sous la forme (x, y) ou $(x\lambda_P)$.
Résultat : $[\frac{1}{2}]P = (u, v) \in S$ sous la forme $(u, \lambda_{[\frac{1}{2}]P})$.
début
  Calculer une solution $\lambda_{[\frac{1}{2}]}$ de l'équation (2.7)
  Calculer u^2 défini par l'équation (2.8)
  Vérifier s'il existe un $\lambda \in \mathbb{F}_{2^n}$ tel que $\lambda^2 + \lambda = a^2 + u^2$
  Si un tel λ n'existe pas calculer : $u^2 = u^2 + x$ et $\lambda_{[\frac{1}{2}]} = \lambda_{[\frac{1}{2}]} + 1$
  Calculer $u = \sqrt{u^2}$
fin
retourner $(u, \lambda_{[\frac{1}{2}]})$

En admettant que a^2 est pré-calculé, effectuer une division de point requiert donc :

1. 1 résolution d'équation du second degré,
2. 1 multiplication,
3. 1 test d'existence,
4. 1 racine carrée.

Une multiplication supplémentaire est nécessaire si l'on a besoin de la quantité v, cette dernière s'obtenant grâce à l'équation 2.9. Cela signifie également qu'il est possible d'économiser cette multiplication au cours des calculs à partir du moment où les divisions se font en conservant l'écriture de type (x, λ).

Rappelons enfin qu'un doublement requiert une inversion, deux multiplications et un carré. Cela signifie que l'utilisation de la division de point devient rentable dès lors que la résolution d'une équation du second degré, le test d'existence d'une solution et le calcul d'une racine carrée sont moins coûteux qu'une inversion, une multiplication et un carré.

Pour que l'algorithme de multiplication soit complet, nous présentons maintenant des formules permettant l'addition de deux points d'une courbe en coordonnées affines $P = (x, y)$ et $Q = (u, u(u + \lambda_Q))$, où Q est donné sous la forme $Q = (u, \lambda_Q)$.

Addition

Soient $P = (x, y)$ et $Q = (u, u(u + \lambda_Q))$ donné sous la forme $Q = (u, \lambda_Q)$ deux points donnés en coordonnées affines. Alors en posant :

$$\lambda = \frac{y + u(u + \lambda_Q)}{x + u},$$

on a

$$P + Q = (\lambda^2 + \lambda + a + x + u, (s + x)\lambda + s + y).$$

Le résultat $P + Q$ est en coordonnées affines et le coût de l'opération est I+3M+C.

Terminons par une étude du coût de calcul que représente cette méthode. Pour cela, nous reprenons directement les analyses faites dans le paragraphe 5 ainsi que dans l'appendice C du papier de Knudsen [Knu99]. En ce qui concerne la représentation elle-même, si le scalaire k est un entier de l bits, alors le rationnel obtenu à partir de la proposition 7 pourra être représenté lui aussi par l bits et, tout comme avec les entiers, il est possible d'obtenir une densité de bits non nuls de 1/3 à l'aide d'une représentation signée (type NAF). Si l'on utilise des points en coordonnées affines, cela donne ainsi la complexité suivante :

- $l/3$ inversions ($l/3$ additions),
- $2l$ multiplications (l divisions + $3 \times l/3$ additions),
- $l/3$ carrés ($l/3$ additions),
- l résolutions d'équation du second degré,
- l tests d'existence,
- l calculs d'une racine carré.

Afin de donner une estimation en termes de multiplications sur le corps de bases de la complexité de cette méthode de multiplication de point, nous reprenons l'étude faite dans [HLHM00], [FLM03] et [Knu99]. On a, dans ce cadre, les ordres de grandeurs suivants :

- le carré est négligeable devant la multiplication,
- 1 inversion $\sim$ 8 multiplications,
- résolution de $\lambda_Q^2 + \lambda_Q = x + a$ + test d'existence + calcul d'une racine carré $\sim$ 1 multiplication + 1 carré.

Ainsi, si le scalaire k est un entier de 163 bits, le coût de calcul de l'algorithme de multiplication par un scalaire est de 924M. Cette complexité est à comparer avec le meilleur algorithme sans précalculs (utilisant la représentation NAF) dont le coût est de 1137M ; ce qui représente une amélioration d'environ 19%.

Comme toujours, il est possible d'améliorer cette approche, d'une part, par l'utilisation de coordonnées projectives, et d'autre part, par l'emploi de méthodes à fenêtres. La combinaison des deux permet d'obtenir un algorithme très efficace dont le coût de calculs est d'environ 680M [FLM03].

Courbes de Koblitz

Une courbe de Koblitz (car introduite pour la première fois en cryptographie par Neal Koblitz [Kob92]) est une courbe elliptique définie sur $\mathbb{F}_{2^n}$ et dont les coefficients sont à valeurs dans $\mathbb{F}_2$. Plus précisément, il en existe exactement deux : $E_0 : y^2 + xy = x^3 + 1$ et $E_1 : y^2 + xy = x^3 + x^2 + 1$. L'intérêt de ces courbes est qu'il va être possible d'effectuer la multiplication de points en remplaçant les doublements par un opérateur beaucoup plus efficace : le morphisme de Frobenius.

Pour ce faire, nous allons tout d'abord donner la définition générale du morphisme de Frobenius, puis expliquer en quoi celui-ci devient très intéressant sur les courbes de Koblitz.

Définition 9 Soit $E(\mathbb{F}_{q^n})$ une courbe elliptique le morphisme de Frobenius est l'application suivante :

$$\begin{array}{rrl} \tau: & E(\mathbb{F}_{q^n}) & \to E(\mathbb{F}_{q^n}) \\ & (x,y) & \to (x^q, y^q) \end{array}$$

Proposition 8 Soient $a \in \mathbb{F}_2$ et $E_a : y^2 + xy = x^3 + ax^2 + 1$ une courbe de Koblitz. Le morphisme de Frobenius vérifie alors :

$$\forall P \in E_a(\mathbb{F}_{2^n}), \quad \tau^2(P) - [\mu]\tau(P) + [2]P = \mathcal{O}, \tag{2.10}$$

où $\mu = (-1)^{1-a}$

Autrement dit, le polynôme $X^2 - \mu X + 2$ est un polynôme annulateur de τ. De cette manière, τ peut être vu comme un nombre complexe, c'est-à-dire comme l'une des deux racines, dans $\mathbb{C}$, du polynôme $X^2 - \mu X + 2 \in \mathbb{Z}[X]$. A partir de là, on peut alors considérer l'anneau $\mathbb{Z}[\tau]$ dont les éléments sont de la forme $z = \sum_{i=0}^{l-1} u_i \tau^i$, $u_i \in \mathbb{Z}$. Il est alors possible d'étendre le morphisme multiplication par un scalaire à $\mathbb{Z}[\tau]$ en définissant

$$\forall z \in \mathbb{Z}[\tau], [z]P = \sum_{i=0}^{l-1} [u_i]\tau^i(P). \tag{2.11}$$

Si maintenant on prend en compte la question de l'efficacité de cette multiplication par un scalaire, l'idée est de trouver une bonne représentation du scalaire k dans $\mathbb{Z}[\tau]$, c'est-à-dire une représentation courte et avec de petits coefficients dont la plupart sont nuls.

On trouve donc des algorithmes permettant de calculer les équivalents respectifs de la représentation binaire et de la forme non-adjacente avec et sans fenêtres [HLHM00].

A partir d'une de ces formes il est alors très simple d'écrire un algorithme de multiplication par un scalaire en remplaçant simplement les doublements par des appels au morphisme de Frobenius (c'est-à-dire juste deux carrés modulaires).

Afin d'illustrer ce qui vient d'être présenté, nous allons voir maintenant comment calculer une représentation τ-adique d'un entier k. Celle-ci est en quelque sorte l'équivalent dans $\mathbb{Z}[\tau]$ de la représentation binaire classique. D'ailleurs, l'algorithme utilisé ici est largement inspiré de celui permettant l'écriture en base 2 d'un entier.

L'idée est la suivante : considérons un élément z de $\mathbb{Z}[\tau]$; celui-ci peut toujours s'écrire sous la forme $z_0 + \tau z_1$ où z_0 et z_1 sont des entiers (car $\tau^2 = \mu\tau - 2$). Considérons maintenant la division euclidienne de z_0 par 2, c'est-à-dire $z_0 = u_0 + 2 \times u'_0$. On a alors

$$\begin{aligned} z &= z_0 + \tau z_1 \\ &= u_0 + 2 \times u'_0 + \tau z_1 \\ &= u_0 + u'_0 \mu \tau - u'_0 \tau^2 + \tau z_1 \\ &= u_0 + \tau(u'_0 \mu + z_1 - u'_0 \tau) \\ &= u_0 + \tau(z_2 + \tau z_3) \end{aligned}$$

On peut donc répéter l'opération avec $z_2 + \tau z_3$, et ainsi de suite, construire une représentation τ-adique de z. Reste encore à justifier qu'un tel processus prend toujours fin. Pour cela, il faut considérer l'application

$$\begin{array}{rccc} N: & \mathbb{Z}[\tau] & \longrightarrow & \mathbb{N} \\ & z_0 + \tau z_1 & \longrightarrow & z_0^2 + \mu z_0 z_1 + 2 z_1^2. \end{array}$$

C'est en fait une norme sur $\mathbb{Z}[\tau]$, le munissant au passage d'une structure d'anneau euclidien, et l'on peut vérifier que $N(z_0 + \tau z_1) > N(z_2 + \tau z_3)$. On obtient ainsi une suite d'entiers positifs strictement décroissante, ce qui assure bien que l'algorithme précédemment suggéré prend toujours fin.

Exemple 7 Considérons la courbe de Koblitz $E_1 : y^2 + xy = x^3 + x^2 + 1$. Le morphisme de Frobenius vérifie donc $\tau^2 - \tau + 2 = 0$. On peut obtenir la représentation τ-adique de l'entier 9 de la manière suivante :

$$\begin{aligned} 9 &= 1 + 2 \times 4 \\ &= 1 + \tau(4 - 4\tau) \\ &= 1 + \tau(2 \times 2 - 4\tau) \\ &= 1 + \tau^2(-2 - 2\tau) \\ &= 1 + \tau^3(-3 + \tau) \\ &= 1 + \tau^3(1 - 2 \times 2 + \tau) \\ &= 1 + \tau^3 + \tau^4(-1 + 2\tau) \\ &= 1 + \tau^3 + \tau^4(1 - 2 + 2\tau) \\ &= 1 + \tau^3 + \tau^4 + \tau^5(1 + \tau) \\ &= 1 + \tau^2 + \tau^4 + \tau^5 + \tau^6 \end{aligned}$$

A partir de là on peut donc calculer

$$[9]P = P + \tau^2(P) + \tau^4(P) + \tau^5(P) + \tau^6(P). \tag{2.12}$$

De nombreuses améliorations ont été apportées à cette approche. Une fois de plus, les principales consistent à utiliser des équivalents des méthodes NAF et w-NAF (que l'on note classiquement TNAF et w-TNAF) dont les densités de bits non nuls sont les mêmes que dans le cas binaire. Au final, l'avantage de toute ces méthodes est que les doublements sont remplacés par le Frobenius, donc par une opération quasiment gratuite sur un corps binaire et seules restent les additions de points. Ainsi pour un scalaire k, codé sur de l bits, l'utilisation de la méthode TNAF requiert seulement $l/3$ additions sur la courbe. Ce nombre descendant à $l/(w+1)$ avec le w-TNAF.

Finissons cette partie avec le tableau 2.3 qui résume le coût de calcul des algorithmes spécifiques de multiplication de points sur des courbes définies sur des corps binaires. Les résultats sont donnés pour un scalaire de l bits et dans le cas particulier $l = 163$, la lettre H symbolise les divisions par 2 (Halving). Enfin, comme dans les tableaux précédents, le coût des précalculs n'est pas pris en compte dans la complexité des méthodes à fenêtres.

Algorithme	coord.	nbr. dbl/div/add	nbr. op. sur $\mathbb{F}_{2^n}$	$l = 163$
NAF, division	affine	lH + $\frac{l}{3}$A	$\frac{17l}{3}$	924M
4-NAF, division	affine	lH + $\frac{l}{5}$A	$\frac{21l}{5}$	685M
TNAF	affine	$\frac{l}{3}$A	$\frac{10l}{3}$	543M
TNAF	mixte	$\frac{l}{3}$A	$3l$	489M
4-TNAF	affine	$\frac{l}{5}$A	$2l$	326M
4-TNAF	mixte	$\frac{l}{5}$A	$\frac{9l}{5}$	294M
NAF Double-and-add	mixte	$(l-1)$D + $\frac{l}{3}$A	$(7l-4)$M	1137M
w-NAF Double-and-add	mixte	$(l-1)$D + $\frac{l}{w+1}$A	$(\frac{4w+13}{w+1}l-4)$M	941M

Tab. 2.3 – Complexité moyenne des différents algorithmes sur $\mathbb{F}_{2^n}$

Deuxième partie

Nouvelles Formules d'Addition et Conséquences

Chapitre 3

Nouvelles formules d'addition de points et représentation de Zeckendorf

Nous avons vu dans les chapitres précédents une description des principales techniques de multiplication par un scalaire. Elles reposent quasiment toutes sur un schéma de Horner (seule l'échelle de Montgomery fait exception), c'est-à-dire que chaque algorithme suit un schéma de β-and-add, où β est la base utilisée pour représenter le scalaire (2, 1/2, τ). L'idée fondamentale derrière chaque nouvel algorithme proposé est de trouver une nouvelle opération plus efficace que le doublement, et d'en déduire une nouvelle représentation pour le scalaire.

Dans le chapitre qui suit, nous allons suivre cette démarche en proposant de nouvelles formules d'addition de points pour les courbes définies sur un corps de caractéristique supérieure ou égale à cinq, qui dans un cadre précis, se révèlent plus efficaces que les formules de doublement. Nous verrons qu'en effet, lorsque la somme P_3 de deux points P_1 et P_2 vient d'être calculée, il est alors possible de calculer P_3+P_1 (ou de manière symétrique P_3+P_2) plus rapidement qu'un doublement. A partir de là deux types d'approche sont alors possibles. Une première approche facile à mettre en oeuvre est basée sur les nombres de Fibonacci. Elle consiste principalement à remplacer la représentation binaire du scalaire (somme de puissance de 2) par sa représentation de Zeckendorf (somme de nombres de Fibonacci), et à adapter les algorithmes de multiplication en conséquence. La seconde, beaucoup plus prometteuse en termes d'efficacité, consiste à utiliser exclusivement les nouvelles formules d'additions. La contre-partie est qu'il est difficile de trouver des chaînes d'additions pour le calcul de la multiplication par un scalaire, basés sur ces nouvelles formules, suffisamment courtes pour rester efficaces.

Notons, pour finir, que le même type de démarche peut être effectué sur des courbes définies sur des corps de caractéristique 2 ou 3. Le même genre de formules d'addition de points peut être obtenu. Cependant, en aucun cas ces fomrules n'ont montrées un quelconque intérêt en termes d'efficacité. C'est pourquoi le présent chapître ne traite que des courbes définies sur des corps de caractéristique supérieure ou égale à 5.

3.1 Somme de points avec la même coordonnée z

Soient $\mathbb{K}$ un corps de caractéristique supérieure ou égale à cinq, E une courbe elliptique définie sur $\mathbb{K}$, $P_1 = (X_1 : Y_1 : Z_1)$ et $P_2 = (X_2 : Y_2 : Z_2)$ deux points de E donnés en coordonnées jacobiennes ayant la même coordonnée z. Posons alors $Z = Z_1 = Z_2$ et notons maintenant $P_1 + P_2 = P_3 = (X_3 : Y_3 : Z_3)$. En utilisant les formules d'addition en coordonnées jacobiennes on a :

$$\begin{aligned} X_3 &= (Y_2Z^3 - Y_1Z^3)^2 - (X_2Z^2 - X_1Z^2)^3 - 2X_1Z^2(X_2Z^2 - X_1Z^2)^2 \\ &= ((Y_2 - Y_1)^2 - (X_2 - X_1)^3 - 2X_1(X_2 - X_1)^2)Z^6 \\ &= ((Y_2 - Y_1)^2 - (X_1 + X_2)(X_2 - X_1)^2)Z^6 \\ &= X_3'Z^6 \end{aligned}$$

$$\begin{aligned} Y_3 &= -Y_1Z^3(X_2Z^2 - X_1Z^2)^3 + (Y_2Z^3 - Y_1Z^3)(X_1Z^2(X_2Z^2 - X_1Z^2)^2 - X_3) \\ &= (-Y_1(X_2 - X_1)^3 + (Y_2 - Y_1)(X_1(X_2 - X_1)^2 - X_3'))Z^9 \\ &= Y_3'Z^9 \end{aligned}$$

$$\begin{aligned} Z_3 &= Z^2(X_2Z^2 - X_1Z^2) \\ &= Z(X_2 - X_1)Z^3 \\ &= Z_3'Z^3 \end{aligned}$$

Ainsi $(X_3 : Y_3 : Z_3) = (X_3'Z^6 : Y_3'Z^9 : Z_3'Z^3)$. Or en coordonnées jacobiennes, les points $(X : Y : Z)$ et $(X\lambda^2 : Y\lambda^3 : Z\lambda)$ sont équivalent pour tout $\lambda \in \mathbb{K}^*$. Nous avons donc

$$(X_3 : Y_3 : Z_3) \sim (X_3' : Y_3' : Z_3'). \tag{3.1}$$

En résumé, si P_1 et P_2 ont la même coordonnée z, P_1+P_2 peut être calculé en utilisant les formules suivantes :

Addition :
$P_1 = (X_1, Y_1, Z), P_2 = (X_2, Y_2, Z)$ et $P_1 + P_2 = (X_3', Y_3', Z_3')$

$$A = (X_2 - X_1)^2, \quad B = X_1A, \quad C = X_2A, \quad D = (Y_2 - Y_1)^2$$

et

$$X_3' = D - B - C, \quad Y_3' = (Y_2 - Y_1)(B - X_3) - Y_1(C - B), \quad Z_3' = Z(X_2 - X_1). \tag{3.2}$$

Le coût de ces formules est de 5M+2C.

Il peut sembler assez improbable que deux points P_1 et P_2 pris au hasard se trouvent avoir la même coordonnée z. Cependant, si l'on prête attention aux quantités $X_1A = X_1(X_2 - X_1)^2$ et $Y_1(C - B) = Y_1(X_2 - X_1)^3$ apparaissant lors des calculs, on peut les interpréter comme les coordonnées x et y du point $(X_1(X_2 - X_1)^2, Y_1(X_2 - X_1)^3, Z(X_2 - X_1)) \sim (X_1, Y_1, Z)$. On se retrouve ainsi avec des points P_1 et P_3 qui partagent la même coordonnée z. Il est alors possible de calculer la somme P_3+P_1 avec nos nouvelles formules.

Remarque 2 Il est possible de faire la même remarque concernant les formules de doublement en coordonnées jacobiennes. En effet les quantités $A = X_1(2Y_1)^2$ et $8Y_1^4 = Y_1(2Y_1)^3$ peuvent être interprétées comme les coordonnées en x et en y du point $(X_1(2Y_1)^2, Y_1(2Y_1)^3, 2Y_1Z_1) \sim (X_1, Y_1, Z_1)$ permettant alors de calculer $P + [2]P$ avec les formules précédentes.

Cette remarque est important dans la mesure où, comme nous le verrons par la suite, les schémas de multiplications par un scalaire auxquels nous allons nous intéresser débutent toujours par les opérations $P + P \to [2]P; [2]P + P \to [3]P$.

Comparons ces formules avec celles déjà existantes. Premièrement, il apparaît que ces formules sont les formules d'addition les plus efficaces. Cela n'a rien d'étonnant dans la mesure où elles tirent partie d'une contrainte très forte sur les points : avoir la même coordonnée z. Ce qui peut paraître plus surprenant est le fait que ces formules sont plus efficaces que toutes les formules de doublement connues. En effet, le tableau 1.1 (page 10) montre que le doublement le plus rapide, obtenu avec les coordonnées jacobiennes modifiées, requiert 4M+4C. Mais là encore, en y regardant de plus près, nous pouvons nous rendre compte que les formules de composition de points en coordonnées affines possèdent déjà cette particularité. En effet une addition de point requiert, dans ce cas, une multiplication de moins que le doublement. Ces nouvelles formules sont donc, en quelque sorte, un juste retour des choses.

La comparaison avec les formules de Montgomery est beaucoup plus pertinente. A priori les deux types de formules sont quasiment identiques. Avec les formules de Montgomery, pour faire la somme de deux points P_1 et P_2 il faut connaître la différence P_1-P_2. Les formules que nous venons d'introduire permettent, quand à elles, de calculer efficacement la somme $P_1 + (P_1 + P_2)$ à partir du calcul de $P_1 + P_2$. La différence vient de deux éléments. D'une part, les formules de Montgomery ne sont valables que sur les courbes de Montgomery, alors que les formules 3.2 le sont sur toutes courbes. D'autre part les formules de Montgomery ne permettent que le calcul des coordonnées x et z, alors que les formules que nous proposons calculent les trois coordonnées, ce qui leur confère, a priori, une plus grande flexibilité d'utilisation. Cet avantage est par contre contrebalancé par le fait que les formules de Montgomery sont plus efficaces (4M+2C). Il est, de plus possible de récupérer la coordonnée y en fin de calcul, mais uniquement dans le cas d'une multiplication par un scalaire utilisant l'échelle de Montgomery. Nous verrons de plus qu'avec nos nouvelles formules, il est possible de ne calculer que la coordonnée x dans le cas d'une multiplication de point utilisant les chaînes d'additions euclidiennes. Ceci permet d'avoir une addition équivalente à celle utilisant les formules de Montgomery (4M+2C) tout en conservant le fait que ces nouvelles formules sont valables sur n'importe quelle courbe.

La comparaison précédente reste valable concernant les formules de Brier et Joye [BJ02], qui généralisent l'approche de Montgomery à n'importe quelle courbe elliptique. La seule différence est que les formules d'additions sont alors bien moins compétitive (9M+2C).

3.2 Représentation de Zeckendorf

Résumons ce qui précède. Nous disposons désormais d'un algorithme NewADD fonctionnant de la façon suivante : soient P_1 et P_2 deux points ayant la même coordonnée z, alors NewADD$(P_1, P_2) = (P_1+P_2, P_1)$, où P_1+P_2 et P_1 partagent la même coordonnée z. Le coût de l'opération est de 5M+2C. Cet algorithme est plus efficace qu'un doublement. Nous allons maintenant chercher des méthodes de multiplication de points l'utilisant au mieux.

Exemple 8 On peut calculer le point $[25]P$ en utilisant exclusivement l'algorithme NewADD de la façon suivante :
- NewADD(P, P)=$([2]P, P)$
- NewADD$([2]P, P)$=$([3]P, [2]P)$
- NewADD$([2]P, [3]P)$=$([5]P, [2]P)$
- NewADD$([2]P, [5]P)$=$([7]P, [2]P)$
- NewADD$([7]P, [2]P)$=$([9]P, [7]P)$
- NewADD$([9]P, [7]P)$=$([16]P, [9]P)$
- NewADD$([16]P, [9]P)$=$([25]P, [16]P)$

L'exemple précédent montre qu'il est possible de calculer $[25]P$ en utilisant uniquement l'algorithme NewADD. La chaîne de calcul est la suivante $P \to [2]P \to [3]P \to [5]P \to [7]P \to [9]P \to [16]P \to [25]P$. Si l'on ne s'intéresse qu'à la chaîne des scalaires, on obtient alors ce qu'on appelle une chaîne d'additions calculant l'entier 25. Dans notre cas précis, il s'agit d'une chaîne d'additions euclidienne. Nous étudierons précisément ce type de chaîne dans le paragraphe 4.1, tout ce que nous admettrons pour le moment est qu'il est toujours possible d'effectuer une multiplication de points en utilisant exclusivement l'algorithme NewADD, à condition de connaître une chaîne d'addition euclidienne calculant le scalaire concerné. Nous verrons aussi qu'il est très difficile, en pratique, de trouver de telles chaînes suffisamment courtes pour rendre la multiplication de point efficace.

Malgré tout, il existe un cas bien connu pour lequel trouver une chaîne d'additions euclidienne optimale est aisé, conduisant ainsi à une multiplication de points très efficace : c'est le cas des nombres de Fibonacci.

Définition 10 On définit la suite de Fibonacci $(F_n)_{n\geq 0}$ de la façon suivante :

$$F_n = \begin{cases} 0 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ F_{n-1} + F_{n-2}, \text{ si } n \geq 2 \end{cases}$$

La suite de Fibonacci est très certainement l'une des suites les plus étudiées. On trouve ainsi des livres entiers dédiés aux centaines de propriétés qu'elle possède. On pourra ainsi se reporter à [Knu73, Vor02] pour une liste (non exhaustive) de ces propriétés. Parmi les plus connues, rappelons la formule de Binet :

Théorème 5

$$\forall n \in \mathbb{N},\ F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \tag{3.3}$$

où $\phi = \frac{1+\sqrt{5}}{2}$ est la racine positive du polynôme à coefficients réels $X^2 - X - 1$.

A partir de cette formule, il est alors aisé de déduire les propriétés suivantes :

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \phi,$$

et

$$F_n = \left\lceil \frac{\phi^n}{\sqrt{5}} \right\rfloor,$$

où $\lceil x \rfloor$ représente l'entier le plus proche de x.

Comme dit précédemment, les nombres de Fibonacci forment une classe d'entiers particulièrement intéressante dans le cadre d'une multiplication de points utilisant l'algorithme NewADD. En effet, La suite de Fibonacci est une chaîne d'addition euclidienne permettant de calculer n'importe quel nombre de Fibonacci. Il est, de plus, facile de montrer que cette chaîne est toujours optimale (c'est-à-dire la plus courte possible).

Exemple 9 On a $F_9 = 34$. On peut donc calculer le point $[34]P$ en utilisant exclusivement l'algorithme NewADD de la façon suivante :

- NewADD(P, P)=$([2]P, P)$
- NewADD$([2]P, P)$=$([3]P, [2]P)$
- NewADD$([3]P, [2]P)$=$([5]P, [2]P)$
- NewADD$([5]P, [3]P)$=$([8]P, [5]P)$
- NewADD$([8]P, [5]P)$=$([13]P, [8]P)$
- NewADD$([13]P, [8]P)$=$([21]P, [13]P)$
- NewADD$([21]P, [13]P)$=$([34]P, [21]P)$

La chaîne d'addition correspondante est $(1 \to 2 \to 3 \to 5 \to 8 \to 13 \to 21 \to 34)$, qui n'est autre que $(F_2 \to F_3 \to F_4 \to F_5 \to F_6 \to F_7 \to F_8 \to F_9)$.

On obtient un algorithme très simple permettant de calculer $[k]P$, où k est un nombre de Fibonacci :

Algorithme 12 : Multiplication par un nombre de Fibonacci

Données : $P \in E$ et F_n le n-ième nombre de Fibonacci
Résultat : $[F_n]P \in E$
début
  $(U_1, U_2) \leftarrow (P, P)$
  pour $i = 1 \ldots n-2$ faire
    $(U_1, U_2) \leftarrow \text{NewADD}(U_1, U_2)$
  fin
  retourner U_1
fin

Nous venons de voir qu'il est facile d'effectuer une multiplication de point par un nombre de Fibonacci. Nous allons donc chercher maintenant une représentation du scalaire permettant de tirer partie de cette propriété. Pour cela, nous allons utiliser le théorème suivant [Zec72] :

Théorème 6 Soient k un entier et $(F_i)_{i\geq 0}$ la suite de Fibonacci. Alors k peut s'écrire de manière unique sous la forme :

$$k = \sum_{i=2}^{l} d_i F_i, \tag{3.4}$$

avec $d_i \in \{0, 1\}$ et $d_i d_{i+1} = 0$

Zeckendorf a été le premier, en 1972, à proposer d'écrire les entiers sous cette forme ; c'est pourquoi l'on parle de représentation de Zeckendorf d'un entier. Dans la suite, lorsque un entier k sera donné sous cette forme, nous adopterons la notation $k = (d_{l-1}, \ldots, d_2)_{\mathcal{Z}}$. Cette représentation s'obtient simplement, à l'aide d'un algorithme glouton : on cherche le nombre de Fibonacci le plus proche par valeur inférieure de l'entier concerné, on soustrait le premier au second et on réitère l'opération.

La représentation de Zeckendorf requiert 44% de chiffres de plus que la représentation binaire. Pour s'en convaincre commençons par une propriété classique des nombres de Fibonacci.

Proposition 9

$$\forall l \in \mathbb{N}, \sum_{i=1}^{l} F_i = F_{l+2} - 1$$

Cette propriété se démontre facilement par récurrence sur l (rappelons juste que $F_1 = F_2 = 1$ et $F_3 = 2$). Cela signifie qu'en la représentation de Zeckendorf, avec l chiffres, il est impossible de représenter un entier supérieur à $F_{l+2} - 1$.

Soit F_{l+2} le plus grand nombre de Fibonacci inférieur à 2^{n-1}. Nous pouvons déduire de la propriété précédente que la représentation de Zeckendorf d'un entier de n bits requiert, au moins, l chiffres.

De plus, nous savons que $F_{l+2} \simeq \frac{\phi^l}{\sqrt{5}}$, donc,

$$(l+2)\log_2(\phi) - \log_2(\sqrt{5}) \leq n-1 \leq (l+3)\log_2(\phi) - \log_2(\sqrt{5}), \tag{3.5}$$

$$l+2 \leq \frac{n-1+\log_2(\sqrt{5})}{\log_2(\phi)} \leq l+3. \tag{3.6}$$

Nous pouvons approcher le résultat précédent par

$$l+2 \leq 1.44n + 0.23 \leq l+3. \tag{3.7}$$

A partir de là, nous pouvons conclure que l est, au moins asymptotiquement, de l'ordre de $1.44n$.

Par exemple, le plus grand nombre de Fibonacci inférieur à 2^{160} est F_{232}. Cela veut dire que la représentation de Zeckendorf d'un entier de 160 bits nécessite au moins 230 chiffres, soit environ 160×1.44.

La contrepartie de ce surplus de chiffres est la plus faible proportion de chiffres valant 1. On sait, en effet, que la densité moyenne de 1 dans la représentation de Zeckendorf est environ de 0.2764. Concrètement, cela signifie, qu'en moyenne, un entier de 160 bits est la somme de 80 puissances de 2, mais de seulement 64 nombres de Fibonacci ($\simeq 230 \times 0.2764$). On se reportera à [Cap88] pour les preuves des résultats précédents.

3.3 L'algorithme Fibonacci et addition

Dans le paragraphe précédent nous avons vu qu'il était possible de représenter un entier comme une somme de nombres de Fibonacci. L'approche développée dans cette section est maintenant de proposer un algorithme de multiplication par un scalaire exploitant cette nouvelle représentation, puis de prendre en compte le cas particulier des courbes elliptiques, en essayant d'optimiser l'utilisation de l'algorithme NewADD.

Tout d'abord, nous allons voir un algorithme générique de multiplication de point par un scalaire donné en représentation de Zeckendorf. Cet algorithme est une adaptation directe de l'algorithme de doublement et addition à cette représentation.

Algorithme 13 : Fibonacci et addition

Données : $P \in G$ où G est un groupe quelconque, $k = (d_n, \ldots, d_2)_{\mathcal{Z}}$
Résultat : $[k]P \in G$
début
  $(U, V) \leftarrow (P, P)$
  pour $i = n - 1 \ldots 2$ faire
    si $d_i = 1$ alors
      $U \leftarrow U + P$ (addition)
    fin
    $(U, V) \leftarrow (U + V, U)$ (Fibonacci)
  fin
fin
retourner U

Exemple 10 Prenons $k = 25 = 21 + 3 + 1 = (1000101)_{\mathcal{Z}}$, l'algorithme précédent conduit à la suite d'opérations suivantes :

- initialisation : $(U, V) \leftarrow (P, P)$
- $d_7 = 0 : (U, V) \leftarrow ([2]P, P)$
- $d_6 = 0 : (U, V) \leftarrow ([3]P, [2]P)$
- $d_5 = 0 : (U, V) \leftarrow ([5]P, [3]P)$
- $d_4 = 1 : U \leftarrow [6]P$ puis $(U, V) \leftarrow ([9]P, [6]P)$
- $d_3 = 0 : (U, V) \leftarrow ([15]P, [9]P)$
- $d_2 = 1 : U \leftarrow [16]P$ puis $(U, V) \leftarrow ([25]P, [16]P)$
- retour : $U = [25]P$

En terme de complexité, supposons que k soit un entier de l bits ; dans le paragraphe précédent nous savons que l'algorithme 13 requiert en moyenne $1.83 \times l$ opérations ($1.44 \times l$ « Fibonacci » et $(1.44 \times l) \times 0.2764 = 0.398 \times l$ additions), alors que l'algorithme 2 requiert dans le même temps $1.5 \times l$ opérations (l doublements et $\frac{l}{2}$ additions). Autrement dit, l'algorithme 13 demande naturellement 23% d'opérations supplémentaires. Nous devons donc d'essayer d'optimiser le plus possible les appels à l'opérateur NewADD afin de réduire au maximum cet écart.

Tout comme l'algorithme de doublement peut être vu comme une série de doublements entrecoupée d'additions, l'algorithme 13 peut être vu comme une série d'étape de « Fibonacci » entrecoupée d'additions.

Commençons par traiter le cas des étapes de « Fibonacci ». Plaçons-nous à la i-ème étape de la boucle « pour » en considérant que $d_i = 0$. Nous avons en entrée deux éléments U et V, et nous devons calculer les quantités $U + V$ et U. Supposons que U et V aient la même coordonnée z, alors NewADD(U, V) renvoie bien $U + V$ et U avec la même coordonnée z. Ce qui signifie que l'on peut enchaîner de nouveau sur une étape de « Fibonacci ». En conclusion, dès lors que $Z_U = Z_V$, une série de t 0, dans la représentation de Zeckendorf du scalaire, revient à effectuer t fois l'opération $(U, V) \leftarrow$NewADD(U, V).

Il nous faut maintenant être capable de calculer $U + P$ sans trop de pertes. Plus exactement on veut être capable de prendre en entrée deux points U et V et de renvoyer en sortie les points $U + P$ et V avec la même coordonnée en z, afin de pouvoir enchaîner sur l'étape suivante de "Fibonacci". Pour ce faire, nous proposons le schéma de calcul suivant :

- les données sont $U = (X_U, Y_U, Z)$, $V = (X_V, Y_V, Z)$ et $P = (x, y, 1)$,
- calculer le point$P' = (xZ^2, yZ^3, Z) \sim P$ (3M+C),
- calculer ensuite NewADD(U, P) (5M+2S),
- on obtient ainsi le point $U + P$ dont la coordonnée z est $Z \times (X_U - xZ^2)$,
- les quantités $(X_U - xZ^2)^2$ et $(X_U - xZ^2)^3$ ont été calculées pendant le calcul de $U + P$,
- on peut donc calculer

$$(X_V(X_U - xZ^2)^2, Y_V(X_U - xZ^2)^3, Z(X_U - xZ^2)) \sim V \text{ (2M)}.$$

Le coût de l'opération complète est finalement de 10M+3C.

Nous en déduisons l'algorithme de multiplication par un scalaire suivant :

Algorithme 14 : Fibonacci et addition

```
Données : P ∈ E, k = (d_n, ..., d_2)_Z ;
Résultat : [k]P ∈ E ;
début
    (U, V) ← (P, P)
    pour i = n − 1 ... 2 faire
        si d_i = 1 alors
            P' ← (xZ^2, yZ^3, Z)
            (U, .) ← NewADD(U, P')
            V ← (X_V(X_U − xZ^2)^2, Y_V(X_U − xZ^2)^3, Z(X_U − xZ^2))
        fin
        (U, V) ←NewADD (U, V)
    fin
fin
retourner U
```

Nous avons dit, qu'en moyenne, cet algorithme effectue à peu près $1.44 \times l$ « Fibonacci » et $0.398 \times l$ additions (où l est la taille binaire de k). Ainsi le nombre d'opérations moyen de cet algorithme est de $(11.18 \times l)$M + $(4.07 \times l)$C, soit $(14.4 \times l)$M si l'on estime que C = 0.8M. Si l'on compare ceci au coût d'une multiplication de points utilisant le schéma classique doublement et addition qui était de l'ordre de $(13.2 \times l)$M, on se rend compte que le surplus de calculs a été significativement réduit (de 23% à un peu plus de 9%).

3.4 Raffinements

Tout comme avec la représentation binaire, il est possible de creuser la représentation de Zeckendorf (c'est-à-dire d'en diminuer le nombre de chiffres non nuls) en élargissant l'ensemble de valeurs que peuvent prendre les chiffres.

Le premier raffinement consiste, comme dans le cas binaire, à tirer avantage du fait que le calcul de l'opposé d'un point est très peu coûteux en autorisant l'utilisation du chiffre -1. On obtient un équivalent de la représentation signée en binaire pour la représentation de Zeckendorf. De façon toujours similaire il existe de nombreuses représentations signées pour un même entier, et ce sont bien sûr celles de poids minimal qui vont nous intéresser, c'est-à-dire les représentations comportant le moins de chiffres non nuls.

Avant toute chose, commençons par remarquer qu'il n'existe pas une unique représentation de poids minimal pour un entier. Pour s'en convaincre, il suffit de considérer l'exemple suivant :

$$7 = F_6 - F_2 = F_5 + F_3. \tag{3.8}$$

7 n'est pas un nombre de Fibonacci $F_6 - F_2$ et $F_5 + F_3$ sont bien deux représentations minimales du même entier. Cela n'est pas gênant en soit mais il est plus pratique, en général, de se restreindre à une classe particulière de représentation. Cela facilite, d'une part, le calcul de la dite représentation et, d'autre part, la preuve de minimalité.

Ainsi, nous allons voir que l'on peut définir une représentation à la fois facile à calculer, et minimale parmi les représentations utilisant les chiffres $\{-1, 0, 1\}$.

Définition 11 Soit $k = \sum_{i=1}^{n} d_i F_i$ $d_i \in \{0, \pm 1\}$. On dira que la représentation $(d_n \dots d_1)_{\mathcal{Z}}$ est admissible pour k si

1. $d_1 = 0$
2. les séquences suivantes (ou leurs opposés) n'apparaissent pas :
   - $(1\ \overline{1})_{\mathcal{Z}}$,
   - $(1\ 1)_{\mathcal{Z}}$,
   - $(\overline{1}\ 0\ 1)_{\mathcal{Z}}$,
   - $(1\ 0\ 1)_{\mathcal{Z}}$,
   - $(1\ 0\ 0\ 1)_{\mathcal{Z}}$.

Une telle représentation est bien unique et peut être très facilement calculée grâce à l'algorithme 3.4. On se reportera à [Heu04] pour les preuves des affirmations précédentes.

Algorithme 15 : calcul de la représentation admissible d'un entier

Données : k un entier positif
Résultat : $(d_n \dots d_1)_{\mathcal{Z}}$ vérifiant la définition 11
début
  tant que $k \neq 0$ faire
    Choisir i tel que

$$\left\lfloor \frac{F_{i+2} + F_i}{5} \right\rfloor < |k| \leq \left\lfloor \frac{F_{i+3} + F_{i+1}}{5} \right\rfloor$$

    $d_i = \text{sign}(k)$
    $k = k - d_i F_i$
  fin
fin
retourner $(d_n \dots d_1)$

A partir de là, il est très simple de modifier l'algorithme 14 pour pouvoir effectuer une multiplication de point utilisant la représentation admissible du scalaire.

Enfin, comme prévu, la densité de chiffres non nuls diminue bien : elle passe de 0.2764 à 0.2 [FS07]. La complexité de l'algorithme 14 passe quant à elle à $10.1 \times l$ M+ $3.7 \times l$ C (pour un scalaire de l bits).

Comme d'habitude, il est possible de diminuer encore le nombre de chiffres non nuls en utilisant un équivalent des méthodes par fenêtres glissantes. Il est, par exemple, possible de modifier directement la représentation admissible d'un entier avec les quelques règles suivantes :

Proposition 10

1. $F_{n+3} + F_n = 2F_{n+2} \rightarrow 1001_{\mathcal{Z}} = 0200_{\mathcal{Z}}$,
2. $F_{n+3} - F_n = 2F_{n+1} \rightarrow 100\overline{1}_{\mathcal{Z}} = 0020_{\mathcal{Z}}$,
3. $F_{n+4} + F_n = 3F_{n+2} \rightarrow 10001_{\mathcal{Z}} = 00300_{\mathcal{Z}}$,
4. $F_{n+6} - F_n = 4F_{n+3} \rightarrow 1000001_{\mathcal{Z}} = 0004000_{\mathcal{Z}}$.

Les expérimentations tendent à montrer que l'utilisation de ces règles de recodage permettent de réduire la densité de chiffres non nuls à environ 0.135, de telle sorte que le nombre d'additions de l'algorithme 14 passe à 0.194 par bit. Ici apparaît la limite de l'approche par la représentation de Zeckendorf. En effet, même en utilisant toutes les astuces de réduction précédentes, on peut se rendre compte que le nombre d'additions est finalement très proche de ce que l'on pouvait obtenir dans le cas binaire avec la forme non adjacente de largeur w (0.2 addition par bit). Or si l'on compare les coûts fixes des deux méthodes (c'est-à-dire le coût des doublements d'une part et celui des Fibonacci de l'autre), on peut voir que celui des méthodes binaires (1 doublement par bit, soit 4M+4C avec les coordonnées jacobiennes modifiées) est largement moindre que celui des méthodes

utilisant la représentation de Zeckendorf (1.44 "Fibonacci" par bit soit 1.44×(5M+2C)). Ainsi le nombre d'additions devient sensiblement le même dans les deux cas, l'avantage des nouvelles formules disparaît.

Remarque 3 Il est bien sûr possible de trouver bien d'autres propriétés de ce genre dans la littérature dédiée aux nombres de Fibonacci. Cependant les quelques règles précédemment citées sont suffisantes tant que l'on à faire à des nombres de 160 bits. La raison principale étant qu'au delà, le coût des précalculs devient beaucoup trop important en comparaison du gain apporté concernant le nombre d'additions.

Chapitre 4

Chaînes d'additions différentielles

Nous venons de voir que l'approche par la représentation de Zeckendorf permettait de tirer parti de nos nouvelles formules d'addition. Cependant le surplus de calcul inhérent à cette représentation, ainsi qu'à ses différentes variantes, la rend peu intéressante en pratique. Le but de chapitre est donc de présenter une approche moins intuitive mais plus efficace afin de tirer tout le bénéfice possible des nouvelles formules.

Reprenons l'algorithme NewADD. Celui-ci permet de calculer la somme de deux points P_1 et P_2 partageant la même coordonnée z (en coordonnées jacobiennes), ainsi qu'un point équivalent à P_1 (ou P_2 par symétrie des formules) ayant la même coordonnées z que $P_1 + P_2$. Nous avons vu de plus dans l'exemple 9 qu'il était possible de calculer $[25]P$ en utilisant uniquement cet opérateur. On pouvait le faire en suivant le schéma de calcul suivant :

- NewADD$([2]P, P)$=$([3]P, [2]P)$
- NewADD$([2]P, [3]P)$=$([5]P, [2]P)$
- NewADD$([2]P, [5]P)$=$([7]P, [2]P)$
- NewADD$([7]P, [2]P)$=$([9]P, [7]P)$
- NewADD$([9]P, [7]P)$=$([16]P, [9]P)$
- NewADD$([16]P, [9]P)$=$([25]P, [16]P)$

La question qui se pose alors naturellement est de savoir s'il est toujours possible d'effectuer une multiplication de point suivant un tel schéma et, le cas échéant, si ce schéma est facilement calculable. Nous allons voir que la réponse à ces deux interrogations est oui, mais que d'autres problèmes, bien plus difficiles, vont alors se poser.

4.1 Chaînes d'additions euclidiennes

Le but de cette section est de présenter les chaînes d'additions euclidiennes et de montrer qu'elles sont parfaitement adaptées à l'algorithme NewADD.

Pour cela commençons par quelque définitions.

Définition 12 Une chaîne d'additions calculant un entier k est une suite finie $v = (v_1, \ldots, v_s)$ vérifiant :

- $v_1 = 1,\ v_s = k$
- $\forall\ 1 \leq i \leq s,\ \exists\ i_1,\ i_2 < i :\ v_i = v_{i_1} + v_{i_2}$

Définition 13 Une chaîne d'additions euclidienne calculant un entier k est une chaîne d'additions telle que :

- $v_1 = 1, v_2 = 2, v_3 = v_2 + v_1$
- $\forall\ 3 \leq i \leq s-1,$ si $v_i = v_{i-1} + v_j$ pour un certain $j < i-1$, alors :
  - ou bien $v_{i+1} = v_i + v_{i-1}$ (cas 1)
  - ou bien $v_{i+1} = v_i + v_j$ (cas 2)

Dans le cas 1 nous parlerons de grand pas (puisque l'on ajoute le plus grands des deux entiers à v_i) et dans le cas 2 de petit pas.

Par exemple $(1, 2, 3, 5, 7, 9, 16, 25)$ est une chaîne d'additions euclidienne calculant 25. Pour s'en convaincre, nous pouvons voir qu'à l'étape 4 nous avons calculé $5 = 3 + 2$. Ainsi à l'étape 5 les seules possibilités sont de calculer 8= 5 + 3 ou 7=5+2. Dit différemment, après avoir calculé 5 = 3+2, seuls 2 et 3 peuvent être ajoutés à 5. Dans cet exemple nous avons choisi de calculer 7=5+2 de telle sorte que, à l'étape suivante, il est possible de calculer soit 12 = 7+5 soit 9 = 7+2 etc. Remarquons du coup que ce type de chaînes d'additions est parfaitement adapté au fonctionnement de l'algorithme NewADD.

La suite de Fibonacci est également un bon exemple de chaîne d'additions euclidienne. Celle-ci est d'ailleurs remarquable puisqu'elle n'est constituée que de grand pas.

Trouver une chaîne d'additions euclidienne calculant un entier k donné est très facile, il suffit pour cela de choisir un entier g premier avec k et d'appliquer la version additive de l'algorithme d'Euclide.

Exemple 11 Prenons $k = 25$ et $g = 16$

$$\begin{aligned} 25 - 16 &= 9 \quad \text{(grand pas)} \\ 16 - 9 &= 7 \quad \text{(grand pas)} \\ 9 - 7 &= 2 \quad \text{(petit pas)} \\ 7 - 2 &= 5 \quad \text{(petit pas)} \\ 5 - 2 &= 3 \quad \text{(grand pas)} \\ 3 - 2 &= 1 \\ 2 - 1 &= 1 \\ 1 - 1 &= 0 \end{aligned}$$

On peut alors retrouver la chaîne d'addition correspondante simplement en lisant le premier nombre de chaque ligne. On obtient ici la chaîne

$$(1, 2, 3, 5, 7, 9, 16, 25). \tag{4.1}$$

Pour finir, afin de faciliter l'écriture des algorithmes, nous adopterons la notation suivante : si $v = (1, 2, 3, v_4, \ldots, v_s)$ est une chaîne d'additions euclidienne alors nous ne la considérerons qu'à partir de v_4. De plus nous remplacerons les v_i par 0 si ceux-ci ont été calculés lors d'un grand pas et par 1 sinon.

$$\begin{array}{rl} \text{Par exemple la suite :} & (1,\ 2,\ 3,\quad 5,\quad 7,\quad 9,\quad 16,\quad 25) \\ \text{s'écrira :} & \qquad\qquad\quad (0,\quad 1,\quad 1,\quad 0,\quad 0). \end{array}$$

Enfin on notera la chaîne $c = (c_4, \ldots, c_s)$ au lieu de v afin d'éviter toute confusion entre les deux notations.

Il nous est maintenant possible d'écrire un premier algorithme de multiplication de point par un scalaire pour lequel on dispose d'une chaîne d'additions euclidienne le calculant.

Algorithme 16 : Multiplication de point à l'aide d'une chaîne euclidienne

Données : $P \in E$ et $c = (c_4, \ldots, c_s)$ une chaîne d'additions euclidienne calculant k ;
Résultat : $[k]P \in E$;
début
  $(U_1, U_2) \leftarrow \text{NewADD}(P, P)$
  pour $i = 4 \ldots s$ faire
    si $c_i = 0$ alors
      $(U_1, U_2) \leftarrow \text{NewADD}(U_1, U_2)$;
    sinon
      $(U_1, U_2) \leftarrow \text{NewADD}(U_2, U_1)$;
    fin
  fin
  $(U_1, U_2) \leftarrow \text{NewADD}(U_1, U_2)$;
  retourner U_1
fin

Remarque 4 L'algorithme précédent possède la propriété intéressante de toujours effectuer la même opération. En effet, à part la première étape qui est un doublement, chaque étape de l'algorithme consiste juste à calculer $\text{NewADD}(U_{1+i}, U_{2-i})$. Cela rend cet algorithme, au moins théoriquement, résistant aux attaques par canaux cachés.

Exemple 12 Nous donnons ici le calcul détaillé de $[34]P$, à l'aide de l'algorithme 16, en utilisant ma chaîne $c = (1, 0, 0, 1, 1, 0)$ calculant 34.

	on calcule d'abord	$\text{NewADD}(P, P) = ([2]P, P)$
$c_4 = 1$	$\rightarrow$	$\text{NewADD}(P, [2]P) = ([3]P, P)$
$c_5 = 0$	$\rightarrow$	$\text{NewADD}([3]P, P) = ([4]P, [3]P)$
$c_6 = 0$	$\rightarrow$	$\text{NewADD}([4]P, [3]P) = ([7]P, [4]P)$
$c_7 = 1$	$\rightarrow$	$\text{NewADD}([4]P, [7]P) = ([11]P, [4]P)$
$c_8 = 1$	$\rightarrow$	$\text{NewADD}([4]P, [11]P) = ([15]P, [4]P)$
$c_9 = 0$	$\rightarrow$	$\text{NewADD}([15]P, [4]P) = ([19]P, [15]P)$
	et enfin	$\text{NewADD}([19]P, [15]P)$ donne $[34]P$

En considérant que le point P est donné en coordonnées affines ($Z = 1$) le premier doublement peut s'effectuer en 2M et 4C (voire formules de doublements page 6). Le coût total de l'algorithme devenant $(5s - 8)\text{M} + 2s\text{C}$. Il est même possible de faire encore mieux sous certaines conditions. En effet, dans certains cas, le calcul de la coordonnée y n'est pas nécessaire. Plus exactement celle-ci n'apporte que peu d'information. Prenons l'exemple d'un échange de clé utilisant le protocole Diffie-Hellman (voir exemple ?? page ??), le but est que les deux parties possèdent un secret commun, dans notre cas un point d'une courbe elliptique. Dans ce cas, le fait de connaître la coordonnée y du point n'apporte que peut d'information, dans le sens où elle peut être facilement déduite de la coordonnée x et n'augmente donc pas le nombre de clés potentielles. Par exemple, si la courbe est définie sur un corps de caractéristique supérieure à 5, l'équation de la courbe est alors de la forme $y^2 = x^3 + ax + b$. Ainsi à tout élément x du corps correspond au plus deux points sur la courbe (dont les coordonnées y sont les deux racines potentielles du polynôme $f(y) = y^2 - (x^3 + ax + b)$). Au final, le fait de ne calculer que la coordonnée x ne fait, dans le pire des cas, que diviser par 2 le nombre de clés potentielles.

Dans ce cas il est possible d'économiser une multiplication par étape de l'algorithme 16. Pour cela reprenons les formules d'additions de la page 40. Soient $P_1 = (X_1, Y_1, Z)$, $P_2 = (X_2, Y_2, Z)$ et $P_1 + P_2 = (X_3, Y_3, Z_3)$ trois points en coordonnées jacobiennes. Nous avons vu que :

- $X_3 = (Y_2 - Y_1)^2 - X_1(X_2 - X_1)^2 - X_2(X_2 - X_1)^2$
- $Y_3 = (Y_2 - Y_1)(X_1(X_2 - X_1)^2 - X_3) - Y_1(X_2 - X_1)^3$

Nous avons également vu que le point $P_1' = (X_1(X_2 - X_1)^2, Y_1(X_2 - X_1)^3, Z_3)$ était équivalent au point P_1 et possédait la même coordonnée z que $P_3 = P_1 + P_2$. Or comme nous venons de le voir, lors du calcul des coordonnées x et y de la somme de deux points la coordonnées z n'intervient pas, dès lors que celle-ci est commune à ces deux points. Il est ainsi possible de faire la somme de P_3 et P_1' tout en ignorant la valeur Z_3. Au final cela nous donne un algorithme de multiplication de point par un scalaire ne calculant que les coordonnées x et $y$en coordonnées jacobiennes.

Supposons maintenant que nous ayons calculé les coordonnées x et y du point $[k]P =$

(X_k, Y_k, Z_k) à l'aide de cet algorithme. A priori, il n'est pas possible de récupérer un point en coordonnées affines (puisque cela nécessite le calcul de $(X_k/Z_k^2, Y_k/Z_k^3)$ et l'on ne dispose pas de Z_k). Cependant, la propriété suivante va nous montré qu'il est possible de retrouver la valeur Z_k^2, et de calculer la coordonnée x de $[k]P$ en coordonnées affines.

Proposition 11 Soient $P_1 = (X_1, Y_1, Z)$, $P_2 = (X_2, Y_2, Z)$ et
$P_1 + P_2 = (X_3, Y_3, Z_3)$ trois points donnés en coordonnées jacobiennes, alors

$$Z^2 = \frac{a}{2b}\left[\frac{(X_1 - X_2)(X_3 + 2Y_2Y_1 - X_1X_2(X_1 + X_2))}{Y_1^2 - Y_2^2 + X_2^3 - X_1^3} - (X_1 + X_2)\right]$$

Preuve : P_1 et P_2 vérifient $Y^2 = X^3 + aXZ^4 + bZ^6$ donc

$$\begin{aligned} Y_1^2 - Y_2^2 &= X_1^3 - X_2^3 + aX_1Z^4 - aX_2Z^4 + bZ^6 - bZ^6 \\ \text{ce qui donne} & \\ Z^4 &= \frac{Y_1^2 - Y_2^2 + X_2^3 - X_1^3}{a(X_1 - X_2)} \end{aligned}$$

de plus

$$\begin{aligned} X_3 &= (Y_2 - Y_1)^2 - (X_1 + X_2)(X_2 - X_1)^2 \\ &= Y_2^2 - 2Y_2Y_1 + Y_1^2 - X_2^3 + X_2^2X_1 + X_1^2X_2 - X_1^3 \\ &= Y_2^2 - X_2^3 + Y_1^2 - X_1^3 - 2Y_2Y_1 + X_1X_2(X_1 + X_2) \\ &= aX_1Z^4 + bZ^6 + aX_2Z^4 + bZ^6 - 2Y_2Y_1 + X_1X_2(X_1 + X_2) \\ &= Z^4(a(X_1 + X_2) + 2bZ^2) - 2Y_2Y_1 + X_1X_2(X_1 + X_2) \end{aligned}$$

et donc

$$Z^2 = \frac{a}{2b}\left[\frac{(X_1 - X_2)(X_3 + 2Y_2Y_1 - X_1X_2(X_1 + X_2))}{Y_1^2 - Y_2^2 + X_2^3 - X_1^3} - (X_1 + X_2)\right]$$

Le coût total pour retrouver Z^2 est 7M+4C+I.

4.2 Question de longueur

Nous avons vu que trouver une chaîne d'additions euclidienne calculant un entier k donné est très facile, il suffit pour cela de choisir un entier g premier avec k et d'appliquer l'algorithme d'Euclide additif. Nous allons maintenant voir que la recherche de chaînes courtes est, elle, beaucoup plus ardue.

Commençons, pour se donner une première idée, par un théorème démontré par Knuth et Yao en 1975 [KY75].

Théorème 7 Soit $S(k)$ le nombre moyen d'étapes pour calculer le pgcd de k et g en utilisant l'algorithme d'Euclide additif, avec g uniformément distribué dans l'intervalle $1 \leq g \leq k$, alors

$$S(k) = 6\pi^{-2}(\ln k)^2 + O(\log k(\log\log k)^2)$$

Ce théorème montre en substance que le fait de choisir l'entier g de façon aléatoire conduira en pratique à des chaînes d'additions dont la longueur sera de l'ordre de $(\ln k)^2$. Cela n'est pas du tout satisfaisant dans l'optique d'une application à la multiplication de point. Par exemple, pour un scalaire de 160 bits, le théorème nous dit que la longueur moyenne d'une chaîne euclidienne est aux alentours de 7000 (plutôt 2500 en pratique). Nous avons vu page 53 que la complexité de l'algorithme 16 utilisant les chaînes euclidiennes est de $(5s-8)$M+$2s$C (soit $6.6s$M-8 avec C=0.8M), où s est la longueur de la chaîne. D'après le tableau 2.2 page 28, pour une courbe définie sur un corps premier, l'algorithme de doublement et addition requiert environ 2100M. Ce qui signifie que, pour rester compétitif, l'algorithme 16 nécessite des chaînes de longueur au plus 320 (c'est-à-dire à peu près 2100/6.6). Cela en fait, en ce moment là, un algorithme aussi efficace que la méthode usuelle de multiplication de points par un scalaire, avec cependant l'avantage non négligeable d'être naturellement protégé contre les attaques par canaux cachés.

Une façon classique de limiter la longueur des chaînes euclidiennes est de choisir g « proche » de $\frac{k}{\phi}$, où $\phi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or. En effet cela assure que les premières étapes de l'algorithme d'Euclide additif correspondront à des grands pas. Pour s'en convaincre commençons par chercher une condition assurant que la dernière étape soit un grand pas. On cherche donc une condition sur g assurant que $k = g+(k-g)$ avec $k-g < g < k$. Cela se traduit simplement par la condition $\frac{k}{2} < g < k$. Maintenant étudions la condition assurant deux grands pas successifs. On cherche donc g vérifiant $2g-k < k-g < g < k$, ce qui se traduit par $\frac{k}{2} < g < \frac{2k}{3}$. Pour trois pas consécutifs on obtient de façon similaire la condition $\frac{3k}{5} < g < \frac{2k}{3}$ etc. On peut alors montrer par récurrence qu'une condition nécessaire et suffisante pour qu'une chaîne d'additions euclidienne finisse par n grand pas est

$$\frac{F_n}{F_{n+1}}k < g < \frac{F_{n+1}}{F_{n+2}}k \tag{4.2}$$

si n est pair et

$$\frac{F_{n+1}}{F_{n+2}}k < g < \frac{F_n}{F_{n+1}}k \tag{4.3}$$

si n est impair.

Nous savons également que

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \phi.$$

Tout ceci permet de justifier le choix de g proche de $\frac{k}{\phi}$.

De façon complémentaire, il est évident possible de tester différents g autour de $\frac{k}{\phi}$ et de garder celui donnant la meilleure chaîne.

Ces remarques ont déjà été faites dans un cadre plus général [Mon83] par Montgomery.

Pour ce donner un ordre d'idée, nous avons effectuer quelques tests expérimentaux. La procédure est la suivante :

- nous choisissons, de façon « aléatoire », un entier k de 160 bit,
- nous calculons $g = \left\lceil \frac{k}{\phi} \right\rfloor$,
- si $\text{pgcd}(k, g) = 1$, nous calculons la longueur de la chaîne euclidienne correspondante,
- nous recommençons avec $g + 1$.

Nous comptons ainsi le nombre d'itérations nécessaires pour obtenir une chaîne de longueur inférieure à une valeur donnée. Le tableau 4.1 donne une estimation du nombre d'essaies nécessaires pour différentes tailles de chaînes. Estimation faite après avoir testé 10 000 000 d'entiers de 160 bits.

longueur de chaîne	320	300	280	260
moyenne	29	121	2353	7,795,840
pire cas	521	3,454	44,254	79,402,210

Tab. 4.1 – Nombre d'essaies pour trouver des chaînes d'additions euclidiennes d'une longueur donnée, pour des entiers de 160 bits

Nous pouvons ainsi voir que, s'agissant d'entiers de 160 bits, trouver une chaîne d'additions euclidienne de longueur 320 demande, en moyenne de, tester 29 valeurs de g. trouver des chaînes plus courtes devient par contre beaucoup plus difficile, par exemple des chaînes de longueur 270 requiert souvent plus de 45 000 essaies.

Tout cela signifie qu'une telle recherche ne peut pas être intégré à un protocole cryptographique, dès lors que ce dernier ne permet pas la réutilisation du scalaire. En effet le coup de la recherche peut rapidement devenir prohibitif. A moins de pouvoir effectuer des pré-calculs hors-ligne, l'utilisation des chaînes euclidiennes doit donc se limiter à celles de longueur entre 300 et 320 (pour lesquelles le coût d'un recherche reste raisonnable comparé à celui de la multiplication de point elle-même).

4.3 Supprimer la dépendance à la longueur de la chaîne

Nous l'avons vu dans la section précédente les chaînes d'additions euclidiennes s'accommodent très bien avec notre nouvel algorithme d'addition de points. Cependant l'efficacité de l'algorithme de multiplication de point qui en résulte dépend largement de la longueur de la chaîne utilisée. Cette section présente donc une seconde version de

l'algorithme 16 de multiplication de point dont la complexité ne dépend plus aussi dramatiquement de la chaîne, supprimant dans le même temps la coûteuse phase de recherche itérative décrite plus haut.

Dans un premier temps remarquons que le principal problème quant à l'efficacité des chaînes euclidiennes vient des longues séries de petits pas (c'est à dire des longues séries de 1 avec la notation de la section 4.1).

Exemple 13 Prenons $k = 15502$, nous avons donc $g = 9581 \simeq \frac{k}{\phi}$. L'algorithme d'Euclide donne la chaîne suivante :

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 29, 43, 57, 71, 85, 99, 113,$$

$$212, 325, 537, 862, 1399, 2261, 3660, 5921, 9581, 15502$$

ou avec la notation de la section 4.1 :

$$1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0$$

qui est une chaîne de longueur 32, alors qu'une recherche plus approfondie permet de montrer qu'il existe des chaînes euclidiennes de longueur 22 calculant k (en prenant $g = 9909$ par exemple).

Analysons l'exemple précédent. Prenons la première série de petits pas partant du calcul de 4 jusqu'à celui de 15. En 12 étapes de calculs nous nous retrouvons finalement avec le nombre $3 + 12 \times 1$. Plus généralement si $v_i = v_{i-1} + v_j$ est calculé à l'étape i et qu'une série de t petits pas débute à l'étape $i + 1$ cela signifie que l'algorithme 16 va additionner l'entier v_j t fois au résultat courant, ce qui revient donc à calculer $v_{i+t+1} = v_i + t \times v_j$. Tout le problème des petits pas apparaît dans cette équation, en effet nous avons dans le cas présent une augmentation linéaire du résultat courant en fonction du nombre d'étapes alors qu'une série de grand assure une croissance exponentielle. Le but est donc de limiter le sur-coût occasionné par ces longues séries de petits pas. L'idée que nous allons développer maintenant est de calculer directement $t \times v_j$ dès que t est trop grand de telle manière que nous pourrons ensuite reprendre le cours de l'algorithme 16 sans trop de perte. Le but est donc de calculer à la fois la valeur $v_i + t \times v_j$, mais aussi $v_{i-1} + t \times v_j = v_i + (t - 1) \times v_j$ qui est la valeur nécessaire pour reprendre l'algorithme 16.

Pour ce faire nous allons utiliser le schéma de calcul suivant :
- à l'étape $i - 1$ nous avons v_{i-1} et v_j
- nous calculons $t \times v_j$ en utilisant une chaîne euclidienne courte pour t
- nous additionnons v_{i-1} à $t \times v_j$
- nous calculons enfin $v_j + v_{i-1} + t \times v_j = v_i + t \times v_j$

Exemple 14 Reprenons notre chaîne calculant 15502 et considérons la série de petits pas entre 4 et 15. Le nombre suivant étant 29 il nous faut donc calculer 14 et 15, en suivant le schéma proposé cela donne :

- nous avons $v_{i-1} = 2$ et $v_j = 1$
- nous devons calculer 12×1 en utilisant une chaîne courte, par exemple $(0, 1, 0)$
- nous additionnons 2 et 12 pour obtenir 14
- il ne reste plus qu'à calculer $15 = 14 + 1$

Maintenant il faut adapter le schéma précédent à l'algorithme NewADD. Cette nouvelle approche repose sur les deux remarques suivantes :

Remarque 5 Si $P_1 = (X_1, Y_1, Z \times Z')$ et $P_2 = (X_2, Y_2, Z)$ sont deux points d'une courbe alors il ne faut que 3 multiplications sur le corps de base (4 si l'on possède Z' mais pas $Z \times Z'$) pour obtenir le point $P_2' = (X_2 \times Z'^2, Y_2 \times Z'^3, Z \times Z')$ équivalent au point P_2 et partageant la même coordonnée z que le point P_1.

Remarque 6 Soient $P_1 = (X_1, Y_1, Z \times Z')$ et $P_2 = (X_2, Y_2, Z \times Z')$ deux points. Posons $P_3 = (X_3, Y_3, Z_3) = \text{NewADD}(P_1, P_2)$ et $P_3' = (X_3', Y_3', Z_3') = \text{NewADD}(P_1', P_2')$ où $P_1' = (X_1, Y_1, Z)$ et $P_2' = (X_2, Y_2, Z)$,
alors $(X_3, Y_3) = (X_3', Y_3')$ et $Z_3 = Z_3' \times Z'$.

La première remarque est triviale. Pour illustrer la seconde, reprenons à nouveau les formules d'additions de points. En prenant les notations de la remarque 6 nous avons :

- $X_3 = (Y_2 - Y_1)^2 - X_1(X_2 - X_1)^2 - X_2(X_2 - X_1)^2$
- $Y_3 = (Y_2 - Y_1)(X_1(X_2 - X_1)^2 - X_3) - Y_1(X_2 - X_1)^3$
- $Z_3 = ZZ'(X_2 - X_1)$

La coordonnée z n'intervenant pas dans les calculs de X_3 et Y_3, il est logique que les sommes de P_1 et P_2 d'une part, et de P_1' et P_2' d'autre part possèdent les même coordonnées x et y. De plus, en remplaçant ZZ' par Z, il est aisé de voir que $Z_3' = Z(X_2 - X_1)$ et que nous avons bien $Z_3 = Z_3' \times Z'$.

Nous pouvons maintenant proposer un schéma de calcul de points utilisant l'opérateur NewADD :

1. considérons qu'à l'étape $i-1$ nous avons d'une part $[v_{i-1}]P = (X_{[v_{i-1}]P}, Y_{[v_{i-1}]P}, Z)$ et d'autre part $[v_j]P = (X_{[v_j]P}, Y_{[v_j]P}, Z)$
2. nous calculons $\text{NewADD}([v_j]P, [v_j]P)$ pour obtenir des points $[2]([v_j]P) = [2v_j]P = (X_{[2v_{i-1}]P}, Y_{[2v_{i-1}]P}, Z2Y_{[v_{i-1}]P})$ et $[v_{i-1}]P'$ équivalent à $[v_{i-1}]P$ ayant la même coordonnée z que $[2v_j]P$
3. nous évaluons le point $[t \times v_{i-1}]P'$ à l'aide d'une chaîne d'addition courte calculant t en utilisant les points précédemment calculés en considérant que leur coordonnées z sont égales à $2Y_{[v_{i-1}]P}$. D'après la seconde remarque précédente nous savons que $[t \times v_j]P'$ a les même coordonnées x et y que $[t \times v_j]P$. De plus $Z_{[t \times v_j]P} = Z_{[t \times v_j]P'} \times Z$

4. nous ajoutons $[v_{i-1}]P$ à $[t \times v_j]P$ avec nos formules, avec un sur-coût de 3M+C (d'après la première remarque)
5. nous calculons $[v_j]P + [v_{i-1} + t \times v_j]P$ en utilisant NewADD pour un coût supplémentaire de 4M+C

A partir du moment où ont lieu plus de 7 petits pas consécutifs, notre schéma devient rentable. Grâce à cette remarque, nous pouvons donc écrire un nouvel algorithme de multiplication par un scalaire.

Remarque 7 Nous utilisons la notation NewSCH$(n, [v_{i-1}]P, [v_j]P)$ pour désigner l'opération décrite précédemment permettant d'obtenir les points $[v_i + t \times v_j]P$ et $[v_i + (t-1) \times v_j]P)$ avec la même coordonnée z, ceci afin de simplifier l'écriture de l'algorithme 17. De plus nous remplaçons les séries de 1 par le nombre de 1. Par exemple

$$(1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0)$$

s'écrira

$$(1, 0, 0, 2, 0, 0, 6, 0).$$

Algorithme 17 : Nouveau schéma de multiplication de point par chaîne euclidienne

```
Données : P ∈ E et une chaîne c = (c_4, ..., c_s) calculant k
Résultat : [k]P ∈ E
début
    (U_1, U_2) ← NewADD(P, P)
    i ← 4
    tant que i ≤ s faire
        si c_i = 0 alors
            (U_1, U_2) ← NewADD(U_1, U_2)
        sinon
            si c_i < 7 alors
                pour t = 1 ... c_i faire
                    (U_1, U_2) ← NewADD(U_2, U_1)
                fin
            sinon
                (U_1, U_2) ← NewSCH(c_i, U_1, U_2)
            fin
        fin
        i ← i + 1
    fin
    (U_1, U_2) ← NewADD(U_1, U_2)
    retourner U_1
fin
```

Quelques remarques

Le schéma précédent propose de calculer $[t \times v_j]P'$ à l'aide d'une chaîne d'additions euclidienne pré-calculée. Bien entendu il est inconcevable de pré-calculer des chaînes courtes pour tous les entiers pouvant intervenir au court de l'algorithme 17. Cependant les expériences semblent montrer que pour une chaîne $c = (c_4, \ldots, c_s)$ calculant un entier aléatoire de 160 bits, les c_i sont en moyenne relativement petits : seulement 5.8% des chaînes possèdent un c_i supérieur à 2^{10} et seulement 0.3% en possèdent un supérieur à 2^{14}.

Ainsi il devient envisageable de pré-calculer une chaîne optimale pour tous les entiers inférieurs à un certain seuil S uniquement (par exemple 2^{10}) tout en garantissant que l'algorithme pourra s'effectuer de façon optimale la plupart du temps.

Néanmoins si l'on ne dispose pas d'une chaîne minimale pour un certain c_i (c'est à dire que ce c_i est plus grand que le seuil choisi) on peu toujours utiliser l'une des méthodes suivantes :

1. effectuer la division euclidienne de c_i par S : $c_i = c'_i \times S + r_i$ (si S est une puissance de 2 l'opération ne coûte rien), puis calculer $[S]P$ et $[r_i]P$ avec une chaîne pré-calculée. Si $c'_i < S$ nous pouvons calculer $[c_i]P$, sinon on recommence avec c'_i,
2. appliquer de façon récursive l'algorithme 17 pour calculer $[c_i]P$, ce qui signifie que l'on doit être capable de calculer l'entier le plus proche de $\frac{c_i}{\phi}$ et calculer la chaîne euclidienne correspondante,
3. si le protocole le permet choisir un autre scalaire k (dans la mesure où il y a peu de chances de tomber sur un chaîne posant problème).

Comparaisons

Ici nous présentons quelques résultats pratiques sur la complexité des différents algorithmes proposés précédemment ainsi que des comparaisons avec les méthodes classiques de l'état de l'art.

Le tableau 4.2 permet de comparer notre nouvel algorithme NewADD utilisé avec des chaînes d'additions euclidiennes avec l'échelle de Montgomery et les chaînes d'additions euclidiennes sur les courbes de Montgomery.

Le tableau 4.3 permet de comparer notre nouvelle méthode utilisant la représentation de Zeckendorf et ses différentes améliorations avec leurs équivalents binaires (soit le doublement et addition, le NAF et le 4-NAF). Nous avons également ajouté à nos comparaisons la seconde méthode utilisant les chaînes euclidiennes. Dans ce cas précis, n'ayant pas de résultats théoriques sur la complexité moyenne de l'algorithme, le résultat donné est la complexité moyenne obtenue à l'aide de 10^5 entiers aléatoires de 160 bits.

Nous considérons ici que C=0.8M, que k est un entier de 160 bits et on reprend les résultats donnés dans [CMO98] concernant la complexité de la méthode binaire à fenêtre.

Le tableau 4.2 montre que l'opérateur NewADD permet en quelque sorte de généraliser l'utilisation des chaînes d'additions euclidiennes sans perte d'efficacité. De plus il est

Algorithme	type de courbe	calcul de y	nbr. Mult.
Ech. Montgomery	Montgomery	avec	1463
EAC-320	Montgomery	sans	1792
EAC-270	Montgomery	sans	1512
EAC-320	Weiestraß	avec	2112
EAC-270	Weiestraß	avec	1782
EAC-320	Weiestraß	sans	1792
EAC-270	Weiestraß	sans	1512

Tab. 4.2 – Comparaisons avec les courbes de Montgomery définies sur des corps premiers de 2^{160} éléments

Algorithme	Coord.	nbr. Mult.
Dbl-et-add	Mixtes	2104
NAF	Mixtes	1780
4-NAF	Mixtes	1600
Fibonacci-et-add	NewADD	2311
Fib-et-add signé	NewADD	2088
Fib-et-add fenêtre	NewADD	1960
EAC-V2	NewADD	1846

Tab. 4.3 – Comparaisons entre les méthodes binaires et la représentation de Zeckendorf sur des courbes elliptiques définies sur des corps premiers de 2^{160} éléments

même possible de calculer à la fois les coordonnées x et y avec notre nouvelle méthodes, moyennant une légère perte d'efficacité, alors que cela n'est pas possible sur les courbes de Montgomery. Malgré tout l'échelle de Montgomery reste encore et toujours l'algorithme le plus efficace.

Si l'on compare les algorithmes similaires du tableau 4.3 on se rend compte que l'utilisation des formules introduites dans le chapitre précédent ne permet pas de compenser à aucun moment le sur-coût inhérent à la représentation de Zeckendorf. Ce sur-coût variant entre 10 et 23 % selon la méthode utilisée.

4.4 Chaînes d'additions différentielles

Dans les sections précédentes nous avons vu que l'opérateur NewADD était parfaitement adapté à la multiplication de point par un scalaire utilisant des chaînes d'additions euclidiennes. Nous avons également vu que trouver des chaînes efficaces, c'est-à-dire courtes, était particulièrement difficile. L'idée développée dans cette section est donc d'étudier une classe plus large de chaîne d'additions ayant des propriétés similaires à celles des chaînes euclidiennes. Ce qui suit se base largement sur les études faites dans [Mon83] et [Ber06].

Définition 14 Une chaîne d'additions différentielles calculant un entier k est une chaîne d'additions $v = (v_1, \ldots, v_s)$ calculant k telle que :

$$\forall i \geq j \geq 1, \forall t \geq 2, \quad v_t = v_i + v_j \Rightarrow v_i - v_j \in \{v_1, \ldots, v_{t-1}\}$$

Cela signifie en substance que si l'on veut faire la somme de deux éléments P et Q d'une telle chaîne, la différence $P - Q$ doit déjà se trouver dans la chaîne.

Exemple 15 $(1, 2, 3, 5, 7, 10, 17, 27)$ est une chaîne d'additions différentielle :

$$\begin{aligned} 2 &= 1 + 1 \quad \text{différence} \quad 0 \\ 3 &= 2 + 1 \quad \text{différence} \quad 1 \\ 5 &= 3 + 2 \quad \text{différence} \quad 1 \\ 7 &= 5 + 2 \quad \text{différence} \quad 3 \\ 10 &= 5 + 5 \quad \text{différence} \quad 0 \\ 17 &= 10 + 7 \quad \text{différence} \quad 3 \\ 27 &= 17 + 10 \quad \text{différence} \quad 7 \end{aligned}$$

L'avantage de ces chaînes est qu'elles permettent d'obtenir des algorithmes d'exponentiation très efficace dès lors que l'opération

$$P, Q, P - Q \rightarrow P + Q$$

est efficace. C'est évidemment le cas sur les courbes de Montgomery, c'est un peu plus compliqué s'agissant des formules introduites dans ce mémoire page 40. En effet, non seulement la différence des deux points concernés doit avoir été calculée dans la chaîne, mais celle-ci doit en plus avoir servi dans le calcul précédent. Autrement dit, pour que le calcul de $P + Q$ soit efficace il faut que le calcul $(P - Q) + Q$ ai été effectué auparavant.

Jusqu'à présent, les seuls exemples de chaînes d'additions différentielles que nous ayons vus sont les chaînes euclidiennes ainsi que celles obtenues grâce à l'échelle de Montgomery. Nous avons ainsi vu que, dans le premier cas, il était difficile d'obtenir des chaînes efficaces et que, dans le second, le nombre d'opérations était fixe et n'était vraiment efficace que s'il

était utilisé sur les courbes de Montgomery. L'idée est donc de nous intéresser à d'autres constructions de chaînes différentielles, naturellement plus courtes. Plus précisément nous allons voir quelques techniques permettant de limiter les longues séquences de petits pas au sein des chaînes d'additions euclidiennes.

Chaînes de Bleichenbacher et Tsuruoka

Nous avons vu qu'une manière simple d'obtenir une chaîne d'additions euclidienne calculant un scalaire k était d'appliquer l'algorithme d'Euclide additif à k et g, où g est un entier inférieur à k. Choisir g proche de $\frac{k}{\phi}$ permet de réduire la taille de la chaîne obtenue. Cela assure que la moitié des bits de k soit obtenue grâce à des grand pas.

Exemple 16 Reprenons l'exemple avec $k = 15502$ et $g = 9581 \simeq \frac{k}{\phi}$. Nous avons vu que cela conduisait à la chaîne

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 29, 43, 57, 71, 85, 99, 113,$$

$$212, 325, 537, 862, 1399, 2261, 3660, 5921, 9581, 15502$$

Si l'on part de la fin de la chaîne (c'est-à-dire de k) on peut s'apercevoir que 113 est le premier entier à ne pas être obtenu par un grand pas ($113 \neq 99 + 85$). Cela signifie qu'il suffit de trouver une chaîne d'additions différentielle efficace calculant à la fois 113 et 99 (entiers dont les tailles en bit sont, grosso modo, de l'ordre de la moitié de celle de k) pour ensuite obtenir k en utilisant uniquement des grands pas.

A partir de maintenant nous allons donc énumérer quelques méthodes classiques de modification des chaînes euclidiennes permettant ainsi d'obtenir des chaînes différentielles plus courtes.

Une première façon de limiter les comportements coûteux est d'utiliser les chaînes d'additions définies récursivement par la procédure suivante :

$$S(g,k) = \begin{cases} 0, k & \text{si } g = 0 \\ S(k-g, k) & \text{si } k/2 < g \\ S(g, k/2), k-g, k & \text{si } 0 < g < k/4 \text{ et } k \in 2\mathbb{Z} \\ S(g, k-g), k & \text{sinon} \end{cases} \tag{4.4}$$

La seule différence entre cet algorithme et l'algorithme d'Euclide est la ligne

$$S(g, k/2), k-g, k \text{ si } 0 < g < k/4 \text{ et } k \in 2\mathbb{Z}.$$

Dans ce cas précis k est obtenu par doublement de $k/2$, g apparaît naturellement dans $S(g, k/2)$ et enfin $k - g$ ce calcule grâce à $k/2 - g$, qui apparaît lui aussi dans $S(g, k/2)$, et $k/2$.

Exemple 17 Déroulons la procédure 4.4 appliquée à l'exemple précédent, c'est-à-dire à la recherche d'une chaîne différentielle calculant 113 et 99.

$$\begin{aligned}
S(99,113) &= S(14,113)\\
&= S(14,99),113\\
&= S(14,85),99,113\\
&= S(14,71),85,99,113\\
&= S(14,57),71,85,99,113\\
&= S(14,43),57,71,85,99,113\\
&= S(14,29),43,57,71,85,99,113\\
&= S(14,15),29,43,57,71,85,99,113\\
&= S(1,15),29,43,57,71,85,99,113\\
&= S(1,14),15,29,43,57,71,85,99,113\\
&= S(1,7),13,14,15,29,43,57,71,85,99,113\\
&= S(1,6),7,13,14,15,29,43,57,71,85,99,113\\
&= S(1,3),5,6,7,13,14,15,29,43,57,71,85,99,113\\
&= S(1,2),3,5,6,7,13,14,15,29,43,57,71,85,99,113\\
&= 1,2,3,5,6,7,13,14,15,29,43,57,71,85,99,113
\end{aligned}$$

La longueur de la chaîne a été clairement réduite, cependant la chaîne ainsi créée n'est plus une chaîne euclidienne. Par exemple 13 est obtenu en faisant la somme de 6 et 7, cependant $14 \neq 13+6$ et $14 \neq 13+7$.

Cela signifie que nous ne pouvons plus utiliser l'opérateur NewADD de façon systématique. L'algorithme de multiplication de point requiert quelques aménagements (ce sera l'objet de la section suivante). Pour l'instant, nous allons décrire deux autres constructions de chaînes différentielles plus élaborées. Celles-ci sont plus complexes à mettre en oeuvre mais conduisent, en moyenne, à des chaînes plus courtes.

La première des deux constructions, plus sophistiquée que la précédente, a été proposé par Bleichenbacher [Ble96] et est définie de la manière suivante :

$$B(g,k) = \begin{cases}
0,k & \text{si } g = 0\\
B(k-g,k) & \text{si } k/2 < g\\
B(g,k/2),k-g,k & \text{si } 0 < g < k/5 \text{ et } k \in 2\mathbb{Z}\\
B(g,(k+g)/2),k-g,k & \text{si } 0 < g < k/5 \text{ et } k \notin 2\mathbb{Z} \text{ et } k+g \in 2\mathbb{Z}\\
B(g/2,k-g/2),g,k & \text{si } 0 < g < k/5 \text{ et } k \notin 2\mathbb{Z} \text{ et } g \in 2\mathbb{Z}\\
B(g,k-g),k) & \text{sinon}
\end{cases} \tag{4.5}$$

En reprenant $k = 113$ et $g = 99$ nous obtenons la chaîne différentielle suivante :

$$B(99, 113) = 1, 2, 3, 5, 7, 9, 16, 23, 30, 46, 53, 99, 106, 14, 113 \tag{4.6}$$

Nous obtenons bien une chaîne d'additions différentielle plus courte que les précédentes (la chaîne $B(99, 113)$ est de longueur 15, alors que la chaîne $S(99, 113)$ était de longueur 16).

La seconde construction a été proposée par Tsuruoka [Tsu01]. Elle est plus complexe que les précédentes, mais permet d'obtenir des chaînes encore plus courtes.

$$T(g,k) = \begin{cases} 0, k & \text{si } g = 0 \\ T(k-g, k) & \text{si } k/2 < g \\ T(g, k/2), k-g, k & \text{si } 2g \leq k \leq 2.09g \text{ et } k \in 2\mathbb{Z} \\ T(g, k/2), k-g, k & \text{si } 3.92g \leq k \text{ et } k \in 2\mathbb{Z} \\ T(g, (k+g)/3), (2k-g)/3, k-g, k & \text{sinon et } 5.7g \leq k \text{ et } k+g \in 3\mathbb{Z} \\ T(g, (k-g)/3), (k+2g)/3, (2k-2g)/3, & \text{sinon et } 4.9g \leq k \text{ et } k-g \in 3\mathbb{Z} \\ k-g, k & \\ T(g, (k+g)/2), k-g, k & \text{sinon et } 4.9g \leq k \text{ et } k+g \in 2\mathbb{Z} \\ T(g, k/3), g+k/3, 2k/3, k-g, k & \text{sinon et } 6.8g \leq k \text{ et } k \in 3\mathbb{Z} \\ T(g/2, k-g/2), g, k & \text{si } 9g \leq k \text{ et } g \in 6\mathbb{Z} \\ T(g, k-g), k) & \text{sinon} \end{cases} \tag{4.7}$$

Pour comprendre d'où viennent les différentes valeurs seuils (3.92, 5.7 etc), il faut tout d'abord garder à l'esprit que les étapes correspondantes sont là pour les longues séries de petits pas. Si la condition $t \times g \leq k$ est vérifiée durant l'algorithme d'Euclide additif, alors celui va entamer une série de t petits pas. Les différentes valeurs seuils sont donc pour exprimer quelle quantité de petits pas l'on accepte d'effectuer à la suite. Les valeurs données sont ici purement heuristiques et sujettes à discutions. Elles ont été choisies dans le but de minimiser la longueur, sans tenir compte du coût des différentes opérations mises en jeu.

En reprenant l'exemple $k = 113$ et $g = 99$ on obtient la chaîne :

$$T(99, 113) = 1, 2, 3, 4, 5, 9, 14, 19, 33, 47, 66, 99, 113 \tag{4.8}$$

La longueur a été réduite de nouveau et nous obtenons une chaîne de longueur 13.

Le tableau 4.4 donne quelques résultats empiriques sur les longueurs moyennes des chaîne vues précédemment pour des scalaires de différentes tailles. Les chaînes sont toutes calculées en prenant g l'entier le plus proche de k/ϕ.

taille de k en bits	Euclide(g,k)	$S(g,k)$	$B(g,k)$	$T(g,k)$
16	30	25	24	24
32	92	56	53	51
160	1100	363	274	260
256	1500	630	440	415

Tab. 4.4 – Comparaisons de différentes chaînes différentielles

4.5 Chaînes quasi-différentielles

Dans la section précédente nous avons vu qu'il était possible d'obtenir des chaînes différentielles plus courtes que les chaînes euclidiennes. Maintenant si nous voulons en tirer partie dans le but de proposer un algorithme de multiplication de point efficace il nous faut veiller à optimiser l'utilisation des différentes formules de composition de points. Le but affiché est donc de réaliser le même travail que Cohen et. al. [CMO98] dans le cadre de la multiplication de point utilisant les méthodes de fenêtres glissantes, c'est à dire de réutiliser le plus souvent possible les résultats des calculs intermédiaires d'une étape à l'autre de l'algorithme.

Toutefois, dans certaines circonstances, la structure de chaîne différentielle n'est pas indispensable au bon déroulement de l'algorithme. En effet contrairement aux formules de type Montgomery, où les coordonnées de la différence interviennent dans le calcul, les formules d'additions 3.2 permettent de calculer les trois coordonnées d'un point et autorisent ainsi l'addition de deux points dont la différence ne serait pas connue.

Illustrons ceci par un l'exemple suivant. Considérons la chaîne

$$T(119,120) = 1,2,3,4,7,8,14,15,29,30,59,60,119,120. \tag{4.9}$$

L'algorithme de multiplication se déroule comme suit à partir de 14 et 15 :
- 14+15=29
- 2×15=30
- 29+30=59
- 2×30=60
- 59+60=119
- 2×60=120

Si le but est de calculer 119 et 120, il est facile de voir que le schéma peut être simplifié de la manière suivante :
- 2×15=30
- 2×30=60
- 2×60=120
- 120-1=119

Ce qui correspond à la chaîne d'addition suivante :

$$1, 2, 3, 4, 7, 8, 15, 30, 60, 120, 119. \tag{4.10}$$

Il est même possible de diminuer la longueur de la chaîne en autorisant des successions de triplements et de doublements. En remarquant que 15=3×5, la chaîne 4.10 peut être modifié en la chaîne :

$$1, 2, 3, 4, 5, 15, 30, 60, 120, 119. \tag{4.11}$$

Nous obtenons une chaîne plus courte que la chaîne de départ 4.9, mais celle-là n'est plus une chaîne différentielle. Dans notre exemple, la dernière étape pose problème ; la différence 120-(-1)=121 n'y apparaît pas. Il est, du coup, impossible de parcourir une telle chaîne à l'aide des formules de Montgomery. L'avantage des formules 3.2 que nous proposons est qu'elle permettent de parcourir la chaîne 4.11.

Nous formalisons maintenant la règle de simplification nous ayant permis d'obtenir la chaîne 4.11. Celle-ci est décrite dans l'algorithme 18. Nous y ferons appel dès lors qu'une division par 2, ou par 3, interviendra dans notre futur l'algorithme de construction de chaînes, basé sur celui de Tsuruoka. Cela dans le but de remplacer les coûteuses séquences de doublements (ou triplements) et mises à jour par une seule série de doublements et/ou triplements.

Étant donnés deux entiers g et k, l'algorithme 18 retourne une suite d'entiers $s = (s_1, \ldots s_{i-1}, k)$ telle que, pour tout $j < i$, $s_j = k/(2^{t_1}3^{t_2})$. De telle sorte qu'il suffit ensuite de calculer une chaîne calculant g et s_1 pour obtenir une chaîne calculant g et k. Notons enfin que la condition $4g < k$ (resp. $6g < k$) est là pour exprimer le fait que l'on préfère effectuer des petits ou des grands pas lorsque le rapport entre g et k n'est pas trop petit ; il y a plus de chance de tomber sur une série de grand pas.

Algorithme 18 : Simplification de chaîne

```
Données : g et k deux éléments d'une chaîne en cours calcul
Résultat : une suite s = (s_1, ..., s_{i-1}, k)
début
    s = (k)
    tant que 4g < k et k ≡ 0 mod 2 faire
        k ← k/2
        rajoute k à la chaîne s
    fin
    tant que 6g < k et k ≡ 0 mod 3 faire
        k ← k/3
        rajoute k à la chaîne s
    fin
    retourner s
fin
```

Multiplication de point par les chaînes d'additions quasi-différentielles

Nous venons de voir que nous pouvions modifier les chaînes de Tsuruoka, faisant ainsi apparaître des séries de doublements et de triplement. Si la taille des chaînes a bien été réduite, encore faut-il, pour que cela soit efficace sur les courbes, que ces deux opérations soient elles aussi efficaces. Plus précisément, étant donnés P_1 et P_2 deux points d'un courbes elliptique ayant la même coordonnée z, nous cherchons à optimiser l'opération consistant à appliquer une série de doublements et/ou de triplements à P_1, puis à mettre à jour les coordonnées de P_2 de telle sorte que ce dernier ai la même coordonnée z que le multiple de P_1 précédemment calculé.

Pour cela nous étudions les différentes opérations de bases intervenant dans ce calcul.

a) $P_1, P_2 \rightarrow P_1 + P_2$ (5M+2C) (voir page 40)

Nous obtenons les points $P_1 + P_2$ et P_1 ou P_2 avec la même coordonnée z. Il est possible de mettre à jour le troisième point pour seulement 1M.

b) $P_1 \rightarrow [2]P_1$ (4M+6C)(voir page 6)

Considérons que nous ayons deux points P_1 et P_2 vérifiant $Z_1 = Z_2$. Après doublement de P_1, nous obtenons deux points P_1 et $[2]P_1$ dont la coordonnée z est égale à $2Y_1Z_1$. Au cours du calcul la quantité $4Y_1^2$ est calculée ce qui permet de mettre à jour les coordonnées de P_2 pour un coût de calcul de 3M (en calculant $X_2(4Y_1^2)$ et $Y_2(8Y_1^3)$). Plus généralement, si i doublements successifs doivent être effectués alors chaque doublement suivant ne coûte plus que 4M+4C (coordonnées jacobiennes modifiées page 7). Dans ce cas, la mise à jour du point P_2 devient un peu plus complexe. Le problème vient du fait qu'après i doublements, la coordonnée z du point obtenu sera égale à MZ_1 pour un certain M. Pour mettre à jour P_2 il faut donc calculer ce M (avec une inversion) puis faire la mise à jour. Afin d'éviter cette coûteuse inversion nous proposons d'effectuer les calculs comme suit :

1. lors du premier doublement ne nous calculons pas la coordonnée z de $[2]P_1$ égale à $2Y_1Z_1$. Nous ne calculons que les coordonnées x et y du point $[2]P_1 = (X_1 4Y_1^2, 8Y_1^4, 2Y_1Z_1)$ $= (X_{[2]P_1}, Y_{[2]P_1}, Z_{[2]P_1})$. Nous économisons ainsi une multiplication par rapport aux formules de doublement données page 6 ; le coût du premier doublement est donc de 3M+6C,
2. lors de la précédente étape, nous avons calculé la quantité aZ_1^4 (voir page 7). Cela veut dire qu'il nous est à nouveau possible d'effectuer le doublement du point $[2]P_1$ sans en connaître la coordonnée z ; le coût est 3M+4C (doublement en coordonnées jacobiennes modifiées moins 1 multiplication). Nous obtenons ainsi les coordonnées x et y du point $[4]P_1 = (X_{[2]P_1} 4Y_{[2]P_1}^2, 8Y_{[2]P_1}^4, 2Y_{[2]P_1}Z_{[2]P_1})$ ainsi que la quantité $aZ_{[2]P_1}^4$ permettant de réitérer l'opération. De plus, nous avons $2Y_{[2]P_1}Z_{[2]P_1} = 2Y_{[2]P_1}(2Y_1Z_1) = MZ_1$. Donc pour calculer $M = (2Y_{[2]P_1})(2Y_1)$ nous

n'avons besoin que d'une multiplication ; le coût total de l'opération est finalement de 4M+4C,

3. au bout de i doublements on obtient bien les coordonnées x et y du point $[2^i]P_1$ainsi qu'une quantité $M = Z_{[2^i]P_1}/Z$. On peut donc calculer $Z_{[2^i]P_1} = MZ$ (1M).

Nous pouvons maintenant mettre à jour les coordonnées de P_2 pour 3M+C en calculant M^2, M^3, X_2M^2 et Y_2M^3. Le coût total de l'opération étant de (3M+6C) + $(i-1)$(4M+4C)+ (1M) + (3M+1C) = $(4i+3)$M + $(4i+3)$C.

c)$P_1 \to [3]P_1$ (10M+6C) (voir page 24)

Nous obtenons deux points P_1 et $[3]P_1$ dont la coordonnée z est égale à ZE. Au cours du calcul les quantité E^2 et E^3 sont calculées ce qui permet de mettre à jour les coordonnées de P_2 pour un coût de calcul de 2M (en calculant X_2E^2 et Y_2E^3). Dans le cas où i triplements successifs doivent être effectués, il est possible d'économiser le calcul d'un carré par triplement intermédiaire. Le principe est exactement le même que dans le cas du doublement. Au bout de i triplement la coordonnée z du point obtenu est égale à MZ pour un certain M ; il suffit donc de reprendre le même schéma de calcul vu juste au dessus pour effectuer i triplements successifs pour un coût de (9M+6C)+$(i-1)$(10M+5C)+(1M)+(3M+1C)= $(10i+3)$M+$(5i+2)$C.

De plus, ces formules faisant intervenir la même quantité aZ_1^4 lors des calculs, il est possible d'enchaîner une série de doublements avec une série de triplements. Ceci permettant d'effectuer une série de i doublements, de j triplements et une mise à jour du point P_2 pour un coût de $(10j+4i+3)$M+$(5j+4i+4)$C.

Exemple 18 Dans le tableau 4.5, nous comparons maintenant le calcul de $[119]P$ et $[120]P$, en utilisant d'un part la chaîne 4.9 et d'autre part la chaîne 4.11. Nous pouvons voir que l'utilisation de la chaîne 4.11 permet de diminuer grandement le coût d'une telle opération. Remarquons de plus qu'il est plus efficace de calculer $[119]P$ et $[120]P$ en utilisant la chaîne 4.11 que de calculer seulement $120P$ en utilisant l'algorithme de doublement et addition. En effet, si l'on utilise la représentation NAF, nous avons alors $120 = 1000\bar{1}000$. L'algorithme 4 requiert alors 7 doublements et deux additions, soit un coût total de 42M+38C en coordonnées mixtes. En considérant que C=0.8M, nous obtenons ainsi un coût de 72.4M contre 71.8M avec notre méthode. De plus le coût de celle-ci diminue jusqu'à 65.2M si nous ne calculons pas $[119]P$ (la dernière addition disparaît).

4.6 Aménagement des chaînes de Tsuruoka

Nous allons maintenant voir comment aménager les chaînes de Tsuruoka afin d'obtenir des chaînes quasi-différentielles, parfaitement adaptées à la multiplication de points sur les courbes elliptiques.

Chaîne 4.9		Chaîne 4.11	
P		P	
$[2]P$	2M+4C	$[2]P$	2M+4C
$[3]P$	5M+2C	$[3]P$	5M+2C
$[4]P$	5M+2C	$[4]P$	5M+2C
$[7]P$	5M+2C	$[5]P$	5M+2C
$[8]P$ et màj $[7]P$	7M+7C	$[15]P$	9M+6C
$[15]P$	5M+2C	$[30]P$	4M+4C
$[14]P$ et màj $[15]P$	7M+7C	$[60]P$	4M+4C
$[29]P$	5M+2C	$[120]P$ et màj de P	8M+5C
$[30]P$ et màj $[29]P$	7M+7C	$[119]P$	5M+2C
$[59]P$	5M+2C		
$[60]P$ et màj $[59]P$	7M+7C		
$[119]P$	5M+2C		
$[120]P$ et màj $[119]P$	7M+7C		
Total	72M+53C		47M+31C

Tab. 4.5 – Comparaisons du coût de calcul de $[119]P$ et $[120]P$ avec les chaînes de Tsuruoka et les chaînes de Tsuruoka modifiées

Pour cela nous commençons par définir récursivement les chaînes de Tsuruoka modifiées récursivement de la manière suivante :

$$T_{\mathcal{M}}(g,k) = \begin{cases} 0, k & \text{si } g = 0 \\ T_{\mathcal{M}}(k-g,k) & \text{si } k/2 < g \\ T_{\mathcal{M}}(g,k/2), k-g, k & \text{si } 2g \leq k \leq 2.09g \text{ et } k \in 2\mathbb{Z} \\ T_{\mathcal{M}}(g,k/2), k-g, k & \text{si } 4g \leq k \text{ et } k \in 2\mathbb{Z} \\ T_{\mathcal{M}}(g,(k-2g)/3), k-2g, k-g, k & \text{sinon et } 8g \leq k+g \text{ et } k+g \in 3\mathbb{Z} \\ T_{\mathcal{M}}(g,(k-g)/3), (k+2g)/3, & \text{sinon et } 7g \leq k \text{ et } k-g \in 3\mathbb{Z} \\ (2k-2g)/3, k-g, k & \\ T_{\mathcal{M}}(g,(k-g)/2), k-g, k & \text{sinon et } 5g \leq k \text{ et } k-g \in 2\mathbb{Z} \\ T_{\mathcal{M}}(g,k/3), k/3, k-g, k & \text{sinon et } 6.8g \leq k \text{ et } k \in 3\mathbb{Z} \\ T_{\mathcal{M}}(g/2,k-g/2), g, k & \text{si } 9g \leq k \text{ et } g \in 6\mathbb{Z} \\ T_{\mathcal{M}}(g,k-g), k) & \text{sinon} \end{cases} \tag{4.12}$$

Remarquons que les chaînes de Tsuruoka modifiées restent très proches, dans leur définition même, des chaînes de Tsuruoka. Les conditions sont légèrement différentes afin de prendre en compte le coût des opérations sur les courbes elliptiques. A l'origine ces conditions ont été déterminées, de manière heuristique, afin de minimiser la longueur des chaînes obtenues. Ici les conditions sont déterminées, toujours de manière heuristique, dans le but de minimiser le coût de calcul. La démarche a consisté à partir des conditions déjà existentes, puis à ajuster les valeurs seuils en fonction du coût moyen de l'algorithme. Les valeurs données dans ce document sont donc sujettes à discutions, mais donne, sans la pratique, de très bons résultats.

Nous allons maintenant voir que ces nouvelles chaînes s'adaptent très bien aux calculs sur les courbes elliptiques, notamment à l'aide de l'algorithme 18. Dans la suite de cette section, nous reprenons les différentes étapes de des chaînes de Tsuruoka modifiées 4.12 et détaillons la procédure à effectuer sur la courbe.

Opération : $T_{\mathcal{M}}(g,k/2), k-g, k$

Condition : $(2g \leq k \leq 2.09g \text{ et } k \in 2\mathbb{Z})$ ou $(4g \leq k \text{ et } k \in 2\mathbb{Z})$

La chaîne de calcul est $[g]P, [k/2]P, [k]P, [k-g]P$. Puisque une division par deux est effectuée nous faisons appel à l'algorithme ??.

Si la condition $4g \leq k$ et $k \in 2\mathbb{Z}$ est toujours remplie pour g et $k/2$ alors on rajoute $[k/4]P$ dans la chaîne afin d'obtenir la chaîne

$$[g]P, [k/4]P, [k/2]P, [k]P, [k-g]P$$

et on recommence avec g et $k/4$ etc. On obtient au final une chaîne

$$[g]P, [k/2^i]P, \ldots [k/2]P, [k]P, [k-g]P$$

où i est le plus grand entier non nul tel que $4g \leq k/2^{i-1}$ et $k/2^{i-1} \in 2\mathbb{Z}$.

Ensuite si la condition $6g \leq k$ et $k \in 3\mathbb{Z}$ est remplie pour g et $k/2^i$ alors on rajoute $[k/2^i 3]P$ et ainsi de suite jusqu'à obtenir

$$[g]P, [k/2^i 3^j]P, \ldots, [k/2^i 3]P, [k/2^i]P, \ldots$$

$$\ldots, [k/2]P, [k]P, [k-g]P$$

où j est le plus grand entier non nul tel que $6g \leq k/2^{j-1}$ et $k/2^i 3^{j-1} \in 3\mathbb{Z}$.

Le calcul s'effectue alors de la manière suivante :

- $[g]P$ et $[k/2^i 3^j]P$ ont la même coordonnée z
- on effectue j triplements et i doublements successifs grâce aux méthodes précédemment décrite en mettant $[g]P$ à jour $(10j{+}4i{+}3)$M+$(5j{+}4i{+}4)$C
- on effectue $[k]P, [g]P \to [k-g]P$ (5M+2C)
- si besoin on met à jour le troisième point (1M)

Coût total : $(10j{+}4i{+}8)$M+$(5j{+}4i{+}6)$C (+1M).

Opération : $T_{\mathcal{M}}(g, (k-2g)/3), k-2g, k-g, k$

Condition : $8g \leq k$ et $k+g \in 3\mathbb{Z}$.

La chaîne de calcul est $[g]P, [(k-2g)/3]P, [k-2g]P, [k-g]P, [k]P$. On fait ensuite appel à l'algorithme ?? pour obtenir la chaîne :

$$[g]P, [(k-2g)/2^i 3^{j+1}]P, \ldots, [(k-2g)/2^i 3^2]P, [(k-2g)/2^i 3]P, \ldots$$

$$\ldots, [(k-2g)/2 \times 3]P, [(k-2g)/3]P, [k-2g]P, [k-g]P, [k]P$$

- $[g]P$ et $[(k-2g)/2^i 3^{j+1}]P$ ont la même coordonnée z
- on effectue $j+1$ triplements et i doublements successifs en mettant $[g]P$ à jour $(10j{+}4i{+}13)$M + $(5j{+}4i{+}9)$C
- on effectue $[k-2g]P, [g]P \to [k-g]P$ (5M+2C)
- on effectue $[k-g]P, [g]P \to [k]P$ (5M+2C)

Coût total : $(10j{+}4i{+}23)$M+$(5j{+}4i{+}13)$C

Opération : $T_{\mathcal{M}}(g, (k-g)/3), (k+2g)/3, (2k-2g)/3, k-g, k$

Condition : $7g \leq k$ et $k-g \in 3\mathbb{Z}$

La chaîne de calcul, avant de faire appel à l'algorithme ??, est $[g]P, [(k-g)/3]P, [k-g]P, [k]P$ et le calcul s'effectue de la manière suivante :

- $[g]P$ et $[(k-g)/2^i 3^{j+1}]P$ ont la même coordonnée z
- on effectue on effectue i doublements et $j+1$ triplements en mettant à jour $[g]P$ $(10j{+}4i{+}13)$M + $(5j{+}4i{+}9)$C
- on effectue $[k-g]P, [g]P \to [k]P$ (5M+2C)

Coût total : $(10j{+}4i{+}18)$M+$(5j{+}4i{+}11)$C.

Opération : $T_{\mathcal{M}}(g, (k-g)/2), k-g, k$

Condition : $5g \leq k$ et $k - g \in 2\mathbb{Z}$

La chaîne de calcul, avant de faire appel à l'algorithme ??, est $[g]P, [(k-g)/2]P, [k-g]P, [k]P$ et le calcul s'effectue de la manière suivante :

- $[g]P$ et $[(k-g)/2^i 3^j]P$ ont la même coordonnée z
- on effectue on effectue i doublements et j triplements en mettant à jour $[g]P$ $(10j+4i+3)$M + $(5j+4i+4)$C
- on effectue $[k-g]P, [g]P \rightarrow [k]P$ (5M+2C)

Coût total : $(10j+4i+8)$M+$(5j+4i+6)$C.

Opération : $T_{\mathcal{M}}(g, k/3), k/3, k-g, k$

Condition : $6.8g \leq k$ et $k \in 3\mathbb{Z}$.

La chaîne de calcul, avant de faire appel à l'algorithme ??, est $[g]P, [k/3]P, [k]P, [k-g]P$ et le calcul s'effectue de la manière suivante :

- $[g]P$ et $[k/2^i 3^j]P$ ont la même coordonnée z
- on effectue on effectue i doublements et j triplements en mettant à jour $[g]P$ $(10j+4i+3)$M + $(5j+4i+4)$C
- on effectue $[k]P, [g]P \rightarrow [k-g]P$ (5M+2C)
- si besoin on met à jour le troisième point (1M)

Coût total : $(10j+4i+8)$M+$(5j+4i+6)$C (+1M).

Opération : $T(g/2, k-g/2), g, k$

Condition : $9g \leq k$ et $g \in 6\mathbb{Z}$

La chaîne de calcul est $[g/2]P, [g]P, [k-g/2]P, [k]P$ et le calcul s'effectue de la manière suivante :

- $[g/2]P$ et $[(k-g/2)]P$ ont la même coordonnée z
- on effectue $[(k-g/2)]P, [g/2]P \rightarrow [k]P$ (5M+2C)
- on effectue $[g/2]P \rightarrow [g]P$ (4M+6C)
- on met à jour $[k]P$ (3M)

Coût total : 12M+8C.

Reprenons notre exemple avec $k = 113$ et $g = 99$. nous avions la chaîne :

$$T(99, 113) = 1, 2, 3, 4, 5, 9, 14, 19, 33, 47, 66, 99, 113,$$

qui devient, en utilisant tous les aménagements précédemment proposés :

$$1, 2, 4, 5, 9, 14, 19, 33, 99, 113.$$

Notons que si le calcul de 47 a vraiment été éliminé de la chaîne celui de 66 est en pratique contenu dans l'opération $33 \rightarrow 99$.

Nous obtenons finalement une méthode de multiplication de points, sans pré-calcul, dont le coût est, expérimentalement, d'environ 1780M pour des scalaires de 160 bits. Cela donne au final un algorithme équivalent à la meilleure méthode sans pré-calcul (le NAF) dont le coût est lui aussi d'environ 1780M (voir tableau 2.2).

4.7 Perspectives

Jusqu'à présent nous avons proposé une approche consistant à utiliser des chaînes différentielles déjà connues et à les simplifier en tenant compte de la souplesse que nous autorise l'utilisation des formules d'addition de points 3.2. Cela a donné en pratique ce que nous appelons des chaînes d'additions quasi-différentielles. Cette approche permet d'obtenir une méthode de multiplication de point aussi efficace que les méthodes classiques, à exigence de mémoire équivalente. Néanmoins les performances obtenues restent encore assez inférieures à celles des méthodes les plus rapides. C'est pourquoi nous allons décrire deux pistes de recherche pouvant conduire à une amélioration sensible de l'efficacité de notre approche initiale.

Vers de vraies chaînes quasi-différentielles

Les deux principales caractéristiques des formules d'additions de points 3.2 sont qu'elles permettent, 1) une multiplication de point très efficace dans le cadre des suites de Fibonacci, 2) une grande flexibilité dans le cadre plus générale des chaînes d'additions différentielles. A partir de là, une approche possible consiste non plus à modifier des chaînes différentielles en chaînes quasi-différentielles mais à fabriquer directement des chaînes quasi-différentielles en essayant de maximiser le nombre de grand pas. Pour illustrer ceci considérons l'entier $k = 1240$ et $g = 766$. L'algorithme d'Euclide additif donne dans ce cas la suite :

$$1, 2, 4, 6, 10, 14, 18, 22, 26, 30, 34, 38, 72, 110, 182, 292, 474, 766, 1240. \tag{4.13}$$

A partir de 34, se produit une série de petits pas qui allonge considérablement la chaîne. Plutôt que d'utiliser les règles de calculs vues dans les sections précédentes, une approche consiste à chercher une chaîne calculant directement 34. Pour cela nous calculons l'entier $\lceil \frac{34}{\phi} \rfloor = 21$ et appliquons l'algorithme d'Euclide additif. Ainsi nous assurons ainsi la présence d'un certain nombre de grands pas à partir de 34. Nous obtenons alors la suite :

$$1, 2, 3, 5, 8, 13, 21, 34, 38, 72, 110, 182, 292, 474, 766, 1240. \tag{4.14}$$

Cette nouvelle suite n'est presque constituée que de grands pas. Cependant, ce n'est plus une chaîne d'additions : il n'existe aucun couple d'entiers dont la somme vaut 38 (par la suite nous appellerons ceci une singularité de chaîne). Néanmoins, ce n'est pas vraiment un problème dans la mesure ou la différence entre 34 et 38 est petite et peut être facilement calculée à partir d'éléments de la chaîne. Pour que la chaîne 4.14 redevienne une chaîne il suffit alors d'y rajouter l'entier 4. Nous avons alors la chaîne :

$$1, 2, 3, 4, 5, 8, 13, 21, 34, 38, 72, 110, 182, 292, 474, 766, 1240. \tag{4.15}$$

Nous obtenons une chaîne d'additions ne comportant presque que des grands pas. Cette approche fonctionne très bien sur de petits entiers. Une étude exhaustive nous a

permis voir que, par exemple, 1240 est le premier pour lequel cette approche ne conduit pas à une chaîne d'additions. Appliqué à des nombres inférieurs à 2^{16}, cela conduit à des chaînes ayant au plus une seule singularité ; ces dernières pouvant être supprimées par l'ajout d'au plus 4 entiers dans la chaîne. Cependant, ce procédé montre ses limites dès lors que l'on commence à considérer des entiers relativement grands. Si le nombre de singularités de chaîne n'augmente pas de façon dramatique, combler ces dernières devient pour le moins coûteux. Par exemple, une recherche sur plusieurs millions d'entiers tirés au hasard tend à montrer que le nombre de singularités apparaissant avec des entiers de 32 bits est au plus de 3 (1.8 en moyenne) mais le nombre maximum d'entiers à ajouter à la chaîne initiale passe lui à 15 (4.1 en moyenne). Lorsque l'on passe à des tailles cryptographiques le sur-coût devient carrément rédhibitoire. Là encore, des expérimentations sur plusieurs millions d'entiers montrent que pour des entiers de 160 bits le nombre de singularité augmente peu, 3.7 en moyenne, mais le nombre d'entiers à ajouter passe lui à 182 en moyenne. Sachant que les chaînes ont une longueur de 240 avant correction, on voit à quel point cette correction devient coûteuse.

Une première façon de limiter ce genre de problèmes consiste à ne pas choisir g comme étant l'entier le plus proche de $\frac{k}{\phi}$ mais comme un entier proche de $\frac{k}{\phi}$ mais conduisant à une longue chaîne de petits pas. En effet une série de t petits pas signifie qu'il se trouve deux entier d et e tels que la séquence suivante apparaît dans la chaîne :

$$d, e - td, e - (t-1)d, \ldots, e - d, e.$$

Si t est grand cela veut donc dire que $d = e - (e - d)$ est petit et il est alors plus probable que celui-ci apparaisse dans le reste de la chaîne. Tout du moins, combler cette singularité devrait être, pour un d petit, moins coûteux que pour un d arbitrairement grand.

Une deuxième façon consiste à autoriser les petites séquences de petits pas. C'est le même argument qui prime là aussi. Une petite série de petits pas signifie que le t est petit et que donc la différence d est grande. Il y a donc moins de chance que celle-ci apparaisse dans la chaîne et l'y insérer peut alors s'avérer très coûteux.

Méthode combinée

La grande majorité des méthodes de multiplication de points basées sur des chaînes d'additions différentielles que nous avons présentées avaient la propriété d'avoir une faible demande en mémoire. En effet dans la majorité des cas, aucun pré-calcul n'était nécessaire. Nous pouvons ainsi proposer une méthode de multiplication de points aussi performante que les méthodes les plus efficaces, à exigence de mémoire équivalente. Cependant les méthodes utilisant des pré-calculs (type 4-NAF) restent, elles, beaucoup plus efficaces. Le but de cette sous section est de proposer une nouvelles approche réunissant à la fois chaîne d'additions différentielles et méthodes binaires classiques.

Lorsque l'on utilise une méthode binaire à fenêtres pour effectuer une multiplication de points on commence toujours par pré-calculer un certain nombre de produits par

un scalaire qui seront réutilisés lors des calculs suivants. Par exemple dans le cas du 4-NAF, l'on commence par calculer les points P, $[3]P$, $[5]P$ et $[7]P$ avant d'entamer la multiplication de points proprement dite, selon un schéma classique de doublements et d'additions. Un des désavantage de cette méthode est que, si l'on désire diminuer le nombre d'additions, il faut doubler le nombre de points pré-calculés. Cependant, le nombre de doublement ne diminue pas, ou très peu. Ainsi, pour le 5-NAF, il faut pré-calculer, en plus des points précédents, les points $[9]P$, $[11]P$, $[13]P$ et $[15]P$; le nombre de doublements à effectuer n'est, par contre, diminué que d'au plus une unité.

L'approche que nous proposons ici consiste à décomposer la multiplication de points en deux étapes. Une première concernant seulement les premier bits (par exemple les 16 ou 32 premiers), utilisant une des méthodes basées sur les formules d'additions de points 3.2 ; puis une seconde, plus classique, utilisant une méthode à fenêtres glissantes. L'idée est de garder en mémoire les points calculés lors de la première phase et de s'en servir lors de la seconde. L'intérêt de cette approche est que, contrairement à ce qui se passe dans le cas d'une exponentiation binaire, une exponentiation à base de chaîne différentielles fait apparaître des entiers dont les représentations binaires sont vraiment différentes.

Pour illustrer ceci prenons comme exemple l'entier $1031 = (10000000111)_2$. Alors que l'algorithme de doublements et d'addition ne fait apparaître que des entiers correspondant aux premiers bits de 1031 (l'algorithme calcule successivement 1, $2 = (10)_2$, $4 = (100)_2$, $8 = (1000)_2$, $16 = (10000)_2$ ...) l'algorithme d'Euclide additif appliqué à 1031 et 637 ($\lceil 1031/\phi \rfloor$) conduit au calcul de la chaîne :

$$1, 2, 3, 5, 7, 12, 19, 26, 33, 59, 92, 151, 243, 394, 637, 1031$$

Considérons les écritures en base 2 de ces nombres, nous obtenons alors :

1	$= (1)_2$	7	$= (111)_2$	33	$= (100001)_2$	243	$= (11110011)_2$
2	$= (10)_2$	12	$= (1100)_2$	59	$= (111011)_2$	394	$= (110001010)_2$
3	$= (11)_2$	19	$= (10011)_2$	92	$= (1011100)_2$	637	$= (1001111101)_2$
5	$= (101)_2$	26	$= (11010)$	151	$= (10010111)_2$	1031	$= (10000000111)_2$

Nous pouvons voir que l'on retrouve tous les nombres impairs de 1 à 15 en tant que préfixe d'un des entiers de la chaîne. Cela signifie que, d'une certaine manière, la phase de pré-calcul est intégrée à la multiplication de point elle même.

Ainsi, à partir du moment où les bits de poids forts du scalaire k sont 10000000111, il est possible d'effectuer une multiplication de points de type fenêtres glissantes à l'aide des points :

1	$= (\underline{1})_2$	5	$= (\underline{101})_2$	19	$= (\underline{10011})_2$	92	$= (\underline{1011}100)_2$
3	$= (\underline{11})_2$	7	$= (\underline{111})_2$	26	$= (\underline{11010})$	243	$= (\underline{1111}0011)_2$

Prenons par exemple $k = 67615219 = (10000000111101110011111110011)_2$. La multiplication de point s'effectue de la façon suivante :

– $(\underline{10000000111}1011100111110011)_2$

Le points $[1031]P$ est calculé grâce à la chaîne euclidienne précédente, au cours du calcul nous mettons en mémoire les points P, $[3]P, \ldots, [92]P$, $[243]P$. Nous pouvons maintenant calculer $[(100000001110000000000000)_2]P$, ce qui signifie qu'il nous reste à calculer

$$\begin{array}{rllllllll} & (100 & 0000 & 0111 & 1011 & 1001 & 1111 & 0011)_2 \\ - & (100 & 0000 & 0111 & 0000 & 0000 & 0000 & 0000)_2 \\ = & (000 & 0000 & 0000 & 1011 & 1001 & 1111 & 0011)_2 \end{array}$$

Nous parcourons les bits du scalaires avec une fenêtre de taille 4,

- $(000000000001\underline{011}100111110011)_2$
  Nous cherchons parmi les points mémorisés celui dont les premiers bits correspondent aux bits du scalaires en cours de traitement, ici $[92]P$. Nous multiplions $[1031]P$ par 2^w où w est la taille de 92, c'est à dire par 2^7. Nous ajoutons $[92]P$ à $[1031 \times 2^7]P$ et nous avons calculé le point $[132060]P$ avec $132060{=}(100000001111011100)_2$, nous devons donc calculer

$$\begin{array}{rllllllll} & (100 & 0000 & 0111 & 1011 & 1001 & 1111 & 0011)_2 \\ - & (100 & 0000 & 0111 & 1011 & 1000 & 0000 & 0000)_2 \\ = & (000 & 0000 & 0000 & 0000 & 0001 & 1111 & 0011)_2 \end{array}$$

- $(00000000000000000\underline{111}10011)_2$
  Le point dont les bits de poids forts correspondent au bits en cours de traitement est $[243]P$ ($243 = (\underline{1111}0011)_2$). En recommençant le même procédé nous pouvons calculer le point $[(10000000111101110011110011)_2]P$, ce qui signifie qu'il reste à calculer

$$\begin{array}{rllllllll} & (100 & 0000 & 0111 & 1011 & 1001 & 1111 & 0011)_2 \\ - & (100 & 0000 & 0111 & 1011 & 1001 & 1110 & 0110)_2 \\ = & (000 & 0000 & 0000 & 0000 & 0000 & 0000 & 1101)_2 \end{array}$$

- $(00000000000000000000000\underline{1101})_2$
  Le point dont les bits de poids forts correspondent au bits en cours de traitement est, cette fois, $[26]P$ ($26 = (\underline{11010})_2$). Cependant la longueur du nombre en question est trop grande ; nous essayons donc sur une taille de fenêtre inférieure. Dans le cas présent nous finissons la multiplication de points à l'aide des points $[3]P$ et P, successivement.

Nous venons de la voir il possible d'effectuer une multiplication de points ayant à la fois l'avantage des méthodes à base de fenêtre glissante tout en supprimant la coûteuse phase de pré-calcul. De plus si la mémoire n'est pas une contrainte, rien n'empêche de mémoriser l'ensemble des points calculés lors de la première partie.

Le principal défaut de cette méthode est qu'elle nécessite la mise en évidence de

chaînes euclidiennes courtes pour des entiers de petites tailles. Si cela reste relativement simple pour les entiers inférieurs à 2^{10}, à partir de 2^{16}, certaines chaînes peuvent causer un sur-coût dommageable. Autour de 2^{32} trouver une chaîne courte peut demander un certain temps de recherche compromettant grandement l'efficacité générale du processus.

Troisième partie

La représentaion RNS pour les courbes elliptiques

Chapitre 5

Introduction au RNS

L'arithmétique des courbes elliptiques est une arithmétique à plusieurs niveaux ; jusqu'à présent nous nous sommes intéressés d'une part à l'arithmétique des points des courbes, en introduisant de nouvelles formules d'additions, et d'autre part à l'arithmétique ayant trait à la multiplication de points avec l'étude des chaînes d'additions. C'est donc fort logiquement que nous allons nous tourner vers un nouvel aspect de l'arithmétique des courbes elliptiques en considérant cette fois un niveau inférieur, à savoir l'arithmétique sur le corps de base. Le but cette fois n'est pas de proposer une nouvelle arithmétique sur les corps finis plus efficace, de très nombreux travaux dans ce domaine ont déjà était fait [Mon85, CH07], mais d'adapter un système un peu particulier de représentation des entiers, la représentation RNS. Celle-ci possède en effet des propriétés très intéressantes dans le cadre des contre-mesures aux attaques par canaux cachés. Des travaux dans ce sens ont déjà été mené dans le cadre de RSA [BILT04]. Ils ont permis de montrer l'intérêt de ce système de représentation à la fois du point de vu de la robustesse aux attaques mais aussi de l'efficacité générale. Cependant, contrairement à RSA qui ne requiert qu'une succession de multiplications et de carrés modulaires, l'exponentiation sur les courbes elliptiques fait intervenir des formules plus complexes faites de sommes de produits modulaires et parfois (en coordonnées affines) d'inversion.

Cette partie est ainsi composée de quatre chapitres. Le chapitre 5 est une introduction à la représentation RNS et au calcul modulaire via le RNS. Dans le chapitre 6, nous présentons une méthode permettant d'optimiser le choix des bases RNS afin d'améliorer l'efficacité du produit modulaire. Le chapitre 7 permet d'introduire un premier algorithme d'inversion modulaire en RNS. Enfin le dernier chapitre est consacré à la comparaison entre notre approche utilisant le RNS et les méthodes utilisant une représentation plus classique.

5.1 Le théorème des restes chinois

La représentation RNS (Residue Number System) est basé sur le très ancien théorème des restes chinois, permettant de répartir le calcul modulo un grand entier sur plusieurs anneaux de plus petites tailles.

Théorème 8 Soit $\mathcal{B} = (m_1, m_2, \ldots, m_n)$ un ensemble d'entiers premiers deux à deux. Notons $M = \prod_{i=1}^{n} m_i$ alors,

$$\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_n\mathbb{Z} \simeq \mathbb{Z}/M\mathbb{Z}$$

Autrement dit si l'on considère $(a_1, a_2, \ldots, a_n)$ un ensemble de n entiers tels que pour tout i $a_i < m_i$ alors il existe un unique A tel que :

$$\begin{aligned} &0 \leq A < M \text{ et} \\ &a_i = A \bmod m_i = |A|_{m_i} \text{ pour } 1 \leq i \leq n. \end{aligned}$$

De plus, il est possible de passer de le représentation RNS d'un entier à une représentation classique de position grâce à la formule suivante :

$$A = \left| \sum_{i=1}^{n} a_i \left| M_i^{-1} \right|_{m_i} M_i \right|_M = \sum_{i=1}^{n} a_i \left| M_i^{-1} \right|_{m_i} M_i - \alpha M, \tag{5.1}$$

où pour tout i M_i est l'entier M/m_i.

Dans la suite nous appelons l'ensemble $\mathcal{B} = (m_1, m_2, \ldots, m_n)$ une base RNS. De plus pour éviter toute confusion nous utilisons la notation A_{RNS} ou $A_{\mathcal{B}}$ s'agissant de la représentation RNS d'un entier A dans la base $\mathcal{B}$.

Exemple 19 Considérons par exemple l'ensemble $\mathcal{B} = (3, 7, 13, 19)$. Tous les entiers qui le composent sont bien premiers deux à deux ce qui signifie que $\mathcal{B}$ est bien une base RNS. De plus cette base permet de représenter de manière unique tous les entiers entre 0 et $3 \times 7 \times 13 \times 19 = 5187$. Par exemple si l'on prend $A = 147$ et $B = 31$ alors $A_{RNS} = (0, 0, 4, 14)$ et $B_{RNS} = (1, 3, 5, 12)$.

Une deuxième propriété fondamentale qui découle du théorème 8 est que les opérations sur l'anneau $\mathbb{Z}/M\mathbb{Z}$, c'est à dire l'addition et la multiplication modulo M, peuvent être distribuées en opérations modulo les m_i.

Exemple 20

$$\begin{aligned} A_{RNS} + B_{RNS} &= (\ |0+1|_3\ ,\ |0+3|_7\ ,\ |4+5|_{13}\ ,\ |14+12|_{19}\ ) \\ &= (\ 1\ ,\ 3\ ,\ 9\ ,\ 7\ ) \\ &= 178 \end{aligned}$$

$$\begin{aligned} A_{RNS} \times B_{RNS} &= (\ |0\times 1|_3\ ,\ |0\times 3|_7\ ,\ |4\times 5|_{13}\ ,\ |14\times 12|_{19}\ ) \\ &= (\ 0\ ,\ 0\ ,\ 7\ ,\ 16\ ) \\ &= 4557 \end{aligned}$$

Ainsi le théorème des restes chinois permet de transformer une opération modulo un grand entier en n opération modulo de petits entiers. L'avantage d'une telle redistribution est que les opérations modulo les m_i sont toutes indépendantes les unes des autres, autorisant par là même un haut niveau de parallélisme.

5.2 Arithmétique pour les opérateurs de base

La complexité de tout algorithme utilisant la représentation RNS dépend directement de la complexité des opérateurs de bases modulo les m_i. Nous présentons dans cette section les algorithmes d'addition et de multiplication modulaire que nous utilisons comme opérateurs de base. Nous considérons par ailleurs que tous les moduli sont des entiers de même taille. Ainsi, pour tout i, $m_i = 2^k - c_i$ avec $c_i < 2^k$.

Addition

Considérons deux entiers a et b tels que $0 \leq a, b < m_i$ et soit s leur somme. Nous voulons donc réduire s modulo m_i.

Nous avons la relation suivante :

$$a + b = s = s_1 2^k + s_0 = s_1 m_i + c_i s_1 + s_0$$

Réduire cette équation modulo m_i conduit donc à $a + b \equiv c_i s_1 + s_0 \mod m_i$. De plus, puisque $0 \leq a, b < m_i$, nous savons que $a + b < 2m_i - 1$. En d'autres terme $a+b < 2^k + (2^k - 2c_i - 1)$. Nous pouvons déduire alors directement que $s_1 \leq 1$. Seul deux cas de figure sont alors possible :

- $a + b$ est plus petit que m_i et alors aucune réduction n'est nécessaire
- $a + b$ est plus grand que m_i et le résultat est alors $a + b - m_i = a + b + c_i - 2^k$.

Remarquons que le deuxième cas impose que $a + b + c_i \geq 2^k$, ce qui veut dire que si, comme on le fait en pratique, les m_i sont des entiers de la taille d'un mot machine (32 ou 64 bits en général), il suffit de tester un dépassement de capacité pour décider laquelle des deux solutions possibles est la bonne.

L'addition modulo m_i est donné par l'algorithme suivant :

Algorithme 19 : modadd(a, b, m_i)

Data : $0 \leq a, b < m_i$
Result : $s = a + b \bmod p$
$d_0 \leftarrow a + b$;
$d_1 \leftarrow d_0 + c_i$;
if $d_1 \geq 2^k$ then
  $s \leftarrow d_1 \mod 2^k$;
else
  $s \leftarrow d_0$;
end

En clair, l'addition modulaire précédente est équivalente à deux additions classique.

Multiplication

Tout comme pour l'addition nous considérons deux entiers a et b tels que $0 \leq a, b < m_i$. Notons $p = a \times b$ leur produit, le but est donc de réduire p modulo m_i. Pour ce faire, nous décomposons p de la manière suivante : $p = p_1 2^k + p_0 = p_1 m_i + c_i p_1 + p_0$. Ainsi, $a \times b \equiv c_i p_1 + p_0 \mod m_i$. Notons alors $p' = c_i p_1 + p_0$.

La condition $0 \leq a, b < m_i$ implique que $p < m_i^2 - 1 < 2^{2k}$ et $p_1 < 2^k$. Maintenant en prenant $c_i < 2^t$ nous obtenons, après une première réduction, $p' = c_i p_1 + p_0 < 2^k(2^t - 1) + 2^k = 2^{k+t}$.

Une deuxième réduction est nécessaire. Nous décomposons alors à nouveau p' de telle sorte que $p' = p'_1 2^k + p'_0$ et nous obtenons $a \times b \equiv c_i p'_1 + p'_0 \mod m_i$. Notons $p'' = c_i p'_1 + p'_0$. Nous pouvons remarquer que, puisque $p' < 2^{k+t}$, $p'_1 < 2^t$. En prenant en compte le fait que $2t < k - 1$, nous avons $p'' < 2^k + 2^{2t} < 2m_i$. Il reste maintenant à réduire p'' de la même manière que dans l'algorithme d'addition pour obtenir le bon résultat.

En imposant de plus la contrainte $1 < t < (k-1)/2$, nous obtenons alors l'algorithme de multiplication modulaire suivant :

Algorithme 20 : modprod(a, b, m_i)

Data : $0 \leq a, b < m_i$
avec $m_i = 2^k - c_i$ and $c_i < 2^t$ and $1 < t < (k-1)/2$
Result : $r = a \times b \bmod m_i$
$p \leftarrow a \times b$;
$p_1 \leftarrow p \div 2^k$; $p_0 \leftarrow p \bmod 2^k$;
$p' \leftarrow c_i p_1 + p_0$;
$p'_1 \leftarrow p' \div 2^k$; $p'_0 \leftarrow p' \bmod 2^k$;
$p'' \leftarrow c_i p'_1 + p'_0$;
$\rho \leftarrow p'' + c_i$;
if $\rho \geq 2^k$ then
  $r \leftarrow \rho \mod 2^k$;
else
  $r \leftarrow p''$;
end

Le produit modulaire requiert donc avec l'algorithme 20 une multiplication de deux entiers de k bits, une multiplication d'un entier de k bits avec une constante de t bits et trois additions entre entiers de k bits. Les opérations $\div 2^k$ et $\bmod 2^k$ ne représentent que des décalages.

En considérant que les coûts d'une multiplication d'un entier de k bits par un entier de t bits et deux entiers de t sont de l'ordre de, respectivement, la moitié et le quart de celui d'une multiplication de deux entiers de k bits alors nous pouvons évaluer la complexité de notre opérateur de multiplication modulo m_i à $\frac{7}{4}M(k)$, où $M(k)$ est la complexité de l'opérateur de multiplication d'entiers de k bits. De plus si le poids de Hamming de

m_i est petit alors les deux multiplications par c_i peuvent être remplacées par quelques additions, rendant ainsi le coût d'une multiplication modulaire équivalent au coût d'une simple multiplication.

Dans la suite de cette partie nous considérons que chaque m_i s'écrit $2^k - c_i$ avec $c_i < 2^t$ et $1 < t < (k-1)/2$.

5.3 RNS et courbes elliptiques

La représentation RNS est particulièrement bien adaptée à l'arithmétique modulo un entier M, dès lors que ce dernier est un produit de petit facteur. Malheureusement, dans ce mémoire, l'arithmétique à laquelle nous nous intéressons est celle des corps des premiers ($\mathbb{Z}/p\mathbb{Z}$ avec p premier). Si nous voulons utiliser le RNS pour représenter les éléments d'un corps premier il va donc falloir faire quelques efforts d'adaptations concernant les les algorithmes additions et de multiplications modulaire. Cette section reprend en partie des travaux déjà effectués dans ce sens concernant la multiplication modulaire dans le cadre du protocole RSA.

Une approche naïve du problème de la multiplication modulaire en RNS pourrait être de considérer une base RNS $\mathcal{B} = (m_1, \ldots, m_n)$ telle que $\prod_{i=1}^{n} m_i > p^2$. De cette manière si A et B sont deux entiers plus petits que p leur produit en RNS sera exact. Ou autrement dit, nous avons $A_{RNS} \times B_{RNS} = (A \times B)_{RNS}$. A partir de là, il suffirait d'effectuer une division euclidienne du résultat par p pour terminer l'opération. Malheureusement, cette approche est non seulement peu efficace en arithmétique classique, mais s'adapte très mal au RNS. Le principal problème étant que nous ne disposons pas à l'heure actuelle d'algorithme de division efficace.

La solution adoptée jusqu'à présent pour pallier cet inconvénient est d'adapter un algorithme de multiplication modulaire ne faisant pas intervenir de division. Nous allons donc, dans un premier temps, présenter cet algorithme, puis, donner son équivalent en RNS.

L'algorithme de Montgomery

La façon la plus naturelle de réduire un entier A modulo un nombre p est d'effectuer la division euclidienne de A par p ($A = qp + r$ pour un certain $r < p$) est d'en garder le reste.

L'approche de Montgomery [Mon85] consiste à remplacer la coûteuse division euclidienne par une division par une puissance de la base (en pratique cela sera donc une puissance de 2). Pour cela nous construisons un multiple de p dont c'est cette fois la partie basse est égale à celle de A. En d'autres termes si $A = \sum_0^{2n-1} a_i \beta^i = (a_{2n-1} \ldots a_0)_\beta$ et $P = (p_{n-1} \ldots p_0)_\beta$, l'algorithme de Montgomery calcule un entier q tel que $qp = (x_{2n-1} \ldots x_n a_{n-1} \ldots a_0)_\beta$. De telle sorte que $A - qp = ((x_{2n-1} \ldots x_n 0 \ldots 0)_\beta \equiv A \mod p$. En divisant par β^n nous obtenons une valeur inférieure à β^n et équivalente à $A\beta^{-n}$ modulo p.

Algorithme 21 : Montgomery(A, B)

Données : $C = A \times B < p^2 < \beta^{2n}$ et $\beta^{n-1} \leq p < \beta^n$
et une valeur pré-calculée $(-p^{-1} \mod \beta^n)$
Résultat : $C' \equiv C\beta^{-n} \mod p < 2N$
$q \leftarrow C \times (-p^{-1}) \mod \beta^n$
$C' \leftarrow (C + qp)/\beta^n$
retourner C'

L'algorithme de Montgomery en RNS

Maintenant nous allons voir comment adapter l'algorithme de Montgomery à la représentation RNS. Le principe global va être de faire jouer le rôle de β^n à M.

Plus précisément, nous allons considérer deux bases RNS $\mathcal{B} = (m_1, \ldots, m_n)$ et $\mathcal{B}' = (m'_1, \ldots, m'_n)$ telles que $p < M < M'$, $\text{pgcd}(M, M') = 1$ et $\text{pgcd}(M, P) = 1$ (où M et M' sont les produits respectifs des m_i et m'_i). Considérons des entiers $A, B < p$ dont nous connaissons les représentations RNS sur chacune des deux bases. Si nous effectuons le produit $A \times B$ pour chaque représentation nous obtenons alors la représentation RNS de $C = AB \mod MM'$. A et B étant inférieurs à p nous savons que $AB < p^2 < MM'$, ce qui signifie que $AB \mod MM' = AB$. A partir de là, la réduction de Montgomery s'effectue via l'algorithme suivant :

Algorithme 22 : Réduction de Montgomery en RNS

Données : Les représentations de $C = A \times B < p^2 < MM'$ dans les bases $\mathcal{B}$ et $\mathcal{B}'$
et que deux valeurs pré-calculées : $-p^{-1} \mod M$ dans $\mathcal{B}$ et $M^{-1} \mod M'$ dans $\mathcal{B}'$
Résultat : $C' \equiv CM^{-1} \mod p$ en RNS dans les bases $\mathcal{B}$ et $\mathcal{B}'$
début
  $Q \leftarrow C \times (-p^{-1})$ dans $\mathcal{B}$ (autrement dit modulo M)
  Étendre Q vers $\mathcal{B}'$
  $C' \leftarrow (C + Q \times p) \times M^{-1}$ dans $\mathcal{B}'$ ($C + Q \times p$ est un multiple de M)
  Étendre C' vers la base $\mathcal{B}$
fin
retourner C'

Exemple 21

- $\mathcal{B} = (5, 7, 12),\ M = 420,\ \mathcal{B}' = (11, 13, 17),\ M' = 2431$
- $P = 83 = (3, 6, 11)_{\mathcal{B}} = (6, 5, 15)_{\mathcal{B}'}$
- $A = 25 = (0, 4, 1)_{\mathcal{B}} = (3, 12, 8)_{\mathcal{B}'}$
- $B = 47 = (2, 5, 11)_{\mathcal{B}} = (3, 8, 13)_{\mathcal{B}'}$

$\mathcal{B}$	Opération	$\mathcal{B}'$
	M^{-1} dans $\mathcal{B}'$	$M^{-1} = (6, 10, 10)_{\mathcal{B}'}$
$-p^{-1} = (3, 1, 1)_{\mathcal{B}}$	$-p^{-1}$ dans $\mathcal{B}$	
$A \times B = (0, 6, 11)_{\mathcal{B}}$	$C = A \times B$ dans $\mathcal{B}$ et $\mathcal{B}'$	$A \times B = (9, 5, 2)_{\mathcal{B}'}$
$Q = (0, 6, 11)_{\mathcal{B}}$	$Q = C \times (-p^{-1})$ dans $\mathcal{B}$	
	étend Q à $\mathcal{B}'$	$Q = (5, 10, 12)_{\mathcal{B}'}$
$C' = (0, 0, 0)_{\mathcal{B}}$	$C' = (C + Q \times p)$ dans $\mathcal{B}'$	$C' = (6, 3, 12)_{\mathcal{B}'}$
	$C' = C' \times M^{-1}$ dans $\mathcal{B}'$	$C' = (3, 4, 1)_{\mathcal{B}'}$
$(4, 3, 6)_{\mathcal{B}}$	étend C' à $\mathcal{B}$	

Finalement on a $(4, 3, 6)_{\mathcal{B}} = 69 = 25 \times 47 \times 420^{-1} \mod 83$.

La complexité réside principalement dans les extensions de bases. L'étude de cette opération fait l'objet la section suivante.

5.4 Changement de base en RNS

Il y a principalement deux méthodes permettant d'effectuer un changement de base en RNS : une basée sur le théorème des restes chinois et une autre faisant intervenir une représentation auxiliaire appelé représentation MRS (pour Mixed Radix System).

Avec le théorème des restes chinois

Nous avons vu précédemment qu'il était possible de passer d'une représentation RNS d'un entier A à sa représentation classique en utilisant l'équation 5.1.

Effectuer le changement de base de $\mathcal{B}$ vers $\mathcal{B}'$, c'est à dire calculer la représentation RNS de A dans $\mathcal{B}'$ à partir de sa représentation dans $\mathcal{B}$ ce fait grâce à l'équation 5.1 qui se présente sous la forme $j \in \{1, ..., n\}$:

$$A \mod m'_j = \left| \sum_{i=1}^{n} \left| a_1 \left| M_i \right|_{m_i}^{-1} \right|_{m_i} \left| M_i \right|_{m'_j} \right|_{m'_j} - \left| \alpha \left| M \right|_{m'_j} \right|_{m'_j} \tag{5.2}$$

où α est tel que $A + \alpha M = \sum_{i=1}^{n} \left| a_1 \left| M_i \right|_{m_i}^{-1} \right|_{m_i} M_i$ et vérifie $0 \leq \alpha < n$.

Le principal inconvénient de cette approche est due la difficulté de calculer α [SK89] [HP94, PP95].

Avec la représentation MRS

La seconde méthode de changement de base RNS fait intervenir la représentation MRS comme représentation intermédiaire. De la même manière que nous avions définie une base

RNS, nous définissons une base MRS comme un ensemble d'entiers $(m_1, m_2, \ldots, m_n)$. Un entier A plus petit que M sera représenté par n-uple $(\widetilde{a}_1, ..., \widetilde{a}_n)$ $(\widetilde{a}_i < m_i)$ tel que pour tout $i \in \{1, ..., n\}$:

$$A = \widetilde{a}_1 + \widetilde{a}_2 m_1 + \widetilde{a}_3 m_1 m_2 + ... + \widetilde{a}_n m_1 ... m_{n-1} \tag{5.3}$$

Le but est donc de passer d'une représentation RNS à une représentation MRS utilisant dans les deux cas la base $\mathcal{B}$ puis d'effectuer une deuxième conversion du MRS vers une représentation RNS utilisant la base $\mathcal{B}'$.

Notons $m_{i,j}^{-1}$ l'inverse de m_i modulo m_j, c'est à dire qu'on a $m_i.m_{i,j}^{-1} \equiv 1 \mod m_j$. A partir de là, la conversion RNS vers MRS peut se faire de la manière suivante [ST67] :

$$\begin{cases} \widetilde{a}_1 = a_1 \bmod m_1 \\ \widetilde{a}_2 = (a_2 - \widetilde{a}_1) m_{1,2}^{-1} \bmod m_2 \\ \widetilde{a}_3 = ((a_3 - \widetilde{a}_1) m_{1,3}^{-1} - \widetilde{a}_2) m_{2,3}^{-1} \bmod m_3 \\ \vdots \\ \widetilde{a}_n = (.(a_n - \widetilde{a}_1) m_{1,n}^{-1} \cdots - \widetilde{a}_{n-1}) m_{n-1,n}^{-1} \bmod m_n \end{cases} \tag{5.4}$$

Le coût de cette partie est donc de $\frac{n(n-1)}{2}$ soustractions modulaires et d'autant de multiplications par un inverse, sachant que les m_i sont des constantes, il est possible de pré-calculer les $m_{i,j}^{-1}$.

Remarque 8 Une autre approche [Knu81] consiste à factoriser tous les inverses au sein du système 5.4 mais dans ce cas l'on perd toute possibilité de parallélisme.

Une fois la représentation MRS connue il suffit ensuite d'appliquer la relation 5.3 à chaque modulo :

$$\begin{aligned} a'_j &= A \bmod m'_j \\ &= |\widetilde{a}_1 + m_1(\widetilde{a}_2 + m_2(\widetilde{a}_3 + ... + m_{n-1}\widetilde{a}_n)...)|_{m_j} \end{aligned} \tag{5.5}$$

cette fois le coût est de $n(n-1)$ additions modulaires et d'autant de multiplications. Ce qui nous donne, au total, une complexité de $\frac{3n(n-1)}{2}$ additions et multiplications modulaires.

L'algorithme 23 reprend cette approche et permet d'effectuer un changement de base pour tous les éléments de la base.

Algorithme 23 : Changement de base RNS

Données : Deux bases RNS $\mathcal{B}$ et $\mathcal{B}'$ ainsi que la représentation A_{RNS} d'un entier A dans la base $\mathcal{B}$

Résultat : La représentation A'_{RNS} de A dans $\mathcal{B}'$

```
début
    A_MRS = (ã_1, ..., ã_n) ← A_RNS
    pour i = 1...n faire
        pour j = i+1...n faire
            ã_j ← (ã_j − ã_i) m_{i,j}^{-1}  mod m_j
        fin
    fin
    pour i = 1...n faire
        a'_i ← ã_n
        pour j = n−1...1 faire
            a'_j ← a'_j m_j + ã_j  mod m'_i
        fin
    fin
fin
retourner (a'_1, ..., a'_n)
```

L'avantage principal du changement de base utilisant le MRS est qu'il possible de tirer partie de certaines propriétés des éléments des bases mises en jeu afin de réduire le coût global de l'opération. Le chapitre 6 est consacré à cette étude.

Chapitre 6

Choix de moduli pertinent pour changement de base en RNS

La plupart des études ayant trait au RNS proposent d'utiliser des bases du type $\{2^k - 1, 2^k, 2^k + 1\}$. Ce genre de bases est très utile en analyse du signal où les valeurs manipulées sont bornées et relativement petites. Les algorithmes de changement de base proposés dans la littérature [Pie94, CN99] utilisent, en général, ce type de bases, couplé avec l'approche basée sur le théorème des restes chinois .

Toutefois, le changement de base utilisant la représentation MRS peut s'avérer très efficace dans le cas d'une base à trois éléments. En effet si l'on regarde de près l'équation 5.4 en prenant $m_1 = 2^k + 1$, $m_2 = 2^k - 1$ et $m_3 = 2^k$, nous pouvons alors remarquer que $m_1 \bmod m_2 = 2$, $m_1 \bmod m_3 = 1$ et $m_2 \bmod m_3 = -1$. Nous pouvons donc en déduire que $m_{1,2}^{-1} = 2^{n-1}$, $m_{1,3}^{-1} = 1$ et $m_{2,3}^{-1} = -1$. De cette manière le changement RNS vers MRS se réduit finalement plus à un décalage et quatre additions.

L'idée développée dans ce chapitre est d'étendre la remarque précédente au cadre qui nous intéresse, à savoir la cryptographie. Nous allons voir que l'utilisation de la représentation MRS peut s'avérer là aussi efficace via un choix de bases judicieux. Plus précisément, nous allons faire en sorte de choisir des moduli dont les différences sont toujours petites (pour un ordre de grandeur que nous précirons) afin d'obtenir des propriétés similaires à celles soulevées au dessus.

Comme dans le chapitre précédent nous considérons ici deux bases $\mathcal{B} = (m_1, \ldots, m_n)$ et $\mathcal{B}' = (m'_1, \ldots, m'_n)$ avec pour tout i $m_i = 2^k - c_i$, $m'_i = 2^k - c'_i$ avec c_i et c'_i vérifiant $0 \leq c_i, c'_i < 2^t$ et $1 < t < (k-1)/2$.

Bases RNS efficaces

Il y a principalement deux types d'opérations coûteuses dans un changement de bases via la représentation MRS. La multiplication par un inverse ($x \times m_{i,j}^{-1} \mod m_j$) et la multiplication par une constante ($x \times m_i \mod m'_j$). Nous allons maintenant voir qu'il est possible de choisir les bases $\mathcal{B}$ et $\mathcal{B}'$ de telle sorte que le coût de ces opérations s'en verra diminué.

Pour cela nous proposons de choisir ces deux bases telles que la différence entre deux éléments pris au hasard dans l'ensemble de ces deux bases soit toujours petite. Autrement dit, nous cherchons deux ensembles $(m_1, \ldots, m_n)$ et $(m'_1, \ldots, m'_n)$ tels que :

$$\begin{cases} \forall i \neq j & |m_i - m_j| \leq 2^t \\ \forall i \neq j & |m'_i - m'_j| \leq 2^t \\ \forall i, j & |m_i - m'_j| \leq 2^t \end{cases} \tag{6.1}$$

L'intérêt d'imposer une telle condition est que si t est petit alors les calculs du changement de base vont s'en trouver simplifier. En effet, en remarquant que

$$x \times m_i \mod m_j = x \times (m_i - m_j) \mod m_j,$$

et en considérant le fait que $m_i - m_j \leq 2^t$, nous pouvons voir qu'il est possible de remplacer une multiplication par un entier de k bits par une multiplication par un entier de t. De la même manière, nous allons pouvoir échanger la multiplication modulaire par un inverse de k bits par une multiplication modulaire par un inverse de t bits.

Reste maintenant à s'assurer qu'il est possible de trouver de telles bases en pratique, et à trouver l'ordre de grandeur de t espéré. Nous cherchons donc des ensembles d'entiers vérifiant à la fois le système d'équations 6.1 ainsi que les contraintes

$$\begin{cases} \forall i \neq j & \text{pgcd}(m_i, m_j) = 1 \\ \forall i \neq j & \text{pgcd}(m'_i, m'_j) = 1 \\ \forall i, j & \text{pgcd}(m_i, m'_j) = 1 \end{cases} \tag{6.2}$$

Dans le tableau 6.1 nous évaluons des valeurs de t_{min} pour différentes tailles de moduli (16, 32 et 64 bits) ainsi que différente taille de corps finis (160, 192, 320 et 1024). Attention, les différentes valeurs de t_{min} ne sont pas des minima absolues. Nous n'avons pas testé toutes les bases possibles dans un intervalle donné. Nous avons 'seulement 'testé les bases construites de la façon suivante :

- nous choisissons un entier m_1 dans l'intervalle
- nous choisissons le premier entier m_2 supérieur à m_1 et premier avec celui-ci
- nous choisissons le premier entier m_3 supérieur à m_2 et premier avec m_1 et m_2
- etc jusqu'à obtenir une base de la taille voulue.

		Tailles cryptographiques			
		160	192	320	1024
k	16	6	6	7	10
	32	4	5	6	8
	64	-	3	5	7

Tab. 6.1 – valeur de t_{min} pour différentes tailles de moduli et différentes tailles de corps finis

Exemple 22 Comme précédemment notons pour tout i $m_i = 2^{32} - c_i$ (reps. $m'_i = 2^{32} - c'_i$). Si nous souhaitons effectuer des calculs sur un corps de 160 bits à partir d'une architecture 32 bits nous pouvons utiliser les bases $\mathcal{B}$ et $\mathcal{B}'$ avec $c_i \in \{3, 5, 9, 15, 17\}$ et $c'_i \in \{19, 21, 23, 27, 29\}$. De cette manière on a $t_{min} = 4$.

Exemple 23 Toujours avec une architecture 32 bits il est possible d'effectuer des calculs sur des corps de 1024 bits en prenant les c_i dans l'ensemble {3, 5, 17, 23, 27, 29, 39, 47, 57, 65, 71, 75, 77, 83, 93, 99, 105, 107, 113, 117, 129, 135, 143, 149, 153, 159, 167, 168, 173, 185, 189, 195} et les c'_i dans {285, 297, 299, 303, 309, 315, 323, 327, 329, 339, 353, 359, 363, 365, 369, 383, 387, 395, 413, 419, 429, 437, 453, 465, 467, 479, 483, 485, 489, 497, 507, 509}. Ce qui donne alors $t_{min} = 8$.

Multiplication par un petit inverse

Nous venons de voir qu'il était possible de choisir des bases RNS particulières de telle sorte que la différence entre deux moduli soit toujours petite. L'intérêt d'avoir à effectuer des multiplications par de petites constantes paraît en effet évident en comparaison des multiplications par des constantes de tailles arbitraires. Concernant l'inversion, il ne semble pas y avoir, a priori, d'avantages à utiliser de petits entiers. En effet même si a est un petit entier son inverse modulaire $a^{-1} \mod m$ sera en général un entier de la taille de p.

Cependant nous allons voir que, même si ce n'est pas évident à première vue, il est également intéressant d'avoir à effectuer des multiplications par de petits inverses.

Exemple 24 Considérons les données suivantes : $a = 217$, $b = 2$ et $m = 331$. La méthode naïve pour effectuer l'opération $a \times b^{-1} \mod m$ consiste à calculer $2^{-1} \mod m = 166$ puis $217 \times 166 \mod 331 = 274$, ce qui implique une multiplication entre deux nombres de tailles arbitraires. Il pourtant possible de faire plus simple. En effet il possible de remplacer l'inversion et la multiplication modulaire par une division exacte. L'idée est de calculer un entier équivalent à a modulo m, divisible par b. Considérons pour cela la valeur $r = a \mod b = 217 \mod 2 = 1$. Cela signifie que la division $(a + m)/2$ est exacte ; et nous obtenons bien le résultat 274. De façon plus générale, si nous voulons effectuer l'opération $a \times 2^{-1} \mod m$, pour des entiers quelconques (avec, tout de même, la condition m premier avec 2), deux cas de figures s'offrent à nous :

- si $r = a \mod 2 = 0$ alors a est divisible par 2 et l'opération $a \times b^{-1} \mod m$ revient à un simple décalage,
- si $r = a \mod 2 = 1$ alors $a + m$ est divisible par 2 et l'opération $a \times b^{-1} \mod m$ revient à calculer $\frac{a+m}{2}$, soit une addition et un décalage.

Nous proposons de généraliser cette méthode grâce à l'équation suivante :

$$a \times b^{-1} \mod m = \frac{a - (am^{-1} \mod b)m}{b} \tag{6.3}$$

Cette approche est inspirée grandement de l'algorithme de Montgomery 21. Le but étant simplement de calculer un entier μ tel que $\mu < b$ et $a + \mu m$ est un multiple de b. De telle sorte que la division $\frac{a+\mu m}{b}$ soit exacte et retourne bien la valeur souhaitée.

Ce calcul est décrit dans l'algorithme 24 où scdiv(x, d) retourne un couple d'entiers (r_x, q_x) représentant respectivement le reste et le quotient de la division euclidienne de x par d (on a donc $x = r_x + q_x \times d$ avec $0 \leq r_x < d$). La procédure est scdiv(x, d) est décrite dans l'algorithme 26.

Algorithme 24 : moddiv(a, b, m)

Données : Deux entiers b, m avec $0 < b < m$ and $gcd(m, b) = 1$,
un entier a, $0 \leq a < m$
et des valeurs pré-calculées r, q et I telles que :
$m = r + qb$ avec $r < b$ et
$I = (-m)^{-1} \mod b$
Résultat : un entier $y = a \times |b|_m^{-1} \mod m$
début

1 $(r_a, q_a) \leftarrow$ scdiv(a, b)
2 $(\mu, .) \leftarrow$ bardiv$(r_a \times I, b)$
3 $\nu \leftarrow (ra + \mu \times r)$
4 $(0, \rho) \leftarrow$ bardiv(ν, b)
5 $y \leftarrow \rho + q_a + \mu \times q$

fin
retourner y

Nous allons maintenant analyser la complexité de cet algorithme étape par étape. Notons pour cela $2^{\delta-1} \leq b < 2^{\delta} \leq 2^t$:

1. r_a et q_a sont le reste et le quotient de la division euclidienne d'un entier a, codé sur k bits, par un petit entier b codé sur δ bits. Pour effectuer cette opération, nous faisons appel à une procédure spécifique scdiv(a, b) que nous allons décrire par la suite,
2. r_a et I sont plus petit que b, ainsi μ peut être obtenu grâce à l'algorithme de Barrett [Bar86] appliqué à l'entier de 2δ bits $r_a \times I$. Cela suppose également que nous allons stocker $\left\lfloor \frac{2^{2\delta}}{b} \right\rfloor$ pour toutes les valeurs de b. μ est le reste obtenu grâce à bardiv$(r_a \times I, b)$,
3. ν est le résultat du produit d'un entier de δ bits par un entier de $(k - \delta)$ bits suivi de l'addition d'un entier k bits avec un entier de δ bits,
4. ρ s'obtient grâce à bardiv qui évalue le quotient de la division euclidienne de ν (2δ bits) par b (δ bits),
5. y est le résultat du produit d'un entier de δ bits par un entier de $(k - \delta)$ bits, d'une addition de deux entiers de δ bits et d'une dernière addition entre une entier de k bits et un entier de $(\delta + 1)$ bits.

Remarquons que l'algorithme 24 requiert de pré-calculer et de stocker certaines valeurs. Pour chaque valeur de $b < 2^t$ et chaque élément m_i de la base les valeurs r, q et I. Ce qui conduit en à l'utilisation de table mémoire $\frac{n^2-n}{2} \times (k + 2t)$ bits.

La fonction scdiv

La procédure scdiv permet de calculer le reste r_a et le quotient q_a de la division euclidienne d'un entier par un petit entier b. Le fait que b soit connu permet de se placer dans un cadre proche de celui que requiert l'algorithme de Barrett [Bar86] que nous allons adapter à notre contexte.

L'algorithme de Barrett repose sur la recherche du quotient q de la division par b de $a < 2^{2\delta}$ grâce à l'équation suivante :

$$q = \left\lfloor \frac{\left(\left\lfloor \frac{2^{2\delta}}{d} \right\rfloor \left\lfloor \frac{x}{2^{\delta-1}} \right\rfloor \right)}{2^{\delta+1}} \right\rfloor + \epsilon \text{ , avec, } \epsilon = 0, 1, 2$$

$\left\lfloor \frac{2^{2\delta}}{d} \right\rfloor$ étant une valeur pré-calculée, il reste à effectuer une multiplication de deux nombres de $(\delta + 1)$ bits, le reste des opérations étant des troncatures et la recherche du ϵ nécessitant au plus deux soustractions.

L'algorithme bardiv(a, b) retourne un couple (q, r) tel que : $q = \left\lfloor \frac{\left(\left\lfloor \frac{2^{2\delta}}{b} \right\rfloor \left\lfloor \frac{a}{2^{\delta-1}} \right\rfloor \right)}{2^{\delta+1}} \right\rfloor$ et $r = a - bq$, pour $a < 2^{\delta+l}$ et $2^{\delta-1} \leq b < 2^{\delta}$.

Algorithme 25 : : bardiv(a,b)

Données : $a < 2^{2\delta}$ et $2^{\delta-1} \leq b < 2^{\delta}$
et la valeur pré-calculée $\left\lfloor \frac{2^{2\delta}}{b} \right\rfloor$
Résultat : (r, q) tel que $r = x - dq < 3d$
début

```
1  q ← ( ⌊2^{2δ}/b⌋ ⌊a/2^{δ-1}⌋ )
2  q ← ⌊q/2^{δ+1}⌋
3  r ← a − qb   ( r < 3d)
4  while r > b do
       r ← r − b
       q ← q + 1
   end
```

fin

Exemple 25 Considérons $a = 185 = (10111001)_2$ et $b = 14 = (1110)_2$. Nous avons donc $a < 2^8$ et $b < 2^4$, c'est-à-dire $\delta = 4$. L'algorithme de Barrett s'applique alors de la manière suivante :

- $\left\lfloor \frac{2^{2\delta}}{b} \right\rfloor = 18$,
- $\left\lfloor \frac{a}{2^{\delta-1}} \right\rfloor$ correspond aux $\delta + 1$ bits de poids forts de $a = (\underline{10111}001)_2$ donc $\left\lfloor \frac{a}{2^{\delta-1}} \right\rfloor = (10111)_2 = 23$ et $q = 18 * 23 = 414$,
- nous avons $414 = (\underline{1100}11110)_2$ et donc $\left\lfloor \frac{q}{2^{\delta+1}} \right\rfloor = (1100)_2 = 12$,
- $r \leftarrow a - qb = 17$

- $17 > 14$ donc $q \leftarrow q + 1$ et $r \leftarrow r - b$,
- nous obtenons bien $q = 13$ et $r = 3$.

Analysons la complexité de cet algorithme, nous avons :

1. un produit de deux entiers de $(\delta + 1)$ bits,
2. un décalage,
3. un produit de deux entiers de $(\delta + 1)$ bits ainsi qu'une soustraction d'entiers de $(\delta + 2)$ bits,
4. au plus deux soustractions d'entiers de δ bits,

L'algorithme de Barrett permet donc d'effectuer une division d'un entier de 2δ bits par un entier de δ bits. Cependant, au cours de l'algorithme 24, nous avons besoin d'effectuer la division d'un entier de k-bits (a) par un entier de δ bit (b). C'est pour cela que nous proposons une version légèrement modifiée de l'algorithme de 25 qui prend en compte l'écriture en base 2^δ des entiers concernés. Cette version est simplement une version adaptée à la base 2^δ de l'algorithme scolaire de division dans lequel les division sont effectuées grâce à l'algorithme de Barrett.

Exemple 26 Plaçons nous dans la cas décimal. Prenons $a = 524174$ et $b = 13$. Nous avons donc $\delta = 2, k = 6$ et $\lceil \frac{k}{\delta} \rceil = 3$. A partir de là l'algorithme 26 se déroule de la manière suivante :

- à chaque étape l'on effectue la division euclidienne des 2δ bits de poids forts du dividende par le diviseur,
- à la première étape on effectue donc la division de 5241 par 13, on obtient alors $R' = 2$ et $Q' = 403$. Le quotient intermédiaire Q est donc égal à $403 \times (10^\delta)^i = 40300$ et le reste R égal à 200,
- on ajoute maintenant les bits de poids faible restant du dividende de départ (ici 74) à R, il reste donc à effectuer la division de 274 par 13,
- cette division donne $274 = 21 \times 13 + 1$, le quotient final est donc 40300+21=40321 et le reste est 1.

Algorithme 26 : : Division par une petite constante scdiv(x, d)

Données : deux entiers $a < 2^k$ and $2^{\delta-1} \leq b < 2^\delta$
Résultat : deux entiers R, Q avec $R + Qb = a$ et $0 \leq R < b$
$R \leftarrow a$;
$Q \leftarrow 0$;
début
  pour $i = \left(\lceil \frac{k}{\delta} \rceil - 2\right) \ldots 0$ faire
1 $R' \leftarrow R \div (2^\delta)^i$; $R \leftarrow R \mod (2^\delta)^i$
2 $(R', Q') \leftarrow$bardiv(R', b)
3 $R \leftarrow R'(2^\delta)^i + R$
4 $Q \leftarrow Q'(2^\delta)^i + Q$
  fin
fin
retourner (R, Q)

Analysons la complexité de l'algorithme 26 :

1. considérons que R est divisé en deux parties, 2δ bits de poids forts et $i\delta$ bits de poids faible,
2. nous faisons appel à bardiv afin de diviser R' (2δ bits) par b (δ bits) et d'obtenir Q' le quotient de cette division,
3. nous reconstruisons R avec la partie supérieure R' et puisque R' est un entier de δ bits cela consiste juste en un décalage,
4. de la même manière nous calculons Q à partir de Q'.

Remarquons que Q' peut très bien être un entier de $\delta + 1$ bits ce qui conduit à effectuer une addition pour chaque itération de la boucle de l'algorithme 26. Afin d'éviter ceci, nous allons considérer que $Q = \sum_{i=0}^{(\lceil \frac{k}{\delta} \rceil - 2)} Q'_i (2^\delta)^i$ où Q'_i est la valeur de Q' lors de la i-ème itération de la boucle. Remarquons également que les quantités $L = \sum_{i=0}^{(\lceil \frac{k}{\delta} \rceil - 2)} (Q'_i \mod 2^\delta)(2^\delta)^i$ et $U = \sum_{i=0}^{(\lceil \frac{k}{\delta} \rceil - 2)} (Q'_i \div 2^\delta)(2^\delta)^i$ peuvent être calculées sans la moindre addition. Cela tient simplement au fait que $(Q'_i \mod 2^\delta)$ et $(Q'_i \div 2^\delta)$ sont des entiers de δ bits, ce qui permet de calculer U et L grâce à de simples décalages. Comme de plus, $Q'_i = (Q'_i \mod 2^\delta) + (Q'_i \div 2^\delta)(2^\delta)$, nous avons $Q = L + U(2^\delta)$. Finalement le calcul de Q ne requiert qu'une seule addition et le coût total d'un appel à scdiv de $\lceil \frac{k}{\delta} \rceil - 1$ appels à bardiv d'une addition d'entiers de k-bits.

Nous pouvons maintenant évalué le coût d'un appel à notre opérateur moddiv. Cela commence par un appel à scdiv avec un entier de k bits et un entier de δ bits. Il y a ensuite deux appels à bardiv afin de diviser un entier de 2δ bits par un entier de δ bits. Il faut effectuer de plus deux multiplications d'entiers de δ bits et une multiplication d'entiers de δ bits par un entier de $k - \delta + 1$ bits. Enfin il faut effectuer trois additions entre des entiers de 2δ, $\delta + 1$ et k bits.

Posons $l = \lceil \frac{k}{\delta} \rceil$, nous pouvons donc estimer la complexité de moddiv à $(l + 1)$ multiplications et $(4l + 1)$ additions d'entiers de δ bits. Nous faisons également appel à bardiv

$(l+1)$ fois, donc si nous considérons que le coût d'un tel appel est de deux multiplications d'entiers de δ bits et quatre additions d'entiers de δ bits, nous obtenons une complexité finale de $(3l+3)$ multiplications et $(8l+5)$ additions.

Nous pouvons, par exemple, comparer ce résultat avec l'approche classique consistant à pré-calculer les inverses et à effectuer la multiplication modulaire grâce à l'algorithme de Barrett. Nous obtenons, dans ce dernier cas, une complexité de $2l^2+4l$ multiplications, ce qui valide pleinement notre approche .

Analyse du changement de base en RNS via le MRS

Pour effectuer un changement de base complet nous avons besoin de faire deux conversions.

D'une part, nous devons faire un changement RNS vers MRS comportant $\frac{n(n-1)}{2}$ étapes, chaque étape étant composée d'une addition modulaire sur k bits via modadd et d'une multiplication modulaire entre un entier de k bits et un inverse de δ bits via moddiv.

D'autre part, nous devons effectuer une conversion MRS vers RNS constituée de $n(n-1)$ étapes, chacune d'entre elles faisant intervenir une multiplication modulaire entre un entier de t bits et un entier de k bits et une addition modulaire sur k bits via modadd.

La complexité finale d'un changement de base complet pouvant s'évaluer à $\frac{n(n-1)}{2}(5l+5)$ multiplications et $\frac{n(n-1)}{2}(20l+5)$ additions faisant intervenir des entiers de t bits.

Dans le tableau 6.2, nous donnons les complexités de nos différents algorithmes en nombre de multiplication sur t bits et comparons ces résultats avec la méthode générale consistant à choisir les m'_i sans relation particulière.

Opération	Notre méthode	Méthode classique
modprod	l^2+l+1	$2l^2+4l$
moddiv	$3l+3$	$2l^2+4l$
RNS-MRS	$\frac{3}{2}n(n-1)(l+1)$	$n(n-1)(l^2+2l)$
MRS-RNS	$n(n-1)(l+1)$	$n(n-1)(2l^2+4l)$
RNS-RNS	$\frac{5}{2}n(n-1)(l+1)$	$3n(n-1)(l^2+2l)$

Tab. 6.2 – Comparaison de notre méthode de changement de base RNS avec la méthode classique

Chapitre 7

Inversion modulaire en RNS

Le chapitre 6 nous a permis de mettre en avant l'intérêt d'un choix judicieux de bases RNS afin d'optimiser la phase de réduction intervenant lors d'une multiplication modulaire. Cela pourrait être suffisant pour effectuer les calculs nécessaires à une multiplication de points sur une courbe elliptique à partir du moment où les points sont données en coordonnées projectives. Par contre, si l'on désire utiliser les coordonnées affines nous avons vu qu'il est nécessaire d'effectuer de nombreuses inversions modulaires ; chose difficile en RNS. En effet nous ne disposons pas, à notre connaissance, d'algorithmes d'inversion modulaire en RNS. La raison étant que tous les algorithmes classique d'inversion sont basés sur l'algorithme d'Euclide étendu et que nous ne disposons même pas d'algorithme de division performant en RNS. Afin de combler ce manque, nous proposons dans ce chapitre un algorithme d'inversion modulaire en RNS qui ne consiste pas à recalculer la valeur de l'entier concerné, l'inverser par une méthode classique et calculer la représentation RNS du résultat. L'idée développée ici est de combiner deux techniques d'évaluation distinctes. D'une part l'évaluation des quotients apparaissant dans l'algorithme d'Euclide, et d'autre part l'évaluation d'un nombre en RNS. Nous verrons qu'il est possible de calculer la représentation RNS d'une valeur approchée du quotient de deux entiers, sans pour autant avoir à effectuer une division entre ces deux entiers.

Dans la mesure où il n'y a pas d'ambiguïté concernant les bases RNS utilisées nous notons simplement $\overline{U}$ la représentation RNS d'un entier U. Nous avons donc $\overline{U} = (u_1, \ldots, u_n)$ et toutes les opérations faisant intervenir des entiers écrits de la sorte sont effectuées directement en représentation RNS. A partir de maintenant considérons également :

- une base RNS $\mathcal{B} = (m_1, \ldots, m_n)$ avec $M = \prod_{i=1}^{n} m_i$ et $2^{k-1} < m_i < 2^k$ $(k = 32)$,
- l'entier $e = \left\lfloor \frac{\log n}{\log 2} \right\rfloor$,
- un nombre premier P tel que $2^{l-1} < P < 2^l$ avec $k << l$ $(l = 160)$.

7.1 L'algorithme d'Euclide étendu

La méthode la plus classique pour effectuer une inversion modulaire consiste à faire appel à la version étendue de l'algorithme d'Euclide 27.

Algorithme 27 : Algorithme d'Euclide étendu ExtEucl(A, p)

Données : Deux entiers $A < p$ premiers entre eux
Résultat : $A^{-1} \mod p$
début
  $(U_1, U_3) \leftarrow (0, p)$
  $(V_1, V_3) \leftarrow (1, A)$
  tant que $V_3 > 1$ faire
    $q \longleftarrow \left\lfloor \frac{U_3}{V_3} \right\rfloor$
    $(U_1, U_3) \longleftarrow (U_1, U_3) - q(V_1, V_3)$
    $(U_1, U_3) \longleftrightarrow (V_1, V_3)$
  fin
  retourner V_1
fin

Exemple 27 Prenons $A = 22$ et $p = 31$ l'algorithme ce déroule alors comme suit :
- $(U_1, U_3) = (0, 31)$ et $(V_1, V_3) = (1, 22)$
- $q = \frac{31}{22} = 1$
- $(U_1, U_3) \longleftarrow (0, 31) - 1 \times (1, 22) = (-1, 9)$
- $(U_1, U_3) \longleftrightarrow (V_1, V_3)$
- $(U_1, U_3) = (1, 22)$ et $(V_1, V_3) = (-1, 9)$
- $q = 2$
- $(U_1, U_3) = (3, 4)$
- $(U_1, U_3) \longleftrightarrow (V_1, V_3)$
- $(U_1, U_3) = (-1, 9)$ et $(V_1, V_3) = (3, 4)$
- $q = 2$
- $(U_1, U_3) = (-7, 1)$
- $(U_1, U_3) \longleftrightarrow (V_1, V_3)$

A partir de là, V_3 est égal à 1 et l'algorithme retourne $V_1 = -7$. Nous pouvons alors vérifier que nous avons bien $-7 \times 22 = 1 \mod 31$.

Chaque étape de cette algorithme est constituée d'une division et de deux multiplication. Dès lors que sous sommes capable de calculer la valeur de $q = \frac{U_3}{V_3}$ en RNS le reste de l'algorithme ne pose donc aucun problème.

Malheureusement il n'existe pas de bon algorithme de division en RNS. Nous allons donc adopter une approche un peu particulière afin de contourner ce problème. L'idée est d'introduire une fonction Estim permettant de calculer une approximation par défaut du quotient $\frac{U_3}{V_3}$ en RNS. Plus précisément nous allons voir qu'il est possible de calculer les premiers chiffres significatifs des quotients U_3/M et V_3/M en utilisant des représentation

RNS de U_3 et V_3. A partir de là, il nous sera possible de donner une estimation de la valeur de $\frac{U_3}{V_3}$.

Le fait de ne calculer qu'une approximation du quotient ne change pas fondamentalement l'algorithme 27. En général, cela conduit simplement à augmenter le nombre d'étapes de l'algorithme.

Exemple 28 Reprenons l'exemple 27. A la deuxième étape nous avions obtenu les valeurs $(U_1, U_3) = (1, 22)$ et $(V_1, V_3) = (-1, 9)$, le quotient $q = \frac{22}{9}$ valant alors 2. Supposons qu'au lieu de calculer le quotient exact nous ne puissions obtenir qu'une valeur approchée $\widetilde{q}$ égale à 1. L'algorithme peut alors s'effectuer de la manière suivante :

- $(U_1, U_3) = (1, 22)$ et $(V_1, V_3) = (-1, 9)$
- $\widetilde{q} = 1$
- $(U_1, U_3) \longleftarrow (1, 22) - \widetilde{q} \times (-1, 9) = (2, 13)$
- on teste si $U_3 < V_3$, si ce n'est pas le cas on on calcul $\widetilde{q} = \frac{U_3}{V_3}$ et on recommence , sinon on passe à la suite.
- on a $13 > 9$
- $\widetilde{q} = \frac{U_3}{V_3} = 1$
- $(U_1, U_3) \longleftarrow (1, 22) - 1 \times (-1, 9) = (3, 4)$
- maintenant on a bien $U_3 < V_3$
- on reprend l'algorithme 27 la où il en était
- etc

L'algorithme 7.1 donne une première ébauche du futur algorithme d'inversion en RNS.

Algorithme 28 : Inversion modulaire en RNS

```
début
    tant que V3 ≠ 1 faire
        Q ⟵ Estim ( ⌊ U3 / V3 ⌋ ) si Q ≠ 0 alors
            (U1, U3) ⟵ (U1, U3) − Q(V1, V3)
        sinon
            si Test (U3 > V3) alors
                (U1, U3) ⟵ (U1, U3) − (V1, V3)
            fin
            (U1, U3) ⟷ (V1, V3)
        fin
    fin
    retourner V1
fin
```

Nous montrons maintenant comment construire la procédure Estim qui évalue une approximation de U/V à partir des valeurs approchées de U/M et V/M.

7.2 Estimation du quotient $\frac{U}{M}$

Soient $\overline{U} = (u_1, ...u_n)$. Toute notre approche repose sur l'équation suivante :

$$U = \sum_{i=1}^{n} \left| u_i \times M_i^{-1} \right|_{m_i} \times M_i - \alpha M = \sum_{i=1}^{n} \frac{\lambda_i}{m_i} \times M - \alpha M \tag{7.1}$$

où α est un entier et pour tout i $\lambda_i = \left| u_i \times M_i^{-1} \right|_{m_i}$.

A partir de cette équation nous avons

$$\frac{U}{M} = \sum_{i=1}^{n} \frac{\lambda_i}{m_i} - \alpha \tag{7.2}$$

Nous posons $\sum_{i=1}^{n} \frac{\lambda_i}{m_i} = E + F$, où E est un entier et F vérifie $0 < F < 1$; c'est-à-dire que E et F sont respectivement la partie entière et fractionnaire de $\sum_{i=1}^{n} \frac{\lambda_i}{m_i}$.

Finalement, nous avons

$$\frac{U}{M} = (E - \alpha) + F \tag{7.3}$$

Puisque $(E - \alpha)$ est un entier et que $0 < \frac{U}{M} < 1$ nous pouvons déduire que $E = \alpha$. Ce qui conduit à l'égalité $F = \frac{U}{M}$.

Cela signifie que pour calculer une valeur approchée de $\frac{U}{M}$, il suffit d'estimer $\sum_{i=1}^{n} \frac{\lambda_i}{m_i}$ et d'en prendre la partie fractionnaire. Si l'on souhaite par exemple obtenir les k premiers bits, le calcul s'effectue de la manière suivante :

- nous approchons les k premiers bits significatifs de $\frac{1}{m_i}$ par $\beta_i = \left\lceil \frac{2^{2k-1}}{m_i} \right\rceil$,
- nous calculons $\sum_{i=1}^{n} \lambda_i \times \beta_i$,
- nous réduisons modulo 2^{2k-1} afin de nous débarrasser de la partie entière de $\sum_{i=1}^{n} \frac{\lambda_i}{m_i}$,
- nous divisons le résultat par 2^k afin de conserver un résultat sur k bits,
- nous rajoutons 1 afin d'être sur de faire une approximation par valeur supérieure,

Notons $\widetilde{u}$ le résultat final, nous avons alors :

$$\max\left(\frac{U}{M} \times 2^{k-1},\ 1 \right) \leq \widetilde{u} \leq \frac{U}{M} \times 2^{k-1} + n \tag{7.4}$$

Exemple 29 Prenons les données suivantes :

- $\mathcal{B} = (100\,003\ ,\ 120\,011\ ,\ 150\,001)$
- $M = 1\,800\,231\,006\,410\,033$
- $U = \ 1\,000\,000\,000\,000 = (900, 61741, 33378)_{RNS}$
- $k = 6$

Nous avons alors

- $(\frac{1}{m_1}, \frac{1}{m_2}, \frac{1}{m_3}) = (0.000009999700\ldots, 0.000008332569\ldots, 0.000006666622\ldots)$
- $(\beta_1, \beta_2, \beta_3) = (\left\lceil \frac{10^{11}}{m_1} \right\rceil, \left\lceil \frac{10^{11}}{m_2} \right\rceil, \left\lceil \frac{10^{11}}{m_3} \right\rceil) = (999971, 833257, 666663)$
- $\overline{\lambda} = (37032, 69472, 7705)$

$$\begin{aligned}\sum_{i=1}^{n} \lambda_i \times \beta_i &= 37032 \times 999971 + 69472 \times 833257 + 7705 \times 666603 \\ &= 100\,055\,594\,791 \\ \widetilde{u} &= ((100\,055\,594\,791 \mod 10^{11}) \div 10^6) + 1 \\ &= (55\,594\,791 \div 10^6) + 1 \\ &= 56\end{aligned}$$

A comparer avec la valeur exacte $\frac{U}{M} \times 10^5 = 55.548426\ldots$

Puisque nous travaillons sur des entiers de 6 chiffres, il paraît assez naturel de vouloir un résultat plus précis. Par exemple sur au moins 4 chiffres. De plus, si nous prenons $U = 1000000$ alors la procédure précédente retourne 1 alors que nous avons $\frac{1\,000\,000}{M} \times 10^5 = 0.0000555484...$, ce qui n'est évidemment pas une approximation satisfaisante.

Pour pallier ce problème, nous proposons de calculer $\frac{U \times 2^{t_u}}{M} \times 2^{k-1}$ en lieu et place de $\frac{U}{M} \times 2^{k-1}$, pour un entier t_u bien choisi.

Par exemple en prenant $t_u = 3$ la procédure précédente appliquée à $U \times 10^3$ retourne 55549. Résultat à comparer avec $\frac{U \times 10^3}{M} \times 10^5 = 55548.4...$

Cette procédure est résumée dans l'algorithme 29. Nous introduisons donc un paramètre t_u tel que l'on ai $\frac{M}{8} < U \times 2^{t_u} < \frac{M}{2}$. A partir de là, l'algorithme 29 retourne un entier $\widetilde{u}$ vérifiant :

$$\frac{U \times 2^{t_u}}{M} \times 2^{k-1} \leq \widetilde{u} < \frac{U \times 2^{t_u}}{M} \times 2^{k-1} + n + 1 \tag{7.5}$$

Algorithme 29 : ApproxSup($\overline{U}, \overline{2^{t_u}}$)

Données : Une entier $\overline{U}$ en RNS et
pour $i = 1, ..., n$ des entiers pré-calculés $\beta_i = \left\lceil \frac{2^{2k}}{m_i} \right\rceil$, $\overline{\mu} = (\mu_1, \ldots, \mu_n)$ avec $\mu_i = |M_i|_{m_i}^{-1}$
Résultat : $\widetilde{u}$ vérifiant la double inégalité 7.5
début
  $\overline{\lambda} \leftarrow \overline{U} \times \overline{\mu} \times \overline{2^{t_u}}$;
  $\widetilde{u} \leftarrow 0$;
  pour $i \leftarrow 1 \ldots n$ faire
    $\widetilde{u} \leftarrow \widetilde{u} + \beta_i \times \lambda_i \bmod 2^{2k}$
  fin
  $\widetilde{u} \leftarrow (\widetilde{u} \div 2^k) + 1$
  retourner $\widetilde{u}$
fin

Preuve : Afin de simplifier la preuve nous considérons que $\frac{M}{8} < U < \frac{M}{2}$, c'est-à-dire que $t_u = 0$. Il est facile de voir que si $t_u \neq 0$ il suffit de remplacer U par $U' = U \times 2^{t_u}$

dans le reste de la preuve pour obtenir le résultat général.

Nous avons $U = (u_1, ..., u_n)_{RNS}$ pour tout $i \in \{1..n\}$, $\beta_i = \left\lceil \frac{2^{2k-1}}{m_i} \right\rceil$ et $0 \leq \lambda_i \leq 2^k$. Soit $i \in \{1..n\}$:

$$\frac{2^{2k-1}}{m_i} \leq \beta_i \leq \frac{2^{2k-1}}{m_i} + 1 \tag{7.6}$$

$$\sum_{i=1}^{n} \lambda_i \times \frac{2^{2k-1}}{m_i} \leq \sum_{i=1}^{n} \lambda_i \times \beta_i \leq \sum_{i=1}^{n} \lambda_i \times \frac{2^{2k-1}}{m_i} + \sum_{i=1}^{n} \lambda_i \tag{7.7}$$

Maintenant, posons $\sum_{i=1}^{n} \frac{2^{2k-1}}{m_i} = (E + \frac{U}{M}) \times 2^{2k-1}$ où E est un entier. Puisque $\frac{U}{M} < \frac{1}{2}$ on a alors,

$$(E + \frac{U}{M}) \times 2^{2k-1} \leq \sum_{i=1}^{n} \lambda_i \times \beta_i < (E + \frac{U}{M}) \times 2^{2k-1} + \sum_{i=1}^{n} \lambda_i \tag{7.8}$$

$$(E + \frac{U}{M}) \times 2^{2k-1} \leq \sum_{i=1}^{n} \lambda_i \times \beta_i < E \times 2^{2k-1} + 2^{2k-2} + n2^k \tag{7.9}$$

Puisque $n < 2^{k-2}$ on a $2^{2k-2} + n2^k < 2^{2k-1}$ et ainsi

$$\left| \sum_{i=1}^{n} \lambda_i \times \beta_i \right|_{2^{2k-1}} = \sum_{i=1}^{n} \lambda_i \times \beta_i - E \times 2^{2k-1} \tag{7.10}$$

finalement

$$\frac{U}{M} \times 2^{2k-1} \leq \left| \sum_{i=1}^{n} \lambda_i \times \beta_i \right|_{2^{2k-1}} \leq \frac{U}{M} \times 2^{2k-1} + n2^k \tag{7.11}$$

Une dernière division par 2^k permet d'obtenir le résultat voulu.

7.3 Normalisation

Grâce à l'algorithme ApproxSup nous sommes capable de donner une valeur approcher des premiers bits significatifs de la fraction $\frac{U \times 2^{t_u}}{M}$. L'utilisation de cet algorithme présuppose cependant que nous ayons à notre disposition l'entier t_u vérifiant $\frac{M}{8} < U \times 2^{t_u} < \frac{M}{2}$.

Nous proposons maintenant un algorithme qui intègre à la fois le calcul d'un t_u, de sorte à avoir un maximum de bits significatifs, et le calcul de la valeur approchée de $\frac{U \times 2^{t_u}}{M}$. Plus précisément la procédure va mettre à jour les valeurs $t_u, \widetilde{u}, \overline{2^{t_u}}$ de sorte que

$$\frac{M}{8} < U \times 2^{t_u} < \frac{M}{2}$$

où $\widetilde{u}$ correspond au résultat renvoyé par ApproxSup.

Pour résumer nous cherchons donc t_u tel que $\frac{M}{8} < U \times 2^{t_u} < \frac{M}{2}$. Commençons par estimer l'erreur commise lors d'un appel à la fonction ApproxSup. D'après l'équation ?? on sait que :

$$\max\left(\frac{U}{M} \times 2^{k-1},\ 1\right) \leq \widetilde{u} \leq \frac{U}{M} \times 2^{k-1} + n + 1.$$

Nous pouvons estimer l'erreur commise à au plus $e = \left\lceil \frac{\log n}{\log 2} \right\rceil$ bits faux, le résultat étant toujours supérieur à la valeur exacte.

Nous présentons maintenant le principe général de la fonction de normalisation. Notons que pour simplifier cette présentation nous n'allons pas prendre les contraintes exactes apparaissant dans l'algorithme 30. Ces contraintes étant principalement issues de la preuve de validité de la fonction afin d'assurer la justesse des résultats, en prenant en compte les cas limites.

Le principe de la fonction de normalisation NormSup est le suivant :

- Tant que la valeur $\widetilde{u}$ est inférieure à l'erreur commise cela signifie que la valeur exacte de $\frac{U}{M}$ est bien plus petite que notre résultat $\widetilde{u}$. Plus précisément nous sommes sûrs que $\frac{U2^{t_u}}{M} \times 2^{l-1} \leq \widetilde{u} < 2^e$ et donc que $U \times 2^{t_u+l-e-2} < \frac{M}{2}$. Cela veut dire que que tant que $\widetilde{u} < n$ nous pouvons ajouter $l - e - 2$ à t_u.
- Une fois que $n < \widetilde{u}$ nous calculons juste le nombre de bits que nous pouvons rajouter afin d'avoir un maximum de bits significatifs. Nous calculons donc l'entier $c = (l-2) - \left\lceil \frac{\log \widetilde{u}}{\log 2} \right\rceil$ et mettons la valeur t_u à jour : $t_u \longleftarrow t_u + c$.

Exemple 30 Nous reprenons les données de l'exemple 29. Rappelons que nous considérons dans cette exemple les écritures décimales des entiers, et non les écritures binaires. Nous avons donc :

- $erreur = n = 3$
- $e = \left\lfloor \frac{\log n}{\log 10} \right\rfloor = 1$
- $l - 2 - e = 3$
- $U = 1\,000\,000\,000 = (70003, 68348, 93334)_{RNS}$
- $\text{ApproxSup}(\overline{U}) = 1 < 10^1$
- $\text{ApproxSup}(\overline{U} \times 10^3) = 56 > 10^1$
- $c = (l-2) - \left\lceil \frac{\log \widetilde{u}}{\log 10} \right\rceil = 2$
- $t_u = 5$, $\text{ApproxSup}(\overline{U} \times 10^5) = 5555$ et $\frac{U \times 10^5}{M} \times 10^6 = 5554.84...$

L'algorithme 30 synthétise cette approche.

Algorithme 30 : NormSup($\overline{U}, t_u, \widetilde{u}, \overline{2^{t_u}}$)

```
début
    ũ ← ApproxSup(Ū, 2^{t_u}‾);
    tant que ũ < 2^{e+2} faire
        t_u ← t_u + l − e − 4;
        2^{t_u}‾ ← 2^{l−e−4} × 2^{t_u}‾;
        ũ ← ApproxSup(Ū, 2^{t_u}‾);
    fin
    c ← 0;
    tant que ũ < 2^{l−3} faire
        t_u ← t_u + 1;
        ũ ← ũ << 1;
        c ← c + 1;
    fin
    si c > 0 alors
        2^{t_u}‾ ← 2^c × 2^{t_u}‾;
        ũ ← ApproxSup(Ū, 2^{t_u}‾);
    fin
fin
```

preuve : Supposons que $0 < U \times 2^{t_u} < \frac{M}{2}$ et notons $e = \left\lceil \frac{\log n}{\log 2} \right\rceil$.

D'abord, supposons $\widetilde{u} < 2^{e+2}$ alors

$$\begin{aligned} \frac{U}{M} \times 2^{t_u} \times 2^{l-1} &\leq \widetilde{u} \\ U \times 2^{t_u} \times 2^{l-1} &< M \times 2^{e+2} \\ U \times 2^{t_u} \times 2^{l-e-4} &< \frac{M}{2} \end{aligned}$$

A la fin de chaque itération de la première boucle t_u est remplacé par $t_u + l - e - 4$ et nous avons toujours $U \times 2^{t_u} < \frac{M}{2}$. Nous obtenons ainsi $2^{e+2} \leq \widetilde{u} < 2^{l-2}$ et

$$\frac{U}{M} \times 2^{t_u} \times 2^{l-1} \leq \widetilde{u} < \frac{U}{M} \times 2^{t_u} \times 2^{l-1} + 2^e \tag{7.12}$$

Considérons maintenant la deuxième boucle construisant un entier c tel que : $2^{l-3} \leq \widetilde{u} \times 2^c < 2^{l-2}$.

Nous avons alors

$$2^{l-3} < \widetilde{u} \times 2^c < \frac{U}{M} \times 2^c \times 2^{t_u} \times 2^{l-1} + 2^e \times 2^c$$

$$2^{-3} < \frac{U}{M} \times 2^c \times 2^{t_u} \times 2^{-1} + 2^e \times 2^c \times 2^{-l}$$

Puisque $\widetilde{u} \times 2^c \geq 2^{e+2} \times 2^c$, nous avons $e + c + 2 \leq l - 2$.

Ensuite,

$$2^{-3} < \frac{U}{M} \times 2^{c} \times 2^{t_u} \times 2^{-1} + 2^{-4}$$

et finalement,

$$2^{-4} = 2^{-3} - 2^{-4} < \frac{U}{M} \times 2^{c} \times 2^{t_u} \times 2^{-1}$$

autrement dit,

$$2^{-3} < \frac{U}{M} \times 2^{c} \times 2^{t_u}.$$

De la même manière nous pouvons obtenir la relation,

$$\frac{U}{M} \times 2^{t_u} \times 2^{c} \times 2^{l-1} \leq \widetilde{u} \times 2^{c} \leq 2^{l-2}$$

qui donne,

$$\frac{U}{M} \times 2^{t_u} \times 2^{c} \leq 2^{-1}$$

Il suffit alors de poser $t_u = t_u + c$ pour obtenir,

$$\frac{1}{8} < \frac{U}{M} \times 2^{t_u} < \frac{1}{2}. \tag{7.13}$$

7.4 Estimation du quotient

Nous sommes maintenant capable d'estimer la valeur du quotient $\frac{U}{V}$ en utilisant uniquement la représentation RNS de chacun des deux nombres. Pour cela il nous suffit de calculer $\widetilde{U}$,$\widetilde{V}$ puis de calculer le quotient

$$q = \left\lfloor \frac{(\widetilde{u} - n) \times 2^{t_v - t_u}}{\widetilde{v}} \right\rfloor \leq \left\lfloor \frac{u}{v} \right\rfloor$$

Exemple 31 Reprenons à nouveau les données des exemples 29 et 30.

- $U = 1000000000 = (70003, 68348, 93334)_{RNS}$, $V = 3654025 = (53917, 53695, 54001)_{RNS}$
- $\widetilde{u} = 5555$, $t_u = 5$ et $\widetilde{v} = 2030$, $t_u = 7$

Dans ce cas si nous calculons $q = \left\lfloor \frac{(\widetilde{u}-n)\times 2^{t_v - t_u}}{\widetilde{v}} \right\rfloor$ nous obtenons 273, c'est-à-dire la valeur exacte du quotient ($\left\lfloor \frac{U}{V} \right\rfloor = 273$).

En prenant une autre valeur pour V, par exemple $V = 215$, et en effectuant à nouveau les calculs nous obtenons $q = 4\,648\,535$, alors que $\left\lfloor \frac{U}{V} \right\rfloor = 4\,651\,162$. Bien que le résultat ne soit pas exact, nous pouvons voir que notre estimation est très proche de la valeur exacte.

Cette procédure est donnée par l'algorithme 31 et permet de calculer une approximation du quotient $\left\lfloor \frac{U}{V} \right\rfloor$ telle que la valeur retournée est nulle si et seulement si $\left\lfloor \frac{U}{V} \right\rfloor < 2$.

Dans tous les autres cas, dès lors que $t_v - t_u - (l-2) \geq 0$, le résultat $\widetilde{q}$ vérifie $2^{l-5} < \widetilde{q} < 2^l$.

Algorithme 31 : Estim$(\widetilde{u}, \widetilde{v}, t_u, t_v, \overline{2^{t_u}}, \overline{2^{t_v}})$

```
début
    si t_v − t_u − (l − 2) ≥ 0 alors
        s ⟵ l − 2
    sinon
        s ⟵ t_v − t_u
    fin
    q̃ ⟵ ((ũ − n)2^s) / ṽ
    Q̃ ⟵ q̃ × 2^{t_v − t_u − (s)}
    retourner (Q̃, q̃)
fin
```

Preuve : La preuve va se décomposer en deux parties. Nous allons d'abord montrer que $2^{s-3} < \widetilde{q} < 2^{s+2}$. Pour cela considérons que

$$\frac{M}{8} < U \times 2^{t_u} < \frac{M}{2} \text{ et } \frac{M}{8} < V \times 2^{t_v} < \frac{M}{2} \tag{7.14}$$

Nous obtenons alors

$$2^{-2} < \frac{U}{V} \times 2^{t_u - t_v} < 2^2 \tag{7.15}$$

Maintenant nous cherchons à borner $\widetilde{q}$. Pour cela, nous utilisons l'équation 7.5. Nous notons de plus $\widetilde{u} = \widetilde{u} - n$ afin de simplifier les notations.

$$\frac{U}{M} \times 2^{t_u+l-1} - n \leq \widetilde{u} < \frac{U}{M} \times 2^{t_u+l-1} \tag{7.16}$$

$$\frac{V}{M} \times 2^{t_v+l-1} \leq \widetilde{v} < \frac{V}{M} \times 2^{t_v+l-1} + n \tag{7.17}$$

Ainsi nous obtenons :

$$\frac{U \times 2^{t_u+l-1} - nM}{V \times 2^{t_v+l-1} + nM} \leq \frac{\widetilde{u}}{\widetilde{v}} \leq \frac{U}{V} 2^{t_u - t_v} \tag{7.18}$$

$$\frac{U - nM \times 2^{-(t_u+l-1)}}{V + nM \times 2^{-(t_v+l-1)}} 2^{t_u - t_v} \leq \frac{\widetilde{u}}{\widetilde{v}} \leq \frac{U}{V} 2^{t_u - t_v} \tag{7.19}$$

Nous avons également : $\frac{M}{8} < U \times 2^{t_u} < \frac{M}{2}$ et $\frac{M}{8} < V \times 2^{t_v} < \frac{M}{2}$. Puisque pour tout i $2^{l-1} < m_i < 2^l$ et $n < 2^e$, l'équation (7.19) devient :

$$\frac{1 - 2^{e+3-(l-1)}}{1 + 2^{e+3-(l-1)}} \times \frac{U}{V} \times 2^{t_u - t_v} \leq \frac{\widetilde{u}}{\widetilde{v}} \leq \frac{U}{V} \times 2^{t_u - t_v} \tag{7.20}$$

Nous savons que $\frac{1}{1+2^{e+3-(l-1)}} < 1 - 2^{e+3-(l-1)}$ quand $e + 3 - (l-1) \leq -2$, autrement

dit, quand $l \geq e+6$.

L'équation devient alors 7.21 :

$$(1-2^{e+3-(l-1)})^2 \times \frac{U}{V} \times 2^{t_u-t_v} \leq \frac{\widetilde{u}}{\widetilde{v}} \leq \frac{U}{V} \times 2^{t_u-t_v} \tag{7.21}$$

Si $l \geq e+6$ alors, $(1-2^{e+3-(l-1)})^2 > \frac{1}{2}$. Nous en déduisons que :

$$\left\lfloor \frac{U}{V} \times 2^{t_u-t_v+s-1} \right\rfloor \leq \widetilde{q} \leq \left\lfloor \frac{U}{V} \times 2^{t_u-t_v+s} \right\rfloor \tag{7.22}$$

et

$$\left\lfloor 2^{-1} \times \frac{U}{V} \right\rfloor \leq \widetilde{Q} \leq \left\lfloor \frac{U}{V} \right\rfloor \tag{7.23}$$

Si $s = l-2$ et $l \geq e+6$ nous obtenons $2^{l-5} \leq \widetilde{q} \leq 2^l$.

Si $s = t_v - t_u$ alors $\widetilde{Q} = \widetilde{q}$. Dans ce cas, $\widetilde{q}$ peut être égal à 0 si $(1-2^{e+3-(l-1)})^2 \times \frac{U}{V} < 1$. En d'autres termes, $\frac{U}{V} < (1-2^{e+3-(l-1)})^{-2}$ est plus petit que 2 puisque $e+3-(l-1) \leq -3$, c'est-à-dire $l > e+7$. Donc si $l > e+7$, Estim renverra une valeur nulle seulement si $U < 2 \times V$.

Signe d'une différence

Nous sommes maintenant capable d'évaluer en RNS la valeur du quotient $\frac{U}{V}$. Cependant nous avons vu que dans un cas particulier notre fonction Estim renvoie la valeur 0. Sachant que dans ce cas nous avons forcément $U < 2 \times V$ que le quotient en question vaut 1 ou 0. S'il vaut 1 cela signifie que $U > V$ et qu'il faut soustraire V à U, s'il vaut 0 cela signifie que $U < V$ et qu'il faut alors soustraire U à V. Au final cela veut dire qu'il nous faut, dans ce cas précis, être capable d'estimer le signe de la différence $D = U - V$.

Notre but est d'évaluer le signe de D en RNS, c'est-à-dire le signe de $\overline{D} = \overline{U} - \overline{V}$. Deux cas se présentent : soit $U \geq V$ et alors $\overline{D} = D$, soit $U < V$, et alors, $D < 0$ et $\overline{D} = M + D$. Sachant que, par hypothèse, les entrées de notre algorithme vérifient $0 < V < P < \frac{1}{4}M$ nous pouvons déduire que si D est positif alors nous aurons $\overline{D} = D < \frac{1}{4}M$ alors que si $D < 0$ et $\overline{D} = M + D > \frac{1}{2}M$.

Notons $\widetilde{d}$ la valeur retournée par ApproxSup$(\overline{D}, \overline{1})$, nous avons,

$$\frac{D}{M}2^{l-1} < \widetilde{d} < \frac{D}{M}2^{l-1} + n. \tag{7.24}$$

Si $\widetilde{d} \geq 2^{l-2}$ alors $\overline{D} = M + D$ sinon $\overline{D} = D$ et $\widetilde{d} < 2^{l-3} + n$.

Il reste un problème à gérer car la fonction ApproxSup ne permet de faire des approximation que pour des entiers plus petit que $\frac{M}{2}$. La solution que nous adoptons ici pour assurer cette condition consiste simplement à évaluer non pas D mais $\frac{D}{2}$. Dans ce cas il suffira de comparer $\widetilde{d}$ à 2^{l-3}. Si D est pair, la division se fait en multipliant le résultat courant par l'inverse de 2 modulo M. Si D est impair, nous remplaçons simplement D

par $D-1$. Pour tester la parité de D il suffit que l'un des m_i soit pair, le test revenant alors à tester la parité du reste modulo ce m_i.

Tout cela conduit à une procédure Red calculant $\frac{D}{2}$ ou $\frac{D-1}{2}$ selon que D soit pair ou impair.

La fonction finale Test est donnée par l'algorithme suivant :

Algorithme 32 : Test $(\overline{U}, \overline{V})$

Données : $0 \leq U, V < p < \frac{1}{4}M$
Résultat : un booléen B valant vrai si $U \geq V$ et faux sinon
début
  $\overline{D} \leftarrow \overline{U} - \overline{V}$
  $\overline{D} \leftarrow$ Red$(\overline{D})$
  $\widetilde{d} \leftarrow$ ApproxSup$(\overline{D}, \overline{1})$ si $\widetilde{d} \geq 2^{l-3}$ alors
    $B \leftarrow$ faux
  sinon
    $B \leftarrow$ vrai
  fin
fin
retourner B

A l'aide des procédures que nous venons de décrire nous pouvons maintenant énoncer l'algorithme 33 d'inversion modulaire.

Algorithme 33 : ExtEuclRNS($\overline{A}, \overline{P}$)

Données : $A < p < \frac{M}{4}$ avec p un nombre premier
Résultat : $\overline{A^{-1}} \bmod P$;
$(\overline{U_1}, \overline{U_3}) \leftarrow (\overline{0}, \overline{P})$;
$(\overline{V_1}, \overline{V_3}) \leftarrow (\overline{1}, \overline{A})$;
$t_u \leftarrow 0$; $t_v \leftarrow 0$; $\overline{2^{t_u}} \leftarrow (\overline{1})$; $\overline{2^{t_v}} \leftarrow (\overline{1})$;
NormSup($\overline{U_3}, t_u, \widetilde{u}, \overline{2^{t_u}}$);
NormSup($\overline{V_3}, t_v, \widetilde{v}, \overline{2^{t_v}}$);
counter $\leftarrow 0$;
1 tant que $\overline{V_3} \neq 0$ faire
  $(\widetilde{Q}, \widetilde{q}) \longleftarrow$ Estim($\widetilde{u}, \widetilde{v}, t_u, t_v, \overline{2^{t_u}}, \overline{2^{t_v}}$) ;
2 si $\widetilde{q} > 0$ alors
    $(\overline{U_1}, \overline{U_3}) \longleftarrow (\overline{U_1}, \overline{U_3}) - \widetilde{Q}(\overline{V_1}, \overline{V_3})$;
    NormSup($\overline{U_3}, t_u, \widetilde{u}, \overline{2^{t_u}}$);
  sinon
3 si Test($\overline{U_3}, \overline{V_3}$) alors
      $(\overline{U_1}, \overline{U_3}) \longleftarrow (\overline{U_1}, \overline{U_3}) - (\overline{V_1}, \overline{V_3})$;
      NormSup($\overline{U_3}, t_u, \widetilde{u}, \overline{2^{t_u}}$);
    fin
4 $(\overline{U_1}, \overline{U_3}, t_u, \overline{2^{t_u}}) \longleftrightarrow (\overline{V_1}, \overline{V_3}, t_v, \overline{2^{t_v}})$;
    counter $\leftarrow$ *counter* $+ 1$;
  fin
fin
si *counter* est pair alors
  $\overline{U_1} \longleftarrow \overline{P} + \overline{U_1}$
fin
retourner $\overline{U_1}$

Chapitre 8

Étude des sommes de produits modulaires

Nous avons vu que le système de représentation RNS est caractérisé par une multiplication peu coûteuse, mais une réduction modulaire difficile à mettre en oeuvre. D'une manière analogue les systèmes de représentations classiques sont, en général, caractérisés par une multiplication assez coûteuse, mais une réduction modulaire pouvant être très efficace très efficace dans certain cas.

Le but de ce chapitre est ainsi d'analyser la complexité des deux approches dans le cadre des formules intervenant lors d'une multiplication de point sur une courbe elliptique. En général il est plus intéressant d'utiliser une représentation classique pour effectuer une multiplication modulaire, cependant les formules de composition de points sur une courbe elliptique sont des expressions plus complexes et font intervenir plusieurs sommes de produits. Cela nous amène à étudier les expressions de la forme $\sum_{i=1}^{n} A_i B_i \mod p$. En effet, un tel calcul requiert en pratique n multiplications, $n-1$ additions et une réduction modulaire. Le coût d'un multiplication en RNS étant bien moindre que celui en représentation classique, il paraît assez naturel d'espérer que cela suffise pour compenser le sur-coût de la réduction modulaire en RNS vis-à-vis de celle en représentation classique.

Dans ce chapitre, nous allons tout d'abord faire une étude des complexités des différentes méthodes de multiplication modulaire afin de les comparer dans le cadre des sommes de produits modulaires et nous finirons par voir comment l'on peut modifier les formules de compositions de points afin de tirer partie des propriétés de la multiplication modulaire en RNS.

8.1 Problèmes de multiplication et de réduction

S'agissant du calcul modulaire les cas de l'addition et de la multiplication sont très différents. En effet une série de plusieurs additions impliquent une augmentation relativement lente de la taille des opérandes, c'est pourquoi l'on peut se permettre en pratique d'effectuer plusieurs additions avant d'effectuer une réduction modulaire. Par contre une multiplication entre deux nombres de l bits conduit à un résultat codé sur $2l$ bits, ce qui signifie que l'on ne peut pas se permettre, en pratique, d'effectuer plusieurs multiplications avant une réduction modulaire. D'une part, cela requiert une grande capacité de

stockage pour pouvoir conserver les résultats partiels et, d'autre part, en ce qui concerne le RNS, le coût d'une réduction modulaire ne croit en général pas linéairement avec la taille du nombre mais plutôt de manière quadratique. Dans le cas des formules d'addition de points, les multiplications sont, la plupart du temps, suivies d'additions. C'est la raison pour laquelle étudier les sommes de produits apparaît comme pertinent.

Coût de la multiplication

Concernant la multiplication de grands nombres, plusieurs algorithmes existent déjà. Cela dit, dans la mesure où nous nous plaçons dans le contexte de la cryptographie basée sur les courbes elliptiques, nous allons nous intéresser à des nombres dont la taille va varier de 160 à 512 bits. Si l'on travaille avec des architectures de 32 bits (ou 64 bits)cela veut donc dire que les données manipulées tiendront sur au plus 16 mots machine (ou 8). A partir de là, nous allons donc restreindre notre étude à l'algorithme de multiplication scolaire en nous basant sur les recommandations faites pour la librairie GMP [GMP06]. Le tableau 8.1 récapitule les seuils pour lesquels les algorithmes de Karatsuba puis de Tom et Cook deviennent plus rentables, le tout en fonction du type d'architecture de processeur et de la taille des données manipulées.

Architecture	taille de mot	Karatsuba	Tom-Cook
AMD K7	32	26 (832)	202
Pentium 4	32	18 (576)	139
PowerPC 32	32	20 (640)	226
SPARC v7 32	32	8 (256)	466
PowerPC 64	64	8 (512)	57 (3648)
ultrasparc-II	64	22 (1408)	98
IA64	64	47 (3008)	288

Tab. 8.1 – Seuils d'utilisation des algorithmes de Karatsuba et Tom-Cook selon différentes architectures

Dans ce chapitre, nous considérons que l'architecture utilise des entiers codés sur k bits, k pouvant valoir 16, 32 ou 64. Nous considérons également l'écriture en base β des grands entiers

$$A = \sum_{i=0}^{n-1} a_i \beta^i,$$

où $\beta = 2^k$.

Multiplication scolaire

C'est la méthode la plus simple que l'on puisse mettre en oeuvre pour effectuer une multiplication ; elle est donné par l'algorithme 8.1. A chaque étape $\overline{P}$ représente les nk

bits de poids fort de $a_i \times B$ et $\underline{P}$ les k bits de poids faible.

Algorithme 34 : Multiplication scolaire

Données : $A = \sum_{i=0}^{n-1} a_i \beta^i$ et $B = \sum_{i=0}^{n-1} b_i \beta^i$
Résultat : P tel que $P = A \times B$
$P \leftarrow 0$ pour $i = n-1 \ldots 0$ faire
  $(\overline{P}, \underline{P}) \leftarrow a_i \times B$
  $P \leftarrow (P + \overline{P})\beta + \underline{P}$
fin
retourner P

Une étude rapide de cet algorithme classique permet de voir qu'il requiert n^2 produits d'entiers de k bits et $n^2 - n$ additions de k bits.

Réduction modulaire

Le cas des pseudo-Mersenne

Dans ses recommandations le NIST suggère de choisir des moduli de type pseudo-Mersenne, c'est-à-dire des entiers vérifiant :

$$N = \beta^n - c, \text{ avec } c < \beta^{\frac{n}{2}}, \text{ et } w_H(c) < t$$

où w_H représente le poids de Hamming et t est un petit nombre.

La réduction peut alors s'effectuer grâce à trois additions et deux produit par c. Si c est composé de $w_H(c)$ chiffres non nuls alors une telle multiplication peut s'effectuer en $w_H(c)$ additions.

Prenons $X < \beta^{2n}$ alors

$$X = X_1\beta^n + X_0 \equiv c \times X_1 + X_0 = X' \tag{8.1}$$

Maintenant, $X' < \beta^{\frac{3}{2}n}$, on recommence alors le processus

$$X' = X'_1\beta^n + X'_0 \equiv c \times X'_1 + X'_0 = X" \tag{8.2}$$

.

Ainsi, on obtient $X'' < 2\beta^n$, une dernière réduction peut alors être nécessaire et dans ce cas il suffit de considérer r le bit de poids β^n :

- si $r = 1$ alors $X \mod p = (X'' + c) \mod \beta^n$,
- sinon $\widehat{X} = (X'' + c) \mod \beta^n$ et $r' = (X'' + c)$ div β^n, si $r' = 1$ alors $X \mod p = \widehat{X}$ sinon $X \pmod p = X''$.

En résumé le coût de cette réduction est de deux multiplications d'entiers de $\frac{n}{2}$ mots par des entiers de n mots et de trois additions sur n mots. Si t est petit le coût alors inférieur à $3 + 2t$ additions sur n mots.

On peut trouver dans la littérature de nombreuses approches concernant cette classe de nombre [CH03, Sol99].

Via l'algorithme de Montgomery

Nous avons déjà vu qu'un algorithme de réduction efficace a été proposé par Montgomery [Mon85]. Cet algorithme a l'avantage d'être un algorithme généraliste dans la mesure où il ne dépend pas de l'écriture du modulo en question.

Nous avons déjà vu l'algorithme de Montgomery dans le chapitre ??, nous donnons maintenant sa version itérative :

Algorithme 35 : Montgomery(R)

Données : $R = \widetilde{X} \times \widetilde{Y} < p^2 < \beta^{2n}$ et $\beta^{n-1} \leq p < \beta^n$
ainsi qu'une valeur pré-calculée $(-n_0^{-1} \mod \beta)$
Résultat : $R' = R\beta^{-n} \mod p < 2p$
début
  $R' \leftarrow R$
  pour $i = 0 \ldots n-1$ faire
1   $q \leftarrow r'_0 \times n_0^{-1} \mod \beta$
2   $R' \leftarrow (R' + qN)/\beta$
  fin
  retourner R'
fin

Cet algorithme est l'un des plus utilisés en pratique [Mon85, BGV94, SSC04]. C'est un algorithme particulièrement intéressant pour des architectures généralistes qui ne sont pas dédiées à un seul modulo.

Sa complexité est de $n^2 + n$ produits et $2n(n-1)$ additions sur des mots.

Cas du RNS

La manière la plus efficace d'effectuer une réduction modulaire RNS est d'utiliser une version adaptée de l'algorithme de réduction de Montgomery [BIK01]. Nous avons déjà décrit cet algorithme dans le chapitre 5, c'est pourquoi nous rappelons juste qu'une multiplication modulaire en RNS est constituée d'une multiplication classique (soit $2n$ multiplications modulo les m_i et m'_i) et d'une réduction constituée de deux changement de bases. Dans le chapitre 5, nous avons vu qu'il était possible d'effectuer cette réduction en utilisant soit en passant par la représentation MRS, soit la formule de reconstruction d'un entier à partir de sa représentation RNS. Bien que nous ayons proposé une manière d'optimiser les bases bases lors de l'utilisation du MRS, l'approche la plus efficace reste quand même celle décrite dans ??.

Le coût total de cette approche est de $2n^2+8n+1$ produits sur des mots et $2n^2+2n-1$ additions modulaires sur des mots. Ainsi, la complexité est de $2n^2 + 8n + 1$ produits et $2n^2 + 2n - 1 + a(2n^2 + 6n)$ additions sur des mots.

Méthode	produits sur des mots	additions sur de mots
Scolaire	n^2	$2n^2 - n$
Pseudo Mersenne	0	$(3 + 2t)n$
Montgomery	$n^2 + n$	$2n(n - 1)$
RNS multiplication	$2n$	$2an$
RNS Montgomery	$2n^2 + 8n + 1$	$2n^2 + 2n - 1 + a(2n^2 + 6n)$

Tab. 8.2 – Coûts de différentes méthodes de multiplication et réduction en nombre de multiplications sur des mots

8.2 Étude des sommes de produits modulaires

Dans cette section, nous comparons les différents coûts des approches décrites plus haut en termes de multiplications sur des mots. Pour cela, les complexités des différentes méthodes de multiplications et de réductions sont récapitulées dans le tableau 8.2. Afin de simplifier l'étude, nous ne prenons pas en compte les additions, dans la mesure où ces dernières sont bien moins coûteuses que les multiplications.

Formules du type $\sum_{i=1}^{s} A_i B_i \bmod p$

Lorsque l'on étudie la complexité des formules de compositions de points sur une courbe elliptique, ou plus généralement la complexité des algorithmes de multiplication par un scalaire, celle-ci est toujours donnée en nombre d'opérations sur le corps de base. On se restreint d'ailleurs en général à ne compter que les multiplications et les inversions modulaires, l'additions étant en pratique négligeable. Une telle comptabilité sous-entend que chaque opération modulaire est effectué indépendamment des autres. Or, ce n'est pas le cas en pratique, par exemple le calcul de l'expression $AB + CD \mod p$ s'effectue en pratique via deux multiplications et une seule réduction au lieu de deux multiplications modulaires. Cependant cela ne pose pas de problèmes dans la mesure où tous les algorithmes de multiplications modulaires en représentation classique utilise le même algorithme de multiplication ; la seule différence étant l'algorithme de réduction.

Avec la représentation RNS une telle distinction devient par contre critique. En effet en RNS la multiplication est une opération très efficace et seule la phase de réduction est vraiment gourmande en calcul, alors que c'est l'exact opposé avec les représentations classiques.

Afin de voir à quel point cette distinction peu influencer les performances d'un système, nous allons étudier les formules du type

$$\sum_{i=1}^{s} A_i B_i \mod p.$$

Ainsi, la complexité en terme d'opérations est de s produits, s additions et une réduction finale.

Le tableau 8.3 permet de résumer le coût de calcul d'une telle expression en fonction de s et de la méthode de calcul choisie.

Méthode	# produits
Pseudo Mersenne	$(s) \times n^2$
Montgomery	$(s+1) \times n^2 + n$
RNS	$(2s+8) \times n + 2n^2 + 1$

Tab. 8.3 – Coût de calcul pour l'évaluation de $\sum_{i=1}^{s} A_i B_i \bmod N$ en nombre de multiplication sur des mots

Nous comparons maintenant le RNS avec les deux autres approches afin de déterminer à partir de quelles conditions celui-ci devient plus avantageux.

Commençons par l'approche utilisant les pseudo Mersenne :

$$(2s+8) \times n + 2n^2 + 1 < s \times n^2 \tag{8.3}$$

$$s > \frac{2n^2 + 8n + 1}{n^2 - 2n} \tag{8.4}$$

Ainsi dès lors que $s = 3$ le RNS devient plus intéressant tant que $15 \leq n$. Si $9 \leq n \leq 14$ alors il faut que l'on ai au moins $s = 4$.

Maintenant, si nous comparons le RNS avec la méthode basée sur la réduction de Montgomery nous obtenons :

$$(2s+8) \times n + 2n^2 + 1 < (s+1) \times n^2 n \tag{8.5}$$

$$s > \frac{n^2 + 7n + 1}{n^2 - 2n} \tag{8.6}$$

Le RNS devient plus avantageux si $s = 2$ et $n \geq 12$, $s = 3$ et $7 \leq n \leq 11$ ou $s = 4$ et $n = 6$.

Application aux courbes elliptiques

Soit p un nombre premier et $E : y^2 = x^3 + ax + b$ une courbe elliptique définie sur $\mathbb{F}_p$. Soient $P = (X_p, Y_p, Z_p)$ et $Q = (X_q, Y_q, Z_q) \in E(\mathbb{F}_p)$ donnés en coordonnées projectives. Nous rappelons ici les formules de Brier et Joye, généralisant les formules de Montgomery à n'importe quelle courbe.

Nous supposons que la la différence $P - Q = (x, y)$ est connue, dans ce cas

$$\begin{aligned} X_{p+q} &= -4bZ_pZ_q(X_pZ_q + X_qZ_p) + (X_pX_q - aZ_pZ_q)^2, \\ Z_{p+q} &= x\left((X_pZ_q + X_qZ_p)^2 - 4X_pX_qZ_pZ_q\right), \\ X_{2p} &= \left(X_p^2 - aZ_p^2\right)^2 - 8bX_pZ_p^3, \\ Z_{2p} &= 4X_pZ_p\left(X_p^2 + aZ_p^2\right) + 4bZ_p^4. \end{aligned}$$

Nous proposons de calculer X_{p+q} et Z_{p+q} en utilisant les opérations suivantes :

$$1.\ \alpha = Z_pZ_q \qquad 2.\ \beta = X_pZ_q + X_qZ_p \qquad 3.\ \gamma = X_pX_q \qquad 4.\ \delta = -4b\alpha$$
$$5.\ X_{p+q} = \beta\delta + (\gamma - a\alpha)^2 \qquad 6.\ \epsilon = \beta^2 - 4\alpha\gamma \qquad 7.\ Z_{p+q} = x\epsilon$$

Pour calculer X_{2p} et Z_{2p} nous effectuons les opérations suivantes :

$$1.\ \alpha = Z_p^2 \qquad 2.\ \beta = 2X_pZ_p \qquad 3.\ \gamma = X_p^2 \qquad 4.\ \delta = -4b\alpha$$
$$5.\ X_{2p} = \beta\delta + (\gamma - a\alpha)^2 \qquad 6.\ Z_{2p} = 2\beta(\gamma + a\alpha) - \alpha\delta$$

Pour effectuer une multiplication de points avec ces formules nous utilisons l'échelle de Montgomery qui fait intervenir 17 multiplications et 13 réductions à chaque étape (une addition et un doublement de points).

Ainsi, chaque étape de l'algorithme requiert $18(2n)+13(2n^2+8n)$ opérations en RNS, $17n^2+14(n^2+n)$ en utilisant la multiplication de Montgomery et $17n^2$ si p est un pseudo Mersenne.

$\lvert p\rvert_2$	mot	RNS	Montgomery	Mersenne
160	5	1350	845	425
192	6	1776	1200	612
256	8	2784	2096	1088
320	10	4000	3240	1700
512	16	8896	8160	4352

Tab. 8.4 – Nombre de multiplication sur des mots pour une étape de l'algorithme de Montgomery

Nous montrons que le RNS reste plus lent que les deux autres méthodes tant que l'on reste sur des tailles standards (160-192 bits). Néanmoins nous pouvons remarquer que :

- premièrement l'approche RNS est meilleure asymptotiquement que l'approche utilisant l'algorithme de Montgomery,
- deuxièmement le principal problème tient au fait que ces formules ne font intervenir que quatre coordonnées indépendantes, ce qui limite fortement la possibilité

d'obtenir de longues sommes de produits. Cela signifie qu'une telle approche aurait probablement tout intérêt à être étendue aux courbes de genre supérieur (courbes hyperelliptiques de genre 2 par exemple) où les formules font intervenir un nombre de coordonnées bien supérieur (douze en genre 2).

Conclusion

Dans ce manuscrit, nous nous sommes intéressés au problème de l'implantation de la multiplication de point par un scalaire sur les courbes elliptiques définies sur des corps premiers. Nous avons abordé ce problème aussi bien au niveau des algorithmes de multiplication de points, que de l'arithmétique de la courbe ou du corps sous-jacent. L'originalité des travaux présentés ici est qu'ils ne traitent pas de chaque aspects séparément. En effet nous avons toujours cherché à développer l'arithmétique à un niveau donné en gardant à l'esprit son lien avec les niveaux inférieurs ou supérieurs.

Concernant l'arithmétique des courbes elliptiques, nous avons, dans un premier temps, proposé de nouvelles formules d'addition de points, dans un contexte particulier. À partir de ces formules, nous avons pu établir une approche originale du problème de la multiplication de point par un scalaire. En effet, la méthodologie habituelle consiste à trouver un morphisme de la courbe facilement calculable, puis à réécrire le scalaire dans la base de ce morphisme. Si c'est bien cette approche qui a été la notre au début, avec l'utilisation de la représentation de Zeckendorf et la description de algorithme dit de « Fibonacci » et addition, nous avons rapidement dérivé vers une approche basée sur les chaînes d'additions. A ce stade notre volonté était de décorréler la chaînes de calculs à effectuer pour calculer le scalaire de son écriture (binaire ou autre) ; ceci afin d'exploiter au mieux les formules d'addition de points. Nous avons pu voir que ces formules étaient particulièrement adaptées multiplication de points utilisant des algorithmes basées sur les chaînes d'additions euclidiennes. Ceux-ci permettent, dans les meilleurs cas, d'effectuer des multiplication de points aussi efficace que les meilleurs algorithmes existants dans le cadre classique ; avec l'avantage de ne pas nécessiter de précalculs. De plus le fait de décorréler les calculs de l'écriture du scalaire, ainsi que la structure particulière de notre opérateur NewADD, ont permis de rendre l'implantation de ces algorithmes de multiplication de points beaucoup plus résistants aux attaques par canaux cachés. L'inconvénient est qu'il est très difficile de trouver des chaînes efficaces, c'est-à-dire courtes, dans un temps raisonnable.

Afin de contourner ce problème, nous avons proposé d'étudier une classe de chaînes d'additions plus large que les chaînes euclidiennes : les chaînes d'additions différentielles. Ces dernières, en effet, autorisent une plus large flexibilité quant à leurs constructions. Combinées aux formules d'additions de points proposées dans ce mémoire, nous avons pu introduire la notion de chaînes quasi-différentielles. Nous avons, ensuite, proposé un algorithme de multiplication de points aussi efficace que les meilleurs algorithmes existants, à exigence de mémoire équivalente.

Après l'étude au niveau de l'arithmétique des courbes elles-mêmes, nous nous sommes intéressés à l'arithmétique du corps sous-jacent. Plutôt que d'essayer de proposer une nouvelle arithmétique sur les corps finis, nous avons proposé l'adaptation d'un système de représentation des nombres, le RNS, ayant déjà fait ses preuves quant à la protection des protocoles cryptographiques, dans le cadre de RSA, aux attaques par canaux cachés. Le but a été, une fois encore, non pas de d'optimiser au maximum les calculs en RNS, mais d'avoir une réflexion globale sur les améliorations que l'on pouvait apporter dans le cadre des courbes elliptiques. C'est notamment l'objet des travaux effectués dans le cadre de l'inversion modulaire. Opération pour laquelle il n'existait pas d'équivalent en RNS et nécessaire à l'arithmétique des courbes elliptiques. C'est également le cas concernant notre étude des sommes de produits en RNS. Là encore nous avons cherché non pas à améliorer cet opération en temps que telle, mais à tirer partie des propriétés du RNS en reformulant les équations d'addition de points. En effet, la multiplication modulaire en RNS est caractérisée par un multiplication peu coûteuse mais une réduction modulaire qui l'est beaucoup plus, alors que c'est le contraire dans la plupart des autres systèmes de représentation. Nous avons donc remanier les formules de compositions de points afin de factoriser, au maximum, les réductions modulaires, au lieu des multiplications. Cela a permis de montrer que l'utilisation RNS permettait d'améliorer le coût d'une multiplication de points, au moins asymptotiquement, par rapport à l'utilisation de l'algorithme de multiplication modulaire de Montgomery.

Bibliographie

[ACS04] R. Avanzi, M. Ciet, and F. Sica. Faster scalar multiplication on koblitz curves combining point halving with the frobenius endomorphism. In Public Key Cryptography, volume 2947 of LNCS, pages 28–40. Springer, February 2004.

[ADDS06] R. Avanzi, V. Dimitrov, C. Doche, and F. Sica. Extending scalar multiplication using double bases. In ASIACRYPT, volume 4284 of LNCS, pages 130–144. Spinger, 2006.

[ADMRK02] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman. A new addition formula for elliptic curves over GF(2 n). IEEE Transactions on Computers, 51 :972–975, 2002.

[Bar86] P. Barrett. Implementing the rivest, shamir and adleman public key encryption algorithm on a standard digital processor. In A. M. Odlyzko, editor, Advances in Cryptology, Proceedings of Crypto'86, pages 311–323, 1986.

[Ber06] D.J. Bernstein. Differential addition chains, 2006. Available at cr.yp.to/ecdh/diffchain-20060219.pdf.

[BGV94] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In CRYPTO, volume 773 of LNCS, pages 175–186, 1994.

[BI04] V. Berthé and L. Imbert. On converting numbers to the double-base number system. In Advanced Signal Processing Algorithms, Architecture and Implementations XIV, volume 5559 of Proceedings of SPIE, pages 70–78. SPIE, 2004.

[BIK01] J.C. Bajard, L. Imbert, and P. Kornerup. Modular multiplication and base extension in residuenumber systems. In 15th IEEE Symposium on Computer Arithmetic, pages 59–65. IEEE Computer Society Press, 2001.

[BILT04] J.C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In Cryptographic Hardware and Embedded Systems, (CHES), pages 62–75, 2004.

[Bir06] P. Birkner. Efficient divisor class halving on genus two curves. In Selected Area in Cryptography (SAC), 2006.

[BJ02] E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In Public Key Cryptography, (PKC), volume 2274 of LNCS, pages 335–345, 2002.

[Ble96] D. Bleichenbacher. Efficiency and Security of Cryptosystems Based on Number Theory. PhD thesis, ETH Zürich, 1996.

[Boo51] A. Booth. A signed binary multiplication technique. The Quarterly Journal of Mechanics and Applied Mathematics, 4 :236–240, 1951.

[Cap88] R.M. Capocelli. A generalization of fibonacci trees. In Third In. Conf. on Fibonacci Numbers and their Applications, 1988.

[CF06] H. Cohen and G. Frey, editors. Handbook of Elliptic and Hyperelliptic Cryptography. Chapman & Hall, 2006.

[CH03] J. Chung and A. Hasan. More generalized mersenne numbers. In Selected Area in Cryptography, LNCS, pages 335–347, 2003.

[CH07] J. Chung and M. A. Hasan. Low-weight polynomial form integers for efficient modular multiplication. IEEE Trans. Computers, 56(1) :44–57, 2007.

[CMO98] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In ASIACRYPT, LNCS. Springer, 1998.

[CN99] R. Conway and J. Nelson. Fast converter for 3 moduli rns using new property of crt. IEEE Transactions on Computers, 48(8) :852–860, 1999.

[DI06] C. Doche and L. Imbert. Extended double-base number system with applications to elliptic curve cryptography. In INDOCRYPT, volume 4329 of LNCS, pages 335–348. Springer, 2006.

[DIM07] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. Mathematics of Computations, 2007. à paraître.

[DJM98] V. S. Dimitrov, G. A. Julien, and W. C. Miller. An agorithm for modular exponentiation. Information Procesing Letters, 66 :155–159, May 1998.

[FHLM04] K. Fong, D. Hankerson, J. Lopez, and A. Menezes. Field inversion and point halving revisited. IEEE Transactions on Computers, 53(8) :1047–1059, august 2004.

[FLM03] K. Fong, D. Hankerson J. López, and A. J. Menezes. Field inversion and point halving revisited. Technical report, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2003.

[FS07] C. Frougny and V. Steiner. Minimal weight expansions in some pisot bases. Technical report, Univ. Paris 8, Univ. Paris Diderot, 2007.

[GMP06] The gnu multiple precision arithmetic library, May 2006.

[Heu04] C. Heuberger. Minimal expansions in redundant number systems :fibonacci bases and greedy algorithms. Period. Math. Hungar., 49 :65–89, 2004.

[HLHM00] D. Hankerson, J. Lòpez Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In Cryptographic Hardware and Embedded Systems, (CHES), volume 1965 of LNCS, pages 1–24. Springer, 2000.

[HMV04] D. Hankerson, A. Menezes, and S. Vanstone. Guide to Elliptic Curve Cryptography. Springer, 2004.

[HP94] C. Hung and B. Parhami. An approximate sign detection method for residue numbers and its application to RNS division. Computers and Mathematics with Applications, 27(4) :23–35, Feb. 1994.

[HT00] A. Higuchi and N. Takagi. A fast addition algorithm for elliptic curve arithmetic in gf(2n) using projective coordinataes. Inf. Process. Lett., 76(3) :101–103, 2000.

[JY00] M. Joye and S.-M. Yen. Optimal left-to-right binary signed-digit recoding. IEEE Transactions on Computers, 49(7) :740–748, 2000.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In Advances in cryptology - CRYPTO, volume 1666 of LNCS, pages 388–397. Springer, 1999.

[KKT05] I. Kitamura, M. Katagi, and T. Takagi. A complete divisor class halving algorithm for hyperelliptic curve cryptosystems of genus two. In Information Security and Privacy (ACISP), volume 3574 of LNCS, pages 146–157. Springer, 2005.

[Knu73] D. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley, 1973.

[Knu81] D. Knuth. Seminumerical Algorithms, volume 2 of The Art of Computer Programming. Addison-Wesley, 2 edition, 1981.

[Knu99] E.W. Knudsen. Elliptic scalar multiplication using point halving. In ASIACRYPT, volume 1716 of LNCS, pages 135–149. Springer, 1999.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. Mathematics of Computation, 48 :203–209, 1987.

[Kob92] N. Koblitz. Cm-curves with good cryptographic properties. In Advances in Cryptology - CRYPTO, volume 576 of LNCS, page 279. Springer, February 1992.

[Koc96] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Advances in cryptology - CRYPTO, volume 1109 of LNCS, pages 104–113. Springer, August 1996.

[KR04] B. King and B. Rubin. Improvements to the point halving algorithm. In Information Security and Privacy (ACISP), volume 3108 of LNCS, pages 262–276. Springer, 2004.

[KY75] D. Knuth and A. Yao. Analysis of the subtractive algorithm for greater common divisors. Proc. Nat. Acad. Sci. USA, 72(12) :4720–4722, December 1975.

[Lan04] T. Lange. A note on lópez dahab coordinates. Technical report, Technical university of Denmark, 2004.

[LD98] J. Lopez and R. Dahab. Improved algorithms for elliptic curve arithmetic in GF(2 n). In Selected Areas in Cryptography, SAC, volume 1556 of LNCS, pages 201–212. Springer, 1998.

[LD99] J. Lopez and R. Dahab. Fast multiplication on elliptic curves over GF(2 m) without precomputation. In Cryptographic Hardware and Embedded Systems, (CHES), volume 1717 of LNCS, pages 316–327. Springer, 1999.

[Mil86] V.S. Miller. Uses of elliptic curves in cryptography. In Advaces in Cryptology-CRYPTO, volume 218 of LNCS, pages 417–428. Springer, 1986.

[Mon83] P. Montgomery. Evaluating Recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, 1983. Available at ftp.cwi.nl :/pub/pmontgom/Lucas.ps.gz.

[Mon85] P. Montgomery. Modular multiplication without trial division. Mathematics of Computation, 44 :519–521, 1985.

[Mon87] P. Montgomery. Speeding the pollard and elliptic curve methods of factorization. Mathematics of Computation, 48 :243–264, 1987.

[MWZ96] A. J. Menezes, Y. H. Wu, and R. J. Zuccherato. An elementary introduction to hyperelliptic curves. Technical report, Departement of C&O, University of Waterloo, 1996.

[PB04] B. Phillips and N. Burgess. Minimal weight digit set conversions. IEEE Transactions on Computers, 53(6) :666–677, 2004.

[Pie94] Stanislaw J. Piestrak. Design of high-speed residue-to-binary number system converter based on chinese remainder theorem. In ICCD, pages 508–511, 1994.

[PP95] K. C. Posch and R. Posch. Modulo reduction in residue number systems. IEEE Transaction on Parallel and Distributed Systems, 6(5) :449–454, 1995.

[QS01] J.J. Quisquater and D. Samyde. Electromagnetic analysis (ema) : Measures and counter-measures for smart cards. In Proceedings of the International Conference on Research in Smart Cards : Smart Card Programming and Security, volume 2140 of LNCS, pages 200–210. Springer, 2001.

[Sch00] W. Schindler. A timing attack against rsa with the chinese remainder theorem. In Cryptographic Hardware and Embedded Systems (CHES), volume 1965 of LNCS, pages 109–124. Springer, 2000.

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. In Eurocrypt, volume 1233 of LNCS, pages 256–266, 1997.

[Sil86] J.H. Silverman. The Arithmetic of Ellitpic Curves. Springer, 1986.

[SK89] A. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. IEEE Transactions on Computer, 38(2) :292–296, 1989.

[Sol99] J. Solinas. Generalized mersenne numbers. Technical report, University of Waterloo, 1999.

[SSC04] M. Sanu, C. Swartzlander, and C. Chase. Parallel montgomery multipliers. In ASAP, pages 63–72, 2004.

[ST67] N. S. Szabo and R. I. Tanaka. Residue Arithmetic and its Applications to Computer Technology. McGraw-Hill, 1967.

[Tsu01] Y. Tsuruoka. Computing short lucas chains for elliptic curve cryptosystems. IEICE Transactions on Fundamentals, E84-A :1227–1233, 2001.

[Vor02] N. Vorobiev. Fibonacci Numbers. Birkhäuser, 2002.

[Zec72] E. Zeckendorf. Représentations des nombres naturels par une somme de nombre de fibonacci ou de nombres de lucas. Bulletin de la Société Royale des Sciences de Liège, pages 179–182, 1972.