Contribution de l'arithmétique des ordinateurs aux implémentations résistantes aux attaques par canaux auxiliaires

Fangan Yssouf Dosso

UNIVERSITÉ DE TOULON

MER ET SCIENCES
ÉCOLE DOCTORALE 548

École doctorale 548 - Mer et Sciences

THÈSE présentée par :

Fangan Yssouf DOSSO

Pour obtenir le grade de

Docteur de l'Université de Toulon

Spécialité : Informatique

Contribution de l'arithmétique des ordinateurs aux implémentations résistantes aux attaques par canaux auxiliaires

Soutenue le mardi 02 juin 2020 devant le jury composé de :

M. DIDIER Laurent-Stéphane	Professeur, Université de Toulon	Directeur de thèse
Mme EL MRABET Nadia	Maître assistant HDR, EMSE Gardanne	Invitée
M. LERCIER Reynald	Ingénieur de l'armement HDR, DGA-MI, Bruz	Rapporteur
M. NEGRE Christophe	Maître de conférences HDR, Université de Perpignan	Examinateur
M. RENAULT Guenael	Professeur, École Polytechnique, Palaiseau	Examinateur
M. TISSERAND Arnaud	Directeur de recherche CNRS, Lab-STICC, Lorient	Rapporteur
M. VÉRON Pascal	Maître de conférences, Université de Toulon	Co-encadrant de thèse

Remerciements

Mes premiers remerciements s'adressent à Laurent-Stéphane DIDIER et Pascal VÉRON qui ont encadrés mon travail de thèse. Je les remercie de m'avoir donné l'élan qu'il fallait pour bien entamer ma thèse. Merci également à eux pour leur confiance et leur soutien durant toutes ces années.

Merci à Reynald LERCIER et Arnaud TISSERAND d'avoir accepté de rapporter cette thèse. Je leur suis très reconnaissant de m'avoir fait l'honneur de relire ce manuscrit et pour les rapports qu'ils ont faits. Un grand merci à Nadia EL MRABET, Christophe NEGRE et Guenael RENAULT pour avoir bien voulu faire partie de ce jury de qualité.

Je remercie l'ensemble des membres du laboratoire IMATH et tout particulièrement ceux de l'équipe Informatique et Algèbre Appliquée qui m'ont bien des fois aidés et donnés des conseils à la fois pour mes recherches et activités d'enseignement. Je tiens également à remercier l'ensemble des doctorants des laboratoires IMATH et COSMER que j'ai eu à côtoyer durant ces années de thèse.

Enfin, mes remerciements vont à mes parents sans qui rien n'aurait été possible. Merci en particulier à mon frère qui n'a cessé d'être là pour moi.

Table des matières

Notations et conventions

Notations

Ensembles

$\#\mathcal{A}$: Cardinal de $\mathcal{A}$, avec $\mathcal{A}$ un ensemble.

$\langle g \rangle$: Sous-groupe de $\mathcal{D}$ engendré par $g \in \mathcal{D}$, où $\mathcal{D}$ est un groupe.

$\mathbb{Z}[X]$: Ensemble des polynômes à coefficients dans $\mathbb{Z}$.

$\mathbb{Z}_{n-1}[X]$: Ensemble des polynômes de $\mathbb{Z}[X]$ de degrés strictement inférieurs à n : $\mathbb{Z}_{n-1}[X] = \{C \in \mathbb{Z}[X], \text{ tel que : } \deg(C) < n\}$.

$\mathbb{F}_m$: Corps finis à m éléments, où m est un nombre premier ou une puissance d'un nombre premier.

Nombres

$|x|$: Valeur absolue de x.

$\lfloor x \rfloor$: Partie entière de x.

$\lceil x \rceil$: Partie entière supérieure de x.

$\lfloor x \rceil$: Entier le plus proche de x.

Normes

$\| \cdot \|$: Norme euclidienne.

$\| \cdot \|_\infty$: Norme infinie.

$\| \cdot \|_1$: Norme une.

$\langle \cdot \,|\, \cdot \rangle$: Produit scalaire.

Abréviations

MNS : Modular Number System, cf. définition 1.4.

AMNS : Adapted Modular Number System, cf. définition 1.10.

PMNS : Polynomial Modular Number System, cf. définition 1.11.

EAC : Euclidean Addition Chain, cf. définition 4.3.

ECC : Elliptic Curve Cryptography.

ECSM : Elliptic Curve Scalar Multiplication.

Conventions

- Selon le contexte, le symbole 0 représentera soit l'entier $0 \in \mathbb{N}$, soit le vecteur $(0, \ldots, 0) \in \mathbb{R}^n$.

- Si $v \in \mathbb{Z}^n$ est un vecteur, on supposera que $v = (v_0, \ldots, v_{n-1})$.
  Si $V \in \mathbb{Z}_{n-1}[X]$ est un polynôme, on supposera que $V(X) = v_0 + v_1 X + \cdots + v_{n-1} X^{n-1}$. De plus, nous nous permettrons de considérer indifféremment v ou V selon le contexte, l'équivalence étant évidente.

Liste des algorithmes

Liste des tableaux

Liste des définitions et théorèmes

Introduction

Avec le développement rapide des réseaux de communication numériques ainsi que le nombre toujours grandissant d'objets connectés, la sécurité numérique n'a cessé d'occuper une place essentielle dans de nombreux domaines allant du militaire au bancaire en passant par la protection de la vie privée ainsi que celle des échanges entre différents objets connectés. La cryptographie, qui fut longtemps cantonnée aux sphères militaires et politiques, est aujourd'hui un élément incontournable de la sécurité numérique.

La cryptographie est une science qui permet à des entités d'échanger des messages sans qu'une partie adverse puisse avoir accès à l'information qui est échangée. Elle est utilisée de nos jours pour, entre autres, assurer la confidentialité, l'authenticité ainsi que l'intégrité des données. On distingue deux types de cryptographie. La première, la plus ancienne qui remonte à l'antiquité, est la *cryptographie à clé sécrète* aussi appelée *cryptographie symétrique*. La seconde, qui est beaucoup plus récente, est la *cryptographie à clé publique* aussi appelée *cryptographie asymétrique*.

La principe de la cryptographie à clé sécrète est d'échanger des messages chiffrés à partir d'une clé connue uniquement par les parties concernées. Le chiffrement et le déchiffrement d'un message se font avec la même clé. La cryptographie symétrique remonte jusqu'à 2000 ans avant J.-C. en Égypte, où l'on a des traces de son utilisation. Deux standards ont été proposés pour la cryptographie symétrique. Ce sont le Data Encryption Standard (DES) [FIP99], standardisé en janvier 1977 et déclaré obsolète en octobre 1999, et l'Advanced Encryption Standard (AES) [FIP01], standardisé en décembre 2001 par le NIST (National Institute of Standards and Technology). La principale limite de la cryptographie symétrique est le problème (assez paradoxale) de l'échange de clé. En effet, pour pouvoir échanger secrètement des informations, les parties doivent au préalable partager un secret (la clé).

En 1976, Diffie et Hellman [DH76] posent les bases de la cryptographie à clé publique afin de s'affranchir du problème d'échange de clés. Son principe est d'utiliser non plus une même clé pour le chiffrement et le déchiffrent mais une paire de clés : une clé publique, qui peut être connue de tous, pour le chiffrement et une clé privée, connue uniquement du destinataire, pour le déchiffrement. Le problème d'échange de clé ne se pose donc plus. On se retrouve cependant face à un autre problème qui est la diffusion de la clé publique. Ce problème est résolu par les infrastructures à clés publiques (PKI : Public Key Infrastructure, en anglais). Parmi les standards les plus utilisés de la cryptographie à clé publique, on compte le cryptosystème RSA [RSA78] et le cryptosystème ElGamal [ElG85].

Bien que différentes, la cryptographie symétrique et la cryptographie asymétrique ne sont pas en concurrence mais plutôt complémentaires. En effet, les cryptosystèmes à clé sécrète, qui nécessitent un échange préalable de secret (la clé), sont nettement plus rapides que les cryptosystèmes à clé publique, qui n'ont pas ce problème d'échange de clé. Ainsi, une combinaison typique de ces deux types de cryptosystèmes est d'utiliser un cryptosystème à clé publique pour partager la clé secrète (qui fait quelques centaines de bits) qui servira à chiffrer le message (qui peut être très grand) avec un cryptosystème à clé sécrète.

Les fonctions à sens unique, avec brèche secrète (ou porte dérobée), sont au coeur de la cryptographie asymétrique. Avec une telle fonction, le calcul des images est facile, alors que le calcul des antécédents est extrêmement difficile lorsque la brèche est inconnue. Comme exemples de telles fonctions, on peut citer la fonction exponentielle modulaire, dont la réciproque est le logarithme discret et la multiplication de deux nombres premiers, dont la réciproque est la factorisation d'entiers ; le cryptosystème ElGamal repose sur la première et le cryptosystème RSA sur la seconde.

Avec les progrès effectués dans la domaine de la factorisation et de la résolution du logarithme discret classique, les cryptosystèmes RSA et ElGamal, qui occupaient les premiers rangs parmi les cryptosystèmes à clé publique, font depuis plusieurs années place aux cryptosystèmes basés sur les courbes elliptiques. La cryptographie sur les courbes elliptiques (ECC : Elliptic Curve Cryptography, en anglais) fut introduite par Koblitz [Kob87] et Miller [Mil85]. Le problème sur lequel repose la sécurité de l'ECC est la résolution du logarithme discret sur les courbes elliptiques ; i.e. trouver l'entier k à partir de deux points P et Q d'une courbe, tels que $Q = kP = P + P + \cdots + P$ (k fois). Ce problème demeure jusqu'à présent très difficile (de façon générale). On ne dispose pas d'algorithme de complexité sous-exponentielle pour sa résolution, contrairement au problème de la factorisation ou du logarithme discret sur les corps de nombres. Ainsi, pour un niveau de sécurité équivalent, les cryptosystèmes basés sur les courbes elliptiques requièrent des clés nettement plus petites que RSA ou ElGamal (classique). À titre d'exemple, une clé de 256 bits pour la cryptographie sur les courbes elliptiques est aussi robuste qu'une clé RSA de 3072 bits. Pour un même niveau de sécurité, les protocoles cryptographiques basés sur les courbes elliptiques nécessitent donc moins de mémoire et ont de meilleures performances. Ainsi, la cryptographie sur les courbes elliptiques est particulièrement bien adaptée pour les environnements à ressources (puissance et mémoire) réduites, tels que les cartes à puce.

L'opération principale de l'ECC est la multiplication scalaire (ECSM : Elliptic Curve Scalar Multiplication, en anglais). La multiplication scalaire correspond à l'opération $k\,P$, où k est un entier et P un point d'une courbe. On a :

$$k\,P = \underbrace{P + P + \cdots + P}_{k \text{ fois}}.$$

Le résultat de cette opération est donc un point de la même courbe. Cette opé-

ration est assez proche de l'exponentiation modulaire. En effet, les méthodes *double-and-add* et *square-and-multiply*, deux algorithmes qui permettent respectivement d'effectuer la multiplication scalaire et l'exponentiation modulaire, dérivent de l'algorithme générique (algorithme 1) ci-dessous. Soit $(G, *)$ un groupe commutatif muni de la loi $*$. Notons 1 son élément neutre.

Algorithme 1 *Méthode générique pour l'exponentiation*

Entrée(s) : $a \in G$, $k \in \mathbb{N}$, tel que $k = (k_{n-1}, \ldots, k_0)_2$
Sortie : $s \in G$, tel que $s = a * a * \cdots * a$ (k fois)

```
1: s ← 1
2: for i = n − 1 to 0 do
3:     s ← s ∗ s
4:     if k_i = 1 then
5:         s ← s ∗ a
6:     end if
7: end for
8: retourner s
```

Aussi, certains algorithmes pour effectuer la multiplication scalaire sont de simples modifications d'algorithmes d'exponentiation modulaire (ou inversement). Par exemple, l'*échelle de Montgomery* [Mon87], initialement proposée par Montgomery pour la multiplication scalaire, fut adaptée par Joye et Yen pour l'exponentiation modulaire [JY02].

Le choix de la représentation du scalaire k pour effectuer la multiplication scalaire est essentiel pour l'efficacité de cette opération, car ce choix définit en grande partie le nombre d'additions et de doublements de points qu'il faudra faire pour réaliser cette dernière. Parmi les plus populaires, on compte la représentation binaire, qui est utilisée par la méthode *double-and-add* et l'*échelle de Montgomery* ; on compte également la représentation NAF [Rei60] qui minimise le nombre d'éléments non-nuls dans la représentation, ce qui a pour effet de réduire le nombre d'additions de points lors de la multiplication scalaire. En outre, certains algorithmes utilisent des pré-calculs (basés sur le scalaire k et le point P) pour accélérer cette opération. Selon les ressources (puissance de calcul et mémoire) dont on dispose, certaines représentations peuvent s'avérer plus intéressantes que d'autres.

Certaines courbes possèdent des propriétés qui permettent d'accélérer la multiplication scalaire. On a parmi celles-ci les courbes munies d'endomorphismes efficaces. Sur une telle courbe, il existe (au moins) un endomorphisme ϕ, tel que : pour tout point P de cette courbe, on a $\phi(P) = \lambda P$, avec λ fixé et ϕ efficacement calculable. Ici, efficacement calculable signifie que $\phi(P)$ s'obtient avec très peu d'opérations modulaires ; par exemple, une multiplication modulaire sur les coordonnées de P. La méthode GLV de Gallant et al. [GLV01] utilise ce type de courbe pour réduire à peu près de moitié le coût de la multiplication scalaire. L'utilisation de ces courbes pour accélérer la multiplication scalaire sera abordée dans les chapitres 4 et 5.

En plus de la robustesse des problèmes qui sont au coeur de la sécurité

des cryptosystèmes, l'accent fut longtemps mis sur l'amélioration du temps de calcul et des ressources mémoires, jusqu'à ce que Kocher présente un nouveau type d'attaque [Koc96b] qui passe outre les garanties théoriques qui primaient jusqu'alors. Ces attaques dites par *canaux auxiliaires* (SCA : Side Channel Attacks, en anglais) n'ont pas pour objectif de résoudre le problème théorique sur lequel repose la sécurité du cryptosystème cible ; il s'agit plutôt d'exploiter les fuites d'information (chaleur, consommation d'énergie, temps de calcul, etc.) du système physique qui exécute le cryptosystème afin d'extraire le secret recherché. Avec ces attaques, il n'est donc plus seulement question de vitesse, de ressource mémoire ou de taille de clé, mais aussi de régularité ou de tout autre paramètre qui pourrait être lié à des fuites d'information du matériel. Les mesures supplémentaires qu'il faut désormais prendre pour se prémunir de ces attaques doivent être aussi bien logicielles que matérielles.

La plupart des cryptosystèmes à clé publique nécessitent des opérations modulaires. Pour garantir la robustesse des problèmes sur lesquels reposent la sécurité de ces cryptosystèmes, il est nécessaire que ces opérations modulaires soient faites avec des entiers de (très) grandes tailles. Ainsi, une arithmétique modulaire sûre et efficace sur les grands entiers est indispensable, puisque les cryptosystèmes qui dépendent de cette arithmétique modulaire doivent sûres contre les attaques par canaux auxiliaires et efficaces pour ne pas ralentir les échanges de données.

Contributions

Dans cette thèse, nous nous intéressons d'une part à l'arithmétique modulaire avec de grands entiers et d'autre part à la multiplication scalaire sur les courbes elliptiques (ECSM). L'objectif étant d'effectuer ces opérations de manière sûre et efficace, pour les raisons mentionnées plus haut.

Dans [Mel07], Méloni propose une formule très efficace pour additionner deux points d'une courbe. Cette formule requiert cependant une condition particulière sur les points à additionner. Dans cet article, l'auteur montre que cette formule est particulièrement bien adaptée pour effectuer l'ECSM lorsque le scalaire k est représenté comme une *chaîne d'additions euclidienne* (EAC : Euclidean Addition Chain, en anglais). Une EAC calculant un entier k est définie par une suite finie d'entiers $(v_1, \ldots, v_s)$, telle que :

- $v_1 = 1$, $v_2 = 2$, $v_3 = v_2 + v_1$ et $v_s = k$,
- pour tout entier i tel que $3 \leqslant i < s$, si $v_i = v_{i-1} + v_j$, avec $j < i - 1$, alors $v_{i+1} = v_i + v_{i-1}$ ou $v_{i+1} = v_i + v_j$.

Dans [HLM+10], Herbaut et al. montrent que la multiplication scalaire avec les EAC peut être une alternative sûre et efficace aux méthodes standards pour effectuer la multiplication scalaire, lorsque le point de base P est fixé. Au cours de cette thèse, un travail de généralisation au cas où le point de base varie a été effectué, en utilisant les courbes munies d'un endomorphisme efficace [DHMV18].

Dans [PCRS09], Pontarelli et al. proposent une méthode pour la détection de fautes lors de la multiplication scalaire avec les EAC. Dans cet article, les auteurs présentent une contre-mesure pour un modèle d'attaquant qu'ils définissent. Nous avons amélioré leur proposition en présentant une nouvelle méthode de détection de fautes plus efficace, pour un modèle d'attaquant plus fort [DV17].

L'optimisation de l'arithmétique modulaire peut être faite à deux niveaux. Le premier niveau est arithmétique. Il s'agit, sans changer de système de représentation, de développer des astuces pour minimiser le coût de certaines opérations ou de simplement éviter ces dernières. C'est le cas par exemple des méthodes de Barrett et de Montgomery pour la multiplication modulaire, où on échange les divisions non-triviales contre des opérations nettement moins coûteuses. Le second niveau concerne la représentation des entiers. Il s'agit, dans ce cas, de trouver des systèmes de représentation qui possèdent des propriétés intéressantes pour l'efficacité (et éventuellement la sûreté) des opérations modulaires. Comme exemples de tels systèmes, on a le système de représentation par les restes (RNS) [Gar59] qui transforme les entiers en des restes modulo de petits entiers. L'arithmétique modulaire s'en trouve alors simplifiée, avec d'intéressantes possibilités de parallélisation. Le système de représentation modulaire (MNS : Modular Number System, en anglais) est un autre exemple de tels systèmes. Ce système de représentation fut proposé en 2004 par Bajard et al. [BIP04]. Sa principale caractéristique est que les éléments y sont représentés sous forme de polynômes, avec de petits coefficients. De cette caractéristique découlent plusieurs avantages à la fois pour les performances et la sûreté des opérations arithmétiques dans ce système. Dans cette thèse, nous nous intéressons à l'AMNS (AMNS : Adapted Modular Number System, en anglais), également introduit dans [BIP04]. Un AMNS est un MNS qui possède une propriété particulière permettant de simplifier et d'accélérer les opérations modulaires.

Depuis l'introduction de l'AMNS par Bajard et al., plusieurs travaux ont été faits pour améliorer les performances de ce système [BIP05, NP08] ou l'utiliser pour des opérations plus complexes [EN09, EG12]. Malgré ces travaux, deux questions essentielles demeuraient sans réponse. La première était de savoir si l'AMNS peut être une alternative intéressante aux systèmes de représentation déjà existants, pour l'arithmétique modulaire. La seconde question concernait la génération des paramètres nécessaires pour une utilisation efficace de ce système. En effet, dans [NP08], Negre et Plantard ont présenté une méthode, assez proche de celle de Montgomery, pour la multiplication modulaire dans l'AMNS. Cependant, l'utilisation efficace de cette méthode nécessite un ensemble de paramètres pour lesquels aucun processus de génération n'avait été proposé, excepté un cas particulier traité dans [EG12]. Notre contribution a été d'une part d'apporter des réponses à ces questions et d'autre part de proposer des solutions à d'autres problématiques toutes aussi importantes pour une utilisation efficace de l'AMNS en pratique. Ce travail a fait l'objet d'une publication [DDV20].

Comme mentionné plus haut, il n'est plus question avec les attaques par canaux auxiliaires de se contenter d'optimiser le temps de calcul ou les besoins en

mémoire. Il est également indispensable de mettre en place des contre-mesures contre ces attaques. Pour cela, l'AMNS possède des propriétés très intéressantes. Celles-ci viennent en grande partie du fait que les éléments sont des polynômes dans ce système. La première propriété est qu'il n'y a pas de propagation de retenue entre coefficients lors des opérations arithmétiques. Comme on le verra dans cette thèse, cela permet de développer un ensemble complet d'algorithmes sans branchement conditionnel pour les opérations arithmétiques principales. Ainsi, l'AMNS est intrinsèquement sûr contre les attaques simples de type SPA (Simple Power Analysis). Cette propriété aide également à protéger les cryptosystèmes contre des attaques plus avancées comme celles de type DPA (Differential Power Analysis) [KJJ99]. La seconde propriété de ce système est qu'il est *redondant*. En effet, un même entier peut avoir plusieurs représentations différentes dans l'AMNS. Dans un travail récent [DDEM+19], nous avons exploité cette propriété pour randomiser les opérations arithmétiques au sein de l'AMNS. En outre, nous montrons comment utiliser cette randomisation pour appliquer les contre-mesures classiques suggérées dans la littérature pour protéger l'ECSM contre les attaques de type DPA. Nous montrons également que la randomisation avec l'AMNS protège naturellement la multiplication scalaire contre certaines attaques spécifiques, comme celle de Goubin [Gou03], pour lesquelles il est nécessaire de prendre des mesures supplémentaires lorsqu'on utilise le système de numération positionnel classique.

Plan de la thèse

Le présent document est composé de deux parties. La première partie traite de l'arithmétique modulaire. Nous commençons par un état de l'art de cette dernière dans le chapitre 1. Dans ce chapitre, nous abordons le système de numération positionnel, le système de représentation par les restes (RNS) [Gar59] et plus en détail, les fondements de l'AMNS. Le chapitre 2 porte sur nos contributions pour une arithmétique modulaire efficace dans l'AMNS. Enfin, dans le chapitre 3, nous nous intéresserons à l'utilisation de l'AMNS pour la protection des opérations (telles que l'ECSM) contre les attaques par canaux auxiliaires.

La deuxième partie a pour objet l'arithmétique des courbes elliptiques. Nous commençons par un état de l'art de cette dernière dans le chapitre 4. Dans ce chapitre, nous nous intéresserons particulièrement aux courbes munies d'un endomorphisme efficace. Le chapitre 5 traite de l'utilisation des EAC et des courbes munies d'un endomorphisme efficace pour effectuer la multiplication scalaire efficacement, avec un point de base P quelconque. Enfin, le chapitre 6 présente notre proposition pour la détection de faute lors de la multiplication scalaire en utilisant les EAC.

Première partie

Arithmétique modulaire

Chapitre 1

Arithmétique modulaire, un état de l'art

Sommaire

L'arithmétique modulaire [BZ10] est incontournable pour la plupart des cryptosystèmes à clé publique, tels que les protocoles basés sur les courbes elliptiques ([HMV04], section 4), les algorithmes RSA [RSA78] et ElGamal [ElG85], ou l'échange de clés de Diffie-Hellman [DH76]. Dans ce chapitre, nous effectuons un état de l'art de l'arithmétique modulaire. Nous nous intéresserons tout particulièrement au système de représentation modulaire (MNS), introduit par Bajard et al. en 2004 [BIP04].

Les opérations modulaires principales sont l'addition, la multiplication et l'inversion modulaire. L'addition modulaire est la moins coûteuse de ces opérations. Comme on le verra plus loin, l'inversion modulaire (quand elle est possible) peut être effectuée soit à l'aide de l'algorithme d'Euclide étendu, soit par une exponentiation modulaire. Ainsi, l'opération fondamentale est la multiplication modulaire, car c'est cette dernière qui définit l'essentielle de la complexité calculatoire des cryptosystèmes nécessitant des opérations modulaires.

La multiplication modulaire est composée d'une multiplication simple et d'une réduction modulaire. La réduction modulaire consiste à calculer le reste

de la division euclidienne d'un entier c par un entier p. Pour c et p donnés, on sait qu'il existe deux entiers q et r tels que :

$$c = qp + r, \text{ avec } 0 \leqslant r < p.$$

La réduction modulaire consiste donc à calculer l'entier r.

L'approche triviale qui consiste à effectuer la division euclidienne de c par p pour obtenir r peut s'avérer très coûteuse, en particulier si ces entiers sont grands, ce qui est le cas en cryptographie. Ainsi, plusieurs méthodes ont été proposées pour éviter d'avoir à effectuer cette division à chaque fois que l'on doit faire une réduction modulaire. Certaines exploitent la forme du module p quand cela est possible, on parle alors de modules spéciaux [Cra92, Sol99, CH03]. Pour les modules qui n'ont pas de forme particulière, il existe des méthodes, dites générales, pour effectuer la réduction modulaire sans passer par la division euclidienne. Nous en présentons quelques-unes parmi les plus efficaces dans ce chapitre.

Ce chapitre est organisé comme suit. Nous commençons par présenter les méthodes d'addition, de multiplication et d'inversion modulaire dans le système de numération positionnel. Ensuite, nous présentons le système de représentation par les restes (RNS). Nous terminons par le système de représentation modulaire (MNS).

Soit $p \geqslant 2$ un entier et $n, \beta \in \mathbb{N}$ deux entiers tels que : $\beta^{n-1} \leqslant p < \beta^n$. Sauf mention contraire, on supposera que $\beta = 2$ (et donc la représentation binaire sera considérée). Pour la suite, on s'intéressera aux opérations modulo p.

1.1 Le système de numération positionnel

Dans cette section, nous présentons quelques méthodes pour l'addition, la multiplication et l'inversion modulaires pour les systèmes de numération positionnels.

On notera Add_d et Mul_d respectivement l'addition et le produit de deux entiers de taille d bits (dans la représentation binaire), avec $d \geqslant 1$ un entier. On notera également $C_{algo\ i}$ le coût de l'algorithme i.

1.1.1 L'addition modulaire

L'addition modulaire est composée d'une simple addition et d'une réduction modulaire. Plusieurs propositions ont été faites pour optimiser l'addition modulaire. Nous présentons ici deux méthodes pour effectuer cette opération.

La première méthode est l'approche triviale qui consiste à vérifier si le résultat de la somme est supérieur (ou égal) à p et d'y soustraire p si c'est le cas (algorithme 2). La seconde méthode, de J. Omura [KO90], améliore l'approche précédente en évitant d'avoir à faire une comparaison mais uniquement des additions. La comparaison est remplacée par une simple analyse de la retenue sortante (algorithme 3).

Algorithme 2 *Addition modulaire triviale*

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$

Sortie : $s = (a + b) \bmod p$

1: $s \leftarrow a + b$
2: **if** $s \geqslant p$ **then**
3: $\quad s \leftarrow s - p$
4: **end if**
5: retourner s

Algorithme 3 *Addition modulaire d'Omura*

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $c = 2^n - p$

Sortie : $s = (a + b) \bmod p$

1: $s \leftarrow a + b$
2: $t \leftarrow s + c$
3: **if** $t \geqslant 2^n$ **then**
4: $\quad s \leftarrow t \bmod 2^n$
5: **end if**
6: retourner s

Notons que la ligne 3 de l'algorithme 3 correspond à une simple analyse de la retenue sortante. Cette opération simple se fait très efficacement. Il est facile de vérifier que l'algorithme 3 d'Omura est correct. En effet,

- si $a + b < p$, alors $t = a + b + c < p + c = 2^n$. Donc, la sortie s de cet algorithme est telle que : $s = a + b < p$.
- si $a + b \geqslant p$, alors $t = a + b + c \geqslant p + c = 2^n$. Ainsi, la sortie s de cet algorithme est telle que : $s = a + b + c - 2^n = a + b - p$, avec $0 \leqslant s < p$ car $a, b \in \mathbb{Z}/p\mathbb{Z}$.

Les algorithmes 2 et 3 ont les coûts suivants :

$$C_{algo\ 2} = 2Add_n \quad \text{et} \quad C_{algo\ 3} = 2Add_n.$$

Pour l'algorithme 3, les coûts d'analyse de la retenue sortante et de la réduction modulo 2^n sont ignorés, car très simples et insignifiants par rapport au coût des deux additions.

1.1.2 La multiplication modulaire

Comme expliqué plus haut, la multiplication modulaire est composée d'une simple multiplication et d'une réduction modulaire. Il existe plusieurs méthodes pour effectuer cette opération. Certaines commencent par d'abord faire la simple multiplication, puis la réduction modulaire. D'autres combinent la multiplication et la réduction. Certains entiers, de par leurs formes, permettent d'avoir

des réductions modulaires très efficaces. Les plus célèbres sont ceux de la famille des nombres de Mersenne. Pour les entiers qui n'ont pas de forme particulière, il existe plusieurs méthodes qui sont plus ou moins intéressantes selon le cas d'usage. Dans cette section, nous commençons par présenter quelques méthodes de multiplication modulaires pour des modules quelconques. Ensuite, nous abordons le cas particulier des entiers de la famille de Mersenne.

1.1.2.1 La multiplication simple

Supposons que $\beta = 2^k$. Soient $a, b \in \mathbb{Z}/p\mathbb{Z}$. La multiplication simple consiste à calculer $c = ab$. On a donc $0 \leqslant c < p^2 < \beta^{2n}$.

En général, on prend k comme la taille de mot machine. Dans ce cas, n mots machine sont nécessaires pour stocker les éléments de $\mathbb{Z}/p\mathbb{Z}$ et $2n$ mots machine sont nécessaires pour stocker le produit $c = ab$.

Selon la valeur de n, certaines méthodes peuvent s'avérer plus ou moins intéressantes que d'autres. Lorsque n est assez petit, ce qui est en général le cas en cryptographie, la méthode standard (l'algorithme "scolaire") [Knu97] est la plus appropriée. Le tableau 1.1 donne une liste de méthodes pour effectuer la multiplication simple, avec leur complexité en nombre de multiplications élémentaires, c.-à-d. le nombre de multiplications de deux entiers de k bits. Ce tableau donne également la méthode appropriée selon la valeur de n.

Méthode	Complexité	Quand choisir
Standard [Knu97]	$\mathcal{O}(n^2)$	$n < 10$
Karatsuba [KO63]	$\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$	$10 \leqslant n < 300$
Toom-Cook [Knu97]	$\mathcal{O}(n^{\log_3 5}) \approx \mathcal{O}(n^{1.465})$	$300 \leqslant n < 1000$
Schönhage-Strassen [SS71]	$\mathcal{O}(n \log n \log\log n)$	$1000 \leqslant n$

TABLE 1.1 – Exemples de méthodes pour la multiplication simple de 2 entiers représentés chacun sur n mots machine.

La troisième colonne du tableau 1.1 donne l'intervalle de valeurs pour n pour lequel chaque méthode est la plus intéressante. Ces intervalles proviennent de la thèse de Giorgi [Gio04].

1.1.2.2 Multiplication modulaire avec un module quelconque

Il existe plusieurs méthodes [Tay81, Bla83, Mon85, Bar87, Tak92] pour effectuer la multiplication modulaire lorsque le module n'a pas de forme particulière. Selon le cas d'usage, certaines peuvent s'avérer plus appropriées que d'autres. Ici, nous nous intéressons à deux méthodes parmi les plus efficaces. Il s'agit des méthodes de Barrett et de Montgomery.

Multiplication modulaire de Barrett.

Soit $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $c = ab$. La division euclidienne de c par p nous assure l'existence de deux entiers r et q tels que :

$$c = qp + r \text{ avec } 0 \leqslant r < p.$$

L'idée de la méthode de Barrett [Bar87] est de calculer une approximation du quotient q de cette division. L'intérêt de cette méthode est que d'une part cette approximation est faite sans division complexe et d'autre part cette approximation est assez fine pour que la réduction modulaire soit efficace.

Le quotient q de la division euclidienne est tel que : $q = \left\lfloor \frac{c}{p} \right\rfloor$. Supposons que nos entiers soient représentés en base $\beta = 2^k$, avec $\beta^{n-1} \leqslant p < \beta^n$. Barrett approche le quotient q comme suit :

$$\begin{aligned} q &= \left\lfloor \frac{c}{p} \right\rfloor \\ &= \left\lfloor \frac{\frac{c}{\beta^{n-1}} \times \frac{\beta^{2n}}{p}}{\beta^{n+1}} \right\rfloor \\ &\simeq \left\lfloor \frac{\left\lfloor \frac{c}{\beta^{n-1}} \right\rfloor \times \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor}{\beta^{n+1}} \right\rfloor \end{aligned}$$

Puisque β est une puissance de deux, les divisions par ce dernier sont très simples. Ce sont de simples décalages. L'erreur d'approximation du quotient est le suivant :

$$q - 2 \leqslant \left\lfloor \frac{\left\lfloor \frac{c}{\beta^{n-1}} \right\rfloor \times \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor}{\beta^{n+1}} \right\rfloor \leqslant q$$

Ainsi, au plus deux soustractions seront nécessaires pour une réduction complète. Puisque les entiers β, p et n sont fixés, il est possible de pré-calculer $v = \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor$. On évite ainsi la division coûteuse. L'algorithme 4 correspond à la multiplication modulaire de Barrett.

On a : $\beta^{n-1} \leqslant p < \beta^n$, avec $\beta = 2^k$. Donc, $p < 2^{nk}$. Le coût de l'algorithme 4 est donc :

$$C_{algo\ 4} \simeq 3Mul_{kn} + 3Add_{nk}$$

Son coût en produits élémentaires, en d'autres termes le nombre de produits d'entiers de taille k bits, est :

$$T(n)_{algo\ 4} = 3n^2 Mul_k$$

Pour que cet algorithme soit efficace, il est nécessaire de pré-calculer $v = \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor$. Ainsi, la méthode de Barrett n'est intéressante que si on doit effectuer plusieurs multiplications modulaires avec le même module p.

Algorithme 4 *Multiplication modulaire de Barrett*

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $v = \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor$

Sortie : $s = ab \mod p$

1: $c \leftarrow ab$
2: $u \leftarrow \left\lfloor \frac{c}{\beta^{n-1}} \right\rfloor$
3: $w \leftarrow uv$
4: $q \leftarrow \left\lfloor \frac{w}{\beta^{n+1}} \right\rfloor$
5: $s \leftarrow c - qp$
6: **if** $s \geqslant p$ **then**
7: $\quad s \leftarrow s - p$
8: **end if**
9: **if** $s \geqslant p$ **then**
10: $\quad s \leftarrow s - p$
11: **end if**
12: retourner s

Multiplication modulaire de Montgomery.

En 1985, Montgomery [Mon85] propose une méthode de multiplication modulaire qui, comme celle de Barrett, n'impose pas de condition particulière sur le module p.

Soit $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $c = ab$. La méthode de Barrett, présentée plus haut, calcule un entier q tel que les parties hautes de c et qp correspondent. Ainsi, la quantité $c - qp$ est assez proche de $c \mod p$.

Avec la méthode de Montgomery, l'objectif est de calculer un entier q tel que les parties basses de c et qp correspondent. Plus précisément, on calcule q tel que $c - qp \equiv 0 \pmod{\beta^n}$. Ainsi, on a $\frac{c-qp}{\beta^n} \equiv c\beta^{-n} \pmod{p}$.

Pour que la méthode de Montgomery soit efficace, il est nécessaire que le module p soit impair, afin de n'avoir à effectuer que des divisions triviales par des puissances de deux. Dans ce cas, les entiers r' et p' tels que : $r' = \beta^{-n} \mod p$ et $p' = -p^{-1} \mod \beta^n$ existent, car $\text{pgcd}(p, \beta^n) = 1$. L'algorithme 5 correspond à la multiplication modulaire de Montgomery. Posons, $r = \beta^n$.

On a : $\beta^{n-1} \leqslant p < \beta^n$, avec $\beta = 2^k$. Donc, $p < 2^{nk}$. Le coût de l'algorithme 5 est donc :

$$C_{algo\ 5} \simeq 3Mul_{kn} + 2Add_{nk}$$

Cet algorithme est plus efficace que celui de Barrett. Son coût en produits élémentaires, i.e. le nombre de produits d'entiers de taille k bits est :

$$T(n)_{algo\ 5} = 3n^2 Mul_k$$

On peut remarquer que l'algorithme 5 retourne $abr' \mod p$ au lieu de $ab \mod p$. Ainsi, pour garantir la consistance des opérations, il est nécessaire d'utiliser une représentation particulière appelée *représentation de Montgomery*.

Algorithme 5 MontgomeryMul, *Multiplication modulaire de Montgomery*

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $p' = -p^{-1} \bmod \beta^n$

Sortie : $s = abr' \bmod p$, où $r' = r^{-1} \bmod p$

1: $c \leftarrow ab$
2: $q \leftarrow cp' \bmod r$
3: $s \leftarrow (c + qp)/r$ # division exacte
4: **if** $s \geqslant p$ **then**
5: $\quad s \leftarrow s - p$
6: **end if**
7: retourner s

Soit $a \in \mathbb{Z}/p\mathbb{Z}$ un entier. La représentation de Montgomery de a, qu'on notera $\tilde{a}$, est l'entier $\tilde{a} = ar \bmod p$. Cette représentation est stable pour la multiplication modulaire de Motgomery ainsi que l'addition simple. En effet, si $a, b \in \mathbb{Z}/p\mathbb{Z}$, on a :

$$\begin{aligned} \mathbf{MontgomeryMul}(\tilde{a}, \tilde{b}) &= \tilde{a}\tilde{b}r' \bmod p \\ &= arbrr^{-1} \bmod p \\ &= abr \bmod p \\ &= \widetilde{ab} \end{aligned}$$

Pour l'addition classique, on a :

$$\begin{aligned} \tilde{a} + \tilde{b} &= ar + br \\ &= (a + b)r \\ &\equiv \widetilde{a + b} \end{aligned}$$

La conversion dans et hors du domaine de Montgomery se fait très simplement à l'aide d'une multiplication modulaire. Soit, $m = r^2 \bmod p$. Pour la conversion dans le domaine de Montgomery, on a :

$$\begin{aligned} \tilde{a} &= \mathbf{MontgomeryMul}(a, m) \\ &= ar^2r' \bmod p \\ &= ar \bmod p \end{aligned}$$

Pour la conversion hors du domaine de Montgomery, on a :

$$a = \mathbf{MontgomeryMul}(\tilde{a}, 1) = arr' \bmod p$$

Avec cette nécessité de convertir les entrées dans le domaine de Montgomery ainsi que la quantité $p' = -p^{-1} \bmod \beta^n$ à pré-calculer, la méthode de Montgomery n'est intéressante que si le module p ainsi que les entrées sont utilisés pour plusieurs multiplications modulaires. C'est le cas par exemple lorsqu'on effectue une exponentiation modulaire. Si l'exposant de l'exponentiation est de taille h

bits, alors l'exponentiation modulaire avec la méthode de Montgomery nécessitera entre $(h+2)$ et $(2h+2)$ appels à **MontgomeryMul**. Si h est suffisamment grand, ce qui sera le cas en cryptographie, ce sur-coût de deux multiplications devient négligeable.

Multiplication modulaire de Montgomery par bloc.

En 1991, Dussé et Kaliski [DK91] proposent une amélioration de la multiplication modulaire de Montgomery. L'idée principale de leur amélioration est d'effectuer la réduction modulaire de manière itérative par bloc de k bits, avec $\beta = 2^k$.

Le module p est supposé impair. Pour rappel, on a $\beta^{n-1} \leqslant p < \beta^n$. Donc, un entier $a \in \mathbb{Z}/p\mathbb{Z}$ est représenté en base β comme suit : $a = a_0 + a_1\beta + \cdots + a_{n-1}\beta^{n-1}$. En outre, l'entier q calculé à la ligne 2 de l'algorithme 5 peut également être représenté en base β, tel que $q = q_0 + q_1\beta + \cdots + q_{n-1}\beta^{n-1}$. Dans cet algorithme, ces *petits* entiers q_i sont calculés simultanément, avec n^2 *petites* multiplications d'entiers de taille k bits. Dans [DK91], l'idée principale de l'amélioration est de calculer un *petit* entier q_i à la fois, en une seule petite multiplication. Ce faisant, le coût du calcul de q est réduit à n petites multiplications, au lieu de n^2.

Dans [KAK96], Koc et al. analysent et comparent cinq variantes de l'algorithme de multiplication modulaire de Montgomery. Ces variantes sont : le SOS (Separated Operand Scanning), le CIOS (Coarsely Integrated Operand Scanning), le FIOS (Finely Integrated Operand Scanning), le FIPS (Finely Integrated Product Scanning) et le CIHS (Coarsely Integrated Hybrid Scanning). Toutes ces variantes sont basées sur l'amélioration proposée dans [DK91]. Elles diffèrent principalement sur deux aspects. Le premier aspect est la façon dont les étapes de multiplication et de réduction sont combinées. Ils sont soit séparés (Separated : S) ou grossièrement intégrés (Coarsely Integrated : CI) ou finement intégrés (Finely Integrated : FI). Le deuxième aspect est la façon dont la multiplication de deux (grands) entiers est faite. Il peut s'agir d'un Operand Scanning (OS), où une boucle externe se déplace à travers les mots d'un des opérandes ou d'un Product Scanning (PS), où la boucle se déplace à travers les mots du produit lui-même. Pour le CIHS, Hybrid signifie un mélange d'OS et de PS pour la multiplication.

Dans [KAK96], les auteurs obtiennent que la variante CIOS est la meilleure parmi les cinq, en particulier pour des processeurs non-spécialisés (voir la section 9 et les tableaux 1 et 3 de cet article). Cette variante coûte :

$$C_{algo\ 6} = (2n^2 + n)Mul_k + (4n^2 + 4n + 2)Add_k$$

En base β, le module p s'écrit comme suit : $p = p_0 + p_1\beta + \cdots + p_{n-1}\beta^{n-1}$. Comme p est impair, on a : $\text{pgcd}(p_0, \beta) = 1$. Donc, l'entier $g = -p_0^{-1} \bmod \beta$ existe. Cet entier est nécessaire pour toutes les variantes décrites dans [KAK96]. L'algorithme 6 est la variante CIOS. Dans cet algorithme, on suppose que les entrées et la sortie sont des entiers représentés par des tableaux de taille n, où chaque colonne contient un entier inférieur à β. Plus précisément, si $a \in \mathbb{Z}/p\mathbb{Z}$, alors $a := (a[0], \ldots, a[n-1])$, où $0 \leqslant a_i < \beta$.

Algorithme 6 *Montgomery CIOS* [KAK96]

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $g = -p_0^{-1} \bmod \beta$
Sortie : $ab\beta^{-n} \bmod p$

$t \leftarrow (0, \ldots, 0)$ # tableau de taille $n+2$, initialisation à zéro des colonnes.
$u \leftarrow (0, \ldots, 0)$ # tableau de taille n, initialisation à zéro des colonnes.
for $i = 0 \ldots n-1$ **do**
    $C \leftarrow 0$
    for $j = 0 \ldots n-1$ **do**
        $(C, S) \leftarrow t[j] + a[j].b[i] + C$
        $t[j] \leftarrow S$
    end for
    $(C, S) \leftarrow t[n] + C$
    $t[n] \leftarrow S$
    $t[n+1] \leftarrow C$
    $m \leftarrow g.t[0] \bmod \beta$
    $(C, S) \leftarrow t[0] + m.p[0]$
    for $j = 1 \ldots n-1$ **do**
        $(C, S) \leftarrow t[j] + m.p[j] + C$
        $t[j-1] \leftarrow S$
    end for
    $(C, S) \leftarrow t[n] + C$
    $t[n-1] \leftarrow S$
    $t[n] \leftarrow t[n+1] + C$
end for
$B \leftarrow 0$
for $j = 0 \ldots n-1$ **do**
    $(B, D) \leftarrow t[j] - p[j] - B$
    $u[j] \leftarrow D$
end for
$(B, D) \leftarrow t[n] - B$
if $B = 0$ **then**
    retourner $u \ \# \ (u[0], \ldots, u[n-1])$.
else
    retourner $t \ \# \ (t[0], \ldots, t[n-1])$.
end if

1.1.2.3 La famille des nombres de Mersenne

De par leurs formes, certains entiers permettent d'effectuer la réduction modulaire très efficacement. Nous présentons ici la famille des nombres de Mersenne, qui peut être subdivisée en trois sous-groupes. Le premier sous-groupe correspond aux nombres de Mersenne. Ce sont les nombres de la forme $2^n - 1$, avec $n \geqslant 1$ un entier. Ces nombres assurent les meilleures performances pour la réduction modulaire. Malheureusement, la densité des nombres premiers de Mersenne est très faible. Ce qui est un inconvénient majeur, car beaucoup de cryptosystèmes à clé publique nécessitent des modules premiers. C'est par exemple le cas pour la plupart des protocoles ECC. Ainsi, des généralisations de la classe des nombres de Mersenne ont été proposées. Il s'agit des nombres Pseudo Mersenne [Cra92], des nombres de Mersenne Généralisés [Sol99] et des nombres de Mersenne Plus Généralisés [CH03]. La classe des nombres de Mersenne Plus Généralisés est une extension qui englobe toutes les autres classes. Plusieurs nombres de cette famille des nombres de Mersenne ont été recommandés par

le SEC (Standard for Efficient Cryptography) [SEC14] et le NIST (National Institute of Standards and Technology) [NIS19], pour l'arithmétique modulaire et tout particulièrement pour l'implantation des protocoles ECC, à travers de nombreux standards (nous donnons quelques exemples dans le tableau 1.2, plus bas). Dans ce qui suit, nous présentons les nombres de Mersenne ainsi que les nombres Pseudo Mersenne. Pour les autres sous-classes, nous renvoyons le lecteur vers les articles correspondants [Sol99, CH03].

Les nombres de Mersenne.

Définition 1.1. Un entier p est un nombre de Mersenne s'il est de la forme : $p = 2^n - 1$, avec $n \geqslant 1$ un entier.

Ainsi, on a $2^n \equiv 1 \pmod{p}$. Cette propriété permet d'effectuer la réduction modulo p très efficacement. L'algorithme 7 correspond à la multiplication modulaire modulo un nombre de Mersenne.

Algorithme 7 *Multiplication modulaire modulo un nombre de Mersenne*

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $p = 2^n - 1$

Sortie : $s = ab \mod p$

1: $c \leftarrow ab$
2: Soient c_0 et c_1 tels que : $c = c_1 2^n + c_0$
3: $s \leftarrow c_1 + c_0$
4: **if** $s \geqslant p$ **then**
5: $\quad s \leftarrow s - p$
6: **end if**
7: retourner s

Le coût de l'algorithme 7 est :

$$C_{algo\ 7} = Mul_n + 2Add_n$$

On peut remarquer que la réduction modulaire dans cet l'algorithme coûte deux additions d'entiers de n bits. Ce coût est nettement plus intéressant que ceux des méthodes présentées dans la section 1.1.2.2. Malheureusement, la densité des nombres de Mersenne est très faible. En effet, il est évident que, pour n fixé, il n'existe qu'un seul nombre de Mersenne. De plus, d'après la propriété 1.1 (connue dans la littérature), si n n'est pas premier, alors $2^n - 1$ ne peut pas être premier.

Propriété 1.1. Soit $n \geqslant 2$ un entier.

1. Si $p = \beta^n - 1$ est un nombre premier, alors $\beta = 2$.
2. Si $p = 2^n - 1$ est un nombre premier, alors n est premier.

Démonstration. On a $\beta^n - 1 = (\beta - 1)(1 + \beta + \cdots + \beta^{n-1})$. Ainsi, $p = \beta^n - 1$ premier implique nécessairement que $\beta - 1 = 1$. D'où, $\beta = 2$.

Pour le second point, si n n'est pas premier, alors il existe deux entiers $u, v \geqslant 2$ tels que $n = uv$. Ainsi, $p = (2^u)^v - 1$. Or, d'après le premier point, si p est premier, alors $2^u = 2$. Donc, $u = 1$, i.e. n est premier. □

Les nombres Pseudo Mersenne.

En 1992, Crandall étend la classe des nombres de Mersenne en proposant les nombres Pseudo Mersenne [Cra92].

Définition 1.2. Un entier p est un nombre Pseudo Mersenne s'il est de la forme : $p = 2^n - c$, avec $n \geqslant 1$ et $c < 2^{\frac{n}{2}}$ des entiers.

Ainsi, on a $2^n \equiv c \pmod{p}$. Comme pour les nombres de Mersenne, cette propriété permet d'effectuer la réduction modulo p très efficacement. L'algorithme 8 correspond à la multiplication modulaire modulo un nombre Pseudo Mersenne.

Algorithme 8 *Multiplication modulaire modulo un nombre Pseudo Mersenne*

Entrée(s) : $a, b \in \mathbb{Z}/p\mathbb{Z}$ et $p = 2^n - c$, avec $c < 2^{\frac{n}{2}}$

Sortie : $s = ab \mod p$

1: $c \leftarrow ab$
2: Soient r_0 et r_1 tels que : $c = r_1 2^{\frac{3n}{2}} + r_0$
3: $r \leftarrow r_1 c 2^{\frac{n}{2}} + r_0$
4: Soient s_0 et s_1 tels que : $r = s_1 2^n + s_0$
5: $s \leftarrow s_1 c + s_0$
6: **if** $s \geqslant p$ **then**
7: $\quad s \leftarrow s - p$
8: **end if**
9: retourner s

Soit $k = \lceil \log_2(c) \rceil$ la taille de c. Le coût de l'algorithme 8 est le suivant.

$$C_{algo\ 8} = Mul_n + 2Mul_{k,\frac{n}{2}} + 3Add_n$$

Ici, $Mul_{k,\frac{n}{2}}$ désigne le produit de c par un entier de taille inférieure à $n/2$ bits ; cf. lignes 3 et 5 de l'algorithme. On peut ici encore observer que le coût de cet algorithme est plus intéressant que ceux des méthodes présentées dans la section 1.1.2.2.

Dans [Hia00], Hiasat propose une amélioration de l'algorithme 8. Cette amélioration repose sur l'utilisation d'une table mémoire pour réduire le nombre de multiplications à effectuer.

Quelques exemples.

Nous donnons ici quelques exemples de la famille des nombres de Mersenne qui sont recommandés par le SEC [SEC14] et le NIST [NIS19] pour la cryptographie sur les courbes elliptiques.

Identifiant(s) du module	Valeur
secp128	$2^{128} - 2^{97} - 1$
secp160b	$2^{160} - 2^{31} - 1$
p192, secp192b	$2^{192} - 2^{64} - 1$
p224, secp224b	$2^{224} - 2^{96} + 1$
p256, secp256b	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
p384, secp384	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
p521, secp521	$2^{521} - 1$

TABLE 1.2 – Exemples de nombres de la famille de Mersenne, recommandés par le NIST et le SEC.

1.1.3 L'exponentiation modulaire

Soient $a, h \in \mathbb{Z}/p\mathbb{Z}$. L'exponentiation modulaire correspond à l'opération :

$$s = a^h \bmod p = \underbrace{(a \times a \times \cdots \times a)}_{h \text{ fois}} \bmod p.$$

De nombreuses méthodes ont été proposées pour effectuer cette opération [Mon87, JY02, Gor98, CFG+11, NP17]. Selon la quantité de ressources disponibles ainsi que le niveau de sécurité souhaité, certaines méthodes peuvent s'avérer plus intéressantes que d'autres.

La méthode la plus simple pour effectuer l'exponentiation modulaire est le *square-and-multiply* (algorithme 9). Cette méthode est l'algorithme générique d'exponentiation (algorithme 1) pour le groupe multiplicatif $(\mathbb{Z}/p\mathbb{Z}, x)$. On peut remarquer que cet algorithme n'est pas régulier, car la multiplication modulaire n'a lieu que lorsque le bit courant k_i est égal à 1. Il est donc vulnérable aux attaques par canaux auxiliaires, même les plus basiques telles que les attaques de type SPA (Simple Power Analysis).

Dans [Cor99], Coron suggère d'effectuer un calcul *factice* lorsque le bit courant k_i est égal à 0. Dans le cas du *square-and-multiply*, cela revient a effectuer une multiplication modulaire inutile lorsque $k_i = 0$. On obtient ainsi le *square-and-multiply always* (algorithme 10). Cette solution induit cependant une autre vulnérabilité. Le *square-and-multiply always* est vulnérable aux attaques de type *C safe-error* [SMKLM01]. En effet, si une injection de fautes au cours de la multiplication modulaire à l'itération i ne modifie pas le résultat de la exponentiation, alors on est certain que k_i vaut 0. Sinon, on est certain que k_i vaut 1.

L'*échelle de Montgomery* (*Montgomery ladder*, en anglais) est une autre méthode régulière pour effectuer l'exponentiation modulaire. Cette méthode fut initialement proposée pour effectuer la multiplication scalaire sur les courbes de Montgomery [Mon87]. Dans cette section, nous nous intéressons à l'adaptation de cette méthode aux groupes multiplicatifs présentée dans [JY02].

L'*échelle de Montgomery* (algorithme 11) a la particularité de garantir que $R_1 - R_0 = P$ à la fin de chaque itération. En outre, contrairement au *square-and-multiply always*, cette méthode n'introduit aucun calcul factice, ce qui la rend sûre contre les attaques de type *C safe-error*, comme expliqué dans [JY02]. Dans le même article, les auteurs expliquent que l'*échelle de Montgomery* n'est cependant pas sûre contre les attaques de type *M safe-error*[YJ00] dont l'objectif est d'introduire une faute temporaire sans que cette dernière ne soit détectée. Ils proposent par suite une modification de cette méthode qui est sûre contre les attaques de type *M safe-error*. C'est cette dernière qui est présentée ici.

Notons que l'algorithme 11 reste toutefois vulnérable à certaines attaques plus avancées, comme des attaques de type DPA (Differential Power Analysis) [KJJ99]. En effet, l'affectation du carré et de multiplication modulaires aux registres R_0 et R_1 de l'algorithme dépend du bit courant h_i de l'exposant. Cette fuite peut être exploitée pour des attaques de type DPA. Des contre-mesures indispensables dans ce cas sont la randomisation de l'exposant h, de la base a et/ou des adresses mémoires [MDS99, Cor99].

Enfin, comme expliqué dans [JY02] (section 3.2), les opérations des lignes 5 et 6 de algorithme 11 sont indépendantes à chaque tour de boucle. Elles peuvent donc être parallélisées.

Algorithme 9 *Left-to-right square-and-multiply*

Entrée(s) : $a, h \in \mathbb{Z}/p\mathbb{Z}$, tel que $h = (h_{n-1}, \ldots, h_0)_2$

Sortie : $s \in \mathbb{Z}/p\mathbb{Z}$, tel que $s \equiv a^h \mod p$

1: $s \leftarrow 1$
2: **for** $i = n - 1$ **to** 0 **do**
3: $\quad s \leftarrow s^2 \mod p$
4: $\quad$ **if** $h_i = 1$ **then**
5: $\quad\quad s \leftarrow s\,a \mod p$
6: $\quad$ **end if**
7: **end for**
8: retourner s

Algorithme 10 *Left-to-right square-and-multiply always*

Entrée(s) : $a, h \in \mathbb{Z}/p\mathbb{Z}$, tel que $h = (h_{n-1}, \ldots, h_0)_2$

Sortie : $R_0 \in \mathbb{Z}/p\mathbb{Z}$, tel que $R_0 \equiv a^h \mod p$

1: $R_0 \leftarrow 1$
2: **for** $i = n - 1$ **to** 0 **do**
3: $\quad R_0 \leftarrow R_0^2 \mod p$
4: $\quad R_1 \leftarrow R_0 R_1 \mod p$
5: $\quad R_0 \leftarrow R_{h_i}$
6: **end for**
7: retourner R_0

Algorithme 11 *Échelle de Montgomery*

Entrée(s) : $a, h \in \mathbb{Z}/p\mathbb{Z}$, tel que $h = (h_{n-1}, \ldots, h_0)_2$

Sortie : $R_0 \in \mathbb{Z}/p\mathbb{Z}$, tel que $R_0 \equiv a^h \mod p$

1: $R_0 \leftarrow 1$
2: $R_1 \leftarrow a$
3: **for** $i = n - 1$ **to** 0 **do**
4: $\quad b \leftarrow h_i$
5: $\quad R_{1-b} \leftarrow R_{1-b} R_b \mod p$
6: $\quad R_b \leftarrow R_b^2 \mod p$
7: **end for**
8: retourner R_0

Notons MUL_n et SQR_n respectivement une multiplication et un carré modulaire dans $\mathbb{Z}/p\mathbb{Z}$. Les algorithmes 9, 10 et 11 ont les coûts suivants :

$$\begin{aligned} C_{algo\ 9} &= nSQR_n + \frac{n}{2}MUL_n\,, \\ C_{algo\ 10} &= nSQR_n + nMUL_n\,, \\ C_{algo\ 11} &= nSQR_n + nMUL_n \end{aligned}$$

Pour l'algorithme 9, on suppose qu'en moyenne la moité des bits de l'exposant sont égaux à 1.

1.1.4 L'inversion modulaire

Soient $a \in \mathbb{Z}/p\mathbb{Z}$. L'inversion modulaire consiste à calculer l'entier $x \in \mathbb{Z}/p\mathbb{Z}$ tel que $a.x \equiv 1 \pmod p$. Cet entier existe si et seulement si $\text{pgcd}(a, p) = 1$.

Supposons que $\text{pgcd}(a, p) = 1$. L'inverse de a, notée a^{-1}, peut être calculé de deux manières [Coh93]. La première est l'algorithme d'Euclide étendu. Cette méthode est une extension de l'algorithme proposé par Euclide pour le calcul du PGCD (Plus Grand Commun Diviseur) de deux nombres. Le seconde méthode est une transformation de l'inversion modulaire en une exponentiation modulaire, grâce au petit théorème de Fermat lorsque p est premier et plus généralement le théorème d'Euler pour p quelconque.

L'algorithme 12 est l'algorithme d'Euclide étendu. Il prend comme entrées deux entiers a et b et calcule un tuple (d, u, v) qui satisfait l'*identité de Bézout*, i.e. $d = a.u + b.v$, avec $d = \text{pgcd}(a, b)$. Comme $\text{pgcd}(a, p) = 1$, appliqué a et p, cet algorithme retourne u et v tels que $a.u + p.v = 1$. Donc, $a.u \equiv 1 \pmod p$. Ainsi, $a^{-1} = u \mod p$.

La transformation de l'inversion modulaire en une exponentiation modulaire se fait grâce à deux théorèmes. Le premier est le petit théorème de Fermat et le second est un théorème dû à Euler. Le second généralise le premier.

Théorème 1.1 (**Petit théorème de Fermat**). *Soit p un nombre premier et $a \in \mathbb{Z}/p\mathbb{Z}$ un entier non nul. On a :* $a^{p-1} \equiv 1 \pmod p$.

Algorithme 12 *Euclide étendu*

Entrée(s) : a et b deux entiers.

Sortie : (u, v, d), tel que $d = a.u + b.v$, avec $d = \text{pgcd}(a, b)$

1: $(u_1, u_2, u_3) \leftarrow (1, 0, a)$
2: $(v_1, v_2, v_3) \leftarrow (0, 1, b)$
3: **while** $v_3 \neq 0$ **do**
4: $\quad q \leftarrow \left\lfloor \frac{u_3}{v_3} \right\rfloor$
5: $\quad (t_1, t_2, t_3) \leftarrow (v_1, v_2, v_3)$
6: $\quad (v_1, v_2, v_3) \leftarrow (u_1, u_2, u_3) - q(v_1, v_2, v_3)$
7: $\quad (u_1, u_2, u_3) \leftarrow (t_1, t_2, t_3)$
8: **end while**
9: $(u, v, d) \leftarrow (u_1, u_2, u_3)$
10: retourner (u, v, d)

Ainsi, on a $a^{p-1} = a^{p-2}a \equiv 1 \pmod{p}$. Donc,

$$a^{-1} = a^{p-2} \bmod p\,.$$

La généralisation du petit théorème de Fermat à un module p non nécessairement premier utilise l'*indicatrice d'Euler*.

Définition 1.3 (**Indicatrice d'Euler**)**.** Soit m un entier naturel non nul. L'*indicatrice d'Euler* de m, notée $\varphi(m)$, est le nombre d'entiers compris entre 1 et m (inclus), et premiers avec m.

Théorème 1.2 (**Euler**)**.** *Soient a et m deux entiers non nuls, tels que pgcd(a, m) = 1. On a : $a^{\varphi(m)} \equiv 1 \pmod{p}$, où $\varphi(m)$ est l'indicatrice d'Euler de m.*

Ainsi, on a $a^{\varphi(m)} = a^{\varphi(m)-1}.a \equiv 1 \pmod{p}$. Donc,

$$a^{-1} = a^{\varphi(m)-1} \bmod p\,.$$

Notons que si p est un nombre premier, alors $\varphi(p) = p-1$. Une fois l'inversion modulaire transformée en une exponentiation modulaire, on peut se référer à la section 1.1.3 pour la suite.

1.2 La représentation RNS

Le système de représentation par les restes [Gar59] (RNS : Residue Number System, en anglais) est un système de numération non positionnel. Ce système est basé sur le théorème des restes chinois. Dans le RNS, les nombres sont présentés par leurs restes modulo un ensemble d'entiers premiers entre eux. Dans cette section, nous présentons les éléments théoriques de ce système. Nous n'abordons pas l'aspect pratique de l'utilisation de ce système. Pour cela, il existe une vaste littérature. Nous renvoyons le lecteur vers [BDK98, Did98, BMP05], entre autres, pour une compréhension plus approfondie de ce système.

Théorème 1.3 (**Restes chinois, CRT**). *Soient $m_0, \ldots, m_{n-1}$ des entiers naturels non nuls tels que : pgcd(m_i, m_j) = 1, si $i \neq j$. Posons, $m = \prod_{i=0}^{n-1} m_i$. L'application φ ci-dessous est un isomorphisme d'anneaux.*

$$\begin{array}{rccl} \varphi & : & \mathbb{Z}/m\mathbb{Z} & \to & \mathbb{Z}/m_0\mathbb{Z} \times \cdots \times \mathbb{Z}/m_{n-1}\mathbb{Z} \\ & & x & \mapsto & (x \bmod m_0, \ldots, x \bmod m_{n-1}) \end{array}$$

D'après le théorème 1.3, on a l'équivalence suivante :

$$x \in \mathbb{Z}/m\mathbb{Z} \iff \exists!\ (x_0, \ldots, x_{n-1}) \in \mathbb{Z}/m_0\mathbb{Z} \times \cdots \times \mathbb{Z}/m_{n-1}\mathbb{Z},$$
$$\text{tels que } \begin{cases} x \equiv x_0 \pmod{m_0} \\ \vdots \\ x \equiv x_{n-1} \pmod{m_{n-1}} \end{cases}$$

Le tuple $\mathcal{B} = (m_0, \ldots, m_{n-1})$ est alors une base RNS pour l'entier m. La coordonnée x_i peut être calculée à partir de x comme suit : $x_i = x \bmod m_i$.

Le calcul de x à partir du tuple $(x_0, \ldots, x_{n-1})$ est un peu plus complexe. Pour $0 \leqslant i < n$, posons $\hat{m}_i = \frac{m}{m_i}$. Notons que pgcd($\hat{m}_i$, m_i) = 1, donc d'après le théorème de Bachet-Bézout, il existe deux entiers u_i et v_i tels que : $m_i u_i + \hat{m}_i v_i = 1$. Ces entiers peuvent être calculés assez facilement avec l'algorithme d'Euclide étendu. Posons, $t_i = \hat{m}_i v_i$. L'unique entier $x \in \mathbb{Z}/m\mathbb{Z}$ correspondant au tuple $(x_0, \ldots, x_{n-1})$ est alors :

$$x = \left(\sum_{i=0}^{n-1} x_i t_i \right) \bmod m.$$

Il est facile de le vérifier en remarquant que :

$$t_i \equiv 1 \pmod{m_i} \quad \text{et} \quad t_i \equiv 0 \pmod{m_j}, \text{ pour } i \neq j.$$

Grâce à cette décomposition des éléments de $\mathbb{Z}/m\mathbb{Z}$, il est possible de transformer les opérations arithmétiques dans $\mathbb{Z}/m\mathbb{Z}$ en des opérations équivalentes dans les anneaux $\mathbb{Z}/m_i\mathbb{Z}$ qu'il sera possible d'effectuer de façon indépendante. Ainsi, ces opérations arithmétiques deviennent plus simples tout en permettant de les paralléliser, sans avoir de propagation de retenues à gérer.

Soient $a, b \in \mathbb{Z}/m\mathbb{Z}$, tels que $(a_0, \ldots, a_{n-1})$ et $(b_0, \ldots, b_{n-1})$ sont leurs représentations RNS respectives en la base $\mathcal{B}$.

- La représentation RNS de la somme $(a + b) \bmod m$ est le tuple $((a_0 + b_0) \bmod m_0, \ldots, (a_{n-1} + b_{n-1}) \bmod m_{n-1})$.
- De même, la représentation RNS du produit $(ab) \bmod m$ est le tuple $((a_0 b_0) \bmod m_0, \ldots, (a_{n-1} b_{n-1}) \bmod m_{n-1})$.
- Si pgcd(a, m) = 1, alors a est inversible dans $\mathbb{Z}/m\mathbb{Z}$. Notons que dans ce cas, pgcd(a_i, m_i) = 1, pour $0 \leqslant i < n$. La représentation RNS de l'inverse de a est : $(a_0^{-1} \bmod m_0, \ldots, a_{n-1}^{-1} \bmod m_{n-1})$.

1.3 Le système de représentation modulaire (MNS)

En 2004, Bajard et al. ont introduit le système de représentation modulaire (MNS : Modular Number System, en anglais)[BIP04] comme une extension du système de numération simple de position afin de représenter les entiers modulo p. Comme le RNS, le MNS est un système de numération non positionnel. Sa principale caractéristique est que les éléments y sont représentés sous forme de polynômes. Dans ce système, chaque entier $x \in \mathbb{Z}/p\mathbb{Z}$ est représenté par un polynôme en un entier γ.

Définition 1.4. Un système de représentation modulaire (MNS) $\mathcal{B}$ est défini par un quadruplet (p, n, γ, ρ) d'entiers, tel que pour tout entier $x \in \mathbb{Z}/p\mathbb{Z}$, il existe un vecteur $v = (v_0, \ldots, v_{n-1})$ d'entiers satisfaisant :

$$x \equiv \sum_{i=0}^{n-1} v_i \gamma^i \pmod{p},$$

avec $|v_i| < \rho, \quad \rho \approx p^{1/n} \quad$ et $\quad 0 < \gamma < p$. Dans ce cas, on dit que le polynôme $V(X) = v_0 + v_1 X + \cdots + v_{n-1} X^{n-1}$ (ou de façon équivalente le vecteur v) est une représentation de x dans $\mathcal{B}$. On note : $V \equiv x_{\mathcal{B}}$.

D'après la définition 1.4, on a donc :

$$V \equiv x_{\mathcal{B}} \iff \begin{cases} V(\gamma) \equiv x \pmod{p} \\ \deg(V) < n \\ \|V\|_\infty < \rho \end{cases}$$

Exemple 1.1. Soit $p = 19$. Le tableau 1.3 montre que le tuple $\mathcal{B} = (19, 3, 7, 2)$ est un MNS. On y trouve une représentation de chaque élément de $\mathbb{Z}/19\mathbb{Z}$ dans $\mathcal{B}$. On peut vérifier dans ce tableau que toute représentation A d'un élément $a \in \mathbb{Z}/19\mathbb{Z}$ est telle que : $\deg(A) < 3$, $\|A\|_\infty < 2$ et $A(\gamma) \equiv a \pmod{p}$. Par exemple, $\gamma^2 - \gamma + 1 = 49 - 7 + 1 = 43 \equiv 5 \bmod 19$. Donc, $X^2 - X + 1$ est bien une représentation de 5 dans $\mathcal{B}$.

Remarque 1.1. Dans un souci de cohérence, nous supposerons pour la suite de cette section (et les prochains chapitres) que : $p \geqslant 3$ et $n, \gamma, \rho \geqslant 1$.

1.3.1 Définitions et propriétés du MNS

Dans cette section, nous donnons les propriétés et définitions essentielles du MNS. Ces éléments existent déjà dans la thèse de Thomas Plantard (voir Section 3.1 de [Pla05]). Nous les reprenons ici avec parfois quelques modifications.

La première définition que nous présentons est la notion d'équivalence dans le MNS. Cette notion est importante pour la redondance dans le MNS que nous verrons juste après. Elle nous sera également utile dans le chapitre 3 qui porte sur la randomisation dans l'AMNS.

0	1	2	3	4
0	1	$-X^2-X+1$	X^2-X-1	X^2-X

5	6	7	8	9
X^2-X+1	$X-1$	X	$X+1$	$-X^2+1$

10	11	12	13	14
X^2-1	X^2	X^2+1	$-X+1$	$-X^2+X-1$

15	16	17	18
$-X^2+X$	$-X^2+X+1$	X^2+X-1	-1

TABLE 1.3 – Les éléments de $\mathbb{Z}/19\mathbb{Z}$ dans $\mathcal{B} = (19, 3, 7, 2)$

Définition 1.5 (Équivalence dans le MNS). [Pla05] Soient $U, V \in \mathcal{B}$ deux polynômes. Si U et V représentent le même élément de $\mathbb{Z}/p\mathbb{Z}$. On dit qu'ils sont équivalents. Cette relation d'équivalence est notée $U \overset{\mathcal{B}}{\equiv} V$. Plus précisément :

$$U \overset{\mathcal{B}}{\equiv} V \iff \sum_{i=0}^{n-1} u_i \gamma^i \equiv \sum_{i=0}^{n-1} v_i \gamma^i \pmod{p}.$$

La redondance dans un système de représentation est le fait que certaines de ses représentations correspondent à la même valeur.

Définition 1.6. [Pla05] Un système de représentation est *redondant* si au moins deux de ces représentations correspondent à une même valeur.

Ainsi, s'il existe deux polynômes U et V dans un MNS $\mathcal{B}$ tels que $U \overset{\mathcal{B}}{\equiv} V$, alors on dira que ce MNS est *redondant*. Pour le MNS de l'exemple 1.1, on a $-X-1$ et X^2 qui sont des représentations de 11 dans le MNS $\mathcal{B}$ de cet exemple. Par conséquent, ce MNS est redondant.

La Proposition 1.1 donne une condition nécessaire sur ρ pour qu'un tuple $\mathcal{B} = (p, n, \gamma, \rho)$ soit un MNS, une fois p et n fixés. C'est une modification de la Propriété 3 dans [Pla05], p. 51. Ici, nous tenons compte du fait que les coefficients d'un élément du MNS peuvent être négatifs.

Proposition 1.1. *Pour qu'un tuple $\mathcal{B} = (p, n, \gamma, \rho)$ soit un MNS, il faut nécessairement que :*

$$\lceil (\sqrt[n]{p} + 1)/2 \rceil \leqslant \rho \,.$$

Démonstration. Le nombre d'éléments dans $\mathcal{B}$ est $(2\rho - 1)^n$, car les éléments peuvent avoir des coefficients négatifs et leurs valeurs absolues sont strictement inférieures à ρ (voir Définition 1.4). Nous voulons représenter tous les éléments dans $\mathbb{Z}/p\mathbb{Z}$, donc ρ doit être tel que $p \leqslant (2\rho - 1)^n$. □

Ainsi, la valeur minimale de ρ pour un MNS $\mathcal{B}$ est :

$$\rho_{min} = \lceil (\sqrt[n]{p} + 1)/2 \rceil \ .$$

Cette proposition avec la définition 1.6 permettent de déduire qu'un MNS est redondant dès que :

$$\rho > (\sqrt[n]{p} + 1)/2 \ .$$

Cette notion de redondance peut être encore plus affinée dans le cas du MNS grâce au *rapport de redondance*. La définition 1.7 est une modification de la définition 12 (dans [Pla05], p. 51) pour tenir compte du fait que les coefficients d'un élément du MNS peuvent être négatifs.

Définition 1.7. [Pla05] Le *rapport de redondance*, noté π, est donné par :

$$\pi = \log_2 \frac{2\rho - 1}{\lceil \sqrt[n]{p} \rceil} \ .$$

Ce rapport est un critère de qualité qui permet d'évaluer la *redondance* d'un MNS. Si $\pi = 0$, le MNS n'est pas redondant. Si $\pi > 0$, le surcoût de la représentation dans le MNS est de π bits par coefficient.

La valeur du *rapport de redondance* ne sera en général pas un entier, ce qui n'est pas très intéressant en pratique. Car, un surcoût de 1.3 bits par coefficient (par exemple) n'est pas forcément très parlant. Nous proposons donc la *redondance effective* (définition 1.8), assez proche du *rapport de redondance*, qui donne le nombre de bits supplémentaires utilisés en pratique pour représenter chaque coefficient d'un élément du MNS. Pour rappel, le nombre de bits nécessaires pour représenter un élément de $\mathbb{Z}/p\mathbb{Z}$ dans la représentation classique, avec n chiffres, est $n(\lceil \log_2(p)/n \rceil)$. De même, le nombre de bits nécessaires pour représenter un élément du MNS est $n\lceil \log_2(2\rho - 1) \rceil$.

Définition 1.8. La *redondance effective*, noté ϖ, est donné par :

$$\varpi = \lceil \log_2(2\rho - 1) \rceil - \lceil \log_2(\sqrt[n]{p}) \rceil .$$

Chaque élément du MNS étant composé de n coefficients, le surcoût Π de la représentation d'un élément de ce système est alors :

$$\Pi = n\varpi \ . \tag{1.1}$$

Terminons cette section par la définition 1.9 qui définie l'ensemble des représentations d'un élément $a \in \mathbb{Z}/p\mathbb{Z}$. Cette définition est une modification de la définition 15 dans [Pla05], p. 52.

Définition 1.9. L'ensemble des représentations d'un entier $a \in \mathbb{Z}/p\mathbb{Z}$ dans un MNS $\mathcal{B}$ sera noté $\mathcal{B}_a$. Cet ensemble est défini comme suit :

$$\mathcal{B}_a = \{A \in \mathcal{B}, \text{ tel que : } A \equiv a_{\mathcal{B}}\}.$$

Ainsi, si un MNS $\mathcal{B}$ est redondant, alors il existe au moins un entier $a \in \mathbb{Z}/p\mathbb{Z}$ tel que : $\#\mathcal{B}_a > 1$. Nous rappelons que $A \equiv a_{\mathcal{B}}$ signifie que le polynôme A est une représentation de a dans $\mathcal{B}$ (définition 1.4).

Exemple 1.2. Pour le MNS $\mathcal{B} = (19, 3, 7, 2)$ de l'exemple 1.1, on a : $\mathcal{B}_{11} = \{(-X - 1),\ X^2\}$. En d'autres termes, les polynômes $(-X - 1)$ et X^2 sont les représentations de 11 dans $\mathcal{B}$.

1.3.2 Opérations arithmétiques dans le MNS

Dans le MNS, les calculs arithmétiques sont effectués sur les polynômes. Soient $V \equiv x_{\mathcal{B}}$ et $W \equiv y_{\mathcal{B}}$ deux polynômes d'un MNS $\mathcal{B}$.

Le polynôme $T = VW$ satisfait $T(\gamma) \equiv xy \pmod{p}$. Cependant, T pourrait ne pas être une représentation valide de xy dans $\mathcal{B}$, car son degré pourrait être supérieur ou égal à n. Pour garder le degré inférieur à n, le produit VW doit être calculé modulo un polynôme unitaire E tel que : $E(\gamma) \equiv 0 \pmod{p}$ et $\deg(E) = n$. Cette opération s'appelle la *réduction externe* et E est appelé le *polynôme de réduction externe*. Notons que si $E(\gamma) \equiv 0 \pmod{p}$ et $T = VW \pmod{E}$, alors $T(\gamma) \equiv xy \pmod{p}$ et $\deg(T) < n$. La réduction modulo le polynôme E ne suffit pas pour garantir que le résultat sera dans $\mathcal{B}$. En effet, même si $\deg(T) < n$, T pourrait ne pas être une représentation de $xy \bmod p$ dans $\mathcal{B}$, car ses coefficients pourraient être supérieurs ou égaux à ρ. Afin de garantir le résultat dans $\mathcal{B}$, une opération appelée *réduction interne* doit être appliquée. Nous reviendrons plus tard sur cette opération dans la section 1.3.5.

Le polynôme $S = V + W$ satisfait $S(\gamma) \equiv (x + y) \pmod{p}$ et $\deg(S) < n$. Ici aussi, S pourrait ne pas être une représentation de $x + y \bmod p$ dans $\mathcal{B}$, car ses coefficients pourraient être supérieurs ou égaux à ρ. Donc, une réduction interne pourrait être nécessaire pour garantir que le résultat soit dans $\mathcal{B}$.

Tel que défini, le MNS ne fournit pas les outils nécessaires pour effectuer les réductions interne et externe. Afin de réaliser efficacement ces réductions, des sous-classes de MNS ont été introduites. Il s'agit du système de représentation modulaire adapté (AMNS) [BIP04] et du système de représentation modulaire polynomiale (PMNS) [BIP05].

1.3.3 L'AMNS et le PMNS

Dans [BIP04], Bajard et al. définissent l'AMNS comme un MNS dont le paramètre γ possède une propriété particulière destinée à faciliter la réduction modulaire.

Définition 1.10. [BIP04] Un AMNS est un MNS $\mathcal{B} = (p, n, \gamma, \rho)$ tel que $\gamma^n \equiv \lambda \pmod{p}$, avec $|\lambda| \neq 0$ très petite (par exemple, $\lambda = \pm 1, \pm 2$ ou ± 3).

Donc, γ est une racine modulo p du polynôme $E(X) = X^n - \lambda$. Ce polynôme E est le *polynôme de réduction externe* pour l'AMNS. Ainsi, l'AMNS est défini par le tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$.

Exemple 1.3. Reprenons le MNS $(19, 3, 7, 2)$ donné dans l'exemple 1.1. Nous avons $\gamma^n = 7^3 \equiv 1 \pmod{19}$, qui est très petit. Ainsi, avec $E(X) = X^3 - 1$, le tuple $(19, 3, 7, 2, E)$ définit un AMNS.

Le PMNS est une généralisation de l'AMNS qui permet un choix beaucoup plus large de polynômes de réduction externe.

Définition 1.11. Un PMNS est défini par un tuple (p, n, γ, ρ, E) tel que :

- (p, n, γ, ρ) est un MNS,

- $E \in \mathbb{Z}[X]$ est un polynôme unitaire, tel que : $\deg(E) = n$ et $E(\gamma) \equiv 0 \pmod{p}$, avec $\|E\|_\infty$ très petite.

Remarque 1.2. Dans la définition originale du PMNS [BIP05], le polynôme E est *irréductible* et de la forme $E(X) = X^n - \alpha X - \lambda \in \mathbb{Z}[X]$, avec $E(\gamma) \equiv 0 \pmod{p}$ et $\|E\|_\infty$ très petite. La définition 1.11 correspond à celle proposée dans [DDEM+19]. Cette définition généralise l'originale et permet à l'AMNS d'être une sous-classe du PMNS. De plus, nous verrons que la contrainte d'irréductibilité sur E (contrainte nécessaire uniquement pour le théorème 2 dans [BIP05]) n'est pas indispensable (voir théorème 2.2 et section 2.4.3).

1.3.4 La réduction externe

Le but de cette opération est de maintenir le degré des éléments du PMNS strictement inférieur à n. Soit $C \in \mathbb{Z}[X]$ un polynôme. Cette opération consiste à calculer un polynôme R tel que :

$$R \in \mathbb{Z}_{n-1}[X] \text{ et } R(\gamma) \equiv C(\gamma) \pmod{p}.$$

La division euclidienne de C par E calcule Q et R tels que :

$$C = Q \times E + R \ ,$$

avec $\deg(R) < n$ et $Q \in \mathbb{Z}[X]$. Ainsi, $C(\gamma) = Q(\gamma) \times E(\gamma) + R(\gamma)$. Comme $E(\gamma) \equiv 0 \pmod{p}$, on a $R(\gamma) \equiv C(\gamma) \pmod{p}$. La réduction externe est donc l'opération : $R = C \bmod E$.

La définition 1.11 a l'avantage de permettre un très vaste choix de polynômes de réduction externe, cependant la classe de polynômes de la définition 1.10 (incluse dans celle de la définition 1.11) est, de manière générale, la plus intéressante à la fois pour les performances et le surcoût mémoire de la réduction externe. Nos travaux sur le MNS reposent sur la classe des polynômes E pour l'AMNS. Dans la suite de cette thèse, nous ne considérerons donc que des polynômes E de la forme $E(X) = X^n - \lambda$.

Soit $A, B \in \mathbb{Z}_{n-1}[X]$, $C = AB$ et $R = C \bmod E$. On a donc $\deg(C) < 2n-1$ et $\deg(R) < n$. Dans sa thèse [Pla05], Thomas Plantard propose l'algorithme 13 (algorithme 28, section 3.2.1 dans la thèse) qui permet d'effectuer la réduction externe.

Remarque 1.3. Grâce à la forme du polynôme E, le produit $C = AB$ et la réduction $R = C \bmod E$ se combinent très bien. Si $n = 4$ par exemple, on peut directement calculer $R = AB \bmod E$ comme suit :

- $r_0 = a_0b_0 + \lambda(a_1b_3 + a_2b_2 + a_3b_1)$
- $r_1 = a_0b_1 + a_1b_0 + \lambda(a_2b_3 + a_3b_2)$
- $r_2 = a_0b_2 + a_1b_1 + a_2b_0 + \lambda(a_3b_3)$
- $r_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$

Notons qu'ainsi tous les coefficients du polynôme R peuvent être calculés simultanément. Ce qui est très intéressant pour la parallélisation des opérations.

Algorithme 13 RedExt, *Réduction externe* [Pla05]

Entrée(s) : $C \in \mathbb{Z}[X]$ avec $\deg(C) < 2n-1$ et $E(X) = X^n - \lambda$

Sortie : $R \in \mathbb{Z}_{n-1}[X]$, tel que $R = C \bmod E$

1: **for** $i = 0 \ldots n-2$ **do**

2: $\quad r_i \leftarrow c_i + \lambda c_{n+i}$

3: **end for**

4: $r_{n-1} \leftarrow c_{n-1}$

5: retourner R $\quad \# \; R = (r_0, \ \ldots, \ r_{n-1})$

1.3.5 La réduction interne

Le but de la réduction interne est de s'assurer que la norme infinie des éléments de l'AMNS est strictement inférieure à ρ. Soit $C \in \mathbb{Z}_{n-1}[X]$ un polynôme. Cette opération consiste à calculer un polynôme R tel que :

$$R \in \mathcal{B} \text{ et } R(\gamma) \equiv C(\gamma) \pmod{p}.$$

C'est l'opération la plus complexe et aussi la plus coûteuse dans l'AMNS.

Cinq méthodes ont été proposées pour effectuer cette opération. Chacune d'elles nécessite un processus de génération d'AMNS spécifique avec un ensemble de paramètres qui lui est propre. La première méthode fut proposée par Bajard et al. dans [BIP04]. Cette méthode ramène la réduction interne à une séquence de multiplications vecteur-matrice suivie d'une addition polynomiale. Avec un bon choix de paramètres, cette méthode permet d'obtenir la meilleure performance parmi les cinq. Cependant, le processus de génération associé ne permet pas de choisir la valeur du module, mais sa taille approximative. Les quatre autres méthodes qui suivent permettent l'utilisation d'un module p quelconque. Dans [BIP05], Bajard et al. proposent deux nouvelles méthodes pour effectuer la réduction interne. Une utilisant les tables mémoire et une autre basée sur la réduction modulaire de Barrett [Bar87]. La méthode utilisant les tables mémoire n'est pas très intéressante, car elle nécessite beaucoup d'opérations de lecture et la quantité de mémoire requise augmente très rapidement selon la valeur de n et/ou de p. Dans [NP08], Negre et Plantard proposent une méthode basée sur la multiplication modulaire de Montgomery [Mon85]. Enfin, la cinquième méthode est simplement l'algorithme de Babaï [Bab86]. Dans cette section, nous présentons brièvement toutes ces méthodes, à l'exception de celle utilisant les tables mémoire. Pour une présentation détaillée de cette dernière, nous renvoyons le lecteur à la section 6.2 de [BIP05] ou la section 3.4.3 de [Pla05].

1.3.5.1 Une classe spéciale d'AMNS

Dans [BIP04], Bajard et al. présente pour la première fois l'AMNS (ainsi que le MNS). Ils proposent un ensemble de méthodes pour effectuer les opérations arithmétiques dans l'AMNS. Pour la réduction interne, ces auteurs proposent une méthode qui se résume à une multiplication vecteur-matrice plus une addition polynomiale qu'on répète autant de fois que nécessaire pour obtenir le

résultat dans l'AMNS. L'intérêt de cette méthode est que le processus de génération associé permet de choisir une matrice très creuse dont les éléments non nuls sont des puissances de deux (en valeurs absolues). Ainsi, cette multiplication vecteur-matrice devient très simple et peut s'effectuer très efficacement. Cependant, ce processus de génération ne permet pas de choisir la valeur du module p, mais sa taille (approximative), cf. algorithme 15. Cela a pour conséquence de rendre les AMNS associés à cette méthode de réduction interne non utilisables pour certains standards cryptographiques où la valeur de p est déjà fixée (voir par exemple RFC5903 pour IPSEC [SF10]).

La méthode de réduction interne. Soit $\mathcal{B} = (p, n, \gamma, \rho = 2^{k+1}, E)$ un AMNS. Les coefficients des éléments de $\mathcal{B}$ seront alors représentés sur $k+2$ bits, car ils peuvent être négatifs.

Soit $\xi \in \mathcal{B}$ tel que $\xi \equiv (2^k)_{\mathcal{B}}$, avec $\xi(X) = \xi_0 + \xi_1 X + \cdots + \xi_{n-1}X^{n-1}$.
Pour rappel, $E(X) = X^n - \lambda$. Soit $\boldsymbol{M}$ la matrice $n \times n$, dont les lignes $\boldsymbol{m}_i$ sont telles que : $\boldsymbol{m}_i \equiv (2^k\gamma^i)_{\mathcal{B}}$, pour $i = 0, \ldots, n-1$. La matrice $\boldsymbol{M}$ est appelée la *matrice de réduction* et est telle que :

$$\boldsymbol{M} = \begin{pmatrix} \xi_0 & \xi_1 & \cdots & \xi_{n-2} & \xi_{n-1} \\ \lambda\xi_{n-1} & \xi_0 & \cdots & \xi_{n-3} & \xi_{n-2} \\ \vdots & & \ddots & & \vdots \\ \lambda\xi_2 & \lambda\xi_3 & \cdots & \xi_0 & \xi_1 \\ \lambda\xi_1 & \lambda\xi_2 & \cdots & \lambda\xi_{n-1} & \xi_0 \end{pmatrix} \overset{\mathcal{B}}{\equiv} \begin{pmatrix} 2^k & 0 & \cdots & 0 & 0 \\ 0 & 2^k & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 2^k & 0 \\ 0 & 0 & \cdots & 0 & 2^k \end{pmatrix} \tag{1.2}$$

Le principe de la réduction de coefficients est le suivant. Soit $v \in \mathbb{Z}^n$, avec un coefficient v_i tel que $|v_i| \geqslant \rho$. Posons $v_i = l + 2^k h$, on a alors :

$$v = (v_0, \ldots, v_{i-1}, l, v_{i+1}, \ldots, v_{n-1}) + h(0, \ldots, 0, 2^k, 0, \ldots, 0).$$

Puisque $\boldsymbol{m}_i \equiv (2^k\gamma^i)_{\mathcal{B}}$, on a :

$$v \overset{\mathcal{B}}{\equiv} (v_0, \ldots, v_{i-1}, l, v_{i+1}, \ldots, v_{n-1}) + h.\boldsymbol{m}_i.$$

Donc, si $\|\boldsymbol{m}_i\|_\infty$ est très petite, alors la réduction de coefficients sera très efficace. Car, la norme infinie du vecteur obtenu après la réduction de v sera très petite par rapport à celle de v. L'algorithme 14 permet de réaliser cette opération sur l'ensemble des coefficients de v.

Algorithme 14 *Réduction de coefficients* [BIP04]

Entrée(s) : $v \in \mathbb{Z}^n$ et $\mathcal{B} = (p,\ n,\ \gamma,\ \rho,\ \lambda)$

Sortie : $s \in \mathbb{Z}^n$, tel que $s \overset{\mathcal{B}}{\equiv} v$

1: Soient u, w tels que $v = u2^k + w$
2: $s \leftarrow u\boldsymbol{M} + w$
3: retourner s

Pour obtenir un résultat de norme infinie inférieure à ρ, on appelle autant de fois que nécessaire l'algorithme 14 sur le vecteur à réduire. Dans [Pla05] (section 3.3), Plantard fournit un ensemble d'outils et de propriétés pour estimer et maîtriser le nombre d'appels à l'algorithme 14 pour la réduction interne.

Le processus de génération. Il est évident que l'efficacité de l'algorithme 14 dépend fortement du coût de l'opération $u\boldsymbol{M}$ (ligne 2). Le coût de ce dernier dépend des coefficients de $\boldsymbol{M}$. On peut remarquer que les coefficients de $\boldsymbol{M}$ sont entièrement définis par ξ et λ. Par conséquent, pour une réduction interne très efficace, il suffit par exemple de prendre $\lambda = \pm 2^h$, avec $h \in \mathbb{N}$ et de maximiser le nombre de coefficients ξ_i nuls, avec ceux non nuls égaux à $\pm$ des puissances de deux. L'avantage du processus de génération associé à cette méthode de réduction interne est qu'elle permet d'effectuer ce choix. Pour cet algorithme, la remarque suivante est indispensable.

Remarque 1.4. (À propos des paramètres p et γ)

1. **Le module p :** L'équation 1.2 montre que la matrice de réduction $\boldsymbol{M}$ est équivalente à la matrice $2^k Id$, où Id est la matrice identité. Ainsi, la matrice $2^k Id - \boldsymbol{M}$ est équivalente à la matrice nulle (modulo p) dans $\mathcal{B}$; i.e. $(2^k Id - \boldsymbol{M}) \overset{\mathcal{B}}{\equiv} 0$. On en déduit donc que $\det(2^k Id - \mathcal{M}) \equiv 0 \pmod{p}$. Par conséquent, le module p est un diviseur (premier) du déterminant de la matrice $2^k Id - \mathcal{M}$.
2. **La base γ :** On a par définition $E(\gamma) \equiv 0 \pmod{p}$ et $\xi \equiv (2^k)_{\mathcal{B}}$. Donc, le polynôme $I(X)$ définie par : $I(X) = 2^k - \xi(X)$ est tel que $I(\gamma) \equiv 0 \pmod{p}$. Ainsi, γ est à la fois une racine de E et de I modulo p. Par conséquent, pgcd(E, I) a aussi γ comme racine modulo p. On prendra donc γ comme une racine de $pgcd(E, I) \bmod p$.

L'algorithme 15 est le processus de génération proposé dans [Pla05] (algorithme 35, page 65). Cet algorithme génère (s'il en trouve, selon le choix des paramètres) un AMNS dont le module p est de taille inférieure à nk.

Algorithme 15 *Création d'un système de représentation adapté* [Pla05]

Sortie : $\mathcal{B} = (p, n, \gamma, \rho, \lambda)$

1: Choisir la valeur de k.
2: Choisir le nombre de chiffres n pour avoir nk tel que $p < 2^{nk}$
3: **while** $\mathcal{B}$ *ne convient pas* **do** # i.e. module p de taille souhaitée pas trouvé.
4: Choisir la valeur de λ # donc le polynôme E.
5: Choisir ξ tel que ses coefficients ξ_i sont petits ou nuls
6: Construire $\boldsymbol{M}$
7: $p \leftarrow \det(2^k Id - \boldsymbol{M})$ # ou un facteur de taille convenable.
8: **if** p premier **then**
9: $\gamma \leftarrow$ racine de $pgcd(E, I) \bmod p$
10: Choisir $\rho = 2^{k+1}$
11: **end if**
12: **end while**
13: retourner $\mathcal{B}$

On peut remarquer que l'algorithme 15 ne permet pas de choisir la valeur de p mais seulement sa taille (approximative). De plus, ce processus de génération

ne permet pas d'expliciter une classe bien précise d'entiers pour les AMNS générés.

Terminons cette section avec deux exemples d'AMNS intéressants de cette classe. Les polynômes ξ associés à ces AMNS sont très creux. Les matrices $\boldsymbol{M}$ correspondantes sont donc également très creuses. Par conséquent, la réduction de coefficients (avec l'algorithme 14) sera très efficace.

Exemple 1.4. Pour ces exemples, l désigne la taille (en bits) du module p.

1. $l = 272$
   $n = 5$
   $k = 60$
   $\lambda = 2 \quad \#$ donc $E(X) = X^5 - 2$
   $\xi(X) = -X^4$
   $\gamma = 220855883097298041197912187592864814478435487109452369765200775161577472$
   $p = 615289613528856037467994537197468968883516815174256440810456537360058156426045 1457$

2. $l = 257$
   $n = 5$
   $k = 60$
   $\lambda = 3 \quad \#$ donc $E(X) = X^5 - 3$
   $\xi(X) = -X$
   $\gamma = 173906229561255754244200354041550496362564702771423357439245446699162222002223$
   $p = 173906229561255754244200354041550496362564702771423357439246599620666828849199$

1.3.5.2 Réduction interne avec la méthode de Babaï

Comme mentionné plus haut, cette méthode est l'algorithme de Babaï [Bab86] qui résout le problème α-CVP$_\infty$. Ici, nous présentons brièvement cet algorithme. Une explication détaillée est donnée dans [SD16] (section 5.2). La compréhension de cette méthode nécessite certaines connaissances sur les réseaux euclidiens. Dans l'annexe A.2, un rappel est effectué.

Rappelons que si un ensemble E est un espace vectoriel de dimension d, alors un hyperplan de E est un sous-espace de E de dimension $d-1$. Par exemple, les hyperplans d'un plan vectoriel (qui est de dimension 2) sont des droites vectorielles (qui sont de dimension 1).

Soit $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ une famille libre de vecteurs de $\mathbb{R}^n$ et $\mathcal{L}$ le réseau engendré par cette famille. Soit $\boldsymbol{B}^* = (\boldsymbol{b}_0^*, \ldots, \boldsymbol{b}_{i-1}^*)$ la base orthogonalisée de Gram-Schmidt de $\boldsymbol{B}$. Soit $c \in \mathbb{R}$ et H_c le $\mathbb{R}$-espace vectoriel de dimension $n-1$ définie comme suit :

$$H_c = \left\{ \sum_{i=0}^{n-2} a_i \boldsymbol{b}_i + c\boldsymbol{b}_{n-1} \text{ , avec } a_i \in \mathbb{R} \right\}$$

H_c est un hyperplan de $\mathbb{R}^n$ et $H_c = H_0 + c\boldsymbol{b}_{n-1}$. Puisque les éléments du réseau $\mathcal{L}$ sont les combinaisons linéaires entières des vecteurs de la base $\boldsymbol{B}$, alors :

$$\forall \eta \in \mathcal{L}, \text{ il existe } e \in \mathbb{Z} \text{ tel que } \eta \in H_e$$

En d'autres termes, $\mathcal{L}$ peut être partitionné en *hyperplans réseaux* de dimensions $n-1$. Ces hyperplans peuvent également être partitionnés en hyperplans réseaux de dimensions $n-2$. Et ainsi de suite jusqu'à obtenir des hyperplans réseaux de dimensions 0, c.-à-d. un point du réseau $\mathcal{L}$.

La méthode de Babaï est un algorithme itératif basé sur ce fait. Cette méthode trouve sa solution en cherchant l'hyperplan le plus proche d'un vecteur donné. L'idée générale est la suivante. Soit $t \in \mathbb{R}^n$ le vecteur à réduire par rapport au réseau $\mathcal{L}$. L'algorithme commence par identifier l'hyperplan réseau H_e le plus proche de t. Cet hyperplan H_e (de dimension $n-1$) est constitué d'hyperplans réseaux de dimension $n-2$. L'étape suivante de l'algorithme est de déterminer l'hyperplan réseau de H_e le plus proche de t. On effectue ensuite la même opération sur l'hyperplan obtenu. On peut remarquer que la dimension de l'hyperplan réseau à retrouver diminue à chaque itération. Ainsi, après n itérations, on se retrouve avec un hyperplan réseau de dimension 0, c.-à-d. un vecteur η du réseau $\mathcal{L}$. La sortie de l'algorithme est alors : $t - \eta$. L'algorithme 16 correspond à cette méthode.

Algorithme 16 *Réduction de coefficients - Babaï*

Entrée(s) : $v \in \mathbb{Z}^n$

Sortie : $s \in \mathbb{Z}^n$, tel que $s \overset{\mathcal{L}}{\equiv} v$

1: $s \leftarrow v$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad c \leftarrow \lfloor < s, \widetilde{\boldsymbol{b}}_{n-1-i} > / \|\widetilde{\boldsymbol{b}}_{n-1-i}\|^2 \rceil$
4: $\quad s \leftarrow s - c \times \boldsymbol{b}_{n-1-i}$
5: **end for**
6: retourner s

1.3.5.3 Réduction interne avec la méthode Barrett-like

Nous présentons ici l'essentiel de la réduction interne de type Barrett (Barrett-like) proposée dans [BIP05]. Cette méthode de réduction est basée sur la réduction modulaire de Barrett présentée dans la section 1.1.2.2.

Soit $\mathcal{B} = (p, n, \gamma, \rho, E)$ un AMNS. L'algorithme 17 est la méthode Barrett-like appliquée au polynôme $V \in \mathbb{Z}_{n-1}[X]$. Cet algorithme nécessite deux polynômes $M, \hat{M} \in \mathbb{Z}_{n-1}[X]$ tels que :

$$M(\gamma) \equiv 0 \pmod{p} \text{ et } \hat{M} = \lfloor 2^{2h+l} \times M^{-1} \bmod E \rfloor,$$

avec $h = \lceil \log_2(\|M\|_\infty) + \log_2(4n|\lambda|) \rceil$ et $l = \lceil 2\log_2(n|\lambda|) \rceil$. Le polynôme M est appelé le *polynôme de réduction interne*. Notons que le polynôme $\hat{M}$ existe si et seulement si $\text{pgcd}(E, M) = 1$ dans $\mathbb{Q}[X]$.

Algorithme 17 `Barrett-like` - *Réduction de coefficients* [BIP05]

Entrée(s) : $V \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p, n, \gamma, \rho = 2^h, E)$, les polynômes M et $\hat{M}$
Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv V(\gamma) \pmod{p}$

1: $Q \leftarrow \lfloor \frac{\lfloor V/2^{h-1} \rfloor \times \hat{M} \bmod E}{2^{h+l+1}} \rfloor$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow V - T$
4: retourner S

1.3.5.4 Réduction interne avec la méthode Montgomery-like

Dans [NP08], Negre et Plantard proposent une méthode de réduction interne similaire à la réduction modulaire de Montgomery (voir section 1.1.2.2). L'idée est donc de calculer un polynôme qui permettra d'annuler la partie inférieure des coefficients du polynôme à réduire. Dans cet article, les auteurs combinent la multiplication polynomiale avec la réduction interne (voir l'algorithme 1 dans [NP08]). Ici, nous présentons la réduction interne (Montgomery-like) extraite de leur algorithme, car nous l'utiliserons pour d'autres opérations, telles que les processus de conversion. De plus, cela nous permettra de comparer la méthode Montgomery-like aux autres méthodes de réduction interne que nous avons présentées plus haut (cf. section 1.3.5.5).

Soit $\mathcal{B} = (p, n, \gamma, \rho, E)$ un AMNS. L'algorithme 18 est la méthode Montgomery-like appliquée au polynôme $V \in \mathbb{Z}_{n-1}[X]$. Cet algorithme nécessite trois paramètres supplémentaires : un entier $\phi \in \mathbb{N} \setminus \{0\}$ et deux polynômes $M, M' \in \mathbb{Z}_{n-1}[X]$. Ces polynômes sont tels que :

$$M \in \mathcal{B}, M(\gamma) \equiv 0 \pmod{p} \text{ et } M' = -M^{-1} \bmod (E, \phi).$$

Le polynôme M est appelé le *polynôme de réduction interne*.

Algorithme 18 `Montgomery-like` - *Réduction de coefficients* [NP08]

Entrée(s) : $V \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p, n, \gamma, \rho, E)$, les paramètres ϕ, M et M'.
Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv V(\gamma)\phi^{-1} \pmod{p}$

1: $Q \leftarrow V \times M' \bmod (E, \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: retourner S

Dans la méthode Montgomery-like (algorithme 18), on peut remarquer que plusieurs réductions modulo ϕ (ligne 1) et plusieurs divisions par ϕ (ligne 3) sont effectuées. Dans [NP08], les auteurs prouvent que ces divisions sont exactes. Comme pour le Montgomery classique, il est indispensable que ϕ soit une puissance de deux pour que cet algorithme soit efficace. Cependant, choisir ϕ comme une puissance de deux tout en garantissant l'existence du polynôme M' n'est pas évident. Dans [NP08], Negre et Plantard donnent une condition qui n'est

pas suffisante pour assurer l'existence de M'. Aussi, la construction qu'ils proposent ne garantit pas l'existence de M' (pour un ϕ donné), ni que ϕ puisse être choisi comme une puissance de deux.

Enfin, on peut remarquer que, contrairement aux méthodes de réduction interne précédente, la méthode Montgomery-like retourne un polynôme S tel que $S(\gamma) \equiv V(\gamma)\phi^{-1} \pmod{p}$, au lieu de $S(\gamma) \equiv V(\gamma) \pmod{p}$. Dans la section 2.3.4, nous verrons comment gérer ce facteur ϕ^{-1} avec un principe similaire à celui de la méthode de Montgomery classique.

Remarque 1.5.

- L'opération $Q \leftarrow V \times M' \bmod (E, \phi)$ (algorithme 18) correspond à l'opération $Q \leftarrow V \times M' \bmod E$ avec les coefficients de Q réduits modulo ϕ. En supposant que ϕ est une puissance de deux, une manière judicieuse d'effectuer cette opération est de d'abord réduire les coefficients de V modulo ϕ (très simple opération de masquage) avant d'effectuer le produit $V \times M'$. Aussi, tous les produits partiels lors du calcul de $V \times M'$ devraient être réduits modulo ϕ. Si $\|V\|_\infty \geqslant \phi$ (ce qui sera en général le cas avant la réduction interne), ces réductions conduiront à de meilleures performances en réduisant le coût des produits et additions partiels.
- Comme dans la remarque 1.3, les coefficients du polynôme Q peuvent être calculés simultanément. Idem pour les polynômes T et S (lignes 2 et 3).

1.3.5.5 Coûts et comparaison des méthodes de réduction interne

Dans cette section, nous comparons la méthode de Babaï (algorithme 16), la méthode Barrett-like (algorithme 17) et la méthode Montgomery-like (algorithme 18). Nous n'incluons pas la méthode présentée dans la section 1.3.5.1, car cette dernière n'est intéressante que pour une classe particulière d'AMNS. De plus, son coût est étroitement lié à la matrice de réduction M qui est choisie (indirectement) par l'utilisateur lors du processus de génération, cf. algorithme 15. Une étude de complexité (du meilleur des cas) pour cette méthode est donnée dans la section 4 de [BIP04].

Supposons une architecture de k bits. Soit $\mathcal{B} = (p, n, \gamma, \rho, E)$ un AMNS tel que $\rho \leqslant 2^{k-1}$. Nous comptons ici le nombre d'opérations élémentaires nécessaires pour ces trois algorithmes. Ces opérations sont : l'addition de deux mots de k bits (notée $\mathcal{A}$), le produit de deux mots de k bits (noté $\mathcal{M}$) ainsi que les opérations de décalage. On notera respectivement $\mathcal{S}_l^i$ et $\mathcal{S}_r^i$ un décalage à gauche et un décalage à droite de i bits. Nous y ajoutons également le coût de la division (notée $\mathcal{D}$) nécessaire uniquement pour la méthode de Babaï. $\mathcal{D}$ est la division à la ligne 3 de l'algorithme 16. Elle est faite entre deux entiers de tailles inférieures à $2k + \lceil \log_2(n) \rceil$. $\mathcal{C}$ est le coût de la fonction $\lfloor \cdot \rceil$ dans l'algorithme de Babaï (ligne 3). Le tableau 1.4 donne le coût des trois méthodes de réduction interne. Nous supposons que les entrées v et V sont telles que $\|v\|_\infty < 2^{2k}$, $\|V\|_\infty < 2^{2k}$. Enfin, pour simplifier l'estimation des coûts, nous supposons que les paramètres de ces méthodes sont tels que : $\|\boldsymbol{b}_i\|_\infty < 2^k$ et $\|\widetilde{\boldsymbol{b}}_i\|_\infty < 2^k$ (pour la méthode de Babaï), $\|M\|_\infty < 2^k$ et $\|\hat{M}\|_\infty < 2^k$ (pour la méthode Barrett-like), $\|M\|_\infty < 2^k$, $\|M'\|_\infty < 2^k$ et $\phi = 2^j \leqslant 2^k$ (pour la méthode Montgomery-like).

Méthode	Coût
Babaï	$n\mathcal{D} + 2n^2\mathcal{M} + (6n^2 - 3n)\mathcal{A} + n\mathcal{C}$
Barrett-like	$2n^2\mathcal{M} + (6n^2 - 3n)\mathcal{A} + n(\mathcal{S}_r^{h-1} + \mathcal{S}_r^{h+l+1})$
Montgomery-like	$2n^2\mathcal{M} + (4n^2 - n)\mathcal{A} + n\mathcal{S}_r^j$

TABLE 1.4 – Coûts théoriques des opérations de réduction interne.

Remarque 1.6.

- Si a et b sont des produits de deux entiers de k bits. L'opération $a + b$ est compté comme $3\mathcal{A}$. $2\mathcal{A}$ pour l'addition et $\mathcal{A}$ pour l'éventuelle propagation de retenue.
- La partie entière inférieure d'une division par une puissance de deux est équivalente à un décalage vers la droite ; cf. la méthode Barrett-like. Nous la comptons donc comme telle dans le tableau 1.4.
- Pour la méthode de Babaï, on peut pré-calculer les quantités $\|\widetilde{\boldsymbol{b}}_{n-1-i}\|^2$ (ligne 3, algorithme 16), nous n'avons donc pas introduit le coût de leurs calculs dans le tableau 1.4. Le produit scalaire $< s, \widetilde{\boldsymbol{b}}_{n-1-i} >$ (ligne 3 de l'algorithme de Babaï) coûte $n\mathcal{M} + 3(n-1)\mathcal{A}$.
- De même, pour les méthodes Barrett-like et Montgomery-like, les polynômes (M, $\hat{M}$, M') étant fixés pour l'AMNS, on optimise les multiplications par ces polynômes modulo le polynôme E (qui est aussi fixé) en pré-calculant certaines multiplications par λ. Pour $n = 4$ par exemple, l'opération $R = AM \bmod E$ est optimisée comme suit :
  - $r_0 = a_0m_0 + a_1(\lambda m_3) + a_2(\lambda m_2) + a_3(\lambda m_1)$
  - $r_1 = a_0m_1 + a_1m_0 + a_2(\lambda m_3) + a_3(\lambda m_2)$
  - $r_2 = a_0m_2 + a_1m_1 + a_2m_0 + a_3(\lambda m_3)$
  - $r_3 = a_0m_3 + a_1m_2 + a_2m_1 + a_3m_0$

  Les éléments entre parenthèses sont pré-calculables.
- Si les réductions modulo ϕ pour le calcul Q (ligne 1 de la méthode Montgomery-like) sont faites comme suggéré dans la remarque 1.5 (ce que nous supposons ici), le calcul de Q sera accéléré. En effet, les produits partiels seront inférieurs à $\phi \leqslant 2^k$. L'addition de deux produits partiels (réduits) coûtera alors $\mathcal{A}$, au lieu de $3\mathcal{A}$.

D'après le tableau 1.4, il est clair que la méthode de réduction interne la plus intéressante est la méthode Montgomery-like.

1.3.6 Applications de l'AMNS

L'AMNS a été utilisé dans quelques applications. Dans [EN09], El Mrabet et Negre présentent une nouvelle approche utilisant l'AMNS pour la multipli-

cation dans $\mathbb{F}_{p^k}$. Leur proposition permet de réduire le nombre de multiplications dans $\mathbb{F}_p$ au prix de quelques additions supplémentaires dans $\mathbb{F}_p$. Avec des AMNS bien choisis, les auteurs obtiennent une diminution de 50 % du nombre de multiplications dans $\mathbb{F}_p$. Pour la réduction interne, ils utilisent la méthode Montgomery-like. Cependant, la question de l'existence du polynôme M' n'est pas traitée dans cet article.

Dans [EG12], El Mrabet et Gama proposent d'utiliser l'AMNS pour accélérer la multiplication dans les extensions de corps. Ici également, les auteurs utilisent la méthode Montgomery-like pour la réduction interne dans le corps de base de l'extension. Pour la première fois, ils proposent un processus de génération d'AMNS qui garantit l'existence du polynôme M', avec $E(X) = X^n + 1$ et ϕ une puissance de deux. Le processus proposé permet de construire des systèmes qui rendent les multiplications dans les extensions de corps plus efficaces que certaines solutions existantes. Ils fournissent des résultats d'implémentations logicielles et montrent que leur approche est plus rapide que la librairie NTL [Sa] dans de nombreux cas de multiplications dans les extensions de corps. Dans leur construction, ils distinguent deux cas : lorsque le module p peut être choisi librement et lorsque le module p est déjà connu. Dans ce dernier cas, la preuve de leur construction repose sur une condition qui n'est pas suffisante.

Chapitre 2

Algorithmes et processus de génération pour une arithmétique modulaire efficace dans l'AMNS

Sommaire

Dans la section 1.3, nous avons présenté et fait un état de l'art du MNS, de l'AMNS et du PMNS. Dans ce chapitre, nous faisons un bilan de cet état de l'art

en soulignant un ensemble de questions restées sans réponse pour une utilisation efficace de l'AMNS. Cela servira de contexte à nos travaux de recherches sur l'AMNS ainsi que nos contributions. Avec ces contributions, nous verrons que l'AMNS peut être une alternative intéressante aux systèmes classiques pour l'arithmétique modulaire.

Ce chapitre est organisé comme suit. Nous commençons par faire un bilan de l'état de l'art de l'AMNS. Ensuite, nous donnons des résultats généraux qui établissent le lien entre le MNS et les réseaux euclidiens, suivis par des résultats essentiels pour les opérations dans l'AMNS. Par suite, nous présentons un ensemble complet d'algorithmes sans branchement conditionnel pour l'arithmétique modulaire dans ce système. Nous présentons également un nouveau processus de génération de paramètres pour effectuer efficacement les opérations arithmétiques. Ensuite, nous effectuons une analyse avec quelques résultats de comparaison et d'implémentation. Nous terminons ce chapitre par une conclusion résumant nos contributions avec quelques perspectives au regard de ces contributions. Le travail présenté ici a fait l'objet d'une publication [DDV20]. Le présent chapitre contient des éléments supplémentaires avec quelques améliorations.

Remarque 2.1. Pour comprendre certains éléments présentés dans ce chapitre, la connaissance d'un certain nombre de notions sur les réseaux euclidiens et le résultant de polynômes est requise. Dans l'annexe A, nous rappelons toutes les notions qu'il est indispensable de connaître.

2.1 Bilan de l'état de l'art

La comparaison effectuée à la section 1.3.5.5 montre que la méthode Montgomery-like (en prenant le paramètre ϕ comme une puissance de deux) est la meilleure méthode pour la réduction interne, lorsqu'on ne peut pas utiliser la classe spéciale d'AMNS présentée dans la section 1.3.5.1. Cependant, jusqu'à présent aucun processus exact de génération d'AMNS permettant d'utiliser efficacement cette méthode n'a été proposé. Une de nos contributions est de proposer une méthode de génération du polynôme M garantissant l'existence du polynôme M' pour tout $\phi = 2^k$, avec $k \in \mathbb{N} \setminus \{0\}$ (voir sections 2.4.3 et 2.4.4). Avec cette contribution, nous verrons que, pour tout nombre premier $p \geqslant 3$, il est toujours possible de construire plusieurs AMNS qui permettent d'effectuer de manière efficace les opérations arithmétiques en utilisant la méthode Montgomery-like pour la réduction interne.

Aussi, nous présenterons les bornes optimales pour la consistance des opérations arithmétiques dans l'AMNS en utilisant la méthode Montgomery-like. Ces bornes sont des affinements de celles proposées dans [BIP04, NP08].

Dans [BIP05], Bajard et al. donnent une condition suffisante pour qu'un tuple (p, n, γ, ρ) soit un MNS. Dans le théorème 2 de cet article, les auteurs affirment qu'une fois les paramètres p, n, E et γ fixés (avec $E(X) = X^n + \alpha X + \beta$ *irréductible*), si ρ est tel que $\rho \geqslant (|\alpha| + |\beta|)p^{1/n}$, alors le tuple (p, n, γ, ρ) définit un MNS. Cependant, le déroulement de leur preuve montre qu'un facteur n

est manquant sur la borne de ρ. En effet, dans la preuve du théorème, on peut constater que les auteurs utilisent la réduction des éléments de $\mathbb{Z}^n$ dans le domaine fondamental $\mathcal{H}'$ du réseau définie par la matrice $\boldsymbol{B}$ (de l'équation 9 du même article). La conclusion logique de cette preuve devrait conduire à $\rho \geqslant n(|\alpha| + |\beta|)p^{1/n}$ (au lieu de $\rho \geqslant (|\alpha| + |\beta|)p^{1/n}$). Dans la section 2.2.2, nous donnons une condition suffisante pour qu'un tuple (p, n, γ, ρ, E) soit un AMNS. Cette condition n'impose pas que le polynôme E soit irréductible. Notons que si (p, n, γ, ρ, E) est un AMNS, alors (p, n, γ, ρ) est un MNS.

Comme mentionné dans l'introduction, une réduction interne peut être nécessaire après l'addition de deux éléments de l'AMNS. Cependant, la réduction interne est trop coûteuse par rapport à la simple addition polynomiale. Pour s'en convaincre, nous renvoyons le lecteur vers les algorithmes de réduction interne (section 1.3.5). Nous avons donc introduit un nouveau paramètre δ pour résoudre ce "problème". Ce paramètre δ est tel qu'une seule réduction interne suffit pour ramener dans l'AMNS un polynôme $V = AB \bmod E$, où A et B sont chacun la somme de jusqu'à $(\delta + 1)$ éléments de l'AMNS. Nous verrons que la valeur de δ dépend de l'application cible et sera choisie pendant le processus de génération de l'AMNS.

Enfin, comme mentionné dans la section 1.3.5.4, la réduction interne avec la méthode Montgomery-like ajoute un facteur ϕ^{-1} à la sortie. Nous utiliserons une approche similaire à celle du Montgomery classique pour gérer ce facteur.

2.2 Outils mathématiques

Dans cette section, nous présentons un ensemble de résultats fondamentaux pour la génération des AMNS et la consistance des opérations arithmétiques dans ce système. Nous commençons par des éléments qui font le lien entre le MNS (et donc l'AMNS) et les réseaux euclidiens. Ensuite, nous donnons des résultats qui sont propres à l'AMNS.

2.2.1 Réseaux euclidiens et MNS

À chaque MNS $\mathcal{B} = (p, n, \gamma, \rho)$, on associe le réseau euclidien $\mathcal{L}_\mathcal{B}$ des polynômes de $\mathbb{Z}_{n-1}[X]$, ayant γ comme racine modulo p :

$$\mathcal{L}_\mathcal{B} = \{V \in \mathbb{Z}_{n-1}[X], \text{ tel que : } V(\gamma) \equiv 0 \pmod{p}\}$$

Ce réseau a été proposé par Bajard et al. dans [BIP05].

Le théorème 2.1 montre que $\mathcal{L}_\mathcal{B}$ est bien un réseau euclidien. Ce théorème est une simple généralisation du lemme 1 dans [BIP05]. Cette généralisation nous sera utile pour le processus de génération d'AMNS que nous présentons dans la section 2.4.

Théorème 2.1. *Soit $\mathcal{B} = (p, n, \gamma, \rho)$ un MNS. L'ensemble $\mathcal{L}_\mathcal{B}$ est un réseau*

total de dimension n, dont une base est la matrice $\boldsymbol{A}$ suivante :

$$\boldsymbol{A} = \begin{pmatrix} p & 0 & 0 & \dots & 0 & 0 \\ t_1 & 1 & 0 & \dots & 0 & 0 \\ t_2 & 0 & 1 & \dots & 0 & 0 \\ \vdots & & & \ddots & & \vdots \\ t_{n-2} & 0 & 0 & \dots & 1 & 0 \\ t_{n-1} & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \begin{matrix} \leftarrow p \\ \leftarrow X + t_1 \\ \leftarrow X^2 + t_2 \\ \\ \leftarrow X^{n-2} + t_{n-2} \\ \leftarrow X^{n-1} + t_{n-1} \end{matrix} \tag{2.1}$$

avec $t_i = -\gamma^i + k_i p$, et $k_i \in \mathbb{Z}$ quelconque.

Démonstration. Il est évident que les lignes de la matrice $\boldsymbol{A}$ sont linéairement indépendantes et donc que le réseau $\mathcal{L}$ généré par ces vecteurs est un réseau total de dimension n. De plus, chaque chaque vecteur ligne $\boldsymbol{a}_i$ de cette matrice représente un polynôme ayant γ comme racine modulo p. Par exemple, la deuxième ligne correspond au polynôme $\boldsymbol{a}_1(X) = X + t_1$ et on a bien : $a_1(\gamma) = \gamma + t_1 \equiv 0 \pmod{p}$, $\forall\ k_1 \in \mathbb{Z}$. Donc, $\forall\ V \in \mathcal{L}$, on a $V(\gamma) \equiv 0 \pmod{p}$; i.e. $\mathcal{L} \subseteq \mathcal{L}_{\mathcal{B}}$.

Montrons maintenant que $\mathcal{L}_{\mathcal{B}} \subseteq \mathcal{L}$. Soit $W = (w_0, \dots, w_{n-1}) \in \mathcal{L}_{\mathcal{B}}$. On a $W(\gamma) \equiv 0 \pmod{p}$, donc le vecteur $Z_0 = (W(\gamma)/p, w_1, \dots, w_{n-1}) \in \mathbb{Z}^n$. On a aussi : $Z_0.\boldsymbol{A} = (W(\gamma) + x, w_1, \dots, w_{n-1})$, avec $x = \sum_{i=1}^{n-1} w_i t_i$.

Puisque $t_i = -\gamma^i + k_i p$, on a : $x = -\left(\sum_{i=1}^{n-1} w_i \gamma^i\right) + p\left(\sum_{i=1}^{n-1} w_i k_i\right)$.

Ainsi, $x = (-W(\gamma) + w_0) + p\left(\sum_{i=1}^{n-1} w_i k_i\right)$.

Donc, $Z_0.\boldsymbol{A} = \left(w_0 + p\left(\sum_{i=1}^{n-1} w_i k_i\right), w_1, \dots, w_{n-1}\right)$.

Par suite, $Z_0.\boldsymbol{A} = (w_0, w_1, \dots, w_{n-1}) + p\left(\sum_{i=1}^{n-1} w_i k_i, 0, \dots, 0\right)$.

Soit $Z_1 = \left(-\sum_{i=1}^{n-1} w_i k_i, 0, \dots, 0\right)$. On a alors $(Z_0 + Z_1).\boldsymbol{A} = (w_0, w_1, \dots, w_{n-1})$.
On a ainsi construit un vecteur $Z = (Z_0 + Z_1) \in \mathbb{Z}^n$, tel que $Z.\boldsymbol{A} = W$.
Donc, $W \in \mathcal{L}$; i.e. $\mathcal{L}_{\mathcal{B}} \subseteq \mathcal{L}$. D'où, $\mathcal{L}_{\mathcal{B}} = \mathcal{L}$. □

Pour la génération des paramètres de l'AMNS, nous serons amenés à rechercher des vecteurs courts (i.e. α-SVP$_\infty$) du réseau $\mathcal{L}_{\mathcal{B}}$. Nous verrons que les normes infinies de ces vecteurs influent grandement sur la qualité de l'AMNS. Le corollaire 2.1 nous garantit l'existence d'au moins un vecteur non nul dans le réseau $\mathcal{L}_{\mathcal{B}}$ avec une norme infinie satisfaisante. Ce corollaire est le lemme 2 dans [BIP05].

Corollaire 2.1. *[BIP05] Soit $\mathcal{B} = (p, n, \gamma, \rho)$ un MNS et $\mathcal{L}_{\mathcal{B}}$ le réseau euclidien associé. Il existe (au moins) un vecteur non nul $M \in \mathcal{L}_{\mathcal{B}}$, tel que :*

$$\|M\|_\infty \leqslant p^{1/n}.$$

Démonstration. La matrice $\boldsymbol{A}$ du théorème 2.1 est une base du réseau $\mathcal{L}_{\mathcal{B}}$. Donc, on a : $\det(\mathcal{L}_{\mathcal{B}}) = |\det(\boldsymbol{A})| = p$. D'après le théorème A.4 (de Minkowski), il existe (au moins) un vecteur non nul $M \in \mathcal{L}_{\mathcal{B}}$ tel que : $0 < \|M\|_\infty \leqslant \det(\mathcal{L}_{\mathcal{B}})^{1/n}$. □

Dans la section 2.2.2, nous établirons une condition suffisante pour qu'un tuple (p, n, γ, ρ, E) soit un AMNS. Le lemme qui suit nous sera utile à cette fin.

Lemme 2.1. *Soit* $E(X) = X^n + e_{n-1}X^{n-1} + \cdots + e_1X + e_0 \in \mathbb{Z}[X]$ *un polynôme et* $M(X) = m_0 + m_1X + \cdots + m_{n-1}X^{n-1} \in \mathcal{L}_{\mathcal{B}}$. *Si* $E(\gamma) \equiv 0 \pmod p$ *et* $\gcd(E, M) = 1$ *dans* $\mathbb{Q}[X]$, *alors la matrice* $\boldsymbol{B}$ *(ci-dessous) engendre un réseau total* $\mathcal{L}$ *de dimension* n. *De plus,* $\mathcal{L}$ *est un sous réseau de* $\mathcal{L}_{\mathcal{B}}$ *; i.e.* $\mathcal{L} \subset \mathcal{L}_{\mathcal{B}}$.

$$\boldsymbol{B} = \begin{pmatrix} m_0 & m_1 & \dots & m_{n-1} \\ \dots & \dots & \dots & \dots \\ \vdots & \vdots & & \vdots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{2.2}$$

Note : chaque ligne i de $\boldsymbol{B}$ correspond à la représentation vectorielle du polynôme $X^iM \bmod E$.

Démonstration. Soit $B_i = X^iM \bmod E$, pour $i = 0, \dots, n-1$. Soit $\mathcal{L}$ le réseau engendré par $\boldsymbol{B}$. On a $\deg(E) = n$ et $E(\gamma) \equiv 0 \pmod p$, donc $B_i(\gamma) \equiv 0 \pmod p$ et $\deg(B_i) < n$. Par conséquent, $B_i \in \mathcal{L}_{\mathcal{B}}$. D'où, $\mathcal{L} \subset \mathcal{L}_{\mathcal{B}}$ (par définition du réseau).

Pour montrer que $\mathcal{L}$ est un réseau total de dimension n, il suffit de montrer que les vecteurs lignes b_i (associés aux polynômes B_i) de $\boldsymbol{B}$ sont linéairement indépendants. Supposons le contraire. Alors, il existe un vecteur $z = (z_0, z_1, \dots, z_{n-1}) \in \mathbb{Z}^n$, différent du vecteur nul, tel que $z.\boldsymbol{B} = (0, \dots, 0)$; i.e. $\sum_{i=0}^{n-1} z_i.b_i = (0, \dots, 0)$. Puisque $B_i = X^iM \bmod E$, cela signifie en notation polynomiale que $\sum_{i=0}^{n-1} z_iX^iM \equiv 0 \pmod E$; i.e. $ZM \equiv 0 \pmod E$, avec $Z(X) = z_0 + z_1X + \cdots + z_{n-1}X^{n-1}$. Puisque z est différent du vecteur nul, on a $Z \neq 0$. En outre, $\gcd(E, M) = 1$ dans $\mathbb{Q}[X]$, donc $ZM \equiv 0 \pmod E$ implique que E divise Z dans $\mathbb{Q}[X]$. Or, on a $\deg(Z) < n$ et $\deg(E) = n$. Donc, E ne peut pas diviser Z, car $Z \neq 0$. On a donc une absurdité. Par conséquent, les vecteurs lignes de $\boldsymbol{B}$ sont linéairement indépendants. □

Remarque 2.2. Le réseau $\mathcal{L}$ du lemme 2.1 est le sous-réseau de $\mathcal{L}_{\mathcal{B}}$ des multiples (modulo E) du polynôme M. C-à-d :

$$\mathcal{L} = \{Q.M \bmod E, \text{ avec : } Q \in \mathbb{Z}_{n-1}[X]\}$$

En effet, si $v \in \mathcal{L}$, alors il existe $q \in \mathbb{Z}^n$ tel que $v = q\boldsymbol{B}$. En notation polynomiale, cela signifie que $V = QM \bmod E$; V, Q et M étant respectivement les polynômes associés aux vecteurs v, q et m. Le raisonnement inverse est trivial.

Terminons cette partie par une interprétation, basée sur les réseaux, de la méthode Montgomery-like pour la réduction de coefficients dans l'AMNS. Pour rappel, $E(X) = X^n - \lambda$, avec $E(\gamma) \equiv 0 \pmod p$. La méthode Montgomery-like nécessite deux polynômes M et M' tels que : $M(\gamma) \equiv 0 \pmod p$ et $M' = -M^{-1} \bmod (E, \phi)$. Si $\text{pgcd}(E, M) = 1$ dans $\mathbb{Q}[X]$ (nous verrons que ce sera

toujours le cas avec cette méthode pour assurer l'existence du polynôme M'), alors d'après le lemme 2.1 la matrice $\boldsymbol{B}$ (ci-dessous) engendre un réseau $\mathcal{L}$ de dimension n, avec $\mathcal{L} \subset \mathcal{L}_{\mathcal{B}}$.

$$\boldsymbol{B} = \begin{pmatrix} m_0 & m_1 & \dots & m_{n-1} \\ \lambda m_{n-1} & m_0 & \dots & m_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda m_1 & \lambda m_2 & \dots & m_0 \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{2.3}$$

Ainsi, l'opération $T \leftarrow Q \times M \bmod E$ (ligne 2 de la méthode Montgomery-like) est équivalente au produit vecteur-matrice $t = q\boldsymbol{B}$, où t et q sont respectivement les vecteurs correspondant aux polynômes T et Q. Puisque $q \in \mathbb{Z}^n$, on a $t \in \mathcal{L}_{\mathcal{B}}$. La sortie S de l'algorithme peut alors être réécrite comme $s = (v + t)/\phi$.
En résumé, la méthode Montgomery-like calcule un vecteur t du sous-réseau $\mathcal{L}$ tel que : $(v_i + t_i) \equiv 0 \pmod{\phi}$, pour $i = 0, \dots, (n-1)$. La réduction est alors obtenue en divisant chaque somme $(v_i + t_i)$ par ϕ.

2.2.2 Quelques résultats essentiels sur l'AMNS

Dans cette section, nous donnons la borne optimale sur la norme infinie du résultat d'une multiplication polynomiale modulaire. Ensuite, nous présentons une condition suffisante pour qu'un tuple (p, n, γ, ρ, E) soit un AMNS, ainsi que quelques résultats indispensables pour les algorithmes que nous verrons plus loin.

Soit $A, B \in \mathbb{Z}_{n-1}[X]$ et $R = AB \bmod E$ avec $E(X) = X^n - \lambda$, avec $\lambda \neq 0$. Dans [BIP04] (section 3), Bajard et al. expliquent que $\|R\|_\infty < n|\lambda|\|A\|_\infty\|B\|_\infty$. La proposition 2.1 donne une borne légèrement plus fine sur $\|R\|_\infty$. Cette borne est optimale, car elle peut être atteinte. Nous verrons que cette borne, assez triviale, améliore l'essentiel des paramètres de l'AMNS.

Proposition 2.1. *Soit $A, B \in \mathbb{Z}_{n-1}[X]$ et $E(X) = X^n - \lambda$, avec $\lambda \in \mathbb{Z} \setminus \{0\}$. Si $R = AB \bmod E$, alors :*

$$\|R\|_\infty \leqslant w\|A\|_\infty\|B\|_\infty,$$

avec $w = 1 + (n-1)|\lambda|$

Démonstration. Soit $C = AB$. On a $\deg(C) < 2n - 1$ et il est évident que $\|C\|_\infty \leqslant n\|A\|_\infty\|B\|_\infty$. Plus précisément, on a :

- $|c_i| \leqslant (i+1)\|A\|_\infty\|B\|_\infty$, pour $i = 0, \dots, (n-1)$,
- $|c_i| \leqslant (2n - (i+1))\|A\|_\infty\|B\|_\infty$, pour $i = n, \dots, (2n-2)$.

Ainsi, avec l'algorithme 13, on peut voir que la valeur maximale de $\|R\|_\infty$ ne peut être atteinte qu'en $r_0 = c_0 + \lambda c_n$, avec $|r_0| \leqslant \|A\|_\infty\|B\|_\infty + |\lambda|(n-1)\|A\|_\infty\|B\|_\infty$, car $\lambda \neq 0$. D'où, $\|R\|_\infty \leqslant (1 + (n-1)|\lambda|)\|A\|_\infty\|B\|_\infty$. □

Le théorème qui suit donne une condition suffisante pour qu'un tuple (p, n, γ, ρ, E) soit un AMNS. Nous rappelons que si (p, n, γ, ρ, E) est un AMNS, alors (p, n, γ, ρ) est un MNS.

Théorème 2.2. *Soit $p \geqslant 3$ un entier. Soit $n \geqslant 1$ un entier et $\lambda \in \mathbb{Z} \setminus \{0\}$. Soit $E(X) = X^n - \lambda$ un polynôme tel que $E(\gamma) \equiv 0 \pmod{p}$, avec $\gamma \in \mathbb{Z}/p\mathbb{Z}$. S'il existe un polynôme $M \in \mathbb{Z}_{n-1}[X]$ tel que $M(\gamma) \equiv 0 \pmod{p}$ et gcd(E, M) = 1 dans $\mathbb{Q}[X]$, alors le tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ définit un AMNS si*

$$\rho > \frac{1}{2}\|\boldsymbol{B}\|_1,$$

avec

$$\boldsymbol{B} = \begin{pmatrix} m_0 & m_1 & m_2 & \dots & m_{n-2} & m_{n-1} \\ \lambda m_{n-1} & m_0 & m_1 & \dots & m_{n-3} & m_{n-2} \\ \lambda m_{n-2} & \lambda m_{n-1} & m_0 & \dots & m_{n-4} & m_{n-3} \\ \vdots & & & \ddots & & \vdots \\ \lambda m_2 & \lambda m_3 & \lambda m_4 & \dots & m_0 & m_1 \\ \lambda m_1 & \lambda m_2 & \lambda m_3 & \dots & \lambda m_{n-1} & m_0 \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \leftarrow X^2.M \bmod E \\ \\ \leftarrow X^{n-2}.M \bmod E \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{2.4}$$

Démonstration. Nous reprenons ici l'idée de la preuve du théorème 2 dans [BIP05], avec quelques modifications.

Soit $x \in \mathbb{Z}/p\mathbb{Z}$ un entier et $v \in \mathbb{Z}^n$ un vecteur, tel que $v = (v_0, v_1, \dots, v_{n-1})$. Nous rappelons que v est une représentation de x si le polynôme $V(X) = v_0 + v_1 X + \cdots + v_{n-1}X^{n-1}$ est tel que $V(\gamma) \equiv x \pmod{p}$.
Avec le lemme 2.1, nous avons montré que la matrice $\boldsymbol{B}$ ci-dessus est une base du réseau $\mathcal{L} = \{Q.M \bmod E$, avec : $Q \in \mathbb{Z}_{n-1}[X]\}$, cf. remarque 2.2.
Notons $\boldsymbol{b}_i$ chaque ligne de $\boldsymbol{B}$ et $\boldsymbol{B}_i$ le polynôme associé. Soit $\mathcal{H}'$ l'ensemble défini comme suit :

$$\mathcal{H}' = \{t \in \mathbb{R}^n, \text{ tel que : } t = \sum_{i=0}^{n-1} t_i \boldsymbol{b}_i, -\tfrac{1}{2} \leqslant t_i < \tfrac{1}{2}\}.$$

Dans l'annexe A.2.3, on a vu que $\mathcal{H}'$ est une région fondamentale du réseau $\mathcal{L}$ et donc que tout vecteur $v \in \mathbb{R}^n$ est congru modulo $\mathcal{L}$ à un unique vecteur y de $\mathcal{H}'$ (voir proposition A.4 et théorème A.2). Autrement dit : $\forall v \in \mathbb{R}^n$, $\exists!\ t \in \mathcal{H}' : v \overset{\mathcal{L}}{\equiv} t$. Dans la remarque A.3, nous avons expliqué que si $t \in \mathcal{H}'$, alors $\|t\|_\infty \leqslant \frac{1}{2}\|\boldsymbol{B}\|_1$. Nous avons également souligné le fait que la réduction dans la région $\mathcal{H}'$ est plus intéressante que la réduction dans le domaine fondamental $\mathcal{H}$, car on gagne un facteur $1/2$.

Soit $a \in \mathbb{Z}/p\mathbb{Z}$ un entier et $v = (a, 0, \dots, 0) \in \mathbb{Z}^n$. D'après ce qui a été dit plus haut, il existe un unique vecteur $t \in \mathcal{H}'$ tel que $v \overset{\mathcal{L}}{\equiv} t$. En d'autres termes, il existe un unique point $d \in \mathcal{L}$ tel que : $t = (v - d) \in \mathcal{H}'$.
Soient T et D respectivement les polynômes associés aux vecteurs t et d. Puisque $d \in \mathcal{L}$, on a $D(\gamma) \equiv 0 \pmod{p}$. Ainsi, le polynôme T est tel que : $T(\gamma) \equiv a \pmod{p}$ et $\|T\|_\infty \leqslant \frac{1}{2}\|\boldsymbol{B}\|_1$. Ce qui termine cette preuve. □

Le théorème ci-dessus a plusieurs conséquences. Le corollaire 2.2 donne une borne sur ρ pour que le tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ soit un AMNS, sans faire intervenir la norme de la matrice $\boldsymbol{B}$. C'est cette borne que nous utiliserons par la suite, car elle fait le rapprochement entre le théorème 2.2 et la proposition 2.1. Ce qui nous permettra d'établir l'ensemble des bornes nécessaires pour assurer la

consistance des opérations arithmétiques dans l'AMNS. Cette borne permettra également de garantir que le processus de génération que nous présentons plus loin produit effectivement un AMNS.

Corollaire 2.2. *Soit $p \geqslant 3$ un entier. Soit $n \geqslant 1$ un entier et $\lambda \in \mathbb{Z} \setminus \{0\}$. Soit $E(X) = X^n - \lambda$ un polynôme tel que $E(\gamma) \equiv 0 \pmod{p}$, avec $\gamma \in \mathbb{Z}/p\mathbb{Z}$. S'il existe un polynôme $M \in \mathbb{Z}_{n-1}[X]$ tel que $M(\gamma) \equiv 0 \pmod{p}$ et gcd(E, M) = 1 dans $\mathbb{Q}[X]$, alors le tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ définit un AMNS si*

$$\rho > \frac{1}{2} w \|M\|_\infty,$$

avec $w = 1 + (n-1)|\lambda|$.

Démonstration. D'après le théorème 2.2, le tuple $\mathcal{B}$ est un AMNS dès que $\rho > \frac{1}{2}\|\boldsymbol{B}\|_1$. Il est évident que $\|\boldsymbol{B}\|_1 \leqslant (1+(n-1)|\lambda|)\|M\|_\infty$, car $\lambda \neq 0$. Cette borne ne peut être atteinte qu'avec la première colonne de $\boldsymbol{B}$. D'où, $\mathcal{B}$ est un AMNS si $\rho > \frac{1}{2}(1 + (n-1)|\lambda|)\|M\|_\infty$. □

Le corollaire 2.3 donne un lien intéressant entre $\|M\|_\infty$ et p. Ce résultat nous sera utile pour garantir la consistance des opérations de conversion du binaire vers l'AMNS.

Corollaire 2.3. *Soit $p \geqslant 3$ un entier. Soit $n \geqslant 1$ un entier et $\lambda \in \mathbb{Z} \setminus \{0\}$. Soit $E(X) = X^n - \lambda$ un polynôme tel que $E(\gamma) \equiv 0 \pmod{p}$, avec $\gamma \in \mathbb{Z}/p\mathbb{Z}$. Soit $M \in \mathbb{Z}_{n-1}[X]$ un polynôme tel que $M(\gamma) \equiv 0 \pmod{p}$ et gcd(E, M) = 1 dans $\mathbb{Q}[X]$, alors :*

$$p \leqslant (\|\boldsymbol{B}\|_1 + 1)^n \leqslant (w\|M\|_\infty + 1)^n,$$

où $\boldsymbol{B}$ est la matrice du théorème 2.2 (équation 2.4) et $w = 1 + (n-1)|\lambda|$.

Démonstration. Prenons $\rho = \lfloor \frac{1}{2}\|\boldsymbol{B}\|_1 \rfloor + 1$. D'après le théorème 2.2, le tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ est un AMNS. Le nombre d'éléments de l'AMNS $\mathcal{B}$ est $\#\mathcal{B} = (2\lfloor \frac{1}{2}\|\boldsymbol{B}\|_1 \rfloor + 1)^n \leqslant (\|\boldsymbol{B}\|_1 + 1)^n$; pour rappel, les coefficients des éléments peuvent être négatifs dans ce système.

Supposons que $p > \#\mathcal{B}$. Il existe alors nécessairement un entier $a \in \mathbb{Z}/p\mathbb{Z}$ qui n'a pas de représentation dans $\mathcal{B}$. Cela est absurde, car $\mathcal{B}$ est un AMNS pour p. Donc, $p \leqslant \#\mathcal{B}$, d'où $p \leqslant (\|\boldsymbol{B}\|_1 + 1)^n$.

Comme expliqué dans la preuve du corollaire 2.2, $\|\boldsymbol{B}\|_1 \leqslant w\|M\|_\infty$. Donc, $(\|\boldsymbol{B}\|_1 + 1)^n \leqslant (w\|M\|_\infty + 1)^n$. □

2.3 Système complet pour l'arithmétique modulaire

Dans la section 1.3.5.5 du chapitre précédent, nous avons vu que la méthode Montgomery-like est la méthode de réduction interne la plus performante. Dans cette section, nous présentons l'ensemble des algorithmes nécessaires pour effectuer les opérations arithmétiques dans l'AMNS, en utilisant la méthode

Montgomery-like pour la réduction interne. Plusieurs de ces algorithmes sont des modifications de ceux proposés dans [Pla05], pour tenir compte du facteur ϕ^{-1} introduit par cette méthode lors de la réduction interne. Pour chacun des algorithmes présentés dans cette section, nous donnons les propriétés qui garantissent la consistance des opérations.

Soit $\mathcal{B} = (p, n, \gamma, \rho, E)$ un AMNS. Dans toute la suite de cette section, nous prendrons :

$$w = 1 + (n-1)|\lambda|.$$

2.3.1 La réduction interne avec la méthode Montgomery-like

Nous commençons par rappeler la méthode Montgomery-like (algorithme 18), puis présentons certaines de ses propriétés. Dans toute la suite de cette thèse, nous désignerons l'algorithme 18 par **RedCoeff**.

Soit $\phi \in \mathbb{N} \setminus \{0\}$ et $M, M' \in \mathbb{Z}_{n-1}[X]$ deux polynômes tels que :

$$M \in \mathcal{B}, M(\gamma) \equiv 0 \pmod{p} \text{ et } M' = -M^{-1} \bmod (E, \phi).$$

Le polynôme M est appelé le *polynôme de réduction interne*.

Algorithme 18 RedCoeff [NP08]

Entrée(s) : $V \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p, n, \gamma, \rho, E)$, les paramètres ϕ, M et M'.
Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv V(\gamma)\phi^{-1} \pmod{p}$

1: $Q \leftarrow V \times M' \bmod (E, \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: retourner S

Comme déjà expliqué, cette méthode n'est intéressante que si ϕ est une puissance de deux. Dans la section 2.4.4, nous montrons comment générer le polynôme de réduction interne M en garantissant l'existence du polynôme M', avec $\phi \geqslant 2$ une puissance de deux quelconque. Dans la section 2.3.4, nous voyons comment gérer le facteur ϕ^{-1} introduit par **RedCoeff**.

Soit $A, B \in \mathcal{B}$ et $V = AB \bmod E$. On a donc $\|V\|_\infty < n|\lambda|\rho^2$ (voir section 3 dans [BIP04]). Dans [NP08] (théorème 1), Negre et Plantard montrent que la sortie S de l'algorithme 18 (appliqué à V) est telle que $\|S\|_\infty < \rho$ (i.e $S \in \mathcal{B}$) si les paramètres ρ et ϕ sont tels que :

$$\rho > 2n|\lambda|\|M\|_\infty \quad \text{et} \quad \phi > 2n|\lambda|\rho.$$

Grâce à la proposition 2.1, nous donnons des bornes légèrement plus fines.

Proposition 2.2. *Soit $V \in \mathbb{Z}_{n-1}[X]$ un polynôme. Si ρ, ϕ et V sont tels que :*

$$\rho \geqslant 2w\|M\|_\infty, \quad \phi \geqslant 2w\rho \quad \text{et} \quad \|V\|_\infty \leqslant w\rho^2,$$

alors la sortie S de l'algorithme 18 (avec V comme entrée) est telle que $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$).

Démonstration. D'après la proposition 2.1, le polynôme T (ligne 2 de **RedCoeff**) est tel que $\|T\|_\infty < w\phi\|M\|_\infty$, car $\|Q\|_\infty < \phi$. Donc, $\|S\|_\infty < (w\rho^2 + w\phi\|M\|_\infty)/\phi = (w\rho^2)/\phi + w\|M\|_\infty \leqslant (w\rho^2)/\phi + \rho/2$. On a $\phi \geqslant 2w\rho$, donc $(w\rho^2)/\phi \leqslant \rho/2$. Par conséquent, $\|S\|_\infty < \rho$. □

2.3.2 La multiplication

Comme expliqué en introduction, la multiplication dans l'AMNS est une multiplication polynomiale suivie d'une réduction externe puis d'une réduction interne. L'algorithme 19 est la multiplication proposée dans [NP08], avec ici une extension du domaine des entrées à $\mathbb{Z}_{n-1}[X]$. L'intérêt de cette extension apparaîtra dans la section 2.3.3 où nous expliquons comment éviter d'avoir à effectuer une réduction interne après chaque addition.

Algorithme 19 *Multiplication dans l'AMNS* [NP08]

Entrée(s) : $A, B \in \mathbb{Z}_{n-1}[X]$ et $\mathcal{B} = (p, n, \gamma, \rho, E)$

Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod p$

1: $R \leftarrow A \times B \bmod E$

2: $S \leftarrow$ **RedCoeff**(R)

3: retourner S

Dans l'algorithme 19, on a $R(\gamma) \equiv A(\gamma)B(\gamma) \pmod p$, car $E(\gamma) \equiv 0 \pmod p$. Ainsi, cet algorithme retourne un polynôme S tel que $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod p$.

De la proposition 2.2, on déduit le corollaire 2.4 qui est assez proche du théorème 1 dans [NP08].

Corollaire 2.4. *Soit $A, B \in \mathcal{B}$. Si ρ et ϕ sont tels que :*

$$\rho \geqslant 2w\|M\|_\infty \quad \text{et} \quad \phi \geqslant 2w\rho,$$

alors la sortie S de l'algorithme 19 (avec A et B comme entrées) est telle que $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$).

Démonstration. Le polynôme R (ligne 1 de l'algorithme 19) est tel que $\|R\|_\infty < w\rho^2$, cf. proposition 2.1. La proposition 2.2 permet donc de conclure. □

2.3.3 L'addition

L'addition dans l'AMNS est la simple addition polynomiale. L'algorithme 20 correspond à cette opération.

Comme expliqué en introduction, une réduction interne peut être nécessaire après cette addition, car la norme infinie de la sortie S pourrait être supérieure (ou égale) à ρ. Il en est de même pour la soustraction, cf. algorithme 21.

Algorithme 20 *Addition dans l'AMNS*

Entrée(s) : $A \in \mathcal{B}$, $B \in \mathcal{B}$ et $\mathcal{B} = (p, n, \gamma, \rho, E)$

Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv A(\gamma) + B(\gamma) \pmod{p}$

1: $S \leftarrow A + B$

2: retourner S

Algorithme 21 *Soustraction dans l'AMNS*

Entrée(s) : $A \in \mathcal{B}$, $B \in \mathcal{B}$ et $\mathcal{B} = (p, n, \gamma, \rho, E)$

Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv A(\gamma) - B(\gamma) \pmod{p}$

1: $S \leftarrow A - B$

2: retourner S

Dans [Pla05] (algorithme 29, section 3.2.2.2), Plantard propose une addition suivie d'une réduction interne. Cependant, dans la section 2.1, nous avons fait remarquer qu'une réduction interne (quelque soit la méthode choisie) est trop coûteuse pour une simple addition polynomiale. Nous proposons ici une solution qui permet d'effectuer le nombre voulu d'additions suivies d'une multiplication, avant qu'une réduction interne soit nécessaire.

Soit δ le nombre maximum d'additions successives d'éléments de $\mathcal{B}$ qui doivent être faites avant une multiplication modulaire. Cette valeur dépend de l'application cible et est généralement connue. Par exemple, dans la cryptographie sur les courbes elliptiques, les formules d'addition et de doublement de points sont connues. La valeur de δ peut donc être déterminée très simplement. La proposition 2.3 généralise la proposition 2.2 en intégrant le nouveau paramètre δ dans ses bornes.

Proposition 2.3. *Soit $V \in \mathbb{Z}_{n-1}[X]$ un polynôme. Si ρ, ϕ et V sont tels que :*

$$\rho \geqslant 2w\|M\|_\infty, \quad \phi \geqslant 2w\rho(\delta+1)^2 \quad \text{et} \quad \|V\|_\infty \leqslant w\rho^2(\delta+1)^2,$$

alors la sortie S de l'algorithme 18 (avec V comme entrée) est telle que $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$).

Démonstration. D'après la proposition 2.1, le polynôme T (ligne 2 de **RedCoeff**) est tel que $\|T\|_\infty < w\phi\|M\|_\infty$, car $\|Q\|_\infty < \phi$. Donc, $\|S\|_\infty < (w\rho^2(\delta+1)^2 + w\phi\|M\|_\infty)/\phi = (w\rho^2(\delta+1)^2)/\phi + w\|M\|_\infty$. Ainsi, $\|S\|_\infty < (w\rho^2(\delta+1)^2)/\phi + \rho/2$. On a $\phi \geqslant 2w\rho(\delta+1)^2$, donc $(w\rho^2(\delta+1)^2)/\phi \leqslant \rho/2$. D'où, $\|S\|_\infty < \rho$. □

Soit $A \in \mathbb{Z}_{n-1}[X]$ tel que A est la somme d'au maximum $(\delta+1)$ éléments de l'AMNS. On a alors $\|A\|_\infty < (\delta+1)\rho$. Le corollaire 2.5 donne les bornes qui assurent le retour dans l'AMNS du produit de deux polynômes qui sont les résultats de la somme de jusqu'à $(\delta+1)$ éléments de l'AMNS. Ce corollaire généralise le corollaire 2.4.

Corollaire 2.5. *Soit $A, B \in \mathbb{Z}_{n-1}[X]$ tels que : $\|A\|_\infty < (\delta+1)\rho$ et $\|B\|_\infty < (\delta+1)\rho$. Si ρ et ϕ sont tels que :*

$$\rho \geqslant 2w\|M\|_\infty \quad et \quad \phi \geqslant 2w\rho(\delta+1)^2,$$

alors la sortie S de l'algorithme 19 (avec A et B comme entrées) est telle que $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$).

Démonstration. Le polynôme R (ligne 1 de l'algorithme 19) est tel que $\|R\|_\infty < w\rho^2(\delta+1)^2$, d'après proposition 2.1. La proposition 2.3 permet donc de conclure. □

Remarque 2.3.

- Pour $\delta = 0$, la proposition 2.3 et le corollaire 2.5 sont respectivement les mêmes que la proposition 2.2 et le corollaire 2.4.
- Ces généralisations n'affectent pas le paramètre ρ. Il n'y a donc pas de surcoût mémoire pour la représentation des éléments de l'AMNS.

2.3.4 Les opérations de conversion

L'algorithme 19 de multiplication modulaire utilise **RedCoeff** qui introduit un facteur ϕ^{-1} dans le résultat. Ainsi, le calcul du produit $\alpha_1\alpha_2\cdots\alpha_k$, avec $\alpha_i \in \mathbb{Z}/p\mathbb{Z}$, en utilisant leurs représentations dans $\mathcal{B}$, donne un polynôme S tel que $S(\gamma) \equiv \phi^{-\boldsymbol{k}} \prod_{i=1}^{k} \alpha_i \pmod p$. Il est donc nécessaire de gérer cette accumulation du facteur ϕ^{-1} pour ne pas perdre la valeur qui est calculée, i.e. $\prod_{i=1}^{k} \alpha_i \pmod p$.

Une solution simple pour gérer ce facteur est de convertir les valeurs α_i dans le domaine de Montgomery. Dans ce domaine, tout élément $a \in \mathbb{Z}/p\mathbb{Z}$ est remplacé par $a\phi \bmod p$ et a une représentation $A \in \mathcal{B}$ telle que $A(\gamma) \equiv a\phi \pmod p$, i.e. $A \equiv (a\phi)_{\mathcal{B}}$. Ainsi, si $A, B \in \mathcal{B}$ sont respectivement les représentations dans le domaine de Montgomery de $a, b \in \mathbb{Z}/p\mathbb{Z}$, alors **RedCoeff** appliqué à $V = AB \bmod E$ donne un polynôme $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv ab\phi \pmod p$. Donc, S est aussi dans le domaine de Montgomery.

Afin d'assurer la consistance de la multiplication modulaire dans l'AMNS, nous convertirons (et conserverons) les éléments de $\mathbb{Z}/p\mathbb{Z}$ dans le domaine de Montgomery.

Pour la suite, nous supposons que :

$$\rho \geqslant 2w\|M\|_\infty \quad \text{et} \quad \phi \geqslant 2w\rho(\delta+1)^2.$$

Pour rappel, $w = 1 + (n-1)|\lambda|$.

2.3.4.1 Garantie de consistance

Comme mentionné plus haut, **RedCoeff** n'est intéressant que si ϕ est une puissance de deux. Ce choix doit être fait en assurant l'existence du polynôme $M' = -M^{-1} \bmod (E, \phi)$. Dans les sections 2.4.3 et 2.4.4, nous donnons une condition suffisante pour garantir l'existence de ce polynôme, puis présentons des méthodes de génération du polynôme M qui respectent cette condition. Dans la remarque 2.9 plus loin, nous expliquons que cette condition implique que $\gcd(E, M) = 1$ dans $\mathbb{Q}[X]$. Ainsi, le corollaire 2.3 nous assure que $p \leqslant (w\|M\|_\infty + 1)^n$. Comme $\rho \geqslant 2w\|M\|_\infty$, on a alors $p \leqslant \rho^n$. Cette garantie est indispensable pour les algorithmes de conversions vers l'AMNS que nous présentons ici.

2.3.4.2 Conversion du binaire à l'AMNS

L'idée de l'algorithme 22 est d'utiliser la décomposition en base ρ de l'entier à convertir. Cet algorithme nécessite les représentations $P_i(X)$ de $(\rho^i\phi^2)$ dans $\mathcal{B}$. Nous verrons comment calculer ces représentations. Cet algorithme est identique à l'algorithme 30 dans [Pla05] (section 3.2.3) sauf que la sortie et les polynômes $P_i(X)$ sont différents ici, car les éléments doivent être dans le domaine de Montgomery.

Algorithme 22 *Conversion du binaire à l'AMNS*

Entrée(s) : $a \in \mathbb{Z}/p\mathbb{Z}$ et $\mathcal{B} = (p, n, \gamma, \rho, E)$

Sortie : $A \in \mathcal{B}$, tel que $A \equiv (a\phi)_{\mathcal{B}}$

1: $t = (a_{n-1}, ..., a_0)_\rho$ # la décomposition en base ρ de a

2: $U \leftarrow \sum_{i=0}^{n-1} a_i P_i(X)$

3: $A \leftarrow$ **RedCoeff**(U)

4: retourner A

À la ligne 2 de l'algorithme 22, U est une représentation de $a\phi^2$, avec $\|U\|_\infty < n\rho^2$. On a $w \geqslant n$ et $\delta \geqslant 0$, donc $\|U\|_\infty < w\rho^2(\delta+1)^2$. Ainsi, d'après la proposition 2.3, on a $\|A\|_\infty < \rho$, avec $A \equiv (a\phi)_{\mathcal{B}}$.

À la ligne 1 de l'algorithme 22, une décomposition en base ρ, avec n chiffres, de a est effectuée. Nous avons expliqué que $p \leqslant \rho^n$, cette décomposition est donc unique. Le coût de cette décomposition est très faible si ρ est une puissance de deux. Il est toujours possible de faire ce choix. Par exemple, on peut prendre $\rho = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil}$, car il est seulement nécessaire que $\rho \geqslant 2w\|M\|_\infty$. Ce choix pour ρ ajoute au maximum un bit supplémentaire pour le stockage des coefficients des éléments de l'AMNS. Sachant qu'un élément de l'AMNS est représenté avec n coefficients, cela implique donc au maximum n bits supplémentaires. Puisque n est très petit en pratique (voir les exemples d'AMNS dans l'annexe B.1), ce surcoût mémoire est négligeable.

Cet algorithme est moins coûteux qu'une multiplication modulaire dans AMNS (voir tableau 2.1).

Calcul des représentations P_i. L'algorithme 22 nécessite les représentations $P_i(X)$ de $(\rho^i\phi^2)$ dans $\mathcal{B}$. Pour les calculer, nous avons besoin de la proposition 2.4 qui permet de quantifier la réduction obtenue sur les coefficients d'un polynôme après un appel à **RedCoeff**.

Proposition 2.4. *Soit $V \in \mathbb{Z}_{n-1}[X]$ un polynôme. Si ρ et ϕ sont tels que :*

$$\rho \geqslant 2w\|M\|_\infty \quad et \quad \phi \geqslant 2w\rho(\delta+1)^2,$$

*alors un appel à **RedCoeff** (avec V comme entrée) retourne un polynôme S tel que :*

- $\|S\|_\infty < \rho$ *(i.e. $S \in \mathcal{B}$), si* $\|V\|_\infty < \rho^2$.

- $\|S\|_\infty < \frac{\|V\|_\infty}{\rho}$, *si* $\|V\|_\infty \geqslant \rho^2$.

Démonstration. Soit $S = \text{RedCoeff}(V)$, i.e. S est la sortie de **RedCoeff** avec V comme entrée. On a $\|S\|_\infty \leqslant (\|V\|_\infty + \|T\|_\infty)/\phi$. D'après la proposition 2.1, on a $\|T\|_\infty < w\phi\|M\|_\infty$, car $\|Q\|_\infty < \phi$. Par conséquent, $\|S\|_\infty < (\|V\|_\infty + w\phi\|M\|_\infty)/\phi$. On a $\phi \geqslant 2w\rho(\delta+1)^2 \geqslant 2\rho$ et $\rho \geqslant 2w\|M\|_\infty$. Donc, $\|S\|_\infty < \frac{\|V\|_\infty}{2\rho} + \frac{\rho}{2}$. Ainsi, si $\|V\|_\infty < \rho^2$, alors $\|S\|_\infty < \rho$; i.e. $S \in \mathcal{B}$. Mais, si $\|V\|_\infty \geqslant \rho^2$, alors $\|S\|_\infty < \frac{\|V\|_\infty}{\rho}$; i.e. un appel à **RedCoeff** divise les coefficients de V par au moins ρ, si $\|V\|_\infty \geqslant \rho^2$. □

Soit $\tau = \phi^{n-1} \bmod p$. L'algorithme 23, basé sur la proposition 2.4, décrit une méthode itérative pour calculer une représentation *exacte* dans $\mathcal{B}$ d'un élément $a \in \mathbb{Z}/p\mathbb{Z}$. Cette idée de conversion itérative a déjà été mentionnée dans [BIP04] (section 6.1) et aussi dans [Pla05] (section 3.2.3). À la ligne 2 de

Algorithme 23 *Conversion* ***exacte*** *du binaire à l'AMNS*

Entrée(s) : $a \in \mathbb{Z}/p\mathbb{Z}$, $\mathcal{B} = (p, n, \gamma, \rho, E)$ et $\tau = \phi^{n-1} \bmod p$

Sortie : $A \equiv a_{\mathcal{B}}$, tel que $\|A\|_\infty < 2w\|M\|_\infty$

1: $\alpha = a\,\tau \bmod p$
2: $A = (\alpha, 0, \ldots, 0)$ # un polynôme de degré 0
3: **for** $i = 0$ **to** $n-2$ **do**
4: $\quad A \leftarrow$ **RedCoeff**(A)
5: **end for**
6: retourner A

l'algorithme 23, A est un polynôme de degré 0 dont le coefficient constant est strictement inférieur à p. Puisque $p \leqslant \rho^n$, appeler $n-1$ fois **RedCoeff** sur A assure que l'algorithme retourne $A \in \mathcal{B}$, avec $\|A\|_\infty < 2w\|M\|_\infty$. En effet, dans la section 2.3.4.1, nous avons expliqué que $p \leqslant (w\|M\|_\infty + 1)^n \leqslant (2w\|M\|_\infty)^n$. Donc, d'après la proposition 2.4, après les $n-2$ premières itérations de boucle dans l'algorithme 23, on aura $\|A\|_\infty < (2w\|M\|_\infty)^2$, car $\rho \geqslant 2w\|M\|_\infty$. Pour la dernière itération, on aura alors : $\|A\|_\infty < ((2w\|M\|_\infty)^2 + w\phi\|M\|_\infty)/\phi = (2w\|M\|_\infty)^2/\phi + w\|M\|_\infty$. Donc, $\|A\|_\infty < 2w\|M\|_\infty$, car $\phi \geqslant 2w\rho(\delta+1)^2$ et $\rho \geqslant 2w\|M\|_\infty$.

L'algorithme 23 requiert une multiplication modulaire dans $\mathbb{Z}/p\mathbb{Z}$. Cet algorithme ne convient donc pas pour convertir les éléments de $\mathbb{Z}/p\mathbb{Z}$ dans $\mathcal{B}$. En effet, l'objectif étant d'utiliser l'AMNS pour effectuer les opérations arithmétiques, il serait absurde d'effectuer un opération intermédiaire dans un autre système de représentation. De plus, cet algorithme est beaucoup plus coûteux que l'algorithme 22, car il nécessite $n-1$ appels à **RedCoeff**. Nous ne l'utilisons donc que pour pré-calculer les représentations $P_i(X)$.

Avec l'algorithme 23, les représentations $P_i(X)$ peuvent être calculées de deux manières. La première (la plus simple) est de calculer chaque P_i comme la sortie de l'algorithme 23 avec $\rho^i\phi^2$ comme entrée. La seconde méthode (plus rapide) consiste à d'abord calculer les polynômes $\Phi \equiv (\rho\phi)_{\mathcal{B}}$ et $P_0 \equiv (\phi^2)_{\mathcal{B}}$,

en utilisant l'algorithme 23. Ensuite, pour $1 \leqslant i < n$, P_i est calculé comme le produit de P_{i-1} et Φ, en utilisant l'algorithme 19. Cette seconde approche nécessite $3n-3$ appels à **RedCoeff** plus deux multiplications modulaires dans $\mathbb{Z}/p\mathbb{Z}$, tandis que la première nécessite $(n-1)^2$ appels à **RedCoeff** plus n multiplications modulaires dans $\mathbb{Z}/p\mathbb{Z}$.

2.3.4.3 Conversion de l'AMNS au binaire

Pour convertir un élément de l'AMNS au binaire, nous prenons en compte la conversion dans le domaine de Montgomery qui a été faite pour garantir la consistance des opérations en utilisant **RedCoeff**. Nous présentons ici des modifications de deux algorithmes donnés dans [Pla05] pour la conversion de l'AMNS au binaire.

Remarque 2.4. Soient $A \in \mathbb{Z}_{n-1}[X]$ et $a \in \mathbb{Z}/p\mathbb{Z}$, tels que $A \equiv (a\phi)_{\mathcal{B}}$. Lors du calcul de a à partir de A, on a deux options pour sortir du domaine de Montgomery. On peut d'abord calculer $b = A(\gamma) \bmod p$, puis calculer $a = b\phi^{-1} \bmod p$. On peut également calculer dans un premier temps $B = \mathbf{RedCoeff}(A)$, puis calculer $a = B(\gamma) \bmod p$.
La seconde option est moins coûteuse que la première, car elle nécessite un appel à **RedCoeff** alors que la première requiert une multiplication modulaire dans $\mathbb{Z}/p\mathbb{Z}$, par $\phi^{-1} \bmod p$ (qu'on peut éventuellement pré-calculer). C'est donc la seconde option que nous adoptons ici.

Méthode 1. On peut facilement effectuer cette opération grâce au schéma classique d'Horner [Hor19]. L'algorithme 24 est une légère modification de l'algorithme 31 dans [Pla05] (section 3.2.3). Par rapport à cet algorithme, nous ajoutons la ligne 1 pour sortir du domaine de Montgomery. L'algorithme 24 a donc un coût supplémentaire d'un appel à **RedCoeff**.

Algorithme 24 *Conversion de l'AMNS au binaire*

Entrée(s) : $A \in \mathbb{Z}_{n-1}[X]$ et $\mathcal{B} = (p, n, \gamma, \rho, E)$
Sortie : $a \in \mathbb{Z}/p\mathbb{Z}$, tel que $a = A(\gamma)\phi^{-1} \bmod p$

1: $A \leftarrow \mathbf{RedCoeff}(A)$
2: $a \leftarrow a_{n-1}$
3: **for** $i = n-2$ **to** 0 **do**
4: $\quad a \leftarrow (a\gamma + a_i) \bmod p$
5: **end for**
6: retourner a

Méthode 2. L'algorithme 24 est simple et ne nécessite aucun pré-calcul. Cependant, il nécessite $n-1$ multiplications modulaires et un appel à **RedCoeff**. L'algorithme 25 améliore l'algorithme 24 au prix de quelques pré-calculs. Les valeurs $g_i = \gamma^i \bmod p$, pour $i = 1, \ldots, (n-1)$, doivent être pré-calculées avant de l'utiliser. Cet algorithme est une légère modification de l'algorithme 32 dans

[Pla05] (section 3.2.3). Par rapport à cet algorithme, nous ajoutons la ligne 1 pour sortir du domaine de Montgomery. L'algorithme 25 a donc un coût supplémentaire d'un appel à **RedCoeff**.

Algorithme 25 *Conversion de l'AMNS au binaire*

Entrée(s) : $A \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p, n, \gamma, \rho, E)$ et $g_i = \gamma^i \bmod p$, pour $i = 1, \ldots, (n-1)$

Sortie : $a \in \mathbb{Z}/p\mathbb{Z}$, tel que $a = A(\gamma)\phi^{-1} \bmod p$

1: $A \leftarrow$ **RedCoeff**(A)
2: $a \leftarrow a_0$
3: **for** $i = 1$ **to** $n-1$ **do**
4: $\quad a \leftarrow a + a_i g_i$
5: **end for**
6: $a \leftarrow a \bmod p$
7: retourner a

Dans cet algorithme, $n-1$ multiplications et une réduction modulaire sont effectuées (plus un appel à **RedCoeff**). Si $A \in \mathcal{B}$, alors $|a_i| < \rho$. Ainsi, les multiplications $a_i g_i$ ne sont pas des multiplications complètes dans $\mathbb{Z}/p\mathbb{Z}$. Aussi, à la fin de la boucle (avant la ligne 6), nous avons $|a| < (n\rho)p$. Comme $\rho \approx p^{1/n}$, la réduction modulaire à la ligne 6 est moins chère qu'une réduction modulaire complète dans $\mathbb{Z}/p\mathbb{Z}$.

2.3.5 La réduction exacte de coefficients

Au début de la section 2.3.4, nous avons expliqué la nécessité de convertir et maintenir les éléments de l'AMNS dans le domaine de Montgomery pour assurer la consistance de la multiplication modulaire. Ainsi, un entier $a \in \mathbb{Z}/p\mathbb{Z}$ est représenté dans $\mathcal{B}$ par un polynôme A tel que $A(\gamma) \equiv a\phi \pmod p$.

Une fois dans le domaine Montgomery, un autre problème peut survenir lorsqu'il faut calculer $\alpha_1 + \alpha_2 + \cdots + \alpha_k$, avec $\alpha_i \in \mathbb{Z}/p\mathbb{Z}$, utilisant leurs représentations dans $\mathcal{B}$. Le résultat correspondant est un polynôme R tel que $R(\gamma) \equiv \phi \sum_{i=1}^{k} \alpha_i \pmod p$. Si $k \leqslant (\delta + 1)$, alors, d'après le corollaire 2.5, il n'y a pas besoin d'une réduction interne, avant une multiplication avec R. Cependant, si $k > (\delta + 1)$ et que l'on a besoin ou l'on veut ramener le résultat R dans l'AMNS (pour des besoins de stockage, par exemple), alors une réduction interne doit être faite. **RedCoeff** introduit un facteur ϕ^{-1}. Donc, l'appliquer sur R retournera un polynôme S tel que $S(\gamma) \equiv R(\gamma)\phi^{-1} \equiv \sum_{i=1}^{k} \alpha_i \pmod p$. Par conséquent, S n'est plus dans le domaine de Montgomery.

Pour résoudre ce problème, nous proposons l'algorithme 26, qui prend comme entrée un polynôme V et retourne un polynôme S tel que $S(\gamma) \equiv V(\gamma) \pmod p$. Ainsi, si V est dans le domaine Montgomery, alors S le sera.

À la ligne 1 de l'algorithme 26, nous avons $T(\gamma) \equiv V(\gamma)\phi^{-1} \pmod p$. Comme $P_0(\gamma) \equiv \phi^2 \pmod p$, on a $U(\gamma) \equiv V(\gamma)\phi \pmod p$. Ainsi, $S(\gamma) \equiv V(\gamma) \pmod p$.

Algorithme 26 **ExactRedCoeff** - *Réduction* ***exacte*** *de coefficients*

Entrée(s) : $V \in \mathbb{Z}_{n-1}[X]$, $P_0 \equiv (\phi^2)_{\mathcal{B}}$ and $\mathcal{B} = (p, n, \gamma, \rho, E)$
Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv V(\gamma) \pmod{p}$

1: $T \leftarrow$ **RedCoeff**(V)
2: $U \leftarrow T \times P_0 \bmod E$
3: $S \leftarrow$ **RedCoeff**(U)
4: retourner S

ExactRedCoeff (algorithme 26) est plus coûteux que la multiplication modulaire (algorithme 19). Il a un appel supplémentaire à **RedCoeff**. Aussi, il utilise P_0, une représentation exacte de ϕ^2 dans l'AMNS, i.e. $P_0(\gamma) \equiv \phi^2 \pmod{p}$. Dans la section 2.3.4 (algorithme 23), nous avons expliqué comment calculer P_0, car il est également nécessaire pour la conversion (rapide) dans l'AMNS. La proposition 2.5 donne les conditions sur ρ, ϕ et l'entrée V pour que la sortie S de l'algorithme 26 soit dans $\mathcal{B}$.

Proposition 2.5. *Soit $V \in \mathbb{Z}_{n-1}[X]$ un polynôme. Si ρ, ϕ et V sont tels que :*

$$\rho \geqslant 2w\|M\|_\infty, \quad \phi \geqslant 2w\rho(\delta+1)^2 \quad \text{et} \quad \|V\|_\infty \leqslant w\rho^2(\delta+1)^2,$$

alors la sortie S de l'algorithme 26 (avec V comme entrée) est telle que $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$).

Démonstration. D'après la proposition 2.3, on a $\|T\|_\infty < \rho$ (ligne 1 de l'algorithme 26). Ensuite, les lignes 2 et 3 combinées sont équivalentes à une multiplication modulaire, avec les entrées dans $\mathcal{B}$. Le corollaire 2.5 permet donc de conclure. □

Soit $\mu = w\rho(\delta+1)^2 - 1$. De la proposition 2.5, on déduit que si un polynôme V est le résultat de la somme de jusqu'à μ éléments de l'AMNS, alors un appel à **ExactRedCoeff** suffit pour ramener V dans $\mathcal{B}$, avec le résultat dans le domaine Montgomery. Notons qu'en pratique μ sera toujours très grand, grâce à la présence de ρ dans son expression.

En résumé, **ExactRedCoeff** est censé être utilisé lorsque $\nu \in]\delta, \mu]$ additions consécutives d'éléments de $\mathcal{B}$ ont été faites et que le résultat doit être ramené dans l'AMNS.

2.3.6 L'exponentiation modulaire

Soit $a, k \in \mathbb{Z}/p\mathbb{Z}$ et $A \in \mathbb{Z}_{n-1}[X]$ tel que $A \equiv (a\phi)_{\mathcal{B}}$, i.e. A est une représentation de a dans le domaine de Montgomery. L'exponentiation modulaire consiste à calculer $R \in \mathbb{Z}_{n-1}[X]$ tel que $R(\gamma) \equiv (a^k\phi)_{\mathcal{B}}$. Cette opération est similaire à celle du binaire classique. Toutes les méthodes d'exponentiation classique n'exploitant que les représentations (binaire, NAF, etc.) de l'exposant k peuvent très simplement être utilisées pour l'exponentiation modulaire dans l'AMNS.

Dans cette section, nous présentons la méthode *square-and-multiply* ainsi que l'*échelle de Montgomery* pour l'exponentiation modulaire dans l'AMNS.

Nous renvoyons le lecteur vers la section 1.1.3 où nous discutons du coût et de la sûreté de ces méthodes.

Soit $l = \lceil \log_2(p) \rceil$. Les éléments de $\mathbb{Z}/p\mathbb{Z}$ se codent donc sur l bits. Notons $\otimes$ la multiplication modulaire dans l'AMNS, telle que décrite dans l'algorithme 19. Soit $\Theta \equiv (\phi)_{\mathcal{B}}$, une représentation de l'entier 1 dans le domaine de Montgomery. Remarquons que le polynôme Θ peut être (pré-)calculé très simplement à partir du polynôme de conversion P_0, avec le calcul : $\Theta = \textbf{RedCoeff}(P_0)$, car $P_0 \equiv (\phi^2)_{\mathcal{B}}$. Les algorithmes 27 et 28 présentent le calcul de l'exponentiation modulaire en utilisant respectivement la méthode *square-and-multiply* et l'*échelle de Montgomery*.

Algorithme 27 *Left-to-right square-and-multiply dans l'AMNS*

Entrée(s) : $A \in \mathbb{Z}_{n-1}[X]$ tel que $A \equiv (a\phi)_{\mathcal{B}}$, $\mathcal{B} = (p, n, \gamma, \rho, E)$ et $k = (k_{l-1}, \ldots, k_0)_2$

Sortie : $R \in \mathbb{Z}_{n-1}[X]$, tel que $R(\gamma) \equiv (a^k\phi) \pmod{p}$

1: $R \leftarrow \Theta$
2: **for** $i = l - 1$ **to** 0 **do**
3: $\quad R \leftarrow R \otimes R$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad R \leftarrow R \otimes A$
6: $\quad$ **end if**
7: **end for**
8: retourner R

Algorithme 28 *Échelle de Montgomery dans l'AMNS*

Entrée(s) : $A \in \mathbb{Z}_{n-1}[X]$ tel que $A \equiv (a\phi)_{\mathcal{B}}$, $\mathcal{B} = (p, n, \gamma, \rho, E)$ et $k = (k_{l-1}, \ldots, k_0)_2$

Sortie : $R_0 \in \mathbb{Z}_{n-1}[X]$, tel que $R_0(\gamma) \equiv (a^k\phi) \pmod{p}$

1: $R_0 \leftarrow \Theta$
2: $R_1 \leftarrow A$
3: **for** $i = l - 1$ **to** 0 **do**
4: $\quad b \leftarrow k_i$
5: $\quad R_{1-b} \leftarrow R_{1-b} \otimes R_b$
6: $\quad R_b \leftarrow R_b \otimes R_b$
7: **end for**
8: retourner R_0

Dans les algorithmes 27 et 28, on peut remarquer qu'à chaque itération les polynômes R et R_i sont tels que $R(\gamma) \equiv (a^t\phi) \pmod{p}$ et $R_i(\gamma) \equiv (a^u\phi) \pmod{p}$, avec $t, u \leqslant k$; ceci grâce au fait que l'entrée A est dans le domaine de Montgomery. Notons que l'opération de la ligne 3 du *Square-and-Multiply* peut

être accélérée. C'est une élévation au carré, on peut donc presque diviser par deux le nombre de produits partiels dans l'étape de multiplication. La même chose est faisable pour la ligne 6 de l'*échelle de Montgomery*. Cependant, pour une exponentiation plus sûre avec cette méthode, il est préférable d'utiliser l'algorithme 19 de multiplication pour les carrés modulaires également.

Remarque 2.5. Si ρ, ϕ et l'entrée A des algorithmes 27 et 28 sont tels que :

$$\rho \geqslant 2w\|M\|_\infty, \quad \phi \geqslant 2w\rho(\delta+1)^2 \quad \text{et} \quad \|A\|_\infty < (\delta+1)\rho,$$

alors d'après le corollaire 2.5, les sorties R et R_0 de ces algorithmes sont dans l'AMNS, i.e. $\|R\|_\infty < \rho$ et $\|R_0\|_\infty < \rho$.

2.3.7 L'inversion modulaire

Soient $a \in \mathbb{Z}/p\mathbb{Z}$ tel que $\text{pgcd}(a, p) = 1$ et $r \in \mathbb{Z}/p\mathbb{Z}$ l'inverse de a modulo p, donc $ar \equiv 1 \pmod p$. Soit $A \in \mathbb{Z}_{n-1}[X]$ tel que $A \equiv (a\phi)_\mathcal{B}$, i.e. A est une représentation de a dans le domaine de Montgomery. L'inversion modulaire dans l'AMNS consiste à calculer $R \in \mathbb{Z}_{n-1}[X]$ tel que $R(\gamma) \equiv (r\phi)_\mathcal{B}$.

On a $a^{\varphi(p)} \equiv 1 \pmod p$, où φ est l'*indicatrice d'Euler*, cf. définition 1.3. Nous rappelons que $\varphi(p) = p-1$ si p est premier. Puisque $a^{\varphi(p)} = a.a^{\varphi(p)-1} \equiv 1 \pmod p$, on a : $r = a^{\varphi(p)-1} \bmod p$. L'inversion modulaire dans l'AMNS peut donc être faite en effectuant l'exponentiation modulaire $R = A^{\varphi(p)-1}$, comme expliqué dans la section 2.3.6.

Notons que, si possible, on peut également reporter l'inversion modulaire à la fin des calculs et sortir de l'AMNS avant d'effectuer cette inversion, en utilisant l'algorithme d'Euclide étendu (algorithme 12) par exemple.

Jusque-là, nous avons présenté l'AMNS et donné ses propriétés essentielles. Dans cette section qui s'achève, nous avons donné un ensemble d'algorithmes pour les opérations de conversions ainsi que les principales opérations arithmétiques dans l'AMNS. Cela nous a permis de présenter tous les paramètres nécessaires à ces algorithmes et de donner également les contraintes que ces paramètres doivent respecter. Dans la section qui suit, nous montrons comment générer ces paramètres.

2.4 Processus de génération pour une arithmétique modulaire efficace

Nous commençons par lister l'ensemble des paramètres utilisés dans les algorithmes présentés dans la section 2.3 :

- p : un entier premier, $p \geqslant 3$.
- n : le nombre de coefficients des éléments dans l'AMNS, $n \geqslant 2$.
- λ : un petit entier relatif non nul. Posons, $w = 1 + (n-1)|\lambda|$.
- γ : une racine n-ième (modulo p) de λ.

- E : le polynôme de réduction externe, défini comme : $E(X) = X^n - \lambda$.
- M : le polynôme de réduction interne.
- ρ : la borne supérieure de la norme infinie des éléments de l'AMNS, $\rho \geqslant 2w\|M\|_\infty$.
- δ : le nombre maximum d'additions consécutives qui seront effectuées avant une multiplication modulaire, comme expliqué dans la section 2.3.3.
- ϕ : l'entier utilisé dans **RedCoeff**, $\phi \geqslant 2w\rho(\delta+1)^2$.
- M' : un polynôme, tel que $M' = -M^{-1} \bmod (E, \phi)$.

Certains de ces paramètres sont choisis tandis que les autres sont calculés. Dans la section qui suit, nous présentons le processus de génération des paramètres.

2.4.1 Processus de génération de paramètres

Le processus de génération dépend à la fois de l'application cible et de l'architecture cible. Supposons une architecture de k bits. Soit p le nombre premier pour lequel on souhaite construire un AMNS. Le processus de génération est le suivant.

1. Choisir δ en fonction de l'application cible.
2. L'étape suivante est de choisir le nombre de mots machine qu'on souhaite allouer pour chaque coefficient des éléments de l'AMNS. Soit $t \geqslant 1$ ce nombre. Posons $k' = tk$, le nombre de bits qui sera utilisé pour stocker chaque coefficient.
3. Choisir n, tel que $nk' \geqslant \lceil \log_2(p) \rceil$ afin de s'assurer que chaque coefficient des éléments de l'AMNS puisse tenir dans une variable de k' bits.
4. Après le choix de n, on choisit un petit entier non nul $\lambda \in \mathbb{Z}$ tel qu'une racine n-ième γ modulo p de λ existe. Dans la section 2.4.2, nous montrons qu'il est toujours possible de trouver de tels λ et γ, à partir de p et n. Le polynôme de réduction externe E est alors : $E(X) = X^n - \lambda$.
5. Le polynôme de réduction interne M doit être généré en garantissant l'existence du polynôme $M' = -M^{-1} \bmod (E, \phi)$, pour tout $\phi \geqslant 2$ une puissance de deux. Dans la section 2.4.3, nous donnons des conditions suffisantes sur M qui garantissent l'existence du polynôme M'. Dans la section 2.4.4, nous expliquons comment générer un polynôme M qui remplit ces conditions.
6. Posons, $w = 1 + (n-1)|\lambda|$. Après le calcul de M, il nous faut calculer ρ et ϕ. Comme expliqué dans la section 2.3.4.2 portant sur la conversion du binaire vers l'AMNS, choisir $\rho = 2^{\lceil \log_2(2w\|M\|_\infty) \rceil}$ assure une conversion efficace. De plus, nous avons expliqué que ce choix implique un surcoût de n bits au maximum pour représenter les éléments de l'AMNS, avec n qui est très petit en pratique (voir les exemples d'AMNS dans l'annexe B.1). On calcule donc ρ et ϕ comme suit :

$$\rho = 2^{\lceil \log_2(2w\|M\|_\infty) \rceil} \quad \text{et} \quad \phi = 2^{\lceil \log_2(2w\rho(\delta+1)^2) \rceil},$$

pour assurer la consistance des algorithmes présentés dans la section 2.3. **Note :** À ce niveau, il faut s'assurer que $\rho \leqslant 2^{k'-1}$, puisqu'on souhaite allouer k' bits pour chaque coefficient des éléments de l'AMNS. Si $\rho > 2^{k'-1}$, alors $\|M\|_\infty$ est trop grande. Dans ce cas, il faut choisir un autre candidat M de norme infinie satisfaisante, i.e. remonter à l'étape 5. S'il n'y a aucun candidat M de norme infinie satisfaisante, on peut changer la valeur de λ, i.e. remonter à l'étape 4. Si cette seconde option ne marche pas, alors augmenter la valeur de n (i.e. remonter à l'étape 3) ou celle de t (i.e. remonter à l'étape 2).

7. On calcule ensuite le polynôme M' : $M' = -M^{-1} \bmod (E, \phi)$.
8. Pour la conversion (rapide) du binaire vers l'AMNS (l'algorithme 22), on doit pré-calculer les représentations $P_i(X)$ de $(\rho^i \phi^2)$ dans l'AMNS (comme expliqué dans la section 2.3.4). Nous rappelons que P_0 est également requis pour **ExactRedCoeff** (l'algorithme 26). Si une conversion rapide de l'AMNS vers le binaire est souhaitée, les éléments $g_i = \gamma^i \bmod p$, pour $i = 1, \ldots, (n-1)$, doivent être pré-calculés afin d'utiliser l'algorithme 25.

Remarque 2.6 (**Implémentation logicielle**). Pour une implémentation logicielle, une autre stratégie concernant le choix de ϕ pourrait conduire à de meilleures performances. En supposant qu'on a une architecture de k bits (avec k suffisamment grand), un choix judicieux est de prendre $\phi = 2^k$. Ainsi, la division exacte par ϕ et la réduction modulo ϕ peuvent être réalisées avec de très simples opérations de masquage et de décalage. Ce choix rend donc les réductions modulo ϕ (ligne 1, algorithme 18) ainsi que les divisions exactes par ϕ (ligne 3, algorithme 18) très simples et rapides. De plus, puisque $\rho < \phi$, ce choix implique que $k' = k$. En d'autres termes, il garantit que chaque coefficient d'un élément dans l'AMNS tiendra dans un mot de machine. Ce qui est également bon pour les performances.

Le tableau 2.1 (section 2.5.1) montre que la valeur de n a un impact important sur l'efficacité des opérations arithmétiques dans l'AMNS. Choisir $\phi = 2^k$ implique que $n \geqslant \lfloor \frac{\log_2 p}{k} \rfloor + 1$, car $\phi > \rho$. Par conséquent, la valeur optimale pour n est $\lfloor \frac{\log_2 p}{k} \rfloor + 1$. Le paramètre ρ est calculé comme suit : $\rho = 2^{\lceil \log_2(2w\|M\|_\infty) \rceil}$. Donc, pour garantir qu'on puisse prendre $\phi = 2^k$ avec $\phi \geqslant 2w\rho(\delta+1)^2$, il est nécessaire de calculer le polynôme M avec $\|M\|_\infty$ assez petite pour permettre cela. Dans la section 2.4.4, nous expliquons comment le faire, grâce à la réduction de réseau.

Remarque 2.7 (**Garantie de génération d'un AMNS**). Comme nous l'expliquons plus loin dans la remarque 2.9, le processus de génération du polynôme M assure que $\text{pgcd}(E, M) = 1$ dans $\mathbb{Q}[X]$. On choisit ρ tel que $\rho \geqslant 2w\|M\|_\infty$. Donc, $\rho > \frac{1}{2}w\|M\|_\infty$, car $w \geqslant 1$ et $\|M\|_\infty \neq 0$. Ainsi, d'après le corollaire 2.2, le tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ est bien un AMNS.

2.4.2 Existence de γ

Soit $p \geqslant 3$ un nombre premier et $n \geqslant 2$ un entier. La première contrainte dans la génération d'un AMNS est de trouver un petit entier non nul $\lambda \in \mathbb{Z}$ tel

qu'une racine n-ième γ modulo p de λ existe. Dans cette section, nous donnons quelques résultats qui montrent qu'il est toujours possible de trouver un tel λ.

Proposition 2.6. *Soit $E(X) = X^n - \lambda$, avec $\lambda \in \mathbb{Z} \setminus \{0\}$. Soit g un générateur de $(\mathbb{Z}/p\mathbb{Z})^*$ et y tels que $g^y \equiv \lambda \pmod{p}$. Si $\gcd(n, p-1) \mid y$, alors il existe $\gcd(n, p-1)$ racine(s) γ de $E(X)$ dans $\mathbb{Z}/p\mathbb{Z}$.*

Démonstration. Si $\gcd(n, p-1) \mid y$, alors l'équation $nx \equiv y \bmod (p-1)$ admet k solutions, où $k = \gcd(n, p-1)$. Soit x_0 une de ces solutions, posons $\gamma \equiv g^{x_0} \pmod{p}$. Alors, $\gamma^n \equiv \lambda \pmod{p}$. □

La proposition 2.6 donne une condition sur l'existence des racines n-ième modulo p de λ et leur nombre. Le calcul effectif de l'une de ces racines en utilisant la preuve de cette proposition nécessite de pouvoir calculer le logarithme discret de λ dans $\mathbb{Z}/p\mathbb{Z}$. Cependant, pour p assez grand, ce calcul est de manière générale très difficile.

Ci-dessous, nous donnons des conditions suffisantes (mais pas nécessaires) qui sont faciles à tester et qui garantissent l'existence d'une racine n-ième modulo p de λ, en prenant éventuellement $\lambda = 1$.

Corollaire 2.6. *Si $\gcd(n, p-1) = 1$, alors il existe une unique racine n-ième γ de λ dans $\mathbb{Z}/p\mathbb{Z}$, pour tout $\lambda \in \mathbb{Z}/p\mathbb{Z} \setminus \{0\}$.*

Démonstration. Si $\gcd(n, p-1) = 1$, cette racine n-ième peut être facilement calculée. En effet, en utilisant l'algorithme d'Euclide étendu, on calcule les coefficients de Bézout pour le couple $(n, p-1)$; i.e. $u, v \in \mathbb{Z}$ tels que $nu + (p-1)v = 1$. Donc, $\lambda = \lambda^{nu+(p-1)v} = (\lambda^u)^n (\lambda^{p-1})^v$. Puisque $\lambda^{p-1} \equiv 1 \pmod{p}$, il est évident que $\lambda \equiv (\lambda^u)^n \pmod{p}$. Ce qui signifie que $\lambda^u \bmod p$ est une racine n-ième (modulo p) de λ. □

Si $\gcd(n, p-1) = 1$ et $\lambda = 1$, alors d'après le corollaire précédent, l'unique racine n-ième γ de λ est 1. Avec $\gamma = 1$, on ne peut pas construire d'AMNS. En effet, dans ce cas, la plus grande valeur qui peut être représentée dans cet AMNS est inférieure à $n\rho$, puisque les éléments de l'AMNS sont des polynômes de degré $n-1$ évalués en $\gamma = 1$ et leurs coefficients sont inférieurs à ρ. Il n'est donc pas possible de générer tous les éléments dans $\mathbb{Z}/p\mathbb{Z}$ avec un tel système, car $\rho \approx p^{1/n}$. Le corollaire suivant traite du cas $\lambda = 1$ et $\gcd(n, p-1) > 1$.

Corollaire 2.7. *Si $\gcd(n, p-1) > 1$, alors il existe au moins une racine n-ième non triviale γ de 1. Ainsi, on peut prendre $\lambda = 1$.*

Démonstration. Soit $\lambda = 1$, on cherche $\gamma \neq 1$ tel que $\gamma^n \equiv 1 \pmod{p}$. Soit g un générateur de $(\mathbb{Z}/p\mathbb{Z}) \setminus \{0\}$ et $d = \gcd(n, p-1)$, avec $d > 1$. D'après la proposition 2.6, il y a d racines n-ième de l'unité modulo p, car $1 = g^0$. Soit $h = g^{(p-1)/d} \bmod p$. On a $h^n \equiv 1 \pmod{p}$, avec $h \neq 1$. Donc, h est une racine n-ième non triviale de λ. Les autres racines n-ième sont $h^i \bmod p$, pour $2 \leqslant i \leqslant d$. □

Du corollaire 2.6, on déduit que si $\text{pgcd}(n, p-1) = 1$, alors λ peut être n'importe quel entier aussi petit que l'on le souhaite, hormis 0 et 1. En outre, ce corollaire montre que l'unique racine n-ième de λ est facile à calculer. Lorsque $\text{pgcd}(n, p-1) > 1$, d'après le corollaire 2.7, il suffit de prendre $\lambda = 1$. Ce corollaire montre également comment calculer facilement une racine n-ième non triviale de 1.

Notons que si $\text{pgcd}(n, p-1) > 1$, le corollaire 2.7 n'implique pas qu'il faut nécessairement prendre $\lambda = 1$. Si $\lambda \neq 1$ a des racines n-ième modulo p, alors il existe un algorithme proposé dans [Joh99] qui calcule ces racines n-ième. Cet algorithme ramène le calcul des racines n-ième à la résolution d'un logarithme discret dans un "petit" corps fini. Son efficacité dépend donc de la taille de ce "petit" corps fini. Cet algorithme est utilisé dans la bibliothèque SageMath [Sa18] pour le calcul de racine n-ième. Une autre possibilité est de calculer les facteurs irréductibles de $E(X) = X^n - \lambda$ dans $\mathbb{Z}/p\mathbb{Z}$ et de voir si au moins l'un d'eux est de degré un. On obtient alors, s'il y en a, les racines n-ième modulo p de λ. Un algorithme probabiliste qui fonctionne en temps polynômial pour cette factorisation est donné dans [GH98].

Remarque 2.8. Il est intéressant de noter qu'à l'exception des algorithmes de conversion de l'AMNS vers le binaire, aucun des algorithmes présentés dans la section 2.3 n'utilise le paramètre γ. Ce paramètre n'a donc aucun impact sur l'efficacité des opérations arithmétiques. Il n'a également pas d'influence sur l'efficacité des opérations de conversion du binaire vers l'AMNS.

2.4.3 Existence du polynôme M'

Dans [NP08], les auteurs indiquent que M doit être choisi tel que $\text{pgcd}(E, M) = 1$, mais cela ne garantit pas l'existence du polynôme M' utilisé dans **RedCoeff**. En effet, si $\text{gcd}(E, M) = 1$, alors il existe $J \in \mathbb{Q}[X]$ tel que $MJ \equiv 1 \pmod{E}$, mais rien ne garantit que les coefficients de J sont inversibles modulo ϕ. Par conséquent, la condition $\text{pgcd}(E, M) = 1$ n'est pas suffisante pour garantir l'existence de M'.

Une première tentative pour la génération du polynôme M tout en garantissant l'existence du polynôme M' a été proposée par El Mrabet et Gama dans [EG12] pour le cas spécial $E(X) = X^n + 1$. Dans la section 3.3 de cet article, lorsque la valeur de p est fixée et que ϕ est une puissance de 2, les auteurs montrent comment construire un réseau dont toute base réduite contient au moins un polynôme M inversible modulo (E, ϕ). Dans la preuve du lemme 4 de cet article, les auteurs affirment qu'un polynôme M est inversible modulo (E, ϕ) si l'évaluation de M en tout entier est impaire. Cette condition n'est pas suffisante. Par exemple, si $E(X) = X^6 + 1$ et $M(X) = X^4 - X^2 + 1$, l'évaluation de M en tout entier est impaire ; cependant, $\text{pgcd}(E, M) = M$. Donc, il n'y a aucune garantie que le polynôme M' existe, pour un ϕ quelconque.

Dans cette section, nous donnons des conditions suffisantes sur les polynômes M et E qui garantissent l'existence du polynôme M', avec $\phi \geqslant 2$ une puissance de deux quelconque.

Dans ce qui suit, les éléments présentés requièrent la connaissance de cer-

taines notions sur le résultant de polynômes. Nous rappelons dans l'annexe A.1 ce qui doit être connu du lecteur. La proposition A.1 de cette annexe énonce un résultat fondamental sur le résultant. De ce résultat, nous déduisons le critère suivant d'existence de M' pour $\phi \geqslant 2$ quelconque.

Proposition 2.7 (Critère d'existence de M'). *Soit $M, E \in \mathbb{Z}[X]$ et $\phi \geqslant 2$ un entier. Si $\text{pgcd}(\text{Res}(E, M), \phi) = 1$, alors il existe $J \in \mathbb{Z}_{n-1}[X]$ tel que $JM \equiv 1 \pmod{E, \phi}$. En d'autres termes, $J = M^{-1} \bmod (E, \phi)$ existe.*

Démonstration. Soit $r = \text{Res}(E,M)$ et ψ l'inverse de r modulo ϕ. D'après la proposition A.1, il existe U et V dans $\mathbb{Z}[X]$ tels que $U(X)M(X)+V(X)E(X) = r$. Donc, $\psi U(X)M(X) + \psi V(X)E(X) = \psi r$. Par conséquent, $\psi U(X)M(X) \equiv 1 \bmod (E, \phi)$. □

Notons que la proposition 2.7 établit une condition suffisante d'existence du polynôme $J = M^{-1} \bmod (E, \phi)$. Puisque $M' = -M^{-1} \bmod (E, \phi)$, M' existe si et seulement si J existe.

Puisque **RedCoeff** n'est intéressant que si ϕ est une puissance de deux, on déduit le corollaire suivant de la proposition précédente.

Corollaire 2.8. *Soit $M \in \mathbb{Z}[X]$, $E \in \mathbb{Z}[X]$ et $\phi = 2^j$, avec $j \geqslant 1$ un entier. Si $\text{Res}(E, M)$ est impair, alors il existe $J \in \mathbb{Z}[X]$ tel que $JM \equiv 1$ mod (E, ϕ).*

Démonstration. Puisque ϕ est une puissance de deux, $\text{pgcd}(\text{Res}(E, M), \phi) = 1$ est équivalent à $\text{Res}(E, M)$ est impair. On peut donc conclure avec la proposition 2.7. □

Le paramètre E de l'AMNS est tel que $E(X) = X^n - \lambda$. On a aussi $M \in \mathbb{Z}_{n-1}[X]$; posons $M(X) = m_0 + m_1X + \cdots + m_{n-1}X^{n-1}$. La matrice de Sylvester de E et M est la matrice $(2n-1) \times (2n-1)$ définie comme suit :

$$\mathcal{S}_{E,M} = \begin{pmatrix} 1 & 0 & \dots & 0 & m_{n-1} & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & m_{n-2} & m_{n-1} & \dots & 0 & 0 \\ \vdots & & \ddots & & \vdots & & & & \vdots \\ 0 & 0 & \dots & 1 & m_1 & m_2 & \dots & m_{n-1} & 0 \\ 0 & 0 & \dots & 0 & m_0 & m_1 & \dots & m_{n-2} & m_{n-1} \\ -\lambda & 0 & \dots & 0 & 0 & m_0 & \dots & m_{n-3} & m_{n-2} \\ \vdots & & & & \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & m_0 & m_1 \\ 0 & 0 & \dots & -\lambda & 0 & 0 & \dots & 0 & m_0 \end{pmatrix}$$

Le corollaire 2.8 indique que pour utiliser **RedCoeff**, en prenant ϕ comme une puissance de deux, il suffit que le résultant de E et M soit impair. On a $\text{Res}(E, M) = \det(\mathcal{S}_{E,M})$, cf. définition A.1 (annexe A.1). Donc, $\text{Res}(E, M)$ impair est équivalent à $\det(\mathcal{S}_{E,M})$ impair. Dans la suite, notre objectif sera donc d'établir des conditions suffisantes sur E et M pour que $\det(\mathcal{S}_{E,M})$ soit impair. Pour établir ces conditions, nous avons besoin de la propriété suivante.

Propriété 2.1 (**Parité du déterminant de matrice**). Soit $A, B \in \mathcal{M}_{n\times n}(\mathbb{Z})$ deux matrices, telles que : $A = (a_{ij})_{0\leqslant i,j<n}$ et $B = (b_{ij})_{0\leqslant i,j<n}$, avec $b_{ij} = a_{ij} \bmod 2$. Les déterminants de A et B ont la même parité.

Démonstration. Cette propriété est évidente puisque le déterminant d'une matrice est une combinaison d'additions et de produits de ses éléments. Ainsi, remplacer un élément de cette matrice par un entier quelconque de même parité ne modifie pas la parité du déterminant de la matrice initiale. □

Afin d'établir les conditions sur E et M qui garantissent que $\det(\mathcal{S}_{E,M})$ est impair, nous distinguons deux cas selon la parité du paramètre λ de l'AMNS. Pour la suite, nous supposons que $\phi \geqslant 2$.

2.4.3.1 Existence du polynôme M' lorsque λ est pair.

Dans ce premier cas, la condition pour que $\det(\mathcal{S}_{E,M})$ soit impair est assez simple et triviale. Elle est donnée par la proposition 2.8

Proposition 2.8. *Soit $E(X) = X^n - \lambda$ un polynôme tel que λ est pair. Soit $M(X) = m_0 + m_1X + \cdots + m_{n-1}X^{n-1}$ un polynôme. On a l'équivalence suivante :*

$$\det(\mathcal{S}_{E,M}) \text{ est impair si et seulement si } m_0 \text{ est impair.}$$

Démonstration. On a λ pair. Donc, d'après la propriété 2.1, il est clair que le déterminant de $\mathcal{S}_{E,M}$ a la même parité que la matrice suivante (où $\overline{m_i} = m_i \bmod 2$ et λ est remplacé par 0) :

$$\begin{pmatrix}
1 & 0 & \dots & 0 & \overline{m_{n-1}} & 0 & \dots & 0 & 0 \\
0 & 1 & \dots & 0 & \overline{m_{n-2}} & \overline{m_{n-1}} & \dots & 0 & 0 \\
\vdots & & \ddots & & \vdots & & & & \vdots \\
0 & 0 & \dots & 1 & \overline{m_1} & \overline{m_2} & \dots & \overline{m_{n-1}} & 0 \\
0 & 0 & \dots & 0 & \overline{m_0} & \overline{m_1} & \dots & \overline{m_{n-2}} & \overline{m_{n-1}} \\
0 & 0 & \dots & 0 & 0 & \overline{m_0} & \dots & \overline{m_{n-3}} & \overline{m_{n-2}} \\
\vdots & & & & \vdots & & \ddots & & \vdots \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & \overline{m_0} & \overline{m_1} \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & \overline{m_0}
\end{pmatrix}$$

C'est une matrice triangulaire supérieure avec seulement la valeur 1 ou $\overline{m_0}$ sur la diagonale. Donc, son déterminant est non nul (et égal à 1) si et seulement si m_0 est impair. Par conséquent, $\det(\mathcal{S}_{E,M})$ est impair si et seulement si m_0 est impair. □

De la proposition 2.8, on déduit que, lorsque ϕ est une puissance de deux et λ est pair, le polynôme M' existe si m_0 est impair.

2.4.3.2 Existence du polynôme M' lorsque λ est impair.

Ce cas un peu plus complexe nécessite des résultats supplémentaires. Soit $\mathcal{H}_1$ la matrice obtenue à partir de $\mathcal{S}_{E,M}$ en remplaçant m_i par $\overline{m_i} = m_i \bmod 2$. Nous remplaçons également λ par -1.

$$\mathcal{H}_1 = \begin{pmatrix} 1 & 0 & \dots & 0 & \overline{m_{n-1}} & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & \overline{m_{n-2}} & \overline{m_{n-1}} & \dots & 0 & 0 \\ \vdots & & \ddots & & \vdots & & & & \vdots \\ 0 & 0 & \dots & 1 & \overline{m_1} & \overline{m_2} & \dots & \overline{m_{n-1}} & 0 \\ 0 & 0 & \dots & 0 & \overline{m_0} & \overline{m_1} & \dots & \overline{m_{n-2}} & \overline{m_{n-1}} \\ -1 & 0 & \dots & 0 & 0 & \overline{m_0} & \dots & \overline{m_{n-3}} & \overline{m_{n-2}} \\ \vdots & & & & \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & \overline{m_0} & \overline{m_1} \\ 0 & 0 & \dots & -1 & 0 & 0 & \dots & 0 & \overline{m_0} \end{pmatrix}$$

Puisque λ est impair, la propriété 2.1 nous assure que $\det(\mathcal{S}_{E,M})$ et $\det(\mathcal{H}_1)$ ont la même parité.

L'ajout d'une ligne à une autre ne change pas la valeur du déterminant. Par conséquent, $\det(\mathcal{H}_2) = \det(\mathcal{H}_1)$, avec $\mathcal{H}_2$ définie comme suit :

$$\mathcal{H}_2 = \begin{pmatrix} 1 & 0 & \dots & 0 & \overline{m_{n-1}} & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & \overline{m_{n-2}} & \overline{m_{n-1}} & \dots & 0 & 0 \\ \vdots & & \ddots & & \vdots & & & & \vdots \\ 0 & 0 & \dots & 1 & \overline{m_1} & \overline{m_2} & \dots & \overline{m_{n-1}} & 0 \\ 0 & 0 & \dots & 0 & \overline{m_0} & \overline{m_1} & \dots & \overline{m_{n-2}} & \overline{m_{n-1}} \\ 0 & 0 & \dots & 0 & \overline{m_{n-1}} & \overline{m_0} & \dots & \overline{m_{n-3}} & \overline{m_{n-2}} \\ \vdots & & & & \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 0 & \overline{m_2} & \overline{m_3} & \dots & \overline{m_0} & \overline{m_1} \\ 0 & 0 & \dots & 0 & \overline{m_1} & \overline{m_2} & \dots & \overline{m_{n-1}} & \overline{m_0} \end{pmatrix}$$

Ensuite, on a $\det(\mathcal{H}_3) = \det(\mathcal{H}_2)$, où $\mathcal{H}_3$ est la matrice circulante définie comme suit :

$$\mathcal{H}_3 = \begin{pmatrix} \overline{m_0} & \overline{m_1} & \dots & \overline{m_{n-2}} & \overline{m_{n-1}} \\ \overline{m_{n-1}} & \overline{m_0} & \dots & \overline{m_{n-3}} & \overline{m_{n-2}} \\ \vdots & & \ddots & & \vdots \\ \overline{m_2} & \overline{m_3} & \dots & \overline{m_0} & \overline{m_1} \\ \overline{m_1} & \overline{m_2} & \dots & \overline{m_{n-1}} & \overline{m_0} \end{pmatrix}$$

Par conséquent, $\det(\mathcal{S}_{E,M})$ et $\det(\mathcal{H}_3)$ ont la même parité. On a $\mathcal{H}_3 \in \mathcal{M}_{n\times n}(\mathbb{Z}/2\mathbb{Z})$, donc $\det(\mathcal{H}_3)$ est égal à 0 ou 1. Pour la suite, c'est sur la *matrice circulante* $\mathcal{H}_3$ que nous nous focaliserons pour établir une condition suffisante qui garantie que $\det(\mathcal{S}_{E,M})$ est impair. La définition 2.1 introduit une notation qui nous sera utile pour la suite.

Définition 2.1. Soit $P \in \mathbb{Z}[X]$ un polynôme tel que $P(X) = p_0 + p_1X + \cdots + p_{n-1}X^{n-1}$. On notera $\overline{P}$ le polynôme tel que : $\overline{P}(X) = p'_0 + p'_1X + \cdots + p'_{n-1}X^{n-1}$, où $p'_i = p_i \bmod 2$.

Nous avons également besoin de l'application Υ définie dans la proposition suivante.

Proposition 2.9. *[Bal14, Section 3.4] Soit $R_n = \mathbb{F}_2[X]/(X^n - 1)$ l'algèbre de tous les polynômes modulo $(X^n - 1)$, à coefficients dans $\mathbb{F}_2$. Soit C_n l'anneau des matrices circulantes binaires de dimension $n \times n$. Soit Υ l'application définie comme suit :*

$$\begin{array}{ccccc} \Upsilon & : & R_n & \to & C_n \\ & & a_0 + \cdots + a_{n-1}X^{n-1} & \mapsto & \begin{pmatrix} a_0 & a_1 & \dots & a_{n-2} & a_{n-1} \\ a_{n-1} & a_0 & \dots & a_{n-3} & a_{n-2} \\ \vdots & & \ddots & & \vdots \\ a_2 & a_3 & \dots & a_0 & a_1 \\ a_1 & a_2 & \dots & a_{n-1} & a_0 \end{pmatrix} \end{array}$$

Alors, R_n est isomorphe à C_n. et $\Upsilon(R_n) = C_n$.

Corollaire 2.9. *Soit $A \in R_n$ un polynôme, $det(\Upsilon(A)) = 1$ si et seulement si* $\text{pgcd}(A, X^n - 1) = 1$.

Démonstration. A est inversible ssi $\text{pgcd}(A, X^n - 1) = 1$. Un élément de C_n est inversible ssi son déterminant est 1. D'après la proposition 2.9, R_n est isomorphe à C_n, à travers Υ. Ce qui permet de conclure. □

Avec la définition 2.1 et le corollaire 2.9, la proposition suivante donne un critère suffisant pour que $\det(\mathcal{S}_{E,M})$ soit impair .

Proposition 2.10. *Soit $E(X) = X^n - \lambda$ tel que λ est impair. Soit $M(X) = m_0 + m_1X + \cdots + m_{n-1}X^{n-1}$ un polynôme. On a l'équivalence suivante :*

$$\det(\mathcal{S}_{E,M}) \text{ est impair si et seulement si } \text{pgcd}(\overline{M}, X^n - 1) = 1.$$

Démonstration. La matrice circulante $\mathcal{H}_3$, définie ci-dessus, est telle que $\mathcal{H}_3 = \Upsilon(\overline{M})$. On a $\overline{M} \in R_n$. Donc, d'après le corollaire 2.9, $\det(\mathcal{H}_3) = 1$ ssi $\text{pgcd}(\overline{M}, X^n - 1) = 1$. Puisque $\det(\mathcal{S}_{E,M})$ et $\det(\mathcal{H}_3)$ ont la même parité, on en déduit que $\det(\mathcal{S}_{E,M})$ est impair ssi $\text{pgcd}(\overline{M}, X^n - 1) = 1$. □

De la proposition 2.10, lorsque ϕ est une puissance de deux et λ est impair, le polynôme M' existe si $\text{pgcd}(\overline{M}, X^n - 1) = 1$.

Pour résumer cette section, lorsque $\phi \geqslant 2$ est une puissance de deux, le polynôme M' existe si :

— λ est pair et m_0 est impair,

— λ est impair et $\text{pgcd}(\overline{M}, X^n - 1) = 1$.

2.4.4 Génération du polynôme M

Soit $\phi \geqslant 2$ une puissance de deux. Dans la sous-section précédente, nous avons établi des conditions suffisantes sur le polynôme M qui garantissent l'existence du polynôme M'. Ici, nous verrons comment générer des polynômes M qui remplissent ces conditions. On suppose que $M = m_0 + m_1 X + \cdots + m_{n-1}X^{n-1}$.

Remarque 2.9. Le processus de génération du polynôme M, présenté dans cette section, assure que $\text{pgcd}(E, M) = 1$ dans $\mathbb{Q}[X]$. En effet, on vient de voir dans la sous-section précédente que M' existe si le résultant $\text{Res}(E, M)$ de E et M est impair. Si $\text{Res}(E, M)$ impair, alors $\text{Res}(E, M) \neq 0$. Donc, d'après la propriété A.1 de l'annexe A.1, $\text{pgcd}(E, M) = 1$ dans $\mathbb{Q}[X]$, car $E, M \in \mathbb{Z}[X]$.

Dans l'AMNS, les coefficients des éléments sont bornés en valeurs absolues par ρ. Dans le processus de génération (section 2.4.1), on prend $\rho \geqslant 2w\|M\|_\infty$. Donc, pour minimiser autant que possible la quantité mémoire requise pour représenter les éléments dans l'AMNS, il est indispensable que $\|M\|_\infty$ soit aussi petite que possible. De plus, avoir $\|M\|_\infty$ petite est essentiel pour la stratégie de génération proposée dans la remarque 2.6, pour une implémentation logicielle.

La génération d'un tel polynôme M se fait grâce au réseau $\mathcal{L}_\mathcal{B}$ étudié dans la section 2.2.1. L'idée pour trouver M est d'abord de calculer une base réduite de $\mathcal{L}_\mathcal{B}$ en utilisant un algorithme de réduction de réseau tel que l'algorithme LLL [LLL82], puis de prendre M comme un élément ou une combinaison spéciale d'éléments de cette base réduite. Cependant, une approche naïve ne garantit pas que cette base réduite contienne toujours un polynôme M qui respecte les critères d'existence du polynôme M'. Dans les sections 2.4.4.1 et 2.4.4.2, nous abordons cette question selon la parité λ.

2.4.4.1 Génération de M lorsque λ est pair.

Nous rappelons que lorsque λ est pair, il suffit d'avoir m_0 impair pour que M' existe. Soit $\mathcal{M}_1$ la matrice définie comme suit.

$$\mathcal{M}_1 = \begin{pmatrix} p & 0 & 0 & \dots & 0 & 0 \\ t_1 & 1 & 0 & \dots & 0 & 0 \\ t_2 & 0 & 1 & \dots & 0 & 0 \\ \vdots & & & \ddots & & \vdots \\ t_{n-2} & 0 & 0 & \dots & 1 & 0 \\ t_{n-1} & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \begin{matrix} \leftarrow p \\ \leftarrow X + t_1 \\ \leftarrow X^2 + t_2 \\ \\ \leftarrow X^{n-2} + t_{n-2} \\ \leftarrow X^{n-1} + t_{n-1} \end{matrix} \tag{2.5}$$

avec $t_i = -\gamma^i \bmod p$.

D'après le théorème 2.1, la matrice $\mathcal{M}_1$ est une base de $\mathcal{L}_\mathcal{B}$. En effet, si $t_i = -\gamma^i \bmod p$, alors il existe $k_i \in \mathbb{Z}$ tel que $t_i = -\gamma^i + k_i p$.

Proposition 2.11. *Soit $\mathcal{G} = \{\mathcal{G}_0, \dots, \mathcal{G}_{n-1}\}$ une base réduite du réseau $\mathcal{L}_\mathcal{B}$ obtenue de la base $\mathcal{M}_1$. Au moins un vecteur $\mathcal{G}_i$ de $\mathcal{G}$ est tel que $\mathcal{G}_{i,0}$ est impair.*

Démonstration. La matrice $\mathcal{G}$ est une base de $\mathcal{L}_\mathcal{B}$. Donc, le vecteur $V = (p, 0, \dots, 0)$, la première ligne de $\mathcal{M}_1$, est une combinaison linéaire à coefficients

dans $\mathbb{Z}$ des lignes $\mathcal{G}_i$ de $\mathcal{G}$. Supposons que la première composante de chaque ligne de $\mathcal{G}$ soit paire. Cela signifie que chaque combinaison linéaire d'éléments de $\mathcal{G}$ donnera un vecteur dont la première composante sera paire. Comme p est impair, ceci est en contradiction avec le fait que $V \in \mathcal{L}_\mathcal{B}$ et $\mathcal{G}$ est une base de $\mathcal{L}_\mathcal{B}$. Ainsi, au moins une ligne $\mathcal{G}_i$ de $\mathcal{G}$ est telle que $\mathcal{G}_{i,0}$ est impair. □

De la proposition 2.11, on déduit que tout algorithme de réduction de réseau appliqué à la base $\mathcal{M}_1$ donne une base réduite qui contient au moins un polynôme M tel que m_0 est impair, donc le polynôme M' existe.

2.4.4.2 Génération de M lorsque λ est impair.

Nous rappelons que lorsque λ est impair, il suffit d'avoir $\text{pgcd}(\overline{M}, X^n-1) = 1$ pour que M' existe. Soit $\mathcal{M}_2$ la matrice définie comme suit.

$$\mathcal{M}_2 = \begin{pmatrix} p & 0 & 0 & \dots & 0 & 0 \\ s_1 & 1 & 0 & \dots & 0 & 0 \\ s_2 & 0 & 1 & \dots & 0 & 0 \\ \vdots & & & \ddots & & \vdots \\ s_{n-2} & 0 & 0 & \dots & 1 & 0 \\ s_{n-1} & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \begin{matrix} \leftarrow p \\ \leftarrow X + s_1 \\ \leftarrow X^2 + s_2 \\ \\ \leftarrow X^{n-2} + s_{n-2} \\ \leftarrow X^{n-1} + s_{n-1} \end{matrix} \tag{2.6}$$

avec $s_i = t_i + pd_i$, où $t_i = -\gamma^i \bmod p$ et $d_i = t_i \bmod 2$.

Notons que tous les s_i sont pairs, car p est impair. D'après le théorème 2.1, la matrice $\mathcal{M}_2$ est une base de $\mathcal{L}_\mathcal{B}$. En effet, il existe un entier $c_i \in \mathbb{Z}$ tel que $-\gamma^i \bmod p = -\gamma^i + c_i p$. Donc, $s_i = -\gamma^i + p(c_i + d_i)$.

Proposition 2.12. *Soit $\mathcal{G} = \{\mathcal{G}_0, \dots, \mathcal{G}_{n-1}\}$ une base réduite du réseau $\mathcal{L}_\mathcal{B}$ obtenue de la base $\mathcal{M}_2$. Il existe un tuple $(\beta_0, \dots, \beta_{n-1}) \in \mathbb{F}_2^n$, tel que la combinaison linéaire binaire $M = \sum_{i=0}^{n-1} \beta_i \mathcal{G}_i$ satisfait $pgcd(\overline{M}, X^n - 1) = 1$.*

Démonstration. Puisque tous les s_i dans la matrice $\mathcal{M}_2$ sont pairs, on a :

$$\overline{\mathcal{M}_2} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

avec $\overline{\mathcal{M}_{2ij}} = \mathcal{M}_{2ij} \bmod 2$.
Soit $R_n = \mathbb{F}_2[X]/(X^n - 1)$ l'algèbre de tous les polynômes modulo $(X^n - 1)$, à coefficients dans $\mathbb{F}_2$ (voir proposition 2.9). La matrice $\mathcal{M}_2$ est une base de R_n. En effet, chaque ligne i de cette matrice correspond au polynôme $X^i \in R_n$, pour $0 \leqslant i < n$. Soit $U \in R_n$ un polynôme tel que $\text{pgcd}(U, X^n - 1) = 1$; un tel polynôme U existe toujours, par exemple on a les polynômes X, X^2, …, X^{n-1} qui sont premiers avec $X^n - 1$. Puisque $\overline{\mathcal{M}_2}$ est une base de R_n, il existe $T = (t_0, \dots, t_{n-1}) \in \mathbb{F}_2^n$ tel que $U = T\overline{\mathcal{M}_2}$. Comme $T\overline{\mathcal{M}_2} = \overline{T\mathcal{M}_2}$, on obtient que $U = \overline{T\mathcal{M}_2}$. On a $T\mathcal{M}_2 \in \mathcal{L}_\mathcal{B}$, donc il existe $V = (v_0, \dots, v_{n-1}) \in \mathbb{Z}^n$ tel

que $V\mathcal{G} = T\mathcal{M}_2$, puisque $\mathcal{G}$ est une base de $\mathcal{L}_\mathcal{B}$. Ainsi, $U = \overline{V\mathcal{G}}$. Soit $\beta = (\beta_0, \ldots, \beta_{n-1}) \in \mathbb{F}_2^n$ tel que $\beta_i = v_i \bmod 2$, on a alors $U = \overline{\beta\mathcal{G}}$. Soit $M \in \mathcal{L}_\mathcal{B}$ le polynôme tel que $M = \sum_{i=0}^{n-1} \beta_i \mathcal{G}_i$. On a $\overline{M} = U$, donc $\text{pgcd}(\overline{M}, X^n - 1) = 1$. □

D'après la proposition 2.12, tout algorithme de réduction de réseau appliqué à la base $\mathcal{M}_2$ fournit une base réduite $\mathcal{G}$ telle qu'au moins une combinaison linéaire binaire de ses lignes donne un polynôme M tel que $\text{pgcd}(\overline{M}, X^n - 1) = 1$, donc le polynôme M' existe. Ainsi, il faudra tester au maximum 2^n combinaisons linéaires des lignes de $\mathcal{G}$ pour trouver un polynôme M approprié. Pour les tailles cryptographiques, n est suffisamment petit pour permettre le test de toutes ces combinaisons. Voir par exemple l'annexe B.1 pour certaines valeurs possibles de n.

Soit $\theta = \max_{0 \leqslant i < n} \|\mathcal{G}_i\|_\infty$. Pour chaque combinaison linéaire binaire, le polynôme M correspondant vérifie $\|M\|_\infty \leqslant n\theta$. Par conséquent, si les lignes de $\mathcal{G}$ sont petites en norme infinie, alors $\|M\|_\infty$ sera aussi petite, car n est petit et négligeable par rapport à θ.

2.4.5 Un exemple complet de génération d'AMNS

Pour illustrer tout ce qui a été dit plus haut, nous donnons dans cette section un exemple complet de génération d'AMNS pour un nombre premier de 192 bits, en suivant la stratégie de génération présentée dans la remarque 2.6 pour une implémentation logicielle.

Soit $p = 451976979609104182389808764628662097050362422826890001691 1$, notre nombre premier. Nous considérons une architecture de 64 bits, i.e. $k = 64$ (donc $\phi = 2^{64}$). Nous prenons également $\delta = 10$, ce qui est plus que nécessaire pour la plupart des applications cryptographiques. La valeur optimale pour n dans cet exemple est $4 = \lfloor \frac{192}{64} \rfloor + 1$. Soit $n = 4$.

Prenons $\lambda = 2$. Une racine n-ième modulo p de λ est $\gamma = 211016621950685959256928833139050708940347031034159643483 4$. On a donc :

- $w = 7$
- $E(X) = X^4 - 2$

Puisque λ est pair, on construit la base de $\mathcal{M}_1$ du réseau $\mathcal{L}_\mathcal{B}$ comme suggéré dans la section 2.4.4.1 :

$$\mathcal{M}_1 = \begin{pmatrix} 451976979609104182389808764628662097050362422826890001691 1 & 0 & 0 & 0 \\ 240960357658418223132879931489611388110015391792730358207 7 & 1 & 0 & 0 \\ 340444712049664110992610398354066478726101722686237638914 8 & 0 & 1 & 0 \\ 320221102747260515568290013946268716788066687531882321998 2 & 0 & 0 & 1 \end{pmatrix}$$

En utilisant l'implémentation de l'algorithme LLL [LLL82] fournie dans la bibliothèque SageMath [Sa18] pour réduire cette base, nous obtenons la base réduite suivante :

$$\mathcal{M}'_1 = \begin{pmatrix} -158498747706969 & 167054566018957 & -98192163350595 & -34173855083107 \\ 68347710166214 & 158498747706969 & -167054566018957 & 98192163350595 \\ 196384326701190 & 68347710166214 & 158498747706969 & -167054566018957 \\ 175610384330945 & -29329760682233 & -166539873516809 & -192672602790076 \end{pmatrix}$$

Comme expliqué dans la section 2.4.3.1, pour assurer l'existence du polynôme M', on a seulement besoin que le coefficient constant de M soit impair. Ainsi, la première ligne de la base $\mathcal{M}'_1$ donne un bon candidat. Posons :

$$M(X) = -158498747706969+167054566018957X-98192163350595X^2-34173855083107X^3.$$

On a donc $\rho = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil} = 2^{52}$. On peut vérifier que $\phi \geqslant 2w\rho(\delta+1)^2$. Le dernier paramètre $M' = -M^{-1} \bmod (E, \phi)$ s'obtient avec un calcul simple : $M'(Y) = 15602829753513350283 + 115226280609018716 75Y + 158186106652503961 34Y^2 + 154778556213159098 52Y^3$.

2.5 Analyse et implémentation

Dans cette section, nous étudions les besoins en mémoire ainsi que la complexité des opérations principales dans l'AMNS. Nous discutons également du nombre d'AMNS qui peuvent être construits pour un nombre premier donné. Nous comparons nos implémentations logicielles de la multiplication modulaire dans l'AMNS aux implémentations des bibliothèques multi-précision GNU-MP et OpenSSL.

2.5.1 Performances théoriques et consommations mémoire

Les performances et la quantité de mémoire requises dans un AMNS dépendent principalement de l'architecture cible et de la valeur de n. Considérons une architecture de k bits, les opérations arithmétiques de base sont alors effectuées sur des mots de k bits.

Nous supposons que les entrées de nos algorithmes appartiennent à un AMNS $\mathcal{B} = (p, n, \gamma, \rho, E)$, tel que $\rho = 2^t$, $E(X) = X^n - \lambda$ et $\lambda = \pm 2^i + \varepsilon 2^j$, avec $t, i, j \in \mathbb{N}$ et $\varepsilon \in \{-1, 0, 1\}$. Ce type de λ assure la réduction externe la plus rapide, surtout pour $\varepsilon = 0$. D'après les corollaires 2.6 et 2.7 de la section 2.4.2, il est toujours possible de choisir un tel λ. En effet, si $\mathrm{pgcd}(n, p-1) = 1$, d'après le corollaire 2.6, tout $\lambda = \pm 2^i + \varepsilon 2^j$ (avec $\lambda \neq 1$) possède une unique racine n-ième non triviale modulo p qu'il est facile de calculer. Si $\mathrm{pgcd}(n, p-1) > 1$, d'après le corollaire 2.7, il suffit de prendre $\lambda = 1$. Ce n'est cependant pas une obligation, comme expliqué à la suite de ce corollaire.

Nous donnons ici les performances théoriques et les besoins mémoire selon la stratégie d'implémentation logicielle expliquée dans la remarque 2.6. Donc, nous prenons $\phi = 2^k$ et $\rho = 2^t$, avec $0 < t < k$. Le besoin en mémoire est exprimé en fonction du nombre de mots de k bits utilisés. Les performances des opérations arithmétiques sont exprimées en fonction du nombre de multiplications et d'additions de mots de k bits. Nous y ajoutons également les coûts des opérations de décalage.

Les éléments étant des polynômes dans $\mathcal{B}$, n mots de k bits sont nécessaires pour représenter chacun d'eux. Par conséquent, un élément dans $\mathcal{B}$ nécessite

nk bits pour être représenté.

Nous reprenons ici les notations de la section 1.3.5.5 pour donner les coûts théoriques des principales opérations arithmétiques dans l'AMNS. Ainsi, l'addition de deux mots de k bits est notée $\mathcal{A}$, le produit de deux mots de k bits est noté $\mathcal{M}$. Pour les opérations de décalage, $\mathcal{S}_l^i$ et $\mathcal{S}_r^i$ sont respectivement un décalage à gauche et un décalage à droite de i bits. Pour rappel, si x et y sont chacun le produit de deux mots machines, alors l'opération $x+y$ coûte au maximum $3\mathcal{A}$. Pour plus de détails sur le calcul du coût de la réduction interne (**RedCoeff**), nous renvoyons le lecteur à la remarque 1.6 de la section 1.3.5.5. Le tableau 2.1 donne les coûts des principales opérations arithmétiques dans l'AMNS. Pour la conversion vers l'AMNS (algorithme 22), nous rappelons que ρ peut toujours être pris comme une puissance de deux. Dans ce cas, la décomposition en base ρ (ligne 1 de l'algorithme) se fait très simplement. Nous ignorons son coût ici.

Addition polynomiale (alg. 20)	$n\mathcal{A}$
Multiplication polynomiale	$n^2\mathcal{M} + (3n^2 - 6n + 3)\mathcal{A}$
Réduction externe	$3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^i$ $+ \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$
Réduction interne	$2n^2\mathcal{M} + (4n^2 - n)\mathcal{A} + n\mathcal{S}_r^k$
Multiplication modulaire (alg. 19)	$3n^2\mathcal{M} + (7n^2 - 4n)\mathcal{A} + (n-1)\mathcal{S}_l^i + n\mathcal{S}_r^k$ $+ \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$
Conversion vers l'AMNS (alg. 22)	$3n^2\mathcal{M} + (7n^2 - 4n)\mathcal{A} + n\mathcal{S}_r^k$

TABLE 2.1 – Coûts théoriques des opérations, avec $E(X) = X^n - \lambda$, où $\lambda = \pm 2^i + \varepsilon 2^j$, $\varepsilon \in \{-1, 0, 1\}$ et $\phi = 2^k$.

Remarque 2.10. Indépendamment des coûts mentionnés dans le tableau 2.1, l'AMNS possède quelques avantages du fait de la structure polynomiale de ses éléments :

- À l'exception du *Square-and-Multiply* (algorithme 27), aucun des algorithmes présentés dans la section 2.3 ne contient de branchement conditionnel.
- Les éléments dans l'AMNS sont des polynômes, donc leurs coefficients sont indépendants lors des opérations arithmétiques. Ainsi, il n'y a pas de propagation de retenue entre les coefficients lors de ces opérations.
- L'AMNS possède d'excellentes propriétés de parallélisation. En effet, à l'exception des procédures de conversion de l'AMNS vers le binaire, chacune des lignes des algorithmes présentés dans la section 2.3 calcule un polynôme. Comme les coefficients des polynômes sont indépendants lors des opérations arithmétiques, ils peuvent être calculés simultanément (voir remarque 1.3 de la section 1.3.4, par exemple).

2.5.2 Nombre d'AMNS pour un nombre premier donné

Une question intéressante mais assez complexe est de savoir le nombre d'AMNS qu'on peut générer à partir d'un nombre premier et d'une architecture cible. Il est difficile de répondre à cette question en raison du large éventail de paramètres qui définissent un AMNS et aussi parce qu'elle est liée à l'existence d'une racine n-ième γ modulo p d'un entier λ.

Ici, nous donnons une réponse pour la stratégie d'implémentation logicielle de la remarque 2.6. L'objectif est de déterminer dans un premier temps l'ensemble Ω des valeurs possibles pour le paramètre λ de l'AMNS, en tenant compte du nombre maximum de coefficients qu'on accepte (i.e. les valeurs qu'on accepte pour n). Ensuite, nous déterminons le nombre d'éléments $\gamma \in \mathbb{Z}/p\mathbb{Z}$, tels que $\gamma^n \equiv \lambda \pmod{p}$, avec $\lambda \in \Omega$.

Dans la suite, nous considérons une architecture de k bits. Donc, les coefficients des éléments dans l'AMNS sont représentés sur des mots de k bits. Le paramètre δ est tel que $\delta \geqslant 0$. Ainsi, afin d'inclure tous les choix possibles pour ce paramètre en estimant le nombre d'AMNS, nous prenons $\delta = 0$. Une fois le module p fixé, on choisit dans l'ordre les paramètres n, λ et ϕ. Posons $w = 1 + (n-1)|\lambda|$. Nous supposons que $n \geqslant 2$, ce qui devrait toujours être le cas en pratique pour que l'utilisation de l'AMNS ait un intérêt.

Le paramètre ϕ est tel que $\phi \geqslant 2w\rho$. Dans la section 2.3.4.1, nous avons expliqué que le processus de génération garantit qu'on a $p \leqslant \rho^n$. Comme $n \geqslant 2$, on obtient que $2w > n|\lambda|$. Donc, $\log_2 \phi > \log_2(n) + \log_2(|\lambda|) + (\log_2 p)/n$. Ainsi,

$$\log_2 |\lambda| < \log_2 \phi - \log_2(n) - (\log_2 p)/n\,.$$

Pour la suite, nous considérons la stratégie d'implémentation logicielle de la remarque 2.6. Cela signifie que $\phi = 2^k$ et $n > \frac{\log_2 p}{k}$. Nous rappelons que n est le nombre de mots de k bits utilisés pour représenter les éléments de l'AMNS. Ainsi, on doit choisir n aussi petit que possible pour minimiser les calculs, voir tableau 2.1. Supposons que n soit choisi tel que :

$$\frac{\log_2 p}{k} + 1 + c \geqslant n > \frac{\log_2 p}{k}\,.$$

Cela signifie qu'on accepte au maximum c coefficients de plus que la valeur optimale $\lfloor \frac{\log_2 p}{k} \rfloor + 1$. Par conséquent, on a :

$$\log_2 n > \log_2 \log_2 p - \log_2 k \quad \text{et} \quad \tfrac{\log_2 p}{n} \geqslant \tfrac{k \log_2 p}{\log_2 p + kc + k}.$$

Donc,

$$\log_2 |\lambda| < k + \log_2 k - \log_2 \log_2 p - \tfrac{k \log_2 p}{\log_2 p + kc + k}.$$

Soit $\zeta = \left\lfloor k + \log_2 k - \log_2 \log_2 p - \frac{k \log_2 p}{\log_2 p + kc + k} \right\rfloor$. Le paramètre λ doit donc être tel que $|\lambda| \leqslant 2^\zeta$.

Comme dans la section 2.5.1, nous choisissons λ tel que : $\lambda = \pm 2^i + \varepsilon 2^j$, avec $i, j \in \mathbb{N}$, $\varepsilon \in \{-1, 0, 1\}$, pour accélérer les réductions modulo $X^n - \lambda$.
Soit Ω l'ensemble défini comme suit :

$$\Omega = \{\lambda, \text{ tel que : } \lambda = \pm 2^i + \varepsilon 2^j, \text{ avec } i, j \in \mathbb{N}, \varepsilon \in \{-1, 0, 1\} \text{ et } |\lambda| \leqslant 2^\zeta\}.$$

Le nombre de valeurs pour λ est donc $\#\Omega$. La principale difficulté est de déterminer le nombre de racines n-ième modulo p qui peuvent être calculées avec cet ensemble Ω. Certains éléments de Ω peuvent avoir plusieurs racines, tandis que d'autres aucune. Pour donner une réponse, nous distinguons deux cas selon $\text{pgcd}(n, p-1)$.

2.5.2.1 Cas 1 : $\text{pgcd}(n, p-1) = 1$.

D'après le corollaire 2.6, toute valeur $\lambda \in \mathbb{Z} \setminus \{0, 1\}$ donne une unique racine n-ième modulo p appropriée. Donc, dans ce cas, on peut générer au moins $\#\Omega - 2$ AMNS.

2.5.2.2 Cas 2 : $\text{pgcd}(n, p-1) > 1$.

Ce cas est plus complexe, car il nécessite le calcul de soit les racines n-ième modulo p, soit la factorisation de $X^n - \lambda$. Cependant, d'après le corollaire 2.7, pour $\lambda = 1$, au moins $\text{pgcd}(n, p-1) - 1$ AMNS peuvent être générés.

Remarque 2.11. Dans les deux cas ci-dessus, nous avons donné le nombre minimum d'AMNS qui peuvent être générés. En effet, une fois que λ et $\gamma = \lambda^{1/n} \bmod p$ sont fixés, on calcule le polynôme M en utilisant la réduction de réseau. D'après la section 2.4.4, toute base réduite, obtenue à partir de la base appropriée, fournit au moins un polynôme qui satisfait aux contraintes requises sur M. Nos expériences numériques ont montré qu'il y a en général plus d'un candidat convenable pour M dans la base réduite. De plus, certaines combinaisons linéaires d'éléments de cette base réduite donnent des candidats convenables. Pour un tuple (p, n, λ, γ), des polynômes M distincts donnent des AMNS distincts, car les résultats des opérations dépendent du paramètre M ainsi que du polynôme M' associé. On peut donc générer beaucoup plus d'AMNS que les nombres minimums donnés plus haut, en utilisant quelques combinaisons linéaires des polynômes de la base réduite.

Exemple 2.1. Nous avons généré un ensemble d'AMNS pour des nombres premiers de tailles 192, 224, 256, 384 et 521 bits. Ces tailles correspondent aux tailles de clés recommandées par le NIST pour la cryptographie sur les courbes elliptiques. Pour ce test, nous avons pris $k = 64$ et $c = 2$, i.e. nous nous sommes autorisés au maximum 2 coefficients de plus que la valeur optimale $\lfloor \frac{\log_2 p}{64} \rfloor + 1$, ce qui est assez restrictif mais préférable pour les performances. Donc, $\zeta = \lfloor 70 - \log_2 \log_2 p - \frac{64 \log_2 p}{\log_2 p + 192} \rfloor$. Nous avons choisi $\lambda \in \Omega$, comme défini plus haut.

Nous avons utilisé la bibliothèque SageMath qui implémente l'algorithme proposé dans [Joh99] pour calculer les racines n-ième modulo p de λ. Ce calcul pouvant être très long, nous avons mis dans notre code une minuterie de trente minutes. Ainsi, certaines valeurs de λ qui ont des racines n-ième peuvent avoir été rejetées. Enfin, afin d'étendre le nombre d'AMNS, nous avons vérifié toutes

les 2^n combinaisons binaires des vecteurs de la base réduite calculée lors de la génération du polynôme M. Un nombre plus grand de combinaisons devrait conduire à plus d'AMNS.

Avec ces paramètres et contraintes, le tableau 2.2 donne le nombre moyen d'AMNS obtenu pour chacune des tailles mentionnées plus haut.

Taille des entiers	192	224	256	384	521
Nombre moyen d'AMNS	10514	5578	12933	15777	18215

TABLE 2.2 – Nombre moyen (minimum) d'AMNS distincts pour quelques entiers premiers, en utilisant une minuterie de 30 minutes pour le calcul de la racine n-ième.

2.5.3 Résultats d'implémentation

Nous avons écrit avec la bibliothèque SageMath [Sa18] un programme qui génère, à partir de l'ensemble des paramètres d'un AMNS, le code C pour effectuer les opérations arithmétiques dans cet AMNS ainsi que les opérations de conversion. Nos implémentations de la génération d'AMNS, du générateur de code C ainsi que les AMNS que nous avons utilisés pour nos tests sont disponibles sur GitHub :

`https://github.com/arithPMNS/generalisation_amns`

Pour nos tests, nous avons utilisé un Dell Precision Tower 3620 (Intel Core i7-6700 et 32GB de RAM), sous Ubuntu GNOME 16.04 (64 bits). Nous avons compilé les codes C avec gcc 5.4 en utilisant l'option de compilation -O3. Nous avons comparé nos résultats aux implémentations de GNU MP 6.1.1 et d'OpenSSL 1.0.2g. Ces bibliothèques ont également été compilées avec gcc et l'option -O3.

2.5.3.1 Performances.

Pour chaque taille de clé recommandée par le NIST pour la cryptographie par courbe elliptique (192, 224, 256, 384 et 521 bits), nous avons généré un ensemble d'AMNS pour plusieurs nombres premiers ; voir l'annexe B.1 pour quelques exemples d'AMNS.

Comme dans l'exemple 2.1, nous avons pris $k = 64$ et $\phi = 2^k$. La valeur optimale de n pour la stratégie d'implémentation logicielle suggérée dans la remarque 2.6 est $n_{opt} = \lfloor \frac{\log_2 p}{k} \rfloor + 1$. Nous avons aussi pris $c = 2$, ce qui signifie que, pour chaque entier premier, des AMNS sont générés avec n égal à n_{opt},$n_{opt}+1$ et $n_{opt}+2$. Enfin, λ a été choisi tel que $\lambda = \pm 2^i + \varepsilon 2^j$, avec $i, j \in \mathbb{N}$ et $\varepsilon \in \{-1, 0, 1\}$.

Pour chaque AMNS, nous avons effectué 2^{25} multiplications modulaires en utilisant les représentations AMNS d'éléments pris aléatoirement dans $\mathbb{Z}/p\mathbb{Z}$. Pour la comparaison, nous avons également effectué 2^{25} multiplications modulaires avec les bibliothèques GNU MP [Ga] et OpenSSL [Pro] avec les mêmes

entrées qu'avec les AMNS. Pour OpenSSL, nous avons utilisé à la fois la procédure de multiplication modulaire par défaut et la multiplication modulaire Montgomery. Pour GNU MP, nous avons comparé l'AMNS aux fonctions de bas niveau ainsi qu'aux fonctions de haut niveau pour la multiplication modulaire.

Dans le tableau 2.3, nous donnons le ratio moyen entre les performances obtenues pour AMNS, GNU MP et OpenSSL. Nous calculons ces ratios pour n égal à n_{opt}, $n_{opt}+1$ et $n_{opt}+2$.

Dans le tableau 2.4, nous donnons les meilleurs ratios obtenus avec $n = n_{opt}$ pour AMNS. Ils sont obtenus pour $\lambda \in \{\pm 1, \pm 2, \pm 4\}$.

Remarque 2.12. Pour les entiers de taille 521 bits, la valeur optimale pour n est 9. Cependant, avec nos contraintes sur k, ϕ, λ et la minuterie que nous avons utilisée pour calculer les racines n-ième modulo p de λ, nous n'avons pas trouvé d'AMNS avec $n = 9$. Ainsi, pour cette taille, les ratios ont été calculés avec n égal à $n_{opt}+1$, $n_{opt}+2$ et $n_{opt}+3$.

taille de p	192			224			256		
n	**4**	**5**	**6**	**4**	**5**	**6**	**5**	**6**	**7**
ratio 1	**0.86**	1.41	2.04	**0.57**	**0.98**	1.41	**0.98**	1.42	1.84
ratio 2	0.10	0.17	0.24	0.08	0.14	0.19	0.14	0.20	0.26
ratio 3	0.21	0.34	0.49	0.16	0.27	0.39	0.30	0.43	0.55
ratio 4	0.36	0.58	0.86	0.23	0.39	0.58	0.45	0.67	0.87

taille de p	384			521		
n	**7**	**8**	**9**	**10**	**11**	**12**
ratio 1	**0.98**	1.34	1.67	**0.95**	1.18	1.36
ratio 2	0.19	0.25	0.31	0.25	0.29	0.34
ratio 3	0.43	0.58	0.73	0.56	0.69	0.80
ratio 4	0.61	0.80	1.04	0.69	0.83	0.96

TABLE 2.3 – Comparaison de l'AMNS avec les bibliothèques GNU MP et OpenSSL pour la multiplication modulaire, avec n égal à n_{opt}, $n_{opt}+1$ and $n_{opt}+2$ pour l'AMNS.

ratio 1 : AMNS/OpenSSL (Multiplication modulaire Montgomery).
ratio 2 : AMNS/OpenSSL (Multiplication modulaire par défaut).
ratio 3 : AMNS/GNU MP (Multiplication + réduction modulaire).
ratio 4 : AMNS/GNU MP (Mult. + réduc. modulaire, avec les fonctions de bas niveau).

Dans les deux tableaux 2.3 et 2.4, on peut observer que l'AMNS effectue la multiplication modulaire plus efficacement que la bibliothèque GNU MP et la méthode par défaut dans OpenSSL pour toutes les valeurs n. On peut également observer que la multiplication modulaire dans l'AMNS est légèrement plus rapide que la méthode de Montgomery dans OpenSSL lorsque la valeur de n est optimale. De plus, lorsque n_{opt} est égal au nombre de blocs utilisés pour représenter un entier de $\mathbb{Z}/p\mathbb{Z}$ dans GNU MP et OpenSSL, l'AMNS surpasse largement les deux bibliothèques (la méthode de Montgomery dans OpenSSL

(taille de p, n)	(192, 4)	(224, 4)	(256, 5)	(384, 7)	(521, 10)
ratio 1	**0.77**	**0.56**	**0.91**	**0.92**	**0.91**
ratio 2	0.09	0.08	0.13	0.18	0.24
ratio 3	0.19	0.16	0.28	0.40	0.54
ratio 4	0.33	0.23	0.41	0.57	0.66

TABLE 2.4 – Comparaison de l'AMNS avec les bibliothèques GNU MP et OpenSSL pour la multiplication modulaire, avec n égal à n_{opt} pour l'AMNS (meilleurs ratios).

incluse). C'est le cas lorsque p est de taille 224 ou 521 bits.
Comme expliqué dans la remarque 2.12, nous n'avons pas trouvé d'AMNS pour (taille de p, n) = $(521, 9)$. Cependant, avec (taille de p, n) = $(521, 10)$, l'AMNS est plus performant que les deux bibliothèques. Dans les autres cas, l'AMNS utilise au moins un bloc de plus que GNU MP et OpenSSL mais reste compétitif.

Remarque 2.13. Bien que l'AMNS soit beaucoup plus rapide que GNU MP et la méthode par défaut OpenSSL pour la multiplication modulaire, le ratio le plus pertinent dans les tableaux 2.3 et 2.4 est *ratio 1*, car l'AMNS nécessite à peu près la même quantité de données à pré-calculer que la multiplication modulaire de Montgomery dans OpenSSL.

Dans ces tableaux, l'AMNS est plus performant que les bibliothèques GNU MP et OpenSSL parce que :

- il n'y a pas de retenue à gérer ni de branchement conditionnel, grâce à la structure polynomiale des éléments de l'AMNS,
- la forme du polynôme $E(X) = X^n - \lambda$, rend la réduction externe très rapide avec l'algorithme 13. De plus, le paramètre λ a été choisi égal à $\pm 2^i + \varepsilon 2^j$, ce qui accélère également la réduction externe,
- ϕ a été choisi selon la stratégie d'implémentation logicielle suggérée dans la remarque 2.6.

Ces résultats ont été obtenus sans les possibilités de parallélisation offertes par l'AMNS. Comme expliqué dans la remarque 2.10, les éléments étant des polynômes dans l'AMNS, il est toujours possible d'effectuer un grand nombre d'opérations simultanément. Ainsi, on peut par exemple diviser par n le temps d'exécution de la ligne 1 de l'algorithme 19, en calculant de façon indépendante les n coefficients du résultat R. La même chose peut être faite avec toutes les lignes de **RedCoeff** (algorithme 18), qui est appelée dans l'algorithme 19.

Une addition devrait être toujours plus rapide dans l'AMNS que dans la représentation binaire classique, puisque l'addition dans l'AMNS est une simple addition polynomiale sans retenue à gérer, contrairement à la représentation binaire classique. De plus, la structure polynomiale des éléments dans l'AMNS

permet de calculer simultanément tous les n coefficients du résultat de l'addition. Cela devrait rendre l'addition dans l'AMNS encore plus rapide.

2.5.3.2 Consommation mémoire.

Le tableau 2.5 donne la quantité de mémoire nécessaire pour stocker un entier modulo p dans l'AMNS ainsi qu'avec GNU MP et OpenSSL. Ici, nous donnons le nombre d'entiers de 64 bits utilisés pour représenter les éléments de $\mathbb{Z}/p\mathbb{Z}$. Pour l'AMNS, nous prenons $n = n_{opt}$. En annexe B.3 sont fournis les codes sources correspondant aux structures de données utilisées pour stocker les grands entiers avec GNU MP et OpenSSL. Pour les fonctions de bas niveau de GNU MP, un grand entier est un tableau d'entiers de 64 bits.

Taille de p	192	224	256	384	521
AMNS	4	4	5	7	10
GNU MP (bas niveau)	3	4	4	6	9
GNU MP (mpz_t)	4	5	5	7	10
OpenSSL (BIGNUM)	5	6	6	8	11

TABLE 2.5 – Nombre de mots de 64 bits utilisés pour stocker les éléments de $\mathbb{Z}/p\mathbb{Z}$ dans GNU MP, OpenSSL et l'AMNS avec n égal à n_{opt}.

Dans le tableau 2.5, on peut observer que l'AMNS et le type *mpz_t* de GNU MP utilisent en général la même quantité de mémoire pour représenter les éléments $\mathbb{Z}/p\mathbb{Z}$, alors que le type BIGNUM d'OpenSSL en utilise toujours un peu plus. On peut aussi observer que l'AMNS utilise généralement un entier de 64 bits de plus que ce qui est requis pour les fonctions de bas niveau de GNU MP.

Remarque 2.14. Pour AMNS et OpenSSL Montgomery, certaines données doivent être pré-calculées. Cependant, les besoins en mémoire de ces données sont négligeables par rapport à l'utilisation globale de la mémoire lorsque ces données sont utilisées pour plusieurs opérations arithmétiques.

2.6 Conclusion et perspectives

Dans ce chapitre, nous avons établi les bases indispensables pour une arithmétique efficace dans l'AMNS. Nous avons montré comment générer plusieurs AMNS pour un nombre premier quelconque, afin d'effectuer efficacement les opérations arithmétiques en utilisant la méthode Montgomery-like pour la réduction interne. Nous avons en outre proposé un ensemble complet d'algorithmes qui devraient servir de base pour toutes opérations qu'on pourrait effectuer dans l'AMNS. On a pu constater que l'AMNS peut être une alternative intéressante au système de représentation classique à la fois pour les performances et les possibilités de parallélisation qu'il offre.

Dans le prochain chapitre, nous nous intéresserons aux avantages et possibilités qu'offre l'AMNS pour la protection des opérations contre les attaques par anaux auxiliaires.

Malgré les bases posées dans ce chapitre, il reste un certain nombre d'éléments qu'il pourrait être intéressant d'approfondir. Dans la section 1.3.4, nous avons fait remarquer que la forme du polynôme de réduction externe pour l'AMNS offre de manière générale les meilleures performances, en plus d'un moindre surcoût mémoire lors de la réduction externe. Toutefois, il pourrait s'avérer intéressant de s'intéresser à un ensemble plus vaste de polynômes de réduction externe, i.e. les PMNS (cf. définition 1.11). D'une part, pour pouvoir générer encore plus de PMNS pour un module premier quelconque. D'autre part, pour tirer avantage de certains cas généraux de polynômes de réduction externe qui sont tous autant intéressants (sinon plus) qu'une bonne partie de ceux de l'AMNS. Par exemple, le polynôme $E_1(X) = X^n - X - 1$ est plus intéressant à la fois pour les performances et le surcoût mémoire que le polynôme $E_2(X) = X^n - 7$. En effet, pour assurer la consistance des opérations arithmétiques, le paramètre ρ_1 d'un PMNS associé au polynôme E_1 doit être tel que $\rho_1 \geqslant (4n-2)\|M_1\|_\infty$, alors que le paramètre ρ_2 d'un AMNS associé au polynôme E_2 doit être tel que $\rho_2 \geqslant (14n-12)\|M_2\|_\infty$, où M_1 et M_2 sont les polynômes de réduction interne associés aux différents systèmes.

Le test égalitaire (ou la comparaison de façon générale) reste jusque-là une opération assez coûteuse pour l'AMNS. En effet, si on a $A, B \in \mathcal{B}$ et qu'on souhaite vérifier si A et B représentent le même élément, i.e. $A(\gamma) \equiv B(\gamma) \pmod p$, on dispose de deux options. Posons $C = A - B$, la première option est de d'abord revenir à la représentation classique en évaluant C en γ modulo p (avec l'algorithme 24 ou l'algorithme 25 qui est beaucoup plus rapide mais nécessite des pré-calculs). Les polynômes A et B sont alors égaux si et seulement si cette évaluation est égale à 0. La seconde option est de vérifier si C appartient au réseau euclidien $\mathcal{L}_\mathcal{B}$ associé à $\mathcal{B}$.

Bien que n'apparaissant dans aucun des cas d'usage présenté dans cette thèse, il serait intéressant de pouvoir obtenir facilement une représentation d'un élément $a \in \mathbb{Z}/p\mathbb{Z}$ dans un AMNS $\mathcal{B}_2 = (p, n_2, \gamma_2, \rho_2, E_1)$, à partir d'une de ses représentations dans un autre AMNS $\mathcal{B}_1 = (p, n_1, \gamma_1, \rho_1, E_2)$. Ici encore, la solution pour effectuer cette opération est assez coûteuse. En effet, à moins que $\gamma_1 = \gamma_2$ (auquel cas la représentation reste la même dans $\mathcal{B}_2$, si $\rho_1 \leqslant \rho_2$ et $n_1 \leqslant n_2$), il est nécessaire de revenir dans la représentation classique pour ensuite convertir l'élément obtenu dans l'AMNS $\mathcal{B}_2$. Cette solution coûte donc deux opérations de conversion.

Pour terminer, nos objectifs principaux dans cette thèse étaient d'une part la génération de l'AMNS pour une utilisation efficace de la méthode Montgomery-like, et d'autre part de savoir si ce système pouvait être une alternative intéressante aux méthodes classiques telles que la multiplication modulaire de Montgomery, dont la méthode Montgomery-like est très proche. La comparaison de l'AMNS à des systèmes de numération non positionnels tels que le RNS fera l'objet d'un travail futur.

Chapitre 3

Randomisation des opérations arithmétiques avec l'AMNS

Sommaire

3.1 Introduction

La plupart des cryptosystèmes à clés publiques nécessitent des opérations arithmétiques sur de grands entiers. Ainsi, pour que ces cryptosystèmes soient utilisables en pratique, il est nécessaire que ces opérations arithmétiques se fassent rapidement et aussi de manière sécurisée pour résister aux attaques par canaux auxiliaires (SCA : Side Channel Attacks) [Koc96a]. L'objectif de ces

attaques est d'utiliser les fuites d'informations du matériel qui exécute le protocole cryptographique afin de retrouver totalement ou partiellement des données secrètes. Dans le cadre de la cryptographie sur les courbes elliptiques (ECC), où ces attaques ont été prouvées efficaces [ACL19], la plupart des contre-mesures reposent sur la protection de la multiplication scalaire, car cette dernière est l'opération principale et aussi celle qui assez souvent fait intervenir les données sécrètes dans l'ECC. Si besoin, nous renvoyons le lecteur au chapitre 4 pour une introduction sur les courbes elliptiques ainsi que la multiplication scalaire sur ces dernières.

La multiplication scalaire sur les courbes elliptiques (ECSM : Elliptic Curve Scalar Multiplication) consiste à effectuer l'opération kP, où k est un entier et P un point d'une courbe elliptique. Pour protéger cette opération contre les SCA basiques telles que les attaques de type SPA (Simple Power Analysis), il suffit qu'elle soit faite de façon régulière, i.e. sans branchement conditionnel (exploitable). Pour cela, il y a des algorithmes tels que l'*échelle de Montgomery* [Mon87] et le *double-and-add-always* [Cor99]. Cependant pour les attaques plus poussées, telles que les attaques de type DPA (Differential Power Analysis) [KJJ99], il est nécessaire d'avoir recours à des contre-mesures supplémentaires. Contre les attaques de type DPA par exemple, une bonne partie des contre-mesures [DKJ05, FV12, ACL19] reposent sur la randomisation des données en entrée ainsi que les valeurs intermédiaires, la randomisation de la séquence des instructions, la randomisation des opérations arithmétiques et/ou la randomisation des adresses mémoires.

L'AMNS possède des propriétés intéressantes contre les attaques par canaux auxiliaires. En effet, on a pu constater dans le chapitre précédent qu'aucun des algorithmes pour les opérations principales (réductions, conversions, addition et multiplication) n'a de branchement conditionnel. Ce qui rend l'AMNS intrinsèquement sûr contre les attaques simples de type SPA (au moins au niveau algorithmique). L'absence de branchement conditionnel aide à protéger les implémentations contre les attaques de type DPA. Cependant, cela n'est pas suffisant. Car comme expliqué plus haut, il est également indispensable de randomiser, entre autres, les entrées ainsi que les opérations arithmétiques pour protéger les opérations telles que l'ECSM contre les attaques de type DPA. C'est ce à quoi nous nous intéressons dans ce chapitre.

Deux types de randomisations sont possibles avec l'AMNS. Le premier est une randomisation d'AMNS à AMNS. Cette randomisation tire parti de l'existence de plusieurs AMNS pour un même nombre premier p, afin de randomiser les représentations des éléments de $\mathbb{Z}/p\mathbb{Z}$. Le second type est une randomisation à l'intérieur d'un même AMNS. Nous verrons que cela permet de randomiser les représentations des éléments dans l'AMNS ainsi que les opérations arithmétiques. L'avantage de ces randomisations est qu'elles ont lieu au niveau arithmétique. Ainsi, elles peuvent être combinées avec les autres contre-mesures classiques, telles que la randomisation du scalaire ou le masquage de point [Cor99] (pour randomiser le point P ainsi que les points intermédiaires lors de la multiplication scalaire).

Dans ce chapitre, nous verrons comment randomiser les représentations des éléments à l'aide de plusieurs AMNS pour un même nombre premier p. Nous aborderons également la randomisation à l'intérieur d'un même AMNS. Plus précisément, nous verrons comment randomiser la conversion (du binaire vers l'AMNS) ainsi que la multiplication modulaire dans l'AMNS, en utilisant la méthode Montgomery-like (section 2.3.1) pour la réduction interne. Ensuite, nous verrons que ces randomisations permettent lors de l'ECSM de randomiser le point P en entrée, la sortie ainsi que l'ensemble des opérations arithmétiques effectuées.

Il est possible de randomiser la conversion et la multiplication, en utilisant l'algorithme de Babaï (algorithme 16) comme méthode de réduction interne. Cette randomisation est cependant nettement moins intéressante (en nombre de calculs et en ressource mémoire) que celle avec la méthode Montgomery-like que nous présentons dans ce chapitre. Pour rappel, nous désignons l'algorithme de la méthode Montgomery-like par **RedCoeff**. Dans l'annexe D, nous présentons brièvement la randomisation avec l'algorithme de Babaï.

Ce chapitre est organisé comme suit. Nous commençons par présenter la randomisation à l'aide de plusieurs AMNS. Ensuite, nous nous intéressons à la randomisation dans un même AMNS. Nous expliquons le principe de cette randomisation, puis abordons la randomisation de la conversion et de la multiplication. Par suite, nous montrons comment randomiser la multiplication scalaire sur les courbes elliptiques. Enfin, nous terminons par une conclusion et quelques perspectives.

Le travail présenté ici à fait l'objet d'une publication [DDEM+19]. Dans cet article, l'étude a été faite sur le PMNS de manière générale. Ici, nous nous focalisons seulement sur l'AMNS. Cela nous permettra de présenter des bornes plus intéressantes que celles de l'article, grâce au travail présenté dans le chapitre 2 (que nous supposons connu du lecteur). De plus, nous tenons compte du paramètre δ qui n'apparaît dans cet article. Pour rappel, le paramètre δ représente le nombre maximum d'éléments de l'AMNS à additionner avant une multiplication modulaire (voir section 2.3.3).

Soit $p \geqslant 3$ un nombre premier. Dans toute la suite de ce chapitre, on considérera l'AMNS $\mathcal{B} = (p, n, \gamma, \rho, E)$, avec $E(X) = X^n - \lambda$. Comme dans le chapitre 2, nous posons :

$$w = 1 + (n-1)|\lambda|.$$

3.2 Randomisation avec plusieurs AMNS

Plusieurs AMNS peuvent être construits pour un même entier premier p. En effet, étant donné un tuple (p, n, λ), si γ une racine n-ième modulo p de λ existe, alors d'après la proposition 2.6, le nombre de telles racines est $\gcd(n, p-1)$. Chacune d'elles permet de construire au moins un AMNS pour p. De plus, les corollaires 2.6 et 2.7 montrent que pour tout couple (p, n), il est toujours possible de choisir $\lambda \in \mathbb{Z} \setminus \{0\}$ tel que $\gamma^n \pmod p \equiv \lambda$, avec $\gamma \in \mathbb{Z}/p\mathbb{Z}$ facilement

calculable. Les résultats d'implémentation (tableau 2.2, chapitre 2) montrent qu'il est possible de générer des milliers d'AMNS pour un même nombre premier p, en utilisant le processus de génération d'AMNS présenté dans la section 2.4, même avec des conditions restrictives.

L'existence de plusieurs AMNS distincts pour un même entier premier p permet d'avoir plusieurs représentations distinctes de chaque élément de $\mathbb{Z}/p\mathbb{Z}$. Ceci pourrait être utilisé pour randomiser les données d'entrée des protocoles cryptographiques (les coordonnées du point P au début de la multiplication scalaire $k\,P$, par exemple). Dans ce cas, la randomisation est réalisée en précalculant d'abord un ensemble Ω d'AMNS pour le module p. Ensuite, chaque fois qu'un protocole utilisant ce module p est exécuté, on sélectionne aléatoirement un AMNS dans Ω afin d'effectuer les opérations arithmétiques avec cet AMNS. Ce faisant, chaque exécution de ce protocole, avec les mêmes entrées, devrait conduire à une représentation aléatoire de ces entrées. Dans l'annexe B.2, nous donnons des exemples de représentations pour certains éléments de $\mathbb{Z}/p\mathbb{Z}$ d'un AMNS à un autre. Pour que cette méthode de randomisation des données en entrée soit sûre, il est indispensable qu'il n'y ait pas de corrélations exploitables entre les différentes représentations d'un élément de $\mathbb{Z}/p\mathbb{Z}$ d'un AMNS à un autre. Nous prévoyons de faire cette étude dans un travail futur.

3.3 Randomisation dans un même AMNS

Dans cette section, nous présentons la randomisation des opérations de conversion et de multiplication, avec **RedCoeff** (algorithme 18 ci-dessous) comme méthode de réduction interne. Si besoin, nous renvoyons le lecteur vers la section 2.3.1 du chapitre précédent qui porte sur cet algorithme. Pour rappel, ce dernier nécessite deux polynômes M et M', tels que : $M \in \mathbb{Z}_{n-1}[X]$, $M(\gamma) \equiv 0 \pmod{p}$ et $M' = -M^{-1} \bmod (E, \phi)$.

Algorithme 18 RedCoeff [NP08]

Entrée(s) : $V \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p, n, \gamma, \rho, E)$, les paramètres ϕ, M et M'.
Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv V(\gamma)\phi^{-1} \pmod{p}$

1: $Q \leftarrow V \times M' \bmod (E,\ \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: retourner S

Nous commençons par expliquer le principe de la randomisation, puis présentons les opérations randomisées.

3.3.1 Principe de la randomisation

Dans cette section, nous présentons le principe de la randomisation dans l'AMNS. Soit $a \in \mathbb{Z}/p\mathbb{Z}$. Pour rappel, $\mathcal{B}_a$ correspond à l'ensemble des représentations de a dans $\mathcal{B}$ et $\#\mathcal{B}_a$ est le cardinal de $\mathcal{B}_a$ (définition 1.9).

3.3.1.1 Idée générale

Nous commençons par introduire un nouveau paramètre $z \in \mathbb{N}$ ainsi que l'ensemble $\mathcal{H}$ défini comme suit :

$$\mathcal{H} = \{Z \in \mathbb{Z}_{n-1}[X], \text{ tel que } \|Z\|_\infty \leqslant z\} \tag{3.1}$$

Le paramètre z sera choisi pendant le processus de génération. Les éléments de $\mathcal{H}$ sont appelés *polynômes de randomisation* et on a $\#\mathcal{H} = (2\,z+1)^n$.

La randomisation ici est basée sur la redondance dans l'AMNS et se fait en deux étapes. La première est de générer l'AMNS de telle sorte que chaque élément de $\mathbb{Z}/p\mathbb{Z}$ y possède au minimum $\#\mathcal{H}$ représentations *distinctes*. Ainsi, pour tout élément $a \in \mathbb{Z}/p\mathbb{Z}$, on aura : $\#\mathcal{B}_a \geqslant \#\mathcal{H}$. La seconde étape, qui correspond à la randomisation proprement dite, consiste à parcourir de façon aléatoire ces représentations au cours des opérations arithmétiques. Ce parcours aléatoire se fera grâce aux polynômes de randomisation.

Formellement, la première étape consiste à générer l'AMNS $\mathcal{B}$, tel que :

- pour tout $a \in \mathbb{Z}/p\mathbb{Z}$, chaque élément $Z_i \in \mathcal{H}$ calcule une représentation $A_i \in \mathcal{B}$ de a.
- de plus, si $Z_i \neq Z_j$, alors $A_i \neq A_j$.

Pour la seconde étape, on générera aléatoirement un polynôme de randomisation $Z \in \mathcal{H}$ pour atteindre la représentation associée. C'est ce dont il sera question dans les sections 3.3.2 et 3.3.3.

Pour la suite de ce chapitre, on désignera par **RandPoly** la fonction qui génère aléatoirement un polynôme de randomisation ; i.e. si $Z = \mathbf{RandPoly}(z)$, alors : $Z(X) = z_0 + z_1 X + \cdots + z_{n-1}X^{n-1}$, avec $-z \leqslant z_i \leqslant z$. Nous supposons que cette fonction est sûre ; voir par exemple [BSS+16].

3.3.1.2 Conditions pour une randomisation correcte

Soit $A \in \mathcal{B}$ un polynôme. Pour que la randomisation dans l'AMNS se fasse correctement, trois conditions doivent être satisfaites.

La première est que la sortie d'une opération randomisée appartienne toujours à l'AMNS. Donc, cette sortie doit être de degré et de norme infinie respectivement inférieurs strictement à n et ρ.

La seconde condition est que la randomisation ne modifie pas l'évaluation du résultat en γ. En d'autres termes, si $B \in \mathcal{B}$ est la sortie randomisée d'une opération qui calcule A, alors on doit avoir : $B(\gamma) \equiv A(\gamma) \pmod p$.

La troisième et dernière condition est que deux polynômes de randomisation différents conduisent à deux représentations différentes pour la même opération, avec les mêmes entrées. C'est ce que nous appellerons l'absence de *collision*. Cette condition est indispensable pour pouvoir atteindre les $\#\mathcal{H}$ représentations distinctes de chaque élément de $\mathbb{Z}/p\mathbb{Z}$.

Pour la suite, nous présentons les algorithmes randomisés ainsi que les nouvelles bornes sur les paramètres ρ et ϕ qui garantissent que les conditions de la section 3.3.1.2 seront respectées. Ces nouvelles bornes dépendent du paramètre z introduit dans la section 3.3.1.1.

Avant de présenter ces algorithmes, nous rappelons ces points indispensables qui ont été prouvés dans le chapitre précédent :

- Si $A, B \in \mathbb{Z}_{n-1}[X]$ et $C = AB \bmod E$, alors $\|C\|_\infty \leqslant w\|A\|_\infty\|B\|_\infty$.
- L'entier p est tel que : $p \leqslant (w\|M\|_\infty + 1)^n \leqslant (2w\|M\|_\infty)^n$.
- $\text{pgcd}(E,\ M) = 1$, dans $\mathbb{Q}[X]$.

3.3.2 Randomisation du processus de conversion

Soit $a \in \mathbb{Z}/p\mathbb{Z}$. Notre objectif est de calculer une représentation randomisée $A \in \mathcal{B}$ de a. Pour rappel, l'utilisation de **RedCoeff** comme méthode de réduction interne exige que la représentation A soit telle que $A(\gamma) \equiv a\phi \pmod p$, i.e. $A \equiv (a\phi)_\mathcal{B}$ (section 2.3.4).

Nous commençons par présenter l'algorithme 29 de conversion randomisée, puis terminons avec le théorème 3.1 qui donne les nouvelles bornes que les paramètres ρ et ϕ de l'AMNS doivent respecter.

Soit $\beta = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil}$. L'algorithme 29 nécessite les représentations exactes P_i de $(\beta^i\phi^2)$ dans $\mathcal{B}$. Dans la section 2.3.4 du chapitre précédent, nous avons expliqué comment calculer ces représentations, à l'aide de l'algorithme 23. Nous supposons ici que toutes les représentations P_i ont été calculées avec cet algorithme. Donc, on a :

$$\|P_i\|_\infty < 2w\|M\|_\infty, \text{ pour tout } 0 \leqslant i < n.$$

Algorithme 29 *Conversion randomisée du binaire à l'AMNS*

Entrée(s) : $a \in \mathbb{Z}/p\mathbb{Z}$, $\mathcal{B} = (p,\ n,\ \gamma,\ \rho, E)$, $z \in \mathbb{N}$, les polynômes M et P_i, ainsi que la base $\beta = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil}$

Sortie : $A \in \mathcal{B}$, tel que $A \equiv (a\phi)_\mathcal{B}$

1: $Z \leftarrow$ **RandPoly**(z)
2: $J \leftarrow Z \times M \bmod E$
3: $t = (a_{n-1}, ..., a_0)_\beta$ # la décomposition en base β de a
4: $U \leftarrow \sum_{i=0}^{n-1} a_i P_i(X)$
5: $V \leftarrow U + J$
6: $A \leftarrow$ **RedCoeff**$(V) + J$
7: retourner A

Comme mentionné plus haut, on a : $p \leqslant (2w\|M\|_\infty)^n$. Donc, $p \leqslant \beta^n$. Par conséquent, la décomposition en base β de a avec n coefficients (ligne 3, algorithme 29) est toujours faisable et unique.

Théorème 3.1. *Soit $a \in \mathbb{Z}/p\mathbb{Z}$ l'entier à convertir. Soit $\beta = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil}$. Si les paramètres ρ et ϕ de l'AMNS $\mathcal{B}$ sont tels que :*

$$\rho \geqslant w\,\|M\|_\infty\,(2+z) \quad et \quad \phi \geqslant max(2\,w\,\rho\,(\delta+1)^2,\, 2\,n\,\beta + z)$$

alors, l'algorithme 29 permet de calculer $(2\,z+1)^n$ représentations distinctes de a, toutes appartenant à $\mathcal{B}$.

Démonstration. L'objectif ici est de montrer que les trois conditions de la section 3.3.1.2 sont respectées. Soit $m = \|M\|_\infty$.

Premièrement, on a $M(\gamma) \equiv 0 \pmod p$, donc $J(\gamma) \equiv 0 \pmod p$. Par conséquent, $A(\gamma) \equiv U(\gamma)\,\phi^{-1} \pmod p$. Donc $A(\gamma) \equiv a\,\phi \pmod p$, car $P_i \equiv (\beta^i\phi^2)_{\mathcal{B}}$.

Montrons maintenant que la sortie A est bien dans l'AMNS. On a :

$$\begin{aligned} A &= \phi^{-1} \times (U + J + (Q\,M \bmod E)) + J \\ &= \phi^{-1} \times (U + ((\phi+1)Z\,M + Q\,M) \bmod E), \end{aligned}$$

avec $Q = (U+J)\,M' \bmod (E, \phi)$.

On a également : $\begin{cases} \|U\|_\infty < n\,(2\,w\,m)\,\beta, \text{ car } \|P_i\|_\infty < 2\,w\,m, \\ \|ZM \bmod E\|_\infty \leqslant w\,z\,m, \text{ car } \|Z\|_\infty \leqslant z, \\ \|QM \bmod E\|_\infty < w\,\phi\,m, \text{ car } \|Q\|_\infty < \phi. \end{cases}$

Donc,

$$\begin{aligned} \|A\|_\infty &\leqslant \frac{1}{\phi}\,(\|U\|_\infty + (\phi+1)\|Z\,M \bmod E\|_\infty + \|Q\,M \bmod E\|_\infty) \\ &< \frac{w\,m\,(2\,n\,\beta + z)}{\phi} + w\,m(1+z) \end{aligned}$$

Puisque $\phi \geqslant max(2\,w\,\rho\,(\delta+1)^2,\, 2\,n\,\beta+z)$, on a $\phi \geqslant 2\,n\,\beta + z$ et donc $\|A\|_\infty < w\,m(2+z)$. D'où, $\|A\|_\infty < \rho$, car $\rho \geqslant w\,m\,(2+z)$.

Montrons enfin qu'il n'y a pas de collision. Il s'agit de montrer que, pour la même entrée a et deux polynômes de randomisation Y et Z distincts, l'algorithme 29 retourne deux sorties A_1 et A_2 distinctes. Nous procédons par l'absurde. Supposons donc que $A_1 = A_2$. On a :

$$\begin{aligned} A_1 &= \phi^{-1} \times (U + ((\phi+1)Y\,M + Q_1\,M) \bmod E), \\ A_2 &= \phi^{-1} \times (U + ((\phi+1)Z\,M + Q_2\,M) \bmod E). \end{aligned}$$

Ainsi, $A_1 = A_2$ implique que :

$$((\phi+1)Y + Q_1)\,M \bmod E = ((\phi+1)Z + Q_2)\,M \bmod E.$$

Donc, $(\phi+1)\,(Y-Z) = Q_2 - Q_1$, car $\text{pgcd}(E,\, M) = 1$ dans $\mathbb{Q}[X]$.
Puisque $Y \neq Z$, alors il existe i tel que : $|y_i - z_i| \geqslant 1$. Donc, $|(\phi+1)(y_i - z_i)| \geqslant \phi+1 > \phi$. On a cependant que : $\|Q_2 - Q_1\|_\infty < \phi$, car $\|Q_1\|_\infty, \|Q_2\|_\infty < \phi$; nous rappelons que les coefficients des polynômes Q_1 et Q_2 sont toujours positifs.
Donc, $(\phi+1)\,(Y-Z) \neq Q_2 - Q_1$. Par conséquent, on ne peut pas avoir $A_1 = A_2$. Il n'y a donc pas de collision.

Enfin, notons que $\phi \geqslant max(2\,w\,\rho\,(\delta+1)^2,\, 2\,n\,\beta+z)$ implique $\phi \geqslant 2\,w\,\rho\,(\delta+1)^2$. Cette condition est indispensable pour garantir la consistance de la multiplication dans l'AMNS (voir section 2.3). □

Remarque 3.1.

- Pour que la conversion randomisée soit consistante, il est indispensable que les nouvelles bornes sur ρ et ϕ (du théorème 3.1) soient prises en compte, lors du processus de génération présenté dans la section 2.4.
- Si le paramètre z est tel que $z \geqslant 2$, ce qui devrait toujours être le cas en pratique afin que $\#\mathcal{H} = (2\,z+1)^n$ soit suffisamment grand, alors la borne sur ϕ dans le théorème 3.1 devient simplement $\phi \geqslant 2w\rho(\delta+1)^2$.
  En effet, d'une part, on a $\beta = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil}$, donc $\beta < 4w\|M\|_\infty$. D'autre part, la borne sur ρ implique que $\rho \geqslant 4w\|M\|_\infty + w\|M\|_\infty(z-2)$. Donc, $\rho \geqslant \beta+1+w\|M\|_\infty(z-2)$. Par conséquent, $2w\rho \geqslant 2n\beta+2w+2w^2\|M\|_\infty(z-2)$, car $w \geqslant n \geqslant 1$. Si $z \geqslant 2$, alors $2w + 2w^2\|M\|_\infty(z-2) \geqslant 2z-2 \geqslant z$, car $\|M\|_\infty \geqslant 1$. Ainsi, $2w\rho(\delta+1)^2 \geqslant 2n\beta + z$, car $\delta \geqslant 0$.

3.3.3 Randomisation de la multiplication modulaire

Cette section porte sur la randomisation de la multiplication modulaire dans l'AMNS. Comme dans la section précédente, nous présentons l'algorithme randomisé, puis les nouvelles bornes sur ρ et ϕ de cet algorithme.

L'algorithme 30 est la multiplication randomisée. Afin de randomiser la sortie ainsi que toutes les opérations intermédiaires, cet algorithme commence en randomisant un des opérandes.

Algorithme 30 *Multiplication randomisée dans l'AMNS*

Entrée(s) : $A, B \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p,\, n,\, \gamma,\, \rho, E)$, $z \in \mathbb{N}$ et le polynôme M
Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod p$

1: $Z \leftarrow$ **RandPoly**(z)
2: $J \leftarrow Z \times M \bmod E$
3: $B' \leftarrow B + J$
4: $C \leftarrow A \times B' \bmod E$
5: $R \leftarrow$ **RedCoeff**(C)
6: $S \leftarrow R + 2 \times J$
7: retourner S

Théorème 3.2. *Soit $A, B \in \mathbb{Z}_{n-1}[X]$ tels que : $\|A\|_\infty < (\delta+1)\rho$ et $\|B\|_\infty < (\delta+1)\rho$. Si les paramètres ρ et ϕ de l'AMNS $\mathcal{B}$ sont tels que :*

$$\rho \geqslant w\,\|M\|_\infty\,(2+2\,z) \quad \textit{et} \quad \phi \geqslant \tfrac{3\,\rho^2}{2\,\|M\|_\infty}(\delta+1)^2$$

alors, avec A et B comme entrées, l'algorithme 30 permet de calculer $(2\,z+1)^n$ représentations distinctes de $A(\gamma)B(\gamma)\phi^{-1} \bmod p$, toutes appartenant à $\mathcal{B}$.

Démonstration. Ici aussi, l'objectif est de montrer que les trois conditions de la section 3.3.1.2 sont respectées. Soit $m = \|M\|_\infty$.

Premièrement, on a $M(\gamma) \equiv 0 \pmod p$. Donc, $J(\gamma) \equiv 0 \pmod p$ et $B'(\gamma) \equiv B(\gamma) \pmod p$. Ainsi, $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod p$.

Montrons maintenant que la sortie de l'algorithme est effectivement dans l'AMNS. On a :

$$\begin{aligned} S &= \phi^{-1} \times ((C + (Q\,M \bmod E) + 2\,J \\ &= \phi^{-1} \times ((A\,B' + Q\,M) \bmod E) + 2\,J \\ &= \phi^{-1} \times (A\,B \bmod E + (A\,Z\,M + Q\,M + 2\,\phi\,Z\,M) \bmod E), \end{aligned}$$

avec $Q = (A\,B'\,M') \bmod (E, \phi)$.
On a également :

$$\begin{cases} \rho \geqslant w\,m\,(2 + 2\,z) > 2\,w\,m\,z \;\Rightarrow\; \rho/2 > w\,m\,z, \\ \|J\|_\infty = \|Z\,M \bmod E\|_\infty \leqslant w\,\|M\|_\infty\,\|Z\|_\infty \leqslant w\,m\,z \;\Rightarrow\; \|J\|_\infty \leqslant \rho/2, \\ \|B'\|_\infty = \|B + J\|_\infty \leqslant \|B\|_\infty + \|J\|_\infty \;\Rightarrow\; \|B'\|_\infty < (\delta+1)\,\rho + \rho/2 \leqslant \frac{3}{2}\,\rho\,(\delta+1). \end{cases}$$

Donc,

$$\begin{aligned} \|S\|_\infty &= \|\phi^{-1} \times ((A\,B' + Q\,M) \bmod E) + 2\,J\|_\infty \\ &\leqslant \phi^{-1}\,w\,(\|A\|_\infty\,\|B'\|_\infty + \|Q\|_\infty\,\|M\|_\infty) + 2\|J\|_\infty \\ &< \frac{w}{\phi}\left(\frac{3}{2}\,\rho^2\,(\delta+1)^2 + \phi\,m\right) + 2\,w\,m\,z \\ &= \left(\frac{3\,\rho^2}{2}\,(\delta+1)^2\,w\right)/\phi + w\,m\,(1 + 2\,z). \end{aligned}$$

On a $\phi \geqslant \frac{3\,\rho^2}{2\,m}\,(\delta+1)^2$, donc : $m \geqslant \left(\frac{3\,\rho^2}{2}\,(\delta+1)^2\right)/\phi$. Par conséquent, $w\,m \geqslant \left(\frac{3\,\rho^2}{2}\,(\delta+1)^2\,w\right)/\phi$. D'où, $\|S\|_\infty < \rho$, car $\rho \geqslant w\,m\,(2 + 2\,z)$.

Montrons enfin qu'il n'y a pas de collision. Il s'agit de montrer que, pour les mêmes entrées A et B, et deux polynômes de randomisation Y et Z distincts, l'algorithme 30 retourne deux sorties S_1 et S_2 distinctes. Nous procédons par l'absurde. Supposons donc que $S_1 = S_2$. On a :

$$\begin{aligned} S_1 &= \phi^{-1} \times (A\,B \bmod E + (A\,Z\,M + Q_1\,M + 2\,\phi\,Z\,M) \bmod E), \\ S_2 &= \phi^{-1} \times (A\,B \bmod E + (A\,Z\,M + Q_2\,M + 2\,\phi\,Z\,M) \bmod E). \end{aligned}$$

Ainsi, $S_1 = S_2$ implique que :

$$(A\,Z\,M + Q_1\,M + 2\,\phi\,Z\,M) \bmod E = (A\,Z\,M + Q_2\,M + 2\,\phi\,Z\,M) \bmod E.$$

Comme $\mathrm{pgcd}(E, M) = 1$ dans $\mathbb{Q}[X]$, on en déduit que :

$$A\,Z \bmod E + Q_1 + 2\,\phi\,Z = A\,Y \bmod E + Q_2 + 2\,\phi\,Y.$$

Donc, $Q_1 - Q_2 = 2\,\phi\,(Y - Z) + A\,(Y - Z) \bmod E$.
Soit $T = A\,(Y - Z) \bmod E$. On a : $\|T\|_\infty < 2\,z\,w\,(\delta+1)\,\rho$, car $\|Y - Z\|_\infty \leqslant 2\,z$.
Donc, $\|T\|_\infty < \phi$, car $\phi \geqslant \frac{3\,\rho^2}{2\,m}\,(\delta+1)^2$ et $\rho \geqslant w\,m\,(2 + 2\,z)$.
Puisque $Y \neq Z$, alors il existe i tel que : $|y_i - z_i| \geqslant 1$; et donc, $|2\,\phi\,(z_i - y_i)| \geqslant 2\,\phi$.

Par conséquent, $|2\,\phi\,(z_i - y_i) + t_i| > \phi$.
On a cependant $\|Q_2 - Q_1\|_\infty < \phi$, car $\|Q_1\|_\infty, \|Q_2\|_\infty < \phi$. Pour rappel, les coefficients des polynômes Q_1 et Q_2 sont toujours positifs.
Donc, $2\,\phi\,(Y - Z) + T \neq Q_1 - Q_2$.
Par conséquent, on ne peut pas avoir $S_1 = S_2$. Il n'y a donc pas de collision. □

Remarque 3.2.

- Pour que la multiplication randomisée soit consistante, il est indispensable que les nouvelles bornes sur ρ et ϕ (du théorème 3.2) soient prises en compte, lors du processus de génération présenté dans la section 2.4.
- Les bornes du théorème 3.2 sont plus larges que celles du théorème 3.1. Ainsi, pour à la fois randomiser la conversion et la multiplication dans l'AMNS, ce sont les bornes sur ρ et ϕ du théorème 3.2 qui doivent être utilisées.

3.3.4 Récapitulatif des bornes sur les paramètres ρ et ϕ

Le tableau 3.1 est un résumé des bornes sur les paramètres ρ et ϕ de l'AMNS, pour assurer la consistance des opérations arithmétiques dans l'AMNS, selon le type de randomisation souhaité. Notons que les autres paramètres de l'AMNS sont générés comme expliqué dans le chapitre précédent.
Pour la conversion randomisée, nous rappelons que :

- $\beta = 2^{\lceil \log_2(2w\|M\|_\infty)\rceil}$ et $P_i \equiv (\beta^i \phi^2)_{\mathcal{B}}$
- Si $z \geqslant 2$, la borne sur ϕ du théorème 3.1 devient $\phi \geqslant 2w\rho(\delta+1)^2$ (voir le second point de la remarque 3.1).

Le tableau 3.2 donne l'expression de la *redondance effective* ϖ de l'AMNS (cf. définition 1.8), selon le type de randomisation souhaité. La redondance donnée dans ce tableau est minimale, car elle est calculée avec la valeur minimale de ρ. Nous la désignons donc par ϖ_{min}.

Type de randomisation	Bornes sur ρ et ϕ
Aucune randomisation	$\rho \geqslant 2w\|M\|_\infty$, $\phi \geqslant 2w\rho(\delta+1)^2$
Randomisation de la conversion uniquement	$\rho \geqslant w\|M\|_\infty\,(2+z)$, $\phi \geqslant max(2w\rho(\delta+1)^2, 2n\beta + z)$
Randomisation de la conversion et de la multiplication	$\rho \geqslant w\,\|M\|_\infty(2+2z)$, $\phi \geqslant \frac{3\rho^2}{2\|M\|_\infty}(\delta+1)^2$

TABLE 3.1 – Récapitulatif des bornes sur les paramètres ρ et ϕ, selon le type de randomisation.

3.3.5 Coûts des opérations randomisées

Cette section porte sur le coût des algorithmes présentés plus haut. On suppose une architecture de k bits. Nous reprenons les éléments (notations et

Type de randomisation	Redondance effective minimale ϖ_{min}
Aucune randomisation	$\lceil \log_2(4w\|M\|_\infty - 1)\rceil - \lceil \log_2(\sqrt[n]{p})\rceil$
Randomisation de la conversion uniquement	$\lceil \log_2(2w\|M\|_\infty (2+z) - 1)\rceil - \lceil \log_2(\sqrt[n]{p})\rceil$
Randomisation de la conversion et de la multiplication	$\lceil \log_2(4w\,\|M\|_\infty (1+z) - 1)\rceil - \lceil \log_2(\sqrt[n]{p})\rceil$

TABLE 3.2 – Redondance effective minimale, selon le type de randomisation.

coûts) du tableau 2.1 du chapitre précédent. Pour rappel, on a pris $\rho < \phi = 2^k$ et $\lambda = \pm 2^i + \varepsilon 2^j$, avec $\varepsilon \in \{-1, 0, 1\}$. L'addition de deux mots de k bits est notée $\mathcal{A}$, le produit de deux mots de k bits est noté $\mathcal{M}$. Pour les opérations de décalage, $\mathcal{S}_l^i$ et $\mathcal{S}_r^i$ sont respectivement un décalage à gauche et un décalage à droite de i bits. De plus, si x et y sont chacun le produit de deux mots machines, alors l'opération $x+y$ coûte au maximum $3\mathcal{A}$. En outre, on note $\mathcal{R}$ le coût d'un appel à la fonction **RandPoly**.

L'opération $J \leftarrow Z \times M \bmod E$ (ligne 2 des algorithmes 29 et 30) peut être effectuée très efficacement, car les paramètres M et E sont fixés une fois l'AMNS généré (voir le quatrième point de la remarque 1.6). De plus, avec les bornes du théorème 3.2, on a $\|J\|_\infty < \rho < 2^k$. Donc, lors du calcul de J, chaque addition de produits partiels coûte $\mathcal{A}$.

Les tableaux 3.3 et 3.4 donnent respectivement les coûts de la conversion et de la multiplication randomisées. La première ligne de chacun de ces tableaux donne le coût total de la randomisation pour l'opération correspondante. Pour la conversion randomisée (algorithme 29), nous rappelons que β est une puissance de deux, donc la décomposition en base β (ligne 3 de l'algorithme) se fait très simplement. Nous ignorons son coût ici.

Coût total randomisation	$\mathcal{R} + n^2\mathcal{M} + (n^2+2n)\mathcal{A}$
Étape 1 (ligne 4)	$n^2\mathcal{M} + (3n^2-3n)\mathcal{A}$
Réduction interne	$2n^2\mathcal{M} + (4n^2-n)\mathcal{A} + n\mathcal{S}_r^k$
Coût total de la conversion randomisée	$\mathcal{R} + 4n^2\mathcal{M} + (8n^2-2n)\mathcal{A} + n\mathcal{S}_r^k$

TABLE 3.3 – Coût théorique de la conversion randomisée (alg. 29), avec $\phi = 2^k$.

3.3.6 Quelques exemples

Nous présentons dans cette section un exemple d'étude de redondance ainsi que des exemples de randomisation.

Coût total randomisation	$\mathcal{R} + n^2\mathcal{M} + (n^2+n)\mathcal{A} + n\mathcal{S}_l^1$
Multiplication polynomiale	$n^2\mathcal{M} + (3n^2 - 6n + 3)\mathcal{A}$
Réduction externe	$3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^i$ $+ \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$
Réduction interne	$2n^2\mathcal{M} + (4n^2 - n)\mathcal{A} + n\mathcal{S}_r^k$
Coût total de la multiplication randomisée	$\mathcal{R} + 4n^2\mathcal{M} + (8n^2 - 3n)\mathcal{A} + (n-1)\mathcal{S}_l^i + n\mathcal{S}_l^1$ $+ n\mathcal{S}_r^k + \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$

TABLE 3.4 – Coût théorique de la multiplication randomisée (alg. 30), avec $E(X) = X^n - \lambda$, où $\lambda = \pm 2^i + \varepsilon 2^j$, $\varepsilon \in \{-1, 0, 1\}$ et $\phi = 2^k$.

3.3.6.1 Étude de la redondance de quelques AMNS

Soit $p = 877986631880235280730309946384803882269167525514$ $844705484330134094228264005 53$, un nombre premier de 256 bits.

Avec p, nous avons généré plusieurs AMNS pour différents types de randomisation et différentes valeurs pour les paramètres δ et z, avec $n = 5$ ou 6. Ces AMNS ont été générés suivant la stratégie d'implémentation logicielle présentée dans le chapitre précédent. Les paramètres de ces AMNS sont fournis dans l'annexe C.1. Pour chacun de ces AMNS, le paramètre k, qui est ici la taille de mot machine, est égal 64. Donc, $\phi = 2^{64}$.

Le tableau 3.5 présente une étude de la redondance de ces AMNS. Ce tableau donne le type de randomisation pour lequel chacun de ces AMNS a été généré, leurs redondances effectives (selon la définition 1.8) ainsi que leurs redondances effectives (pratiques) pour l'implémentation logicielle que nous avons effectuée.

Pour rappel, la quantité Π (équation 1.1, chapitre 1) est le surcoût de la représentation d'un élément dans le système considéré. Le tableau 3.5 n'aborde pas les performances des AMNS considérés. Dans la section 2.5.3.1, nous avons vu que pour $n = 5$, 6 ou 7, avec des entiers de taille 256 bits, l'AMNS (sans randomisation) est plus efficace que les bibliothèques GNU MP et OpenSSL (avec la méthode par défaut). Les comparaisons avec la randomisation feront l'objet d'un travail futur.

Les bibliothèques GNU MP et OpenSSL utilisent respectivement les types mpz_t et BIGNUM pour représenter les grands entiers. Avec les résultats du tableau 2.5 du chapitre précédent, on obtient que, pour des entiers de taille 256 bits, ces structures engendrent les surcoûts de représentation suivants :

$$\Pi_{GNUMP} = 64 \quad \text{et} \quad \Pi_{OpenSSL} = 128.$$

Notons que ces surcoûts proviennent des informations annexes conservées dans ces structures, en plus du tableau principal qui contient la valeur de l'entier représenté (voir annexe B.3 pour plus de détails).

Pour rappel, le choix de z pour la randomisation implique un minimum de $(2z+1)^n$ représentations distinctes de chaque élément de $\mathbb{Z}/p\mathbb{Z}$ dans l'AMNS généré. Quelques conventions adoptées pour alléger le tableau 3.5 :

- ID : l'identifiant de l'AMNS. C'est le titre de la sous-section donnant les paramètres de cet AMNS dans l'annexe C.1.
- type rando. : le numéro correspondant au type de randomisation permis par l'AMNS. 0 signifie pas de randomisation (i.e. $z = 0$), 1 signifie la randomisation de la conversion uniquement et 2 signifie que l'AMNS correspondant permet de randomiser à la fois la conversion et la multiplication.
- ϖ_1 : la redondance effective, selon la définition 1.8. On a $\varpi = \lceil \log_2(2\rho - 1) \rceil - \lceil \log_2(\sqrt[n]{p}) \rceil$. Il est important de noter que ρ dépend de $\|M\|_\infty$. Ainsi, choisir un algorithme de réduction de réseau efficace lors de la génération du polynôme M (section 2.4.4) est essentiel pour minimiser la valeur de ρ (et donc celle de ϖ_1). Pour nos tests, nous avons utilisé l'algorithme LLL fourni dans la bibliothèque SageMath.
- ϖ_2 : la redondance effective, avec notre implémentation logicielle. La différence avec ϖ_1 vient du fait que, pour nos implémentations, nous avons alloué un mot machine pour stocker chaque coefficient d'un élément de l'AMNS. Ce choix simplifie l'implémentation, mais entraîne du gaspillage des ressources mémoires, d'où $\varpi_2 > \varpi_1$ dans ce tableau. La taille de mot machine ici est $k = 64$, donc $\varpi_2 = 64 - \lceil \log_2(\sqrt[n]{p}) \rceil$.
- Π_i : le surcoût de la représentation d'un élément dans l'AMNS correspondant, calculé avec ϖ_i. On a $\Pi_i = n\varpi_i$ (voir équation 1.1).

ID	n	type rando.	z	δ	ρ	ϖ_1	Π_1	ϖ_2	Π_2
AMNS 1	5	0	0	4	2^{55}	4	20	12	60
AMNS 2	5	1	43	0	2^{59}	8	40	12	60
AMNS 3	5	1	7	1	2^{57}	6	30	12	60
AMNS 4	5	2	1	0	2^{56}	5	25	12	60
AMNS 5	6	0	0	63	2^{47}	5	30	21	126
AMNS 6	6	1	45	15	2^{51}	9	54	21	126
AMNS 7	6	2	46	0	2^{52}	10	60	21	126
AMNS 8	6	2	10	4	2^{50}	8	48	21	126

TABLE 3.5 – Exemple d'étude de la redondance de quelques AMNS.

3.3.6.2 Exemples de randomisation

Nous donnons ici un autre exemple d'AMNS qui permet de randomiser la conversion et la multiplication. Des exemples d'éléments randomisés sont également présentés.

Les paramètres.

Cet AMNS correspond à un nombre premier p de taille 256 bits.

- $p = 670169794354315094013827593578171900597699829291616528443272745366662113702 99$
- $k = 64$
- $z = 16$
- $\delta = 3$
- $n = 6$
- $\lambda = -2$
- $\rho = 2^{52}$
- $\phi = 2^{64}$
- $\gamma = 605629196346040644240324997293578330830922421334767792674195797886931438375 65$
- $E(X) = X^6 + 2$
- $M(X) = 144638328348.X^5 + 162318870234.X^4 + 2790400703052.X^3 + 1521033975246.X^2 - 1617399112033.X + 4283244644745$
- $M'(Y) = 9139298470883188749.Y^5 + 104128656822937823 01.Y^4 + 13045121646837129301.Y^3 + 3109659280741919731.Y^2 + 12302819413495765981.Y + 14015151231881561941$

Notons que $z = 16$, donc le nombre minimum de représentations distinctes de chaque élément de $\mathbb{Z}/p\mathbb{Z}$ dans cet AMNS est $(2 \times 16 + 1)^6$.

Exemples de conversions randomisées.

Nous donnons ici deux représentations de deux éléments a et b de $\mathbb{Z}/p\mathbb{Z}$, ainsi que les polynômes de randomisation associés. On prend :

- $a = 374145266965280024168687594879007158791230203167984287016284980074389519091 41$
- $b = 277985067398787883510878248014520224584927346620402540694775565331180234997 9.$

Soient $Z1$, $Z2$, $Z3$ et $Z4$ les polynômes de randomisation définis comme suit :

- $Z1 = 4.X^5 - 11.X^4 - 5.X^3 + 15.X^2 - 12.X - 4$
- $Z2 = 9.X^5 + 2.X^4 - 11.X^3 - 13.X^2 + 10.X + 6$
- $Z3 = -15.X^5 - 15.X^4 - 16.X^3 - 4.X^2 + 11.X - 3$
- $Z4 = 13.X^5 - 12.X^4 + 5.X^3 + 6.X - 1.$

L'algorithme de conversion randomisée, avec a comme entrée, et $Z1$ et $Z2$ comme polynômes de randomisations, donne comme sorties :

- $Z1 \Rightarrow A1 = 66608918352434.X^5 - 47019756876330.X^4 - 70010663355356.X^3 + 59762200654687.X^2 - 5319604645365.X + 56053520120083$

- $Z2 \Rightarrow A2 = -15241071988574.X^5 + 37343487756669.X^4 + 5560013159642.X^3 - 110929727826867.X^2 - 4975398782821.X + 111723653053883.$

Le même algorithme, avec b comme entrée, et $Z3$ et $Z4$ comme polynômes de randomisations, donne comme sorties :

- $Z3 \Rightarrow B1 = -68281225705419.X^5 - 7536471908277.X^4 - 40184282652086.X^3 + 53138781251529.X^2 + 184646121559748.X + 74132793586224$
- $Z4 \Rightarrow B2 = 89378425609242.X^5 - 44295095588031.X^4 + 31312203266109.X^3 - 81910339041766.X^2 + 50100294103257.X + 47098432091510.$

Exemples de multiplications randomisées.

Nous présentons ici les sorties de l'algorithme de multiplication randomisée avec $A1$ et $B2$ en entrée, et deux polynômes de randomisation $Z5$ et $Z6$, tels que :

- $Z5 = -14.X^5 - 8.X^4 - 12.X^3 + 12.X^2 + 3.X + 2$
- $Z6 = 2.X^5 - 15.X^4 - 8.X^3 - 16.X^2 + 8.X + 1.$

Les sorties $S1$ et $S2$ obtenues sont :

- $Z5 \Rightarrow S1 = -58489740384399.X^5 + 35092857078006.X^4 - 103746007735160.X^3 + 263229912909337.X^2 + 191740344327628.X + 97998341351674$
- $Z6 \Rightarrow S2 = -41542579707397.X^5 - 104667147182684.X^4 + 24385303541772.X^3 - 172202776149549.X^2 + 232195025120088.X + 206174882189600.$

3.3.7 Quelques remarques sur la randomisation

Certains éléments du travail présenté dans ce chapitre méritent d'être approfondis. Nous les soulignons dans cette section.

3.3.7.1 À propos du polynôme de randomisation

Dans ce chapitre, nous ne traitons pas de comment est généré et sauvegardé le polynôme de randomisation Z.

Nous avons supposé que la fonction **RandPoly** est sûre. Bien qu'indépendante de l'AMNS, il est évident que la distribution et l'uniformité de la randomisation dans l'AMNS dépend de cette fonction. En effet, une fois le polynôme de randomisation fixé, la sortie de la multiplication randomisée (tout comme celle de la conversion randomisée) est aussi fixée, car chaque polynôme de randomisation permet d'atteindre une représentation bien précise ; l'aléa provient uniquement de la génération aléatoire du polynôme de randomisation. Ainsi, il est nécessaire que la sortie de **RandPoly** soit uniformément répartie sur l'ensemble $\mathcal{H}$ des polynômes de randomisation (équation 3.1) pour que la randomisation avec les algorithmes 29 et 30 soit aussi uniforme.

Dans les théorèmes 3.1 et 3.2, on peut remarquer que le paramètre z est très petit par rapport au paramètre ρ de l'AMNS. Par conséquent, $\#\mathcal{H}$ est négligeable par rapport à p, car $\rho \approx p^{\frac{1}{n}}$.

Dans le système de numération positionnel classique, pour randomiser les données en entrée ou les données intermédiaires lors de l'ECSM, on génère aléatoirement des entiers dans $\mathbb{Z}/p\mathbb{Z}$ (voir le survey [ACL19]). Puisque $\#\mathcal{H}$ est négligeable par rapport à p, l'aléa généré avec le système positionnel est par conséquent de plus grande taille que celui généré avec l'AMNS. La gestion de l'aléa (génération et sauvegarde) avec l'AMNS devrait donc être moins coûteuse. Notons que la randomisation avec les méthodes présentées plus haut affecte tous les coefficients de l'élément à randomiser.

Pour avoir une idée de l'impact de la génération du polynôme de randomisation Z sur les performances de la randomisation, nous avons effectué trois types de multiplication avec l'AMNS dont les paramètres sont donnés dans l'annexe C.1.8 : des multiplications sans randomisation (**T0**), des multiplications avec un polynôme de randomisation fixé (**T1**) et enfin des multiplications avec un polynôme de randomisation généré aléatoirement à chaque multiplication (**T2**). Pour chacun de ces types, nous avons calculé une moyenne de 2^{25} multiplications. Les implémentations ont été faites en langage C, avec la machine utilisée pour les tests présentés dans la section 2.5.3.

Le tableau 3.6 donne les ratios entre les temps d'exécution moyens des différents types de multiplication. Dans ce tableau, on peut observer qu'une multiplication de type **T2** est 59% plus lente qu'une multiplication de type **T1**. Il est donc évident qu'une implémentation efficace de la fonction **RandPoly** est indispensable pour une randomisation efficace.

type de ratio	**T1/T0**	**T2/T0**	**T2/T1**
ratio	1.49	2.37	1.59

TABLE 3.6 – Exemple de ratios entre les temps d'exécution moyens de différents types de multiplication.

3.3.7.2 Existence d'éventuelles corrélations

Dans ce chapitre, nous n'avons pas étudié les éventuelles corrélations qui pourraient être exploitées, avec les méthodes de randomisation présentées plus haut. Nous présentons ici certains points qui devraient être étudiés.

Le premier point important est de savoir si l'on peut trouver des corrélations exploitables entre un entier de $\mathbb{Z}/p\mathbb{Z}$ et une de ses représentations dans l'AMNS.

Le second point important est l'existence d'éventuelles corrélations entre les différentes représentations dans l'AMNS d'un entier de $\mathbb{Z}/p\mathbb{Z}$. Ce point n'est pas abordé dans ce chapitre. On peut cependant avancer certains éléments pour son étude. Premièrement, après la génération du polynôme de randomisation Z (ligne 1 des algorithmes 29 et 30), on calcule le polynôme $J = Z \times M \bmod E$. C'est ce dernier qui est par la suite utilisé pour la randomisation. Ce calcul

de J brise l'indépendance des coefficients de Z. Le second élément est que le polynôme J (et donc le polynôme Z) intervient dans la réduction interne (l'appel à **RedCoeff**). Le polynôme de randomisation Z intervient donc dans les opérations modulo ϕ qui sont faites dans la fonction **RedCoeff**. Il reste à savoir si ces deux éléments, en plus de la propagation de l'aléa durant le processus, suffisent pour supprimer toutes corrélations exploitables.

3.4 Randomisation de la multiplication scalaire sur les courbes elliptiques

Soit $E(\mathbb{F}_p)$ une courbe elliptique définie sur $\mathbb{Z}/p\mathbb{Z}$. Cette section traite de la randomisation du calcul de kP, avec $k \in \mathbb{N}$ et $P \in E(\mathbb{F}_p)$. Comme expliqué dans l'introduction de ce chapitre, pour protéger cette opération contre les attaques de type DPA, il est indispensable de randomiser les entrées (k et P), les valeurs intermédiaires et les opérations arithmétiques. Il est également nécessaire de randomiser la séquence des instructions ainsi que les adresses mémoires.

Avec l'AMNS, notre objectif ici est de randomiser les coordonnées du point P, les opérations arithmétiques, les valeurs intermédiaires ainsi que les coordonnées de la sortie kP, pendant la multiplication scalaire.

Notons que l'utilisation de l'AMNS pour effectuer la multiplication scalaire sous-entend que les coordonnées de P ainsi que celles des points intermédiaires lors de cette opération seront représentées dans l'AMNS. Le scalaire k n'a pas à être converti dans l'AMNS. Il est utilisé de la même manière qu'avec les systèmes de numération positionnels.

La randomisation de la séquence des instructions et des adresses mémoires peut être faite avec la randomisation du scalaire k, indépendamment des autres randomisations. Nous renvoyons le lecteur vers les sections 5 et 6 du récent survey [ACL19] pour plus de détails.

3.4.1 Randomisation du point de base

Nous nous intéressons ici à la randomisation du point P au début de la multiplication scalaire. Avec l'AMNS, la randomisation de P peut être faite de quatre manières. Dans l'ordre ci-dessous, les deux premières randomisent les coordonnées de P sans modifier leurs valeurs en γ modulo p. Les deux autres modifient les coordonnées de P de façon aléatoire. Remarquons qu'il est possible de combiner ces possibilités de randomisation.

La première possibilité, la plus simple, est la méthode de randomisation avec plusieurs AMNS que nous avons expliquée dans la section 3.2. Le principe est de générer (pré-calculer) un ensemble Ω d'AMNS pour le module associé à la courbe. Ici, les AMNS peuvent être générés en utilisant les bornes indiquées dans le chapitre 2. Puis, au début de la multiplication scalaire, on sélectionnera aléatoirement un AMNS dans cet ensemble Ω, pour ensuite convertir, avec l'algorithme 22 du chapitre 2, les coordonnées du point P dans cet AMNS.

La seconde possibilité repose sur la méthode de conversion randomisée présentée dans la section 3.3.2. L'idée est de générer l'AMNS en respectant les bornes indiquées dans le théorème 3.1 (ou celles du théorème 3.2 si l'on souhaite également randomiser les opérations arithmétiques). Ensuite, on randomisera les coordonnées de P en utilisant l'algorithme de conversion randomisée (algorithme 29).

La troisième possibilité est le masquage du point P (point blinding) [Cor99]. Il s'agit d'ajouter à P un point aléatoire R au début de la multiplication scalaire. Ensuite, à la fin de l'opération, on effectuera une soustraction pour obtenir le bon résultat. Ici, l'AMNS peut être généré en utilisant les bornes indiquées dans le chapitre 2. Notons que les coordonnées du point R peuvent être générées directement dans l'AMNS.

La dernière possibilité est basée sur une contre-mesure de Joye et Tymen pour la protection de la multiplication scalaire contre certaines attaques de type DPA [JT01]. L'idée principale est de changer les coordonnées affines (x, y) de P par $(u^{-2}x, u^{-3}y)$, avec $u \in (\mathbb{Z}/p\mathbb{Z}) \setminus \{0\}$ choisi aléatoirement. Avec l'AMNS, cette contre-mesure peut être facilement intégrée dans les étapes de conversion. Les algorithmes de conversion, présentés dans le chapitre précédent, sont modifiés pour prendre un argument supplémentaire utilisé pour changer les représentations de x et y. Désignons par `MultMod` et `ConvToAMNS` les algorithmes 19 et 22 respectivement. Pour rappel, les algorithmes 19 et 22 décrivent respectivement la multiplication modulaire dans l'AMNS et la conversion du binaire vers l'AMNS. Les algorithmes 31 et 32 correspondent à la contre-mesure suggérée dans [JT01], avec l'AMNS. Pour cette contre-mesure, l'AMNS peut être généré en utilisant les bornes indiquées dans le chapitre 2. Avant le calcul de $k\,P$, les procédures DPA_Conv_to_AMNS(x, u^{-2}) et DPA_Conv_to_AMNS(y, u^{-3}) sont appelées. Une fois le calcul effectué, un appel à DPA_Conv_to_BIN(x, u^2) et DPA_Conv_to_BIN(y, u^3) permet de retrouver les coordonnées de $k\,P$. Notons que cette contre-mesure de Joye et Tymen est généralisée dans [TJ10].

Algorithme 31 *DPA_Conv_to_AMNS(a,β)*

Entrée(s) : $a \in \mathbb{Z}/p\mathbb{Z}$, $\mathcal{B} = (p,\, n,\, \gamma,\, \rho, E)$ et $\beta \in (\mathbb{Z}/p\mathbb{Z}) \setminus \{0\}$
Sortie : $A \equiv (a\,\beta\,\phi)_{\mathcal{B}}$

1: $C \leftarrow$ `ConvToAMNS`(a) $\#\ C \equiv (a\,\phi)_{\mathcal{B}}$
2: $D \leftarrow$ `ConvToAMNS`(β) $\#\ D \equiv (\beta\,\phi)_{\mathcal{B}}$
3: $A \leftarrow$ `MultMod`(C, D)
4: retourner A

3.4.2 Randomisation des opérations arithmétiques et des valeurs intermédiaires

La randomisation des opérations arithmétiques et des valeurs intermédiaires se fera grâce à la multiplication randomisée présentée dans la section 3.3.3. Ainsi, il est indispensable que l'AMNS soit généré en utilisant les bornes indiquées dans le théorème 3.2. Pour rappel, ces bornes sur ρ et ϕ incluent celles du

Algorithme 32 *DPA_Conv_to_BIN(A,β)*

Entrée(s) : $A \in \mathcal{B}$, $\mathcal{B} = (p, n, \gamma, \rho, E)$ et $\beta \in (\mathbb{Z}/p\mathbb{Z}) \setminus \{0\}$
Sortie : $a = A(\gamma)\beta\phi^{-1} \bmod p$
1: $A \leftarrow \textbf{RedCoeff}(A)$
2: $a \leftarrow a_{n-1}$
3: **for** $i = n-2$ **to** 0 **do**
4: $\quad a \leftarrow (a\gamma + a_i) \bmod p$
5: **end for**
6: $a \leftarrow a\,\beta \bmod p$
7: retourner a

théorème 3.1 ainsi que celles du chapitre 2. Par conséquent, toutes les options de randomisation du point P, présentées dans la section 3.4.1, restent applicables.

Nous présentons ici deux stratégies pour randomiser les opérations arithmétiques ainsi que valeurs intermédiaires, selon la fréquence de génération du polynôme de randomisation Z.

La première stratégie consiste à utiliser l'algorithme de multiplication modulaire randomisée (algorithme 30) pour tous les carrés et multiplications modulaires au cours de la multiplication scalaire. Ce faisant, on est certain de randomiser les opérations arithmétiques ainsi que les valeurs intermédiaires à toutes les étapes de l'ECSM. Cependant, comme nous l'avons fait remarquer dans la section 3.3.7.1, une implémentation sûre et efficace de la fonction **RandPoly** est indispensable pour une randomisation sûre et efficace de la multiplication. En outre, on peut observer dans le tableau 3.6 (ratio **T2**/**T1**) que le coût de cette fonction peut être assez important.

Afin de réduire le nombre d'appels à la fonction **RandPoly**, nous suggérons une seconde stratégie dont l'idée principale est d'utiliser un même polynôme de randomisation Z pour plusieurs carrés et multiplications modulaires, avant d'en générer un autre. Par exemple, le même polynôme Z peut être utilisé pour une (ou plusieurs) addition(s) ou doublement(s) de points, avant qu'un autre appel à la fonction **RandPoly** ne soit effectué. Il est évident que cette stratégie conduit à une randomisation moins importante que la première. Cependant, en plus d'un nombre moins élevé d'appels à la fonction **RandPoly**, cette seconde stratégie réduit grandement le coût total de la randomisation lors de la multiplication randomisée. En effet, la valeur du polynôme Z définie entièrement celle du polynôme J (ligne 2, algorithme 30), car les polynômes M et E sont fixés lors de la génération de l'AMNS. Donc, une fois Z fixé, on peut calculer le polynôme $J = Z \times M \bmod E$, qui sera ensuite utilisé pour toutes les randomisations, avant la génération d'un autre polynôme Z. Dans ce cas, l'algorithme de multiplication randomisée peut être réécrit en prenant le polynôme J comme paramètre supplémentaire. L'algorithme 33 correspond à cette modification. Pour cet algorithme, les bornes sur les paramètres ρ et ϕ restent les mêmes que celles de l'algorithme 30 (i.e. celles du théorème 3.2).

Le tableau 3.7 donne le coût de l'algorithme 33 en reprenant les éléments de la section 3.3.5. On peut remarquer dans ce tableau (à la ligne 1) que le

Algorithme 33 *Multiplication randomisée dans l'AMNS, avec J en paramètre*

Entrée(s) : $A, B \in \mathbb{Z}_{n-1}[X]$, $\mathcal{B} = (p, n, \gamma, \rho, E)$, $J = Z \times M \bmod E$

Sortie : $S \in \mathbb{Z}_{n-1}[X]$, tel que $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod{p}$

1: $B' \leftarrow B + J$
2: $C \leftarrow A \times B' \bmod E$
3: $R \leftarrow \textbf{RedCoeff}(C)$
4: $S \leftarrow R + 2 \times J$
5: retourner S

coût total de la randomisation est nettement inférieur à celui du tableau 3.4, ce qui conduit à un coût total plus intéressant pour la multiplication randomisée. Notons que ce coût est très proche de celui de la multiplication non randomisée.

Coût total randomisation	$2n\mathcal{A} + n\mathcal{S}_l^1$
Multiplication polynomiale	$n^2\mathcal{M} + (3n^2 - 6n + 3)\mathcal{A}$
Réduction externe	$3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^i$ $+ \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$
Réduction interne	$2n^2\mathcal{M} + (4n^2 - n)\mathcal{A} + n\mathcal{S}_r^k$
Coût total de la multiplication randomisée	$3n^2\mathcal{M} + (7n^2 - 2n)\mathcal{A} + (n-1)\mathcal{S}_l^i + n\mathcal{S}_l^1$ $+ n\mathcal{S}_r^k + \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$

TABLE 3.7 – Coût théorique de la multiplication randomisée (alg. 33), avec le polynôme $J = Z \times M \bmod E$ en paramètre, pour $E(X) = X^n - \lambda$, où $\lambda = \pm 2^i + \varepsilon 2^j$, $\varepsilon \in \{-1, 0, 1\}$ et $\phi = 2^k$.

3.4.3 Comparaison de différentes stratégies de randomisation de l'ECSM sur un exemple de courbe

Le tableau 3.6 nous a permis de souligner l'importance d'une implémentation efficace de la fonction **RandPoly** pour la multiplication modulaire randomisée. Nous nous intéressons ici à l'impact de cette fonction dans une application plus concrète. L'objectif est d'étudier l'impact de la randomisation sur le temps d'exécution de la multiplication scalaire, selon différentes stratégies de randomisation. Nous avons utilisé une courbe de Weierstrass courte $E(\mathbb{Z}/p\mathbb{Z})$, telle que $E(\mathbb{Z}/p\mathbb{Z}) : y^2 = x^3 + 5$, avec $p = 2^{256} - 1539$. C'est une courbe d'ordre h, où h un nombre premier de taille 256 bits. Les multiplications scalaires ont été faites avec un point de base $P \in E(\mathbb{Z}/p\mathbb{Z})$ et un scalaire $k < h$ générés aléatoirement.

Chaque calcul de kP a été effectué avec l'algorithme 39, qui fait appel aux méthodes `DBLU`, `ZADDU` et `ZADDC` (section 4.1.3.3). Pour les méthodes `ZADDU` et `ZADDC`, prendre $\delta = 3$ suffit. Pour la méthode `DBLU`, on doit prendre $\delta = 12$. Cependant, en effectuant une réduction **exacte** de coefficients (cf. algorithme 26) sur les résultats des lignes 4 et 5 de cette méthode, on réduit δ à 3. Ce qui

est un choix judicieux ici, car la méthode `DBLU` n'est appelée qu'une seule fois au début de l'algorithme 39. De plus, cela impose une borne inférieure moins grande sur ϕ. Avec ce choix, prendre $\delta = 5$ suffit pour assurer la consistance de la séquence d'appels aux méthodes `ZADDU`, `ZADDC` et `DBLU` dans l'algorithme 39.

Les stratégies de randomisation considérées sont :

- **S0 :** Multiplication scalaire sans randomisation.
- **S1 :** Multiplication scalaire avec randomisation uniquement de la représentation du point P, i.e. randomisation de la conversion uniquement.
- **S2 :** Multiplication scalaire avec randomisation de la représentation du point P ainsi que la randomisation des multiplications modulaires avec un polynôme de randomisation Z **fixé** durant toute la multiplication scalaire. Cependant, le calcul du polynôme $J = Z \times M \bmod E$ sera fait pendant chaque multiplication modulaire. Cette stratégie servira uniquement à relever l'impact de la fonction **RandPoly** sur les performances de l'ECSM. Elle n'a autrement aucun intérêt.
- **S3 :** Multiplication scalaire avec randomisation de la représentation du point P ainsi que la randomisation des multiplications modulaires. Pour cette stratégie, nous générons aléatoirement un polynôme de randomisation Z à **chaque tour** de boucle, puis calculons le polynôme $J = Z \times M \bmod E$. Ensuite, pour le tour de boucle concerné, ce polynôme J sera utilisé pour randomiser les multiplications, avec l'algorithme 33. La stratégie **S3** correspond à la seconde stratégie présentée dans la section 3.4.2, avec ici Z généré à chaque tour de boucle.
- **S4 :** Multiplication scalaire avec randomisation de la représentation du point P ainsi que la randomisation des multiplications modulaires, avec un polynôme de randomisation Z généré aléatoirement à chaque multiplication modulaire. En d'autres termes, c'est l'algorithme 30 qui sera utilisé pour toutes les multiplications modulaires. La stratégie **S4** est la première stratégie présentée dans la section 3.4.2.

Afin de souligner l'importance du paramètre δ pour la multiplication scalaire, nous distinguons, en plus des stratégies de randomisation mentionnées plus haut, le cas $\delta = 0$ et le cas $\delta = 5$ pour le calcul de kP. Pour rappel, $\delta = 0$ signifie qu'une réduction **exacte** de coefficients est toujours effectuée après une (ou plusieurs) addition(s) avant d'effectuer une multiplication et $\delta = 5$ signifie qu'on a la possibilité d'effectuer 5 additions sans avoir à effectuer de réduction de coefficients avant une multiplication.

Les implémentations ont été faites en langage C, avec la machine utilisée pour les tests présentés dans la section 2.5.3. Les paramètres des AMNS utilisés ici sont donnés dans l'annexe C.2. Selon la valeur de δ, nous avons, pour chacune des stratégies de randomisation, calculé une moyenne de 2^{10} multiplications scalaires.

Le tableau 3.8 donne les ratios entre les différentes stratégies de randomisation, selon la valeur de δ. Les ratios **S3**/**S2** et **S2**/**S1** n'apparaissent pas, car ils n'ont aucune signification particulière. Dans ce tableau, on peut observer que,

pour chaque type de ratio, le ratio pour $\delta = 0$ est plus petit que celui pour $\delta = 5$. La raison est qu'il est nécessaire, lors du calcul de kP, d'effectuer un grand nombre de réductions **exactes** de coefficients dans chacune des méthodes `ZADDU`, `ZADDC` et `DBLU`, lorsque $\delta = 0$. La réduction exacte de coefficients, qui n'est pas randomisée, est plus coûteuse qu'une multiplication modulaire non randomisée dans l'AMNS (voir section 2.3.5). Ainsi, ces nombreuses réductions exactes de coefficients "cachent" le surcoût de la randomisation. Lorsque $\delta = 5$, seulement deux réductions exactes de coefficients sont effectuées dans tout le calcul de kP afin de réduire la valeur de δ à 3 pour la méthode `DBLU`, comme expliqué plus haut. Ainsi, lorsque $\delta = 5$, le surcoût de la randomisation est plus visible dans les ratios.

Considérons donc le cas $\delta = 5$ qui est le mieux adapté pour relever l'effet des différentes stratégies de randomisation sur les performances de la multiplication scalaire. Dans le tableau 3.8, deux ratios permettent d'estimer l'impact de la fonction **RandPoly**. Le premier est le ratio **S2/S0** qui correspond au ratio entre le temps d'exécution de l'ECSM randomisée **sans** le coût des appels à **RandPoly** (car Z fixé) et celui de l'ECSM non randomisée. Le second est le ratio **S4/S2** qui donne l'impact des appels à **RandPoly** sur les performances de l'ECSM randomisée selon la première stratégie présentée dans la section 3.4.2. Il est intéressant de noter que les ratios **S2/S0** et **S4/S2** sont respectivement assez proches des ratios **T1/T0** et **T2/T1** du tableau 3.6 ; ce qui est assez logique. Car, une fois la conversion du point de base P effectuée, les opérations arithmétiques les plus coûteuses lors du calcul de kP (i.e. le carré et la multiplication modulaires) sont celles qui sont randomisées ici.

Les performances globales relatives des différentes stratégies de randomisation peuvent être observées avec les autres ratios (ici également avec $\delta = 5$). Le ratio **S1/S0** compare l'ECSM avec randomisation de la conversion uniquement à l'ECSM non randomisée. Le ratio **S4/S3** est une comparaison des deux stratégies présentées dans la section 3.4.2. On peut observer que la première (qui conduit à une randomisation plus importante) est 88% plus lente que la seconde. Notons que les appels à la fonction **RandPoly** participent beaucoup à cette différence de performances. Une implémentation efficace de cette fonction devrait réduire l'écart. Le ratio **S4/S0** compare l'ECSM randomisée selon la première stratégie de la section 3.4.2 à l'ECSM non randomisée. On peut noter que ce ratio est assez proche du ratio **T2/T0** du tableau 3.6 ; ce qui est assez logique, pour les raisons mentionnées dans le paragraphe précédent. Le ratio **S3/S0** compare l'ECSM randomisée selon la seconde stratégie de la section 3.4.2 à l'ECSM non randomisée. Pour chaque calcul de kP à effectuer, nous avons généré aléatoirement un scalaire k de taille 256 bits. Ainsi, pour la stratégie **S3**, 258 appels à la fonction **RandPoly** ont été faits (2 pour la conversion randomisée des coordonnées de P et 256 pour la multiplication scalaire). Donc, 258 calculs du polynôme $J = Z \times M \bmod E$ ont également été faits. Malgré ces opérations, on peut remarquer avec le ratio **S3/S0** que l'ECSM selon la stratégie **S3** n'est pas exagérément plus lente que l'ECSM sans randomisation. Cela vient du fait que l'algorithme 33 (utilisé pour la stratégie **S3**) a un coût très proche de celui de l'algorithme 19 (utilisé pour la stratégie **S0**) ; voir tableaux

2.1 et 3.7. Les ratios **S3/S1** et **S4/S1** s'interprètent facilement à partir des ratios **S3/S0** et **S4/S0** respectivement. Ils sont par ailleurs assez proches.

type de ratio	**S1/S0**	**S2/S0**	**S3/S0**	**S3/S1**
ratio (pour $\delta = 0$)	1.03	1.29	1.19	1.16
ratio (pour $\delta = 5$)	1.04	1.54	1.30	1.24

type de ratio	**S4/S0**	**S4/S1**	**S4/S2**	**S4/S3**
ratio (pour $\delta = 0$)	1.61	1.56	1.25	1.35
ratio (pour $\delta = 5$)	2.43	2.33	1.58	1.88

TABLE 3.8 – Exemple de ratios entre les temps d'exécution moyens des différentes stratégies de randomisation, selon la valeur de δ.

Rappelons que l'efficacité de l'AMNS dépend en grande partie de la valeur du paramètre n. Plus n est petit, mieux c'est (du moins théoriquement, voir tableaux 2.1, 3.3 et 3.4). Dans l'annexe C.2, on peut observer que pour $\delta = 0$, les AMNS générés ont le paramètre n égal à 5, alors que pour $\delta = 5$, les AMNS générés ont le paramètre n égal à 6. Cependant, le tableau 3.9 montre que, pour une même stratégie de randomisation, la multiplication scalaire avec l'AMNS ayant $\delta = 5$ est beaucoup plus efficace que celle avec l'AMNS ayant $\delta = 0$. Cela vient des nombreuses réductions exactes de coefficients effectuées lorsque $\delta = 0$, comme expliqué plus haut.

type de ratio	$\frac{\boldsymbol{S0}, \text{ avec } \delta=5}{\boldsymbol{S0}, \text{ avec } \delta=0}$	$\frac{\boldsymbol{S1}, \text{ avec } \delta=5}{\boldsymbol{S1}, \text{ avec } \delta=0}$	$\frac{\boldsymbol{S2}, \text{ avec } \delta=5}{\boldsymbol{S2}, \text{ avec } \delta=0}$	$\frac{\boldsymbol{S3}, \text{ avec } \delta=5}{\boldsymbol{S3}, \text{ avec } \delta=0}$	$\frac{\boldsymbol{S4}, \text{ avec } \delta=5}{\boldsymbol{S4}, \text{ avec } \delta=0}$
ratio	0.44	0.45	0.52	0.48	0.67

TABLE 3.9 – Exemple de ratios entre les temps d'exécution moyens des multiplications scalaires pour $\delta = 0$ et $\delta = 5$, par stratégie de randomisation.

3.4.4 Spécificité de l'AMNS pour la protection de l'ECSM

Dans cette section, nous soulignons quelques avantages propres à l'AMNS pour la protection de la multiplication scalaire contre les attaques par canaux auxiliaires. Ces avantages viennent d'une part de l'absence de branchement conditionnel dans les algorithmes pour les opérations arithmétiques dans l'AMNS, et d'autre part du fait qu'avec la randomisation, chaque élément de $\mathbb{Z}/p\mathbb{Z}$ possède au moins $(2z+1)^n$ représentations différentes dans l'AMNS. Pour rappel, le paramètre z est choisi lors du processus de génération de l'AMNS.

Dans [ACL19], Abarzúa et al. font un état de l'art des attaques et contremesures pour la multiplication scalaire. Nous relevons ici certaines de ces attaques contre lesquelles l'AMNS permet de se protéger très simplement.

Dans la section 4.1.2 [ACL19], les auteurs présentent l'attaque de Walter [Wal04]. Cette attaque utilise la soustraction conditionnelle qui se fait à la fin de la multiplication modulaire de Montgomery. Comme mentionné plus haut, il

n'y a pas de branchement conditionnel dans les algorithmes pour les opérations arithmétiques dans l'AMNS. Ainsi, l'utilisation de l'AMNS protège naturellement contre ce genre d'attaque.

Dans la section 4.1.3 [ACL19], une attaque due à Amiel et al. [AFT+08] est présentée. Cette attaque est basée sur la différentiation du carré et de la multiplication modulaires. Avec la randomisation dans l'AMNS, on peut très simplement rendre le carré modulaire et la multiplication modulaire indifférenciables. Pour cela, il suffit d'utiliser l'un des algorithmes 30 ou 33 (de multiplication randomisée) pour à la fois les carrés et multiplications modulaires. Puisque ces algorithmes commencent par randomiser un des opérandes, on est certain de rendre le carré et la multiplication modulaires indifférenciables.

Dans la section 4.1.5 [ACL19], il est question de l'attaque HCCA (Horizontal Collision Correlation Analysis) [BJP+15]. Cette attaque peut être mise en oeuvre si l'attaquant peut détecter quand deux multiplications modulaires, au cours de la multiplication scalaire, ont au moins un opérande en commun ; ceci peut arriver par exemple lors du doublement de point avec certaines formules. Pour se prémunir contre cette attaque en utilisant l'AMNS, il suffit de randomiser les opérandes appropriés. Dans l'AMNS, cela peut se faire très simplement avec le polynôme $\Theta \equiv (\phi)_{\mathcal{B}}$, une représentation de l'entier 1 dans le domaine de Montgomery. En effet, si $A \in \mathcal{B}$, pour obtenir une autre représentation $B \in \mathcal{B}$ telle que $B \overset{\mathcal{B}}{\equiv} A$, il suffit d'utiliser l'algorithme 30 avec A et Θ comme entrées. Notons que le (pré-)calcul de Θ peut se faire simplement avec l'opération : $\Theta = \mathbf{RedCoeff}(P_0)$, car le polynôme de conversion P_0 est tel que $P_0 \equiv (\phi^2)_{\mathcal{B}}$.

Dans la section 4.1.6 [ACL19], les auteurs abordent l'attaque HSVA (Horizontal Same Value Attack) [DGH+16] qui permet de différencier l'addition et le doublement de point afin de déterminer les bits du scalaire k. Cette attaque peut être mise en oeuvre lorsque certaines valeurs intermédiaires sont identiques au cours de l'opération de doublement. Une solution simple pour se prémunir contre cette attaque est d'utiliser l'algorithme 30 (de multiplication randomisée) pour à la fois les carrés et multiplications modulaires. Pour les additions et soustractions (modulaires) au cours du doublement ou addition de points, on pourra (quand nécessaire) randomiser les opérandes appropriés, comme expliqué dans le paragraphe précédant.

Enfin, nous terminons par l'attaque RPA (Refined Power Analysis) de Goubin [Gou03] dont il est question dans la section 4.2.2 de [ACL19]. Cette attaque peut être menée sur les courbes elliptiques qui possèdent au moins un point P tel que $P = (x, 0)$ ou $P = (0, y)$ dans le système de coordonnées affine. L'objectif de cette attaque est d'utiliser un tel P sur une telle courbe pour effectuer la multiplication scalaire afin de déterminer la valeur du scalaire k. Dans le système de numération positionnel classique, la simple randomisation des coordonnées du point P et du scalaire k ne suffit par pour protéger la multiplication scalaire contre cette attaque. Dans la section 7.1 de [ACL19], un ensemble de contremesures est fourni pour cette attaque. Dans la section 5.10 du même article, les auteurs soulignent la proposition de Smart et al. [SOP08] qui est d'utiliser la redondance pour se prémunir contre l'attaque RPA. La randomisation dans l'AMNS utilise la même idée et protège naturellement la multiplication scalaire

de cette attaque. En effet, avec la randomisation, il y a au moins $(2z+1)^n$ représentations distinctes de 0 et ces représentations n'ont pas de formes spéciales. Donc, cette attaque demanderait à l'attaquant de considérer toutes ces représentations de 0. Ce qui conduirait à une quantité déraisonnable de traces à exploiter pour mener à bien cette attaque, si z est choisi suffisamment grand. Par conséquent, il devient impossible d'effectuer cette attaque quel que soit le type de courbe.

3.5 Conclusion et perspectives

Dans ce chapitre, nous avons vu comment randomiser les représentations des éléments avec l'AMNS. Nous avons présenté des méthodes pour randomiser la conversion ainsi que la multiplication modulaire dans l'AMNS, en utilisant la méthode Montgomery-like pour la réduction interne. À ces méthodes, nous avons associé de nouvelles bornes aux paramètres ρ et ϕ de l'AMNS, afin de garantir la consistance des opérations arithmétiques dans ce système lorsque ces méthodes de randomisation sont utilisées.

Nous avons également vu comment randomiser le point de base, les opérations arithmétiques ainsi que les valeurs intermédiaires, lors de la multiplication scalaire sur une courbe elliptique. En outre, on a pu voir qu'avec la randomisation, l'AMNS permet de protéger assez naturellement la multiplication scalaire contre certaines attaques par canaux auxiliaires. Enfin, nous avons comparé différentes stratégies de randomisation de la multiplication scalaire sur un exemple de courbe, en tenant compte du paramètre δ de l'AMNS.

Comme mentionné dans la section 3.3.7, certains points restent à éclaircir pour garantir la sûreté des méthodes de randomisation présentées dans ce chapitre. On a par exemple l'étude d'éventuelles corrélations exploitables entre les différentes représentations dans l'AMNS d'un élément de $\mathbb{Z}/p\mathbb{Z}$. Aussi, nous avons vu que l'efficacité et la sûreté de la randomisation dépendent de la fonction **RandPoly**. Une implémentation efficace de cette fonction, qui génère aléatoirement un polynôme de randomisation selon la distribution appropriée, est donc indispensable.

Deuxième partie

Arithmétique des courbes elliptiques

Chapitre 4

Arithmétique des courbes elliptiques, un état de l'art

Sommaire

Comme expliqué en introduction, pour un même niveau de sécurité, les protocoles cryptographiques basés sur les courbes elliptiques (protocoles ECC) requièrent des entiers de tailles plus petites que ce qui est requis pour les cryptosystèmes tels que RSA et ElGamal classique. Cela vient du fait qu'il existe des algorithmes de complexité sous-exponentielle pour la résolution de la factorisation ou du logarithme discret classique, alors qu'un tel algorithme n'a pas encore été trouvé pour résoudre le logarithme discret sur les courbes elliptiques (de façon générale). Ainsi, pour un même niveau de sécurité, les protocoles ECC ont de meilleures performances et nécessitent moins de ressource mémoire. Ils sont donc particulièrement bien adaptés pour les environnements à ressources (puissance et mémoire) réduites, tels que les cartes à puce.

Dans ce chapitre, nous effectuons un état de l'art de l'arithmétique des courbes elliptiques. Nous présentons un ensemble de méthodes pour effectuer la multiplication scalaire sur ces courbes (ECSM). Nous nous intéressons particulièrement aux courbes munies d'un endomorphisme efficace, ainsi qu'à l'utilisation des chaînes d'additions euclidiennes pour effectuer la multiplication scalaire.

Ce chapitre est organisé comme suit. Nous commençons par des généralités sur les courbes elliptiques. Ensuite, nous abordons l'ECSM de façon générale.

Puis, nous nous intéressons à l'utilisation des chaînes d'additions euclidiennes pour effectuer la multiplication scalaire. Nous terminons par la multiplication scalaire avec les courbes munies d'un endomorphisme efficace.

4.1 Généralités

Cette section présente les courbes elliptiques, certains systèmes de cordonnées ainsi que les opérations d'addition et de doublement.

4.1.1 Les bases

Définition 4.1. Une courbe elliptique $E(\mathbb{K})$ est une courbe définie sur un corps $\mathbb{K}$, par l'équation de Weierstrass suivante :

$$E(\mathbb{K}) : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{4.1}$$

avec $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ et $\Delta \neq 0$, où Δ est le *discriminant* de la courbe et est défini comme suit :

$$\begin{aligned}
\Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2.
\end{aligned}$$

On dit que $\mathbb{K}$ est le *corps de base* de la courbe $E(\mathbb{K})$. La condition $\Delta \neq 0$ permet de garantir que la courbe est *lisse*. En d'autres termes, elle possède une seule tangente en chacun de ses points, et n'a ni point double, ni point de rebroussement. Cette condition est indispensable pour garantir la consistance des formules de composition de points que nous verrons plus loin.

Les points de la courbe sont les couples (x, y) solutions de l'équation 4.1. Pour définir une structure de groupe sur l'ensemble des points de la courbe, il est nécessaire d'ajouter à cet ensemble un point particulier appelé le *point à l'infini*, noté $\mathcal{O}$. C'est un point abstrait qu'il est nécessaire de prendre en compte pour définir les opérations arithmétiques sur la courbe. Il joue le rôle d'élément neutre pour ces opérations. On peut l'imaginer comme le point à l'intersection de toutes les droites verticales. Nous y reviendrons dans la section 4.1.3, où seront présentés des systèmes de coordonnées qui permettent de donner à ce point des coordonnées bien précises. La figure 4.1 donne deux exemples de courbes elliptiques définies sur $\mathbb{R}$.

Selon la caractéristique du corps de base, l'équation 4.1 peut être considérablement simplifiée. Trois cas sont distinguables : les corps de caractéristique 2, 3 ou strictement supérieure à 3 (donc $\geqslant 5$). Pour des raisons de sécurité [BLf, PQ12], les courbes basées sur des corps de caractéristiques 2 ou 3 ne sont pas recommandées. Dans cette thèse, nous nous intéressons aux courbes définies sur $\mathbb{F}_p$, avec $p \geqslant 5$ un nombre premier. Nous renvoyons le lecteur vers

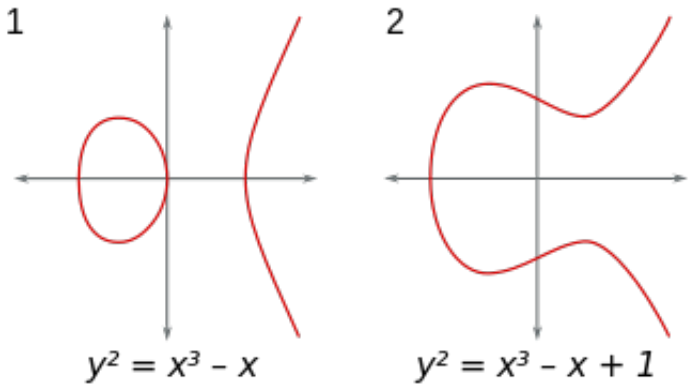


FIGURE 4.1 – Exemples de courbes elliptiques définies sur $\mathbb{R}$.

[HMV04] (section 3.1) pour plus de détails sur la simplification de l'équation 4.1 selon la caractéristique du corps de base.

Soit $p \geqslant 5$ un nombre premier. Pour la suite, nous considérerons la courbe elliptique $E(\mathbb{F}_p)$. L'équation 4.1 peut être réécrite de la façon suivante :

$$E(\mathbb{F}_p) : y^2 = x^3 + ax + b, \tag{4.2}$$

avec $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Cette équation est la *forme de Weierstrass courte* pour $E(\mathbb{F}_p)$. L'ensemble des points de la courbe est alors définie comme suit.

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \,|\, y^2 = x^3 + ax + b\} \cup \mathcal{O}. \tag{4.3}$$

Des exemples de telles courbes sont fournies sur le site SafeCurves [BLe]. Certaines de ces courbes font partie des standards recommandés par l'ANSSI, le SEC ou encore le NIST pour la cryptographie.

Terminons cette partie par le théorème de Hasse qui donne une estimation de l'ordre d'une courbe elliptique. L'ordre de $E(\mathbb{F}_p)$, noté $\#E(\mathbb{F}_p)$, est le nombre de point de la courbe. Ce théorème assure que le nombre de points d'une courbe est du même ordre que le corps de définition de la courbe, i.e. $\#E(\mathbb{F}_p) \approx p$.

Théorème 4.1 (Hasse). *Soit $E(\mathbb{F}_p)$ une courbe elliptique définie sur $\mathbb{F}_p$. On a :*

$$\#E(\mathbb{F}_p) = p + 1 - t,$$

avec $|t| \leqslant 2\sqrt{p}$, où t est appelé la trace de $E(\mathbb{F}_p)$ sur $\mathbb{F}_p$.

4.1.2 Loi de groupe

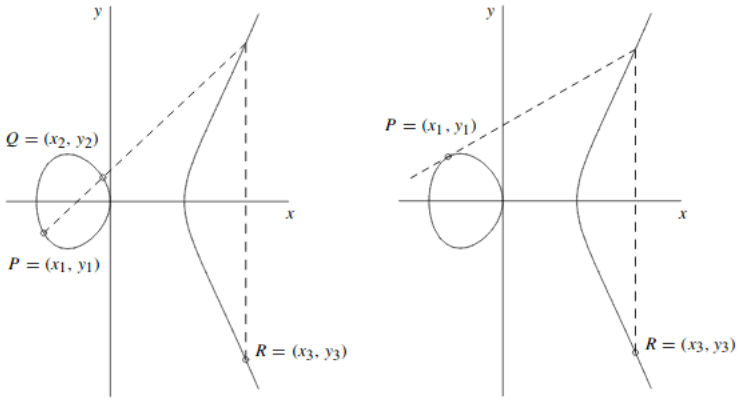
La courbe $E(\mathbb{F}_p)$ peut être muni d'une structure de groupe additif commutatif, dont l'élément neutre est le point à l'infini $\mathcal{O}$. Ce groupe est l'élément de base des cryptosystèmes basés sur les courbes elliptiques.

4.1.2.1 Interprétation géométrique

Soient $P = (x_1, y_1)$ et $Q = (x_2, y_2)$ deux points distincts de la courbe $E(\mathbb{F}_p)$. La loi de groupe peut être interprétée géométriquement grâce à la méthode dite *corde et tangente*, représentée par la figure 4.2.

Pour effectuer l'opération d'addition $R = P+Q$, on trace d'abord une droite passant par P et Q. Cette droite coupe la courbe elliptique en un troisième point. La somme R est alors le symétrique de ce point par rapport à l'axe des x, voir figure 4.2(a).

Pour effectuer l'opération de doublement $R = 2P = P+P$, on trace d'abord la tangente à la courbe, passant par le point P. Cette tangente coupe la courbe elliptique en un deuxième point. Le double R est alors le symétrique de ce point par rapport à l'axe des x, voir figure 4.2(b).



(a) Addition: $P + Q = R$. (b) Doubling: $P + P = R$.

FIGURE 4.2 – Addition et doublement de points sur une courbe elliptique [HMV04].

4.1.2.2 Formules d'addition et de doublement

Le système avec deux coordonnées x et y est appelé le *système de coordonnées affine*. Nous présentons ici les formules mathématiques qui réalisent l'interprétation géométrique présentée plus haut dans ce système de coordonnées.

1. Élément neutre. $P + \mathcal{O} = \mathcal{O} + P = P$, pour tout $P \in E(\mathbb{F}_p)$.
2. Opposé. Soit $P = (x, y) \in E(\mathbb{F}_p)$, on définit l'opposé de P par : $-P = (x, -y)$. De plus, $P - P = (x, y) + (x, -y) = \mathcal{O}$.
3. Addition. Soient $P = (x_1, y_1)$ et $Q = (x_2, y_2)$ deux points de $E(\mathbb{F}_p)$, tels que $P \neq \pm Q$. On a $R = (x_3, y_3) = P + Q$, avec

$$x_3 = \lambda^2 - x_1 - x_2 \quad \text{et} \quad y_3 = \lambda(x_1 - x_3) - y_1,$$

où $\lambda = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)$

4. Doublement. Soit $P = (x_1, y_1) \in E(\mathbb{F}_p)$, tel que $P \neq -P$. On a $R = (x_3, y_3) = 2P = P + P$, avec

$$x_3 = \lambda^2 - 2\,x_1 \quad \text{et} \quad y_3 = \lambda(x_1 - x_3) - y_1,$$

où $\lambda = \left(\frac{3\,x_1^2 + a}{2\,y_1}\right)$

Les formules d'addition et de doublement ci-dessus nécessitent chacune une division dans $\mathbb{F}_p$ pour le calcul de λ. L'inversion dans les corps finis étant en général très coûteuse, d'autres systèmes de coordonnées ont été proposés. Nous en abordons quelques uns dans la section 4.1.3 qui suit.

4.1.3 Systèmes de coordonnées projectifs

Soient c et d deux entiers positifs. On définit sur $\mathbb{F}_p^3 \setminus \{(0,0,0)\}$ la relation d'équivalence $\sim$ suivante :

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \iff \begin{cases} X_1 = \lambda^c X_2, \\ Y_1 = \lambda^d Y_2, \\ Z_1 = \lambda Z_2, \end{cases} \quad \text{avec } \lambda \in \mathbb{F}_p \setminus \{0\}. \tag{4.4}$$

La classe d'équivalence d'un triplet $(X, Y, Z) \in \mathbb{F}_p^3 \setminus \{(0,0,0)\}$ est donc :

$$(X : Y : Z) = \{(\lambda^c X,\ \lambda^d Y,\ \lambda Z) \ :\ \lambda \in \mathbb{F}_p \setminus \{0\}\}.$$

$(X : Y : Z)$ est appelé *point projectif* et tout triplet (X, Y, Z) de cette classe est un *représentant* de ce point projectif. Ainsi, si $Z \neq 0$, alors le triplet $(X/Z^c, Y/Z^d, 1)$ est un représentant du point projectif $(X : Y : Z)$. C'est l'unique représentant ayant la coordonnée Z égale à 1. On établit ainsi une bijection entre l'ensemble des points projectifs $\mathbb{P}(\mathbb{F}_p)^*$ et l'ensemble des points affines $\mathbb{A}(K)$, définies comme suit :

$$\mathbb{P}(\mathbb{F}_p)^* = \{(X : Y : Z) \ :\ X, Y, Z \in \mathbb{F}_p \text{ et } Z \neq 0\},$$

$$\mathbb{A}(K) = \{(x, y) \ :\ x, y \in \mathbb{F}_p\}.$$

L'ensemble des points projectifs $\mathbb{P}(\mathbb{F}_p)^0$ ci-dessous est appelé la *ligne à l'infinie* :

$$\mathbb{P}(\mathbb{F}_p)^0 = \{(X : Y : Z) \ :\ X, Y, Z \in \mathbb{F}_p \text{ et } Z = 0\}.$$

Pour obtenir la *forme projective* de l'équation 4.2 de $E(\mathbb{F}_p)$, on remplace x et y par X/Z^c et Y/Z^d respectivement, puis on réduit au même dénominateur, pour enfin supprimer ce dénominateur. Cette forme est donc :

$$E(\mathbb{F}_p) : Y^2\,Z^{3c-h} = X^3\,Z^{2d-h} + a\,X\,Z^{2(c+d)-h} + b\,Z^{3c+2d-h}, \tag{4.5}$$

avec $h = \min(2\,d,\ 3\,c)$ et $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Si un représentant d'un point projectif $(X : Y : Z)$ est solution de l'équation 4.5, alors tous les représentants de ce point le sont également. On dit donc simplement que $(X : Y : Z)$ appartient à la courbe $E(\mathbb{F}_p)$. Les points de la ligne à l'infinie $\mathbb{P}(\mathbb{F}_p)^0$ qui sont solutions de l'équation 4.5 représentent le point à l'infini $\mathcal{O}$ sur la courbe $E(\mathbb{F}_p)$.

Plusieurs types de coordonnées projectives ont été proposés. Parmi les plus connus, on peut citer les coordonnées projectives standards (avec $c = 1$ et $d = 1$) et les coordonnées jacobiennes (avec $c = 2$ et $d = 3$). Le site [BLd] propose un vaste ensemble de formules (avec les coûts) pour les opérations d'addition et de doublement sur les courbes elliptiques d'équation $y^2 = x^3 + ax + b$. Ce site traite également certains cas particuliers (comme $a = 0$ ou -3) et présente des systèmes de coordonnées étendus assez utilisés en cryptographie.

Ci-dessous, nous présentons brièvement le système de coordonnées projectif standard et le système de coordonnées jacobien. Ensuite, nous abordons les méthodes `ZADDU` [Mel07] et `ZADDC` [GJM+11] pour l'addition de points en coordonnées jacobiennes, ainsi que la méthode `DBLU` [GJM+11] pour le doublement de point lorsque la coordonnée Z est égale à 1.

4.1.3.1 Coordonnées projectives standards

Avec $c = 1$ et $d = 1$, l'équation 4.5 dévient :

$$E(\mathbb{F}_p) : Y^2\,Z = X^3 + a\,X\,Z^2 + b\,Z^3, \tag{4.6}$$

avec $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Le point à l'infini $\mathcal{O}$ est représenté par $(0 : 1 : 0)$ et l'opposé de (X, Y, Z) est $(X, -Y, Z)$. Les formules d'addition et de doublement de points dans le système de coordonnées projectif standard peuvent être trouvées ici [BLc].

4.1.3.2 Coordonnées jacobiennes

Avec $c = 2$ et $d = 3$, l'équation 4.5 dévient :

$$E(\mathbb{F}_p) : Y^2 = X^3 + a\,X\,Z^4 + b\,Z^6, \tag{4.7}$$

avec $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Le point à l'infini $\mathcal{O}$ est représenté par $(1 : 1 : 0)$ et l'opposé de (X, Y, Z) est $(X, -Y, Z)$. Les formules d'addition et de doublement de points dans le système de coordonnées projectif standard peuvent être trouvées ici [BLb].

4.1.3.3 Les méthodes `ZADDU`, `ZADDC` et `DBLU`

Cette section porte sur deux méthodes très efficaces d'addition de points dans le système de coordonnées jacobien (donc $c = 2$ et $d = 3$). Ces méthodes requièrent que les points à additionner aient la même coordonnée Z. Pour rappel, le symbole $\sim$ est la relation d'équivalence de l'équation 4.4. Nous abordons également une méthode de doublement particulière pour le cas où la coordonnée Z est égale à 1.

Dans [Mel07], Méloni propose une formule, nommée `ZADDU`, pour l'addition de deux points P et Q, tels que $P = (X_1, Y_1, Z)$ et $Q = (X_2, Y_2, Z)$ en coordonnées jacobiennes, avec $P \neq Q$. Cette formule calcule deux points R et S tels que $R = (X_3, Y_3, Z_3) = (P + Q)$ et $S = (X_4, Y_4, Z_3) \sim P$. L'algorithme 34 détaille cette formule d'addition.

Dans [Mel07], l'auteur mentionne également une idée similaire à celle de `ZADDU` pour effectuer le doublement de point avec mise à jour de la coordonnée Z de l'opérande sans calcul supplémentaire. On s'intéresse ici au cas spécial $Z = 1$, nommé `DBLU` [GJM+11]. L'algorithme 35 correspond à l'opération `DBLU`.

Dans [GJM+11], Goundar et al. proposent une formule d'addition, nommée `ZADDC`, qui est très proche de `ZADDU`. À partir de deux points P et Q, tels que $P = (X_1, Y_1, Z)$ et $Q = (X_2, Y_2, Z)$ en coordonnées jacobiennes, avec $P \neq Q$, `ZADDC` calcule deux points R et S tels que $R = (X_3, Y_3, Z_3) = (P + Q)$ et $S = (X_4, Y_4, Z_3) = (P - Q)$ (cf. algorithme 36).

Dans le tableau 4.1 (section 4.1.4), les coûts de ces méthodes sont donnés. Nous verrons dans la section 4.2 comment les utiliser pour effectuer la multiplication scalaire.

Algorithme 34 `ZADDU` (Co-Z addition with update) [Mel07]

Entrée(s) : $P, Q \in E(\mathbb{F}_p)$, tels que $P = (X_1, Y_1, Z)$ et $Q = (X_2, Y_2, Z)$
Sortie : $(R, S) \in E(\mathbb{F}_p) \times E(\mathbb{F}_p)$, tel que $R = (P + Q)$ et $S \sim P$

1: $C \leftarrow (X_1 - X_2)^2$
2: $W_1 \leftarrow X_1\, C\,; \quad W_2 \leftarrow X_2\, C$
3: $D \leftarrow (Y_1 - Y_2)^2\,; \quad A_1 \leftarrow Y_1(W_1 - W_2)$
4: $X_3 \leftarrow D - W_1 - W_2$
5: $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$
6: $Z_3 \leftarrow Z(X_1 - X_2)$
7: $X_4 \leftarrow W_1\,; \quad Y_4 \leftarrow A_1$
8: $\#\ R = (X_3, Y_3, Z_3),\ S = (X_4, Y_4, Z_3)$
9: retourner (R, S)

4.1.4 Coûts des opérations d'addition et de doublement

Le tableau 4.1 donne les coûts des opérations d'addition et de doublement dans les systèmes de coordonnées affine, projectif standard et jacobien présentés plus haut. Ces coûts sont donnés en nombre de multiplications ($\mathcal{M}$), de carrés ($\mathcal{S}$) et d'inversions ($\mathcal{I}$) dans $\mathbb{F}_p$. Les multiplications par les constantes a et b de la courbe sont également prises en compte. La notation $t * a$ signifie que la formule associée requiert t multiplication(s) par la constante a ; le même principe s'applique à la constante b.

Les multiplications par les petites constantes fixées (telles que 2, 3, etc.) ne sont pas prises en comptes. En outre, les additions et soustractions dans $\mathbb{F}_p$ ne sont pas prises en compte, car très peu coûteuses par rapport aux autres. Dans la littérature, le coût de ces opérations est assez souvent négligé, d'où ce choix ici. Cependant, on verra dans le prochain chapitre que négliger le coût de

Algorithme 35 `DBLU` (Doubling with update) [GJM+11]

Entrée(s) : $P \in E(\mathbb{F}_p)$, tel que $P = (X_1, Y_1, 1)$
Sortie : $(R, S) \in E(\mathbb{F}_p) \times E(\mathbb{F}_p)$, tel que $R = 2\,P$ et $S \sim P$

1: $B \leftarrow X_1^2$
2: $E \leftarrow Y_1^2$
3: $L \leftarrow E^2$
4: $T \leftarrow 8\,L$
5: $N \leftarrow 2((X_1 + E)^2 - B - L)$
6: $M \leftarrow 3\,B + a$ # le paramètre a de la courbe.
7: $X_2 \leftarrow M^2 - 2\,N$
8: $Y_2 \leftarrow M(N - X_2) - T$
9: $Z_2 \leftarrow 2\,Y_1$
10: # $R = (X_2, Y_2, Z_2)$, $S = (N, T, Z_2)$
11: retourner (R, S)

Algorithme 36 `ZADDC` (Conjugate co-Z addition) [GJM+11]

Entrée(s) : $P, Q \in E(\mathbb{F}_p)$, tels que $P = (X_1, Y_1, Z)$ et $Q = (X_2, Y_2, Z)$
Sortie : $(R, S) \in E(\mathbb{F}_p) \times E(\mathbb{F}_p)$, tel que $R = (P + Q)$ et $S = (P - Q)$

1: $C \leftarrow (X_1 - X_2)^2$
2: $W_1 \leftarrow X_1\,C\,;\quad W_2 \leftarrow X_2\,C$
3: $D \leftarrow (Y_1 - Y_2)^2\,;\quad A_1 \leftarrow Y_1(W_1 - W_2)$
4: $X_3 \leftarrow D - W_1 - W_2$
5: $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$
6: $Z_3 \leftarrow Z(X_1 - X_2)$
7: $\overline{D} \leftarrow (Y_1 + Y_2)^2$
8: $X_4 \leftarrow \overline{D} - W_1 - W_2$
9: $Y_4 \leftarrow (Y_1 + Y_2)(W_1 - X_4) - A_1$
10: # $R = (X_3, Y_3, Z_3)$, $S = (X_4, Y_4, Z_3)$
11: retourner (R, S)

ces opérations lorsqu'on manipule de grands entiers n'est en pratique pas une bonne idée.

Il est possible d'effectuer l'addition de points avec les opérandes dans des systèmes de coordonnées différents. On parle alors d'addition en coordonnées mixtes. Cela permet de réduire le coût de l'addition. Par exemple, dans l'algorithme 37 (section 4.2.1), on peut garder le point P en coordonnées affines pour les additions, avec le point Q en coordonnées jacobiennes.

Les coûts détaillés, avec les formules, peuvent être trouvés dans [BLb, BLc]. Dans le tableau 4.1, les systèmes de coordonnées affine, projectif standard et jacobien sont respectivement représentés par les lettres A, P et J.

4.2 Multiplication scalaire

Soit $k \in \mathbb{N}$ et $P \in E(\mathbb{F}_p)$. La multiplication scalaire (ECSM) est l'opération kP, avec :

Addition		Addition mixte	
Opération	Coût	Opération	Coût
$\mathsf{A} + \mathsf{A} \to \mathsf{A}$	$1\mathcal{I} + 2\mathcal{M} + 1\mathcal{S}$	-	-
$\mathsf{P} + \mathsf{P} \to \mathsf{P}$	$12\mathcal{M} + 2\mathcal{S}$	$\mathsf{P} + \mathsf{A} \to \mathsf{P}$	$9\mathcal{M} + 2\mathcal{S}$
$\mathsf{J} + \mathsf{J} \to \mathsf{J}$	$11\mathcal{M} + 5\mathcal{S}$	$\mathsf{J} + \mathsf{A} \to \mathsf{J}$	$7\mathcal{M} + 4\mathcal{S}$
$\texttt{ZADDU}(\mathsf{J}, \mathsf{J}) \to (\mathsf{J}, \mathsf{J})$	$5\mathcal{M} + 2\mathcal{S}$	-	-
$\texttt{ZADDC}(\mathsf{J}, \mathsf{J}) \to (\mathsf{J}, \mathsf{J})$	$6\mathcal{M} + 3\mathcal{S}$	-	-

Doublement	
Opération	Coût
$2\mathsf{A} \to \mathsf{A}$	$1\mathcal{I} + 2\mathcal{M} + 2\mathcal{S}$
$2\mathsf{P} \to \mathsf{P}$	$5\mathcal{M} + 6\mathcal{S} + 1 * a$
$2\mathsf{J} \to \mathsf{J}$	$1\mathcal{M} + 8\mathcal{S} + 1 * a$
$2\mathsf{J} \to \mathsf{J}$	$2\mathcal{M} + 5\mathcal{S}$ (si $a = 0$)
$\texttt{DBLU}(\mathsf{A}) \to (\mathsf{J}, \mathsf{J})$	$1\mathcal{M} + 5\mathcal{S}$

TABLE 4.1 – Coûts des opérations de doublement, d'addition et d'addition mixte dans les systèmes de coordonnées affine, projectif standard et jacobien, sur une courbe $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$.

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ fois}}.$$

Le résultat de cette opération est un point de la courbe, i.e. $kP \in E(\mathbb{F}_p)$.

Dans [HMV04] (section 3.3), les auteurs présentent plusieurs méthodes pour effectuer la multiplication scalaire efficacement. Dans cette thèse, nous nous intéressons aux méthodes qui sont au minimum sûres contre les attaques de type SPA. Nous ne présenterons donc que des méthodes régulières, i.e. sans branchement conditionnel ou avec des branches équivalentes.

Dans cette section, nous commençons par rappeler quelques méthodes génériques pour effectuer l'ECSM. Ensuite, nous nous intéressons à l'utilisation des chaînes d'additions euclidiennes pour effectuer la multiplication scalaire. Pour terminer, nous abordons l'accélération de cette opération à l'aide des courbes munies d'un endomorphisme efficace.

4.2.1 Méthodes génériques pour l'ECSM

La méthode la plus simple pour effectuer la multiplication scalaire est la méthode *double-and-add*. Cette méthode est l'algorithme générique d'exponentiation (algorithme 1) présenté dans l'introduction de cette thèse pour le groupe additif $(E(\mathbb{F}_p), +)$. On peut remarquer que cet algorithme n'est pas régulier, car l'addition de points n'a lieu que lorsque le bit courant k_i est égal à 1. Dans [Cor99], Coron propose le *double-and-add always* (algorithme 37) qui cor-

respond au *double-and-add* avec une addition de points *factice* lorsque le bit courant k_i est égal à 0. Ainsi, à chaque itération un doublement et une addition de points sont effectués. Ce qui assure la régularité souhaitée. Ces additions factices impliquent cependant des calculs inutiles. Comme mentionné pour le *square-and-multiply always* (section 1.1.3), il en résulte que le *double-and-add always* est vulnérable aux attaques de type *C safe-error*[SMKLM01]. En effet, si une injection de fautes au cours de l'addition à l'itération i ne modifie pas le résultat de la multiplication scalaire, alors on est certain que k_i vaut 0. Sinon, on est certain que k_i vaut 1. Notons que le point P dans cet algorithme reste inchangé pour l'addition (ligne 4), on peut donc effectuer une addition mixte en gardant P en coordonnées affines pour accélérer cette addition.

L'*échelle de Montgomery* [Mon87] est une autre méthode régulière pour effectuer la multiplication scalaire. Cette méthode fut initialement proposée pour effectuer la multiplication scalaire sur les courbes de Montgomery. Ce sont des courbes dont les équations sont de la forme $by^2 = x^3 + ax^2 + x$, avec $a, b \in \mathbb{F}_p$. Ces courbes fournissent une arithmétique de points particulièrement bien adaptée à cette méthode [Mon87]. L'*échelle de Montgomery* a la particularité de garantir que $R_1 - R_0 = P$ à la fin de chaque itération. En outre, contrairement au *double-and-add always*, cette méthode n'introduit aucun calcul factice, ce qui la rend sûre contre les attaques de type *C safe-error*, comme expliqué dans [JY02]. Dans le même article, les auteurs expliquent que l'*échelle de Montgomery* n'est cependant pas sûre contre les attaques de type *M safe-error*[YJ00] dont l'objectif est d'introduire une faute temporaire sans que cette dernière ne soit détectée. Ils proposent par suite une modification de cette méthode qui est sûre contre les attaques de type *M safe-error*. C'est cette modification que nous présentons ici (algorithme 38).

Dans [GJM+11], après avoir introduit la méthode `ZADDC`, Goundar et al. proposent des modifications d'un ensemble d'algorithmes pour effectuer la multiplication scalaire. Ces modifications remplacent les additions et doublements de points par des appels aux méthodes `ZADDU` et `ZADDC`. Il en résulte des algorithmes qui sont à la fois réguliers et plus performants. L'algorithme 39 est leur modification de l'*échelle de Montgomery* basée sur les méthodes `ZADDU` et `ZADDC`. On peut observer dans le tableau 4.2 que cette modification a un coût nettement inférieur à celui de l'original.

Coûts des méthodes génériques.

Le tableau 4.2 donne les coûts des méthodes génériques présentées plus haut. Nous reprenons ici les éléments du tableau 4.1 et supposons que les points sont en coordonnées jacobiennes. Pour le *double-and-add*, nous supposons d'une part qu'en moyenne $n/2$ bits du scalaire valent 1 et d'autre part que l'addition est faite en coordonnées mixtes (avec P en coordonnées affines).

4.2.2 ECSM avec les chaînes d'additions euclidiennes

Cette section porte sur l'utilisation des chaînes d'additions euclidiennes (EAC) pour effectuer la multiplication scalaire. Nous commençons par présen-

Algorithme 37 *Left-to-right double-and-add always*

Entrée(s) : $P \in E(\mathbb{F}_p)$ et $k \in \mathbb{Z}/p\mathbb{Z}$, tel que $k = (k_{n-1}, \ldots, k_0)_2$

Sortie : $R_0 \in E(\mathbb{F}_p)$, tel que $R_0 = kP$

1: $R_0 \leftarrow \mathcal{O}$
2: **for** $i = n - 1$ **to** 0 **do**
3: $\quad R_0 \leftarrow 2\,R_0$
4: $\quad R_1 \leftarrow R_0 + P$
5: $\quad R_0 \leftarrow R_{k_i}$
6: **end for**
7: retourner R_0

Algorithme 38 *Échelle de Montgomery*

Entrée(s) : $P \in E(\mathbb{F}_p)$ et $k \in \mathbb{Z}/p\mathbb{Z}$, tel que $k = (k_{n-1}, \ldots, k_0)_2$

Sortie : $R_0 \in E(\mathbb{F}_p)$, tel que $R_0 = kP$

1: $R_0 \leftarrow \mathcal{O}$
2: $R_1 \leftarrow P$
3: **for** $i = n - 1$ **to** 0 **do**
4: $\quad b \leftarrow k_i$
5: $\quad R_{1-b} \leftarrow R_{1-b} + R_b$
6: $\quad R_b \leftarrow 2\,R_b$
7: **end for**
8: retourner R_0

ter les chaînes d'additions euclidiennes avec quelques propriétés. Ensuite, nous abordons la multiplication scalaire avec ces chaînes.

4.2.2.1 Chaînes d'additions euclidiennes (EAC)

Le problème de la minimisation du nombre d'opérations pour calculer x^k (ou kP, dans le contexte des courbes elliptiques) est étroitement lié à celui de la recherche d'une courte chaîne d'additions menant à k [Knu97].

Définition 4.2. Une chaîne d'additions de longueur s calculant un entier k est une séquence $v_1, \ldots, v_s$ d'entiers, telle que :

- $v_1 = 1$, $v_s = k$,
- $\forall\, i \in [2, s]$, $v_i = v_j + v_t$, avec $1 \leqslant j, t < i$.

Exemple 4.1. La séquence $(1, 2, 3, 4, 5, 7, 11, 12, 23)$ est une chaîne d'additions de longueur 9 calculant l'entier 23. Car, $2 = 1 + 1, 3 = 2 + 1, 4 = 2 + 2, 5 = 3 + 2, 7 = 5 + 2, 11 = 7 + 4, 12 = 7 + 5, 23 = 12 + 11$.

Une chaîne d'additions euclidienne est une chaîne d'additions possédant une propriété particulière qui, comme on le verra plus loin, est plutôt bien adaptée pour la multiplication scalaire.

Algorithme 39 *Co-Z Échelle de Montgomery*

Entrée(s) : $P = (X, Y, 1) \in E(\mathbb{F}_p)$ et $k \in \mathbb{Z}/p\mathbb{Z}$, tel que $k = (1, k_{n-2}, \ldots, k_0)_2$

Sortie : $R_0 \in E(\mathbb{F}_p)$, tel que $R_0 = kP$

1: $(R_1, R_0) \leftarrow \texttt{DBLU}(P)$
2: **for** $i = n - 2$ **to** 0 **do**
3: $\quad b \leftarrow k_i$
4: $\quad (R_{1-b}, R_b) \leftarrow \texttt{ZADDC}(R_b, R_{1-b})$
5: $\quad (R_b, R_{1-b}) \leftarrow \texttt{ZADDU}(R_{1-b}, R_b)$
6: **end for**
7: retourner R_0

Méthode	Coût
Double-and-add always (alg. 37)	$n(8\mathcal{M} + 12\mathcal{S} + 1 * a)$
Échelle de Montgomery (alg. 38)	$n(12\mathcal{M} + 13\mathcal{S} + 1 * a)$
Co-Z Échelle de Montgomery (alg. 39)	$(n-1)(11\mathcal{M} + 5\mathcal{S}) + (1\mathcal{M} + 5\mathcal{S})$

TABLE 4.2 – Coûts des méthodes génériques pour l'ECSM, sur une courbe $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$.

Définition 4.3. Une chaîne d'additions euclidienne (EAC) de longueur s calculant un entier k est une chaîne d'additions qui satisfait les conditions suivantes :

- $v_1 = 1$, $v_2 = 2$, $v_3 = v_2 + v_1$,
- pour tout entier i tel que $3 \leqslant i < s$, si $v_i = v_{i-1} + v_j$, avec $j < i - 1$, alors $v_{i+1} = v_i + v_{i-1}$ (cas 1) ou $v_{i+1} = v_i + v_j$ (cas 2).

Une EAC est une séquence strictement croissante. Dans la définition ci-dessus, le cas 1, où on ajoute le plus grand élément (v_{i-1}) à v_i, est appelé grand pas. Le cas 2, où le plus petit élément (v_j) est ajouté à v_i, est appelé petit pas.

Exemple 4.2. La séquence $(1, 2, 3, 4, 7, 10, 13, 23)$ est une EAC de longueur 8 calculant 23.

La définition qui suit est équivalente à la définition 4.3. Elle permet d'interpréter les EAC comme une séquence binaire. Dans la suite de ce manuscrit, c'est cette définition que nous considérerons.

Définition 4.4. Une chaîne d'additions euclidienne de longueur n est une séquence $c = (c_i)_{i=1\ldots n}$, avec $c_i \in \{0, 1\}$. À cette séquence, on associe la séquence $(v_i, u_i)_{i=0\ldots n}$ telle que :

- $v_0 = 1, u_0 = 2$,
- $\forall\ i \geqslant 1$, $\begin{cases} (v_i, u_i) = (v_{i-1}, v_{i-1} + u_{i-1}), \text{ si } c_i = 1, \\ (v_i, u_i) = (u_{i-1}, v_{i-1} + u_{i-1}), \text{ si } c_i = 0. \end{cases}$

L'entier k calculé à partir de cette séquence est $k = v_n + u_n$.

Exemple 4.3. Soit $c = (10110)$ une EAC de longueur 5. On calcule l'entier k correspondant comme suit :

$$(1,\,2) \xrightarrow{1} (1,\,3) \xrightarrow{0} (3,\,4) \xrightarrow{1} (3,\,7) \xrightarrow{1} (3,\,10) \xrightarrow{0} (10,\,13)$$

Ainsi, $k = 10 + 13 = 23$.

Dans la définition 4.4, on peut remarquer que pour tout $i \in [0, s]$, $v_i < u_i$. En outre, il est évident que le cas $c_i = 1$ correspond au petit pas, car v_i prend la plus petite valeur du couple (v_{i-1}, u_{i-1}). De même, on peut voir que le cas $c_i = 0$ correspond au grand pas, car v_i prend la plus grande valeur du couple (v_{i-1}, u_{i-1}). Le petit pas et le grand pas peuvent être traduits en opérations matricielles.

Définition 4.5. Les matrices S_0 et S_1 ci-dessous correspondent respectivement au grand pas et au petit pas.

$$S_0 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \quad \text{et} \quad S_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Plus précisément, on a :

$$\begin{aligned} (v, u) &\mapsto (u, u + v) = (v, u)S_0 \quad \text{(grand pas)} \\ (v, u) &\mapsto (v, u + v) = (v, u)S_1 \quad \text{(petit pas)} \end{aligned}$$

Notons que S_0 et S_1 sont inversibles avec :

$$S_0^{-1} = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{et} \quad S_1^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

La définition 4.6 présente quelques notations relatives aux EAC.

Définition 4.6. Soit $n > 0$ un entier. On note :

- $\mathcal{M}$ l'ensemble des EAC,
- $\mathcal{M}_n$ l'ensemble des EAC de longueur n,
- e l'unique EAC de longueur 0, i.e. $\mathcal{M}_0 = \{e\}$.
- ψ l'application de $\mathcal{M}$ vers $\mathbb{N} \times \mathbb{N}$, telle que $\psi(c) = (v_n, u_n)$ si $c \in \mathcal{M}_n$,
- χ l'application de $\mathcal{M}$ vers $\mathbb{N}$, telle que $\chi(c)$ est l'entier calculé à partir de $c \in \mathcal{M}$.

Soient $m, n > 0$ des entiers. Si $c \in \mathcal{M}_m$ et $c' \in \mathcal{M}_n$, on note cc' la concaténation des EAC c et c'. Donc :

- $cc' \in \mathcal{M}_{m+n}$,
- $c^h \in \mathcal{M}_{mh}$, avec $h > 0$ un entier,
- e est l'élément neutre de la concaténation.

Soit $c = (c_1, \ldots, c_n) \in \mathcal{M}_n$. D'après les définitions 4.5 et 4.6, on a :

$$\psi(c) = (v_n, u_n) = (1, 2) \prod_{i=1}^{s} S_{c_i},$$
$$\chi(c) = v_n + u_n = (1, 2) \prod_{i=1}^{n} S_{c_i} \left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right).$$

Notons F_i le i-ième nombre de Fibonacci, défini par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour tout $n \in \mathbb{N}$. La proposition suivante donne des bornes sur les EAC et les valeurs qu'elles calculent.

Proposition 4.1. *[HV10] Soit $n > 0$, on a :*

- $S_0^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}$ et $S_1^n = \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix}$.
- $\psi(0^n) = (F_{n+2}, F_{n+3})$, donc $\chi(0^n) = F_{n+4}$,
- $\psi(1^n) = (1, n+2)$, donc $\chi(1^n) = n + 3$,
- $\forall c \in \mathcal{M}_n, \chi(1^n) \leqslant \chi(c) \leqslant \chi(0^n)$,

Le calcul d'une EAC pour un entier k est facile. Il suffit de choisir un entier $g < k$ tel que $\text{pgcd}(g,\ k) = 1$ et ensuite d'appliquer la version soustractive de l'algorithme d'Euclide à g et k.

Exemple 4.4. Soient $k = 23$ et $g = 13$, on a :

$$\begin{aligned} 23 - 13 &= 10 \\ 13 - 10 &= 3 \\ 10 - 3 &= 7 \\ 7 - 3 &= 4 \\ 4 - 3 &= 1 \\ 3 - 1 &= 2 \\ 2 - 1 &= 1 \end{aligned}$$

On reconstitue ensuite les couples (v_i, u_i) de la définition 4.4 en lisant du bas vers le haut les opérandes des soustractions ci-dessus. Pour rappel, on prend toujours $v_i < u_i$, ainsi u_i aura la valeur de l'opérande de gauche et v_i celle de l'opérande de droite. On obtient donc la séquence suivante, avec les types de pas associés :

$$(1,\ 2) \xrightarrow{\text{petit pas}} (1,\ 3) \xrightarrow{\text{grand pas}} (3,\ 4) \xrightarrow{\text{petit pas}} (3,\ 7) \xrightarrow{\text{petit pas}} (3,\ 10) \xrightarrow{\text{grand pas}} (10,\ 13)$$

L'EAC calculant $k = 10 + 13$ est donc : $c = (10110)$.

L'algorithme 40 calcule une EAC pour un entier k passé en paramètre. Cet algorithme est une modification de l'algorithme 2 dans [HV10].

Comme on peut le constater, l'algorithme 40 est plutôt simple. Avec cet algorithme, il est possible de trouver $\varphi(k)$ EAC qui calculent k, où φ est l'*indicatrice*

Algorithme 40 *Calcul d'une EAC pour k*

Entrée(s) : $k \geqslant 4$
Sortie : $c \in \mathcal{M}$, tel que $\chi(c) = k$

1: Générer (aléatoirement) un entier positif $g < k$, tel que $\text{pgcd}(g, k) = 1$.
2: **if** $k \geqslant 2g$ **then**
3: $(v, u) \leftarrow (g, k - g)$
4: **else**
5: $(v, u) \leftarrow (k - g, g)$
6: **end if**
7: $c \leftarrow e$ # l'élément neutre de la concaténation, avec $\mathcal{M}_0 = \{e\}$.
8: **while** $u > 2$ **do**
9: **if** $u \geqslant 2v$ **then**
10: $(v, u) \leftarrow (v, u - v)$
11: $c \leftarrow 1|c$ # concaténation de c et de 1.
12: **else**
13: $(v, u) \leftarrow (u - v, v)$
14: $c \leftarrow 0|c$ # concaténation de c et de 0.
15: **end if**
16: **end while**
17: retourner c

d'Euler. On n'a cependant pas beaucoup d'informations sur la longueur des chaînes calculées. Un résultat asymptotique dû à Yao et Knuth [YK75] affirme que la longueur moyenne de ces chaînes est :

$$S(k) = 6\pi^{-2}(\ln k)^2 + \mathcal{O}(\log k(\log\log k)^2). \tag{4.8}$$

Pour un entier k donné, on ne sait actuellement pas (dans un temps raisonnable, en fonction de k) trouver une EAC courte qui calcule k, ni trouver une EAC de taille donnée qui calcule k. La méthode exhaustive qui consisterait à essayer les $\varphi(k)$ valeurs possibles pour g avec l'algorithme 40 ne peut être envisagée pour un usage cryptographique, car k (et donc $\varphi(k)$) serait trop grand.

4.2.2.2 Multiplication scalaire avec les EAC

Dans [Mel07], Méloni montre que la méthode `ZADDU` qu'il propose est particulièrement bien adaptée aux EAC pour effectuer le calcul de kP.

Soit $c = (c_1, \ldots, c_s)$ une EAC qui calcule k, i.e. $k = \chi(c)$. Dans [Mel07], l'auteur propose l'algorithme 41 qui effectue l'opération $\chi(c)P$, à partir des points P et $2\,P$ tels que $Z_P = Z_{2\,P}$ en coordonnées jacobiennes.

Reprenons les éléments de la section 4.1.4 sur le coût des opérations d'addition et de doublement. Si le point P est donné en coordonnées affines (i.e. $Z = 1$), le calcul de $P' \sim P$ et de $2P$, tels que $Z_{P'} = Z_{2\,P}$, peut se faire avec `DBLU` (algorithme 35). Cette opération coûte donc $1\mathcal{M} + 5\mathcal{S}$. Ainsi, le coût total de l'algorithme `EAC-PointMult` (algorithme 41) est :

$$C_{algo\ 41} = s(5\mathcal{M} + 2\mathcal{S}) + 6\mathcal{M} + 7\mathcal{S}. \tag{4.9}$$

Algorithme 41 `EAC-PointMult`, *Multiplication scalaire avec une EAC* [Mel07]

Entrée(s) : $P, 2P \in E(\mathbb{F}_p)$, tels que $Z_P = Z_{2\,P}$, et $k = \chi(c_1, \ldots, c_s)$

Sortie : $U \in E(\mathbb{F}_p)$, tel que $U = kP$

1: $(U, V) \leftarrow (2P, P)$
2: **for** $i = 1$ **to** s **do**
3: **if** $c_i = 0$ **then**
4: $(U, V) \leftarrow$ `ZADDU`(U, V) # correspond à $(U + V, U)$.
5: **else**
6: $(U, V) \leftarrow$ `ZADDU`(V, U) # correspond à $(U + V, V)$.
7: **end if**
8: **end for**
9: $(U, V) \leftarrow$ `ZADDU`(U, V)
10: retourner U

Comme l'*échelle de Montgomery*, l'algorithme `EAC-PointMult` est régulier et n'effectue pas de calcul factice. Il est donc pertinent de comparer ces deux méthodes. Nous comparons ici l'algorithme `EAC-PointMult` à l'algorithme `Co-Z Échelle de Montgomery` (algorithme 39).

Soit $k > 0$ un entier de n bits et $c = (c_1, \ldots, c_s)$ une EAC qui calcule k, i.e. $k = \chi(c)$. Supposons, pour simplifier, qu'un carré modulaire ait le même coût qu'une multiplication modulaire (i.e. $\mathcal{M} = \mathcal{S}$). Avec l'entier k comme scalaire, on obtient alors que :

$$C_{algo\ 39} = \mathcal{M}(16n - 10),$$
$$C_{algo\ 41} = \mathcal{M}(7s + 13).$$

Donc, pour que l'algorithme `EAC-PointMult` soit au moins aussi efficace que l'algorithme `Co-Z Échelle de Montgomery`, on doit avoir $7s + 13 \leqslant 16n - 10$. Donc, $s \leqslant \frac{16n-23}{7} \approx 2.28n$. Ainsi, pour $n = 160$ par exemple, il faudrait que $s \leqslant 362$.

Dans [PVCTM15], Proy et al. proposent une implémentation matérielle pour la génération de courtes EAC calculant un entier k donné. Les résultats expérimentaux de ces auteurs montrent que pour un entier k de taille n inférieure à 400 bits (environ), leur implémentation produit des EAC de tailles $s \leqslant 2.28n$ avec au maximum 21 appels à l'algorithme 40, en prenant g dans l'intervalle $[\frac{k}{\Phi} - 10, \frac{k}{\Phi} + 10]$ où $\Phi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or. Cependant, aucun résultat théorique ne vient appuyer ces résultats expérimentaux. En outre, le résultat asymptotique de Yao et Knuth (équation 4.8) n'est pas intéressant ici, car pour $n = 160$ par exemple, on a $S(k) \approx 7000$.

Pour contourner cette difficulté, Herbaut et al. propose dans [HLM+10] de générer directement une EAC de taille appropriée pour ensuite effectuer la multiplication scalaire avec l'algorithme `EAC-PointMult`. Cette solution pose cependant un autre problème qui est la distribution des entiers calculés à partir des EAC générées ainsi. La proposition qui suit nous informe qu'une EAC et son image miroir calculent le même entier.

Proposition 4.2. *[HV10] Soient $n > 0$ un entier et $c = (c_1, \ldots, c_n) \in \mathcal{M}_n$ une EAC de longueur n. Alors :*

1. $\chi(c_1, \ldots, c_n) = \chi(c_n, \ldots, c_1)$,
2. *l'application ψ est injective.*

On déduit de cette proposition que l'application χ n'est pas injective. Dans [HV10], Herbaut et Véron relèvent un sous-ensemble particulier de $\mathcal{M}_{2n}$ sur lequel l'application χ est injective.

Proposition 4.3. *[HV10] Soit $\mathcal{M}_n^0$ le sous-ensemble de $\mathcal{M}_{2n}$, dont les éléments commencent par n zéros :*

$$\mathcal{M}_n^0 = \{0^n c, \text{ avec } c \in \mathcal{M}_n\}.$$

La restriction de l'application χ à $\mathcal{M}_n^0$ est injective.

En d'autres termes, si $c, c' \in \mathcal{M}_n^0$ avec $\chi(c) = \chi(c')$, alors $c = c'$. Ainsi, l'ensemble $\mathcal{M}_n^0$ permet de générer 2^n EAC qui calculent 2^n entiers différents. En outre, d'après la proposition 4.1, on a :

$$\forall c \in \mathcal{M}_n^0,\ \chi(0^n 1^n) \leqslant \chi(c) \leqslant \chi(0^{2n})$$

D'après la même proposition, on a $\chi(0^n 1^n) = (n+1)F_{n+2} + F_{n+3}$ et $\chi(0^{2n}) = F_{2n+4}$. Donc,

$$\forall c \in \mathcal{M}_n^0,\ (n+1)F_{n+2} + F_{n+3} \leqslant \chi(c) \leqslant F_{2n+4}$$

Ainsi, en choisissant un point P d'ordre d tel que $d > F_{2n+4}$, on est certain de calculer 2^n points différents avec l'ensemble $\mathcal{M}_n^0$, en utilisant l'algorithme 41.

Soient $c' = 0^n c \in \mathcal{M}_n^0$, avec $c = (c_1, \ldots, c_n)$ et $P \in E(\mathbb{F}_p)$. On peut remarquer que :

$$\begin{aligned}\chi(c')P &= \chi(0^n c)P \\ &= (F_{n+2}P + F_{n+3}P)\prod_{i=1}^{s} S_{c_i}.\end{aligned}$$

Donc, si le point P est **fixé**, on peut pré-calculer les points $F_{n+2}P$ et $F_{n+3}P$, tels qu'ils partagent la même coordonnée Z en coordonnées jacobiennes. Puis, on effectuera la multiplication scalaire $\chi(c')P$ avec l'algorithme 41 en partant du couple $(F_{n+3}P, F_{n+2}P)$, au lieu du couple $(2P, P)$. Cette approche est suggérée par Herbaut et al. dans [HLM+10]. Les auteurs montrent qu'avec un point P fixé d'ordre $d > F_{2n+4}$, cette approche peut être une alternative intéressante aux méthodes classiques, car elle offre le meilleur compromis entre la quantité de pré-calculs et le nombre d'opérations modulaires pour effectuer la multiplication scalaire (voir Table 2 de cet article), tout en bénéficiant du fait que l'algorithme 41 est régulier et n'effectue aucun calcul inutile, tout comme l'*échelle de Montgomery*.

Cette approche n'est cependant plus intéressante lorsque le point P n'est pas fixé. En effet, dans ce cas, il n'est plus possible d'anticiper les n premières étapes

de la multiplication scalaire, ce qui conduit à un nombre trop élevé d'appels à la méthode `ZADDU` dans l'algorithme 41. Dans le chapitre 5, nous verrons comment utiliser efficacement l'algorithme 41 lorsque le point P n'est pas fixé. Cela se fera à l'aide des courbes munies d'un endomorphisme efficace. Dans la section qui suit, nous présentons ce type de courbe.

4.2.3 ECSM avec les courbes munies d'un endomorphisme efficace

Dans [GLV01], Gallant, Lambert et Vanstone propose une nouvelle méthode pour accélérer la multiplication scalaire. Cette méthode repose sur l'utilisation des courbes munies d'un endomorphisme efficace. Il s'agit de la fameuse méthode GLV. Dans cette section, nous effectuons un bref rappel sur les courbes munies d'un endomorphisme efficace. Ensuite, nous présentons la méthode GLV.

4.2.3.1 Courbes munies d'un endomorphisme efficace

Commençons par rappeler l'essentiel sur l'endomorphisme de courbe.

Endomorphisme de courbe. Soit $E(\mathbb{F}_p)$ une courbe elliptique sur $\mathbb{F}_p$. Un endomorphisme ϕ sur cette courbe est une application $\phi : E(\mathbb{F}_p) \to E(\mathbb{F}_p)$ telle que :

- $\phi(\mathcal{O}) = \mathcal{O}$,
- $\phi(P_1 + P_2) = \phi(P_1) + \phi(P_2),\ \forall P_1, P_2 \in E(\mathbb{F}_p)$,
- $\phi(P) = (g(P), h(P)),\ \forall P = (x, y) \in E(\mathbb{F}_p)$,

où g et h sont des fonctions rationnelles à coefficients dans $\mathbb{F}_p$.

Le *polynôme caractéristique* de ϕ est un polynôme univarié et unitaire $f \in \mathbb{Z}[X]$, tel que $f(\phi) = 0$, i.e. $f(\phi)(P) = \mathcal{O},\ \forall P \in E(\mathbb{F}_p)$.

Exemples d'endomorphismes efficaces. Dans [GLV01] (section 2), les auteurs donnent des exemples d'endomorphismes de courbe. Deux d'entre eux sont particulièrement intéressants pour les courbes que nous considérons dans cette thèse.

Exemple 4.5. Soit p un nombre premier, tel que $p \equiv 1 \pmod 4$. Soit $E(\mathbb{F}_p)$ la courbe d'équation :

$$E(\mathbb{F}_p) : y^2 = x^3 + ax \tag{4.10}$$

Soit $\alpha \in \mathbb{F}_p$ un élément d'ordre 4 ; un tel élément existe car $p - 1 \equiv 0 \pmod 4$. On a $\alpha^2 \equiv -1 \pmod p$, donc l'application ϕ ci-dessous est un endomorphisme de la courbe $E(\mathbb{F}_p)$ sur $\mathbb{F}_p$.

$$\begin{array}{rccc} \phi & : & E(\mathbb{F}_p) & \to & E(\mathbb{F}_p) \\ & & (x, y) & \mapsto & (-x, \alpha y) \end{array}$$

Le polynôme caractéristique de ϕ est $f(X) = X^2+1$. On peut en effet remarquer que, pour tout $P = (x, y) \in E(\mathbb{F}_p)$, on a :

$$\begin{aligned} f(\phi)(P) &= \phi(\phi(P)) + P \\ &= (x, \alpha^2 y) + (x, y) \\ &= \mathcal{O}, \text{ car } \alpha^2 \equiv -1 \pmod{p}. \end{aligned}$$

Notons que cet endomorphisme se calcule avec une seule multiplication dans $\mathbb{F}_p$. Il est en outre possible de le calculer directement en coordonnées projectives. Supposons que $P = (X, Y, Z) \in E(\mathbb{F}_p)$. Son représentant en coordonnées affines est $P_a = (X/Z^c, Y/Z^d)$, voir équation 4.4. On a $\phi(P_a) = (-X/Z^c, \alpha Y/Z^d)$. En coordonnées projectives, un représentant de $\phi(P_a)$ est alors $(-X, \alpha Y, Z)$.

Exemple 4.6. Soit p un nombre premier, tel que $p \equiv 1 \pmod 3$. Soit $E(\mathbb{F}_p)$ la courbe d'équation :

$$E(\mathbb{F}_p) : y^2 = x^3 + b \tag{4.11}$$

Soit $\beta \in \mathbb{F}_p$ un élément d'ordre 3 ; un tel élément existe car $p - 1 \equiv 0 \pmod 3$. L'application ϕ ci-dessous est un endomorphisme de la courbe $E(\mathbb{F}_p)$ sur $\mathbb{F}_p$.

$$\begin{array}{rccc} \phi & : & E(\mathbb{F}_p) & \to & E(\mathbb{F}_p) \\ & & (x, y) & \mapsto & (\beta x, y) \end{array}$$

Le polynôme caractéristique de ϕ est $f(X) = X^2 + X + 1$. En effet, pour tout $P = (x, y) \in E(\mathbb{F}_p)$, on a :

$$\begin{aligned} f(\phi)(P) &= \phi(\phi(P)) + \phi(P) + P \\ &= (\beta^2 x, y) + (\beta x, y) + (x, y) \end{aligned}$$

Or $\beta^3 - 1 = (\beta - 1)(\beta^2 + \beta + 1) \equiv 0 \pmod p$, donc $\beta^2 + \beta + 1 \equiv 0 \pmod p$ car $\beta \neq 1$. Ainsi, $(\beta^2 x, y) + (\beta x, y) = (x, -y)$, donc $f(\phi)(P) = \mathcal{O}$. Notons également que cet endomorphisme se calcule avec une seule multiplication dans $\mathbb{F}_p$. Comme pour l'exemple précédent, on peut le calculer directement en coordonnées projectives. Supposons que $P = (X, Y, Z) \in E(\mathbb{F}_p)$. Son représentant en coordonnées affines est $P_a = (X/Z^c, Y/Z^d)$. On a $\phi(P_a) = (\beta X/Z^c, Y/Z^d)$. En coordonnées projectives, un représentant de $\phi(P_a)$ est donc $(\beta X, Y, Z)$.

Remarque 4.1 (**Endomorphisme et groupe cyclique**). Supposons que l'ordre de la courbe $E(\mathbb{F}_p)$ soit tel que $\#E(\mathbb{F}_p) = m \times h$, où m est un nombre premier et $h < m$. Alors, $E(\mathbb{F}_p)$ contient un seul sous-groupe (cyclique) G d'ordre m. Soit $P \in E(\mathbb{F}_p)$ un générateur de ce sous-groupe, donc $G = \langle P \rangle$ et P est d'ordre m. Si ϕ est un endomorphisme sur $E(\mathbb{F}_p)$. Alors $\phi(P) \in \langle P \rangle$. Donc, il existe un entier $\lambda \in [0, m - 1]$ tel que :

$$\phi(P) = \lambda P.$$

Ainsi, $f(\phi)(P) = f(\lambda)(P)$, où f est le polynôme caractéristique de ϕ. Comme $f(\phi)(P) = \mathcal{O}$, on en déduit que $f(\lambda) \equiv 0 \pmod m$, i.e. λ est une racine modulo m du polynôme caractéristique de ϕ.

4.2.3.2 La méthode GLV

Reprenons les éléments de la remarque précédente. Soit $k \in [0, m-1]$ un entier. Pour effectuer l'opération kP, le principe de la méthode GLV est de d'abord décomposer k comme suit :

$$k = k_1 + k_2\lambda, \text{ avec } |k_1|, |k_2| \approx \sqrt{m}.$$

Avec cette décomposition, on obtient que :

$$\begin{aligned} kP &= (k_1 + k_2\lambda)P \\ &= k_1P + k_2(\lambda P) \\ &= k_1P + k_2\,\phi(P) \end{aligned}$$

On peut donc calculer kP en utilisant un algorithme de multiplication simultanée de deux points par deux scalaires, comme l'astuce de Shamir décrite dans [ElG85] ou une des méthodes présentées dans [Möl01]. Ainsi, le nombre de doublements de points pour effectuer cette opération est à peu près réduit de moitié, en échange de quelques additions supplémentaires. Si $\phi(P)$ et la décomposition de k se calculent efficacement, le gain en efficacité est considérable.

Soit g l'homomorphisme défini comme suit :

$$\begin{array}{rccc} g & : & \mathbb{Z}\times\mathbb{Z} & \to & \mathbb{Z}/m\mathbb{Z} \\ & & (i,j) & \mapsto & (i+j\lambda) \bmod m \end{array}$$

Dans [GLV01], l'idée principale de la décomposition de k est de d'abord trouver deux vecteurs courts $v_1, v_2 \in \mathbb{Z}\times\mathbb{Z}$ linéairement indépendants, tels que $g(v_1) = g(v_2) = 0$. Soit $\mathcal{L}$ le réseau euclidien engendré par les vecteurs v_1 et v_2. On calcule un vecteur $\eta \in \mathcal{L}$ proche du vecteur $(k, 0)$. On prend alors $(k_1, k_2) = (k, 0) - \eta$.

Cette décomposition du scalaire k a été améliorée dans plusieurs travaux, dont [PJKL02, SCQ02]. En outre, Sica et al. [SCQ02] montrent qu'il est toujours possible d'effectuer cette décomposition avec $\max(|k_1|, |k_2|) \leqslant \sqrt{1 + |r| + s}\sqrt{m}$, où r et s sont les coefficients du polynôme caractéristique f de ϕ, tel que $f(X) = X^2 + rX + s$.

L'algorithme initial de calcul de point associé à la méthode GLV requiert le stockage des points P, $\phi(P)$, $P + \phi(P)$ et $P - \phi(P)$. Cet algorithme n'est pas régulier et donc est vulnérable aux attaques de type SPA. Dans [FHLS15], Faz-Hernández et al. propose une version régulière de la méthode GLV. Après la décomposition de k en k_1 et k_2, les auteurs réencodent ces derniers dans la représentation `GLV-SAC` (GLV-based Sign-Aligned Colum). Avec cette représentation, ils proposent une méthode de multiplication scalaire régulière qui ne requiert que les points P et $P + \phi(P)$.

Posons $l = \lceil \log_2(m)/2 \rceil + 1$. L'algorithme 42 réencode les scalaires k_1 et k_2 dans la représentation GLV-SAC. L'algorithme 43 décrit la méthode GLV basée sur la représentation GLV-SAC.

Notons que la méthode GLV a été étendue à un ensemble plus large de courbes elliptiques définies sur $\mathbb{F}_{p^2}$ munies de plus d'un endomorphisme. Avec

ces courbes, la décomposition du scalaire avec plusieurs endomorphismes permet d'accélérer encore plus la multiplication scalaire, moyennant un surcoût mémoire. Nous ne traitons pas ces courbes dans cette thèse. Ces courbes sont traitées ici [GLS09, Smi13, GI13, LS14].

Algorithme 42 `GLV-SAC` [FHLS15]

Entrée(s) : $k_1, k_2 \in \mathbb{N}$, tels que $k_1 = (k_{\ell-1}^{(1)}, \ldots, k_0^{(1)})$ et $k_2 = (k_{\ell-1}^{(2)}, \ldots, k_0^{(2)})$, avec k_1 impair.
Sortie : $((b_{\ell-1}^{(1)}, \ldots, b_0^{(1)}), (b_{\ell-1}^{(2)}, \ldots, b_0^{(2)}))$, tel que $\forall i$, $b_i^{(1)} \in \{-1, 1\}$ et $b_i^{(2)} \in \{0, b_i^{(1)}\}$.

$b_{\ell-1}^{(1)} \leftarrow 1$
for $i = 0, \ldots, \ell - 2$ **do**
    $b_i^{(1)} \leftarrow 2k_{i+1}^{(1)} - 1$
    $b_i^{(2)} \leftarrow b_i^{(1)}.k_0^{(2)}$
    $k_2 \leftarrow \lfloor k_2/2 \rfloor - \lfloor b_i^{(2)}/2 \rfloor$
end for
$b_{\ell-1}^{(2)} \leftarrow k_0^{(2)}$
retourner $((b_{\ell-1}^{(1)}, \ldots, b_0^{(1)}), (b_{\ell-1}^{(2)}, \ldots, b_0^{(2)}))$

Algorithme 43 SGLV [FHLS15]

Entrée(s) : k et $PP = (P, P + \phi(P))$
Sortie : $Q = kP$

$(k_1, k_2) \leftarrow$ Décompose(k,ϕ,m, $E(\mathbb{F}_p)$) # la décomposition de k, voir [PJKL02, SCQ02].
$ct \leftarrow k_1 \bmod 2$
if $ct = 0$ **then**
    $k_1 \leftarrow k_1 - 1$
end if
$((x_{l-1}, \ldots, x_0), (y_{l-1}, \ldots, y_0)) \leftarrow$ **GLV-SAC**(k_1, k_2)
$Q \leftarrow (X_{PP[|y_{l-1}|]}, \text{sign}(x_{l-1}).Y_{PP[|y_{l-1}|]})$
$j \leftarrow l - 2$
while $(j \geqslant 0)$ **do**
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + (X_{PP[|y_j|]}, \text{sign}(x_j).Y_{PP[|y_j|]})$
    $j \leftarrow j - 1$
end while
if $ct = 0$ **then**
    $Q \leftarrow Q + (X_{PP[0]}, Y_{PP[0]})$
end if
retourner Q

Coûts de la méthode GLV régulière

En plus du calcul de l'endomorphisme, **SGLV** (algorithme 43) requiert la décomposition du scalaire k en k_1 et k_2, puis le réencodage de ces derniers dans la représentation GLV-SAC. Le tableau 4.3 donne le coût de **SGLV** sans tenir

compte des coûts de ces opérations. Dans ce tableau, nous distinguons deux cas selon la valeur du paramètre a de la courbe.

Pour rappel, $l = \lceil \log_2(m)/2 \rceil + 1$, où m est l'ordre du point P. Donc, $l - 1 = \lceil \log_2(m)/2 \rceil$. On considère que les points P et $P + \phi(P)$ sont donnés en coordonnées affines. Les calculs sont faits en coordonnées jacobiennes, avec des additions en coordonnées mixtes ($\mathsf{J} + \mathsf{A} \rightarrow \mathsf{J}$).

	Coût
Si $a \neq 0$	$(l-1)(8\mathcal{M} + 12\mathcal{S} + 1 * a) + (7\mathcal{M} + 4\mathcal{S})$
Si $a = 0$	$(l-1)(9\mathcal{M} + 9\mathcal{S}) + (7\mathcal{M} + 4\mathcal{S})$

TABLE 4.3 – Coût de la méthode GLV régulière, sur une courbe $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$.

Remarque 4.2. Bien que régulières et donc sûres contre les attaques simples de type SPA, les méthodes de multiplication scalaire présentées dans ce chapitre sont vulnérables à certaines attaques plus avancées. En effet, on peut observer que l'ordre de lecture des données (par exemple) dans ces méthodes est lié au scalaire k. Ce type d'information peut être exploité pour des attaques de type DPA [KJJ99] ou des attaques sur la mémoire cache [BH09, ABG10, BVdPSY14]. Comme mentionné dans le chapitre 3, il est indispensable que la multiplication scalaire soit randomisée pour la protéger des attaques de type DPA. Dans le même chapitre, nous avons vu comment effectuer cette randomisation à l'aide de l'AMNS. Dans les prochains chapitres, nous nous intéresserons à la protection de l'ECSM contre certaines attaques sur la mémoire cache.

Chapitre 5

Généralisation de la multiplication scalaire avec les EAC

Sommaire

Dans la section 4.2.2, nous avons abordé l'utilisation des chaînes d'additions euclidiennes (EAC) pour effectuer la multiplication scalaire. Nous avons vu que l'approche proposée par Herbaut et al. dans [HLM+10] pour effectuer la multiplication scalaire avec les EAC peut être une alternative sûre et efficace aux méthodes classiques, lorsque le point de base est fixé. Dans ce chapitre, nous nous intéressons au cas où le point de base n'est pas fixé. Nous verrons en particulier comment utiliser les courbes munies d'un endomorphisme efficace pour effectuer la multiplication scalaire efficacement avec les chaînes d'additions euclidiennes. Nous comparerons l'efficacité et la sûreté de l'approche proposée ici à celles de la méthode GLV régulière. En outre, nous nous intéresserons à la méthode GLV sur les courbes "tordues" d'Edwards (twisted Edwards curves, en anglais)[BBJ+08]. Ces courbes fournissent un système de coordonnées étendu qui permet des doublements et additions de points très efficaces [HWCD08].

L'accent sera mis sur l'aspect pratique pour justifier l'efficacité du travail présenté ici. Nous nous intéresserons notamment à la bibliothèque Gnu MP, ainsi qu'à la classe `BigInteger` disponible sous le système d'exploitation Android.

Comme mentionné dans les chapitres précédents, il est indispensable que la multiplication scalaire soit faite de façon régulière pour qu'elle soit protégée contre les attaques de type SPA. Il est également nécessaire que cette opération soit randomisée pour la protéger des attaques de type DPA. Dans le chapitre 3, nous avons vu comment effectuer cette randomisation à l'aide de l'AMNS.

Il est possible de mener des attaques sur la mémoire cache du matériel. Ces attaques exploitent les accès à la mémoire cache lors de l'exécution du code pour obtenir des informations sur le secret [Pag02, Pag03]. Dans [BH09], Brumley et al. montrent que les attaques sur la mémoire cache sont effectives contre les protocoles ECC. Ces attaques peuvent être menées sur le cache d'instructions (il s'agit alors de l'*instruction cache attack*)[ABG10] ou sur le cache de données (il s'agit alors du *data cache attack*)[BVdPSY14, BH09]. Pour le *data cache attack*, l'attaquant exploite les éventuelles corrélations entre le secret et le chargement des données dans le cache. Pour l'*instruction cache attack*, ce sont les éventuelles corrélations entre le secret et la séquence d'instructions dans le cache qui sont exploitées par l'attaquant. Ainsi, pour protéger les implémentations contre ces attaques, il est indispensable de rendre les accès au cache (d'instructions et de données) indépendants du secret. Dans ce chapitre, nous verrons une modification de l'`EAC-PointMult` (algorithme 41) qui est sûre contre ces attaques sur le cache.

Ce chapitre est organisé comme suit. Nous commençons par présenter des propriétés sur les chaînes d'additions euclidiennes. Ces propriétés généralisent les éléments de la section 4.2.2.1. Ensuite, nous voyons comment utiliser ces propriétés pour effectuer de manière efficace la multiplication scalaire avec les EAC lorsque le point de base varie. Cela se fera avec les courbes munies d'un endomorphisme efficace. Par suite, nous comparons l'algorithme proposé à la méthode GLV régulière. Enfin, nous terminons par une conclusion et quelques perspectives. Le travail présenté ici a fait l'objet d'une publication [DHMV18].

5.1 Chaînes d'additions euclidiennes calculant différents points

Nous commençons cette section avec une définition qui généralise les applications ψ et χ de la section 4.2.2.1.

Définition 5.1. Soient $(a, b) \in \mathbb{N}^2$, $n \in \mathbb{N}\backslash\{0\}$ et $c \in \mathcal{M}_n$. On définie les deux applications suivantes.

$$\psi_{a,b}(c) = (a, b) \prod_{i=1}^{n} S_{c_i} ,$$

et

$$\chi_{a,b}(c) = (a,b)\prod_{i=1}^{n} S_{c_i}\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right).$$

On peut remarquer que le cas $(a,b) = (1,2)$ correspond aux applications ψ et χ de la définition 4.6, i.e. $\psi_{1,2}(c) = \psi(c)$ et $\chi_{1,2}(c) = \chi(c)$. Ainsi, $\chi_{a,b}(c)P$ est le point obtenu avec l'algorithme 41 en partant du couple (bP, aP), au lieu du couple $(2P, P)$. Dans ce qui suit, nous nous intéressons à l'injectivité de l'application $\chi_{a,b}$.

Proposition 5.1. *Soit $n \in \mathbb{N}\setminus\{0\}$. L'application $\mu : \mathcal{M}_n \to \mathbb{N}^2$ définie par $\mu(c) = \prod_{i=1}^{n} S_{c_i}\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ est injective.*

Démonstration. Soient $c, c' \in \mathcal{M}_n$ deux EAC, telles que $c = (c_1, \ldots, c_n)$ et $c' = (c'_1, \ldots, c'_n)$. Montrons que $\mu(c) = \mu(c')$ implique que $c = c'$. Pour cela, il suffit de montrer que $c_1 = c'_1$. Car, dans ce cas, il suffit de multiplier $\mu(c)$ et $\mu(c')$ par $S_{c_1}^{-1}$ et de reprendre le même raisonnement pour c_2 et c'_2, et ainsi de suite jusqu'à c_n et c'_n.

D'abord, remarquons que pour tout $c \in \mathcal{M}_n$, les deux composantes de $\mu(c)$ sont strictement positives. Ensuite, pour tout couple (x,y), on a :

$$S_0\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) = \left(\begin{smallmatrix}y\\x+y\end{smallmatrix}\right) \quad \text{et} \quad S_1\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) = \left(\begin{smallmatrix}x+y\\y\end{smallmatrix}\right).$$

Donc, si $\mu(c) = \left(\begin{smallmatrix}\alpha\\\beta\end{smallmatrix}\right)$, alors $c_1 = 0$ si et seulement si $\beta > \alpha$.
Ainsi, si $\mu(c) = \mu(c') = \left(\begin{smallmatrix}\alpha\\\beta\end{smallmatrix}\right)$, on peut distinguer deux cas : $\beta > \alpha$ ou $\beta < \alpha$. D'après ce qui a été dit plus haut, on a alors : $c_1 = c'_1 = 0$ ou $c_1 = c'_1 = 1$. □

D'après la proposition 4.1, on a :

$$S_0^n\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right) = \left(\begin{smallmatrix}F_{n+1}\\F_{n+2}\end{smallmatrix}\right) \quad \text{et} \quad S_1 S_0^{n-1}\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right) = \left(\begin{smallmatrix}F_{n+2}\\F_{n+1}\end{smallmatrix}\right)$$

Donc, si $\mu(c) = \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right)$, alors :

$$(x \leqslant F_{n+1} \text{ et } y \leqslant F_{n+2}) \quad \textbf{ou} \quad (x \leqslant F_{n+2} \text{ et } y \leqslant F_{n+1}). \tag{5.1}$$

La proposition qui suit donne une condition d'injectivité de l'application $\chi_{a,b}$.

Proposition 5.2. *Soient n, a et b trois entiers positifs, tels que :*

- $\text{pgcd}(a, b) = 1$,
- $a > F_{n+2}$ ou $b > F_{n+2}$.

Alors, pour tout $(c, c') \in \mathcal{M}_n^2$, on a $\chi_{a,b}(c) = \chi_{a,b}(c')$ si et seulement si $c = c'$.

Démonstration. Soient $(c, c') \in \mathcal{M}_n^2$ telles que $\chi_{a,b}(c) = \chi_{a,b}(c')$. On a $\chi_{a,b}(c) = (a,b)\mu(c)$ et $\chi_{a,b}(c') = (a,b)\mu(c')$.
Posons $\mu(c) = \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right)$ et $\mu(c') = \left(\begin{smallmatrix}x'\\y'\end{smallmatrix}\right)$. Ainsi, $\chi_{a,b}(c) = \chi_{a,b}(c')$ implique que $a(x - x') = b(y' - y)$. On a $\text{pgcd}(a, b) = 1$, donc d'après le lemme de Gauss, a divise $(y' - y)$, et b divise $(x - x')$. D'après l'équation 5.1, $|y' - y| \leq F_{n+2}$, donc si $a > F_{n+2}$, on en déduit que $y' - y = 0$ et donc que $x - x' = 0$, i.e. $\mu(c) = \mu(c')$. De la même manière, si $b > F_{n+2}$, on obtient que $\mu(c) = \mu(c')$.
Ainsi, la proposition 5.1 permet de conclure. □

Corollaire 5.1. *Soient $E(\mathbb{F}_p)$ une courbe elliptique et $P \in E(\mathbb{F}_p)$ un point d'ordre d. Soient n, a et b trois entiers positifs, tels que :*

- $\text{pgcd}(a, b) = 1$,
- $a > F_{n+2}$ ou $b > F_{n+2}$,
- $aF_{n+1} + bF_{n+2} < d$ et $aF_{n+2} + bF_{n+1} < d$.

Alors, l'ensemble des EAC $\mathcal{M}_n$ permet de calculer 2^n points différents avec l'algorithme 41 en partant du couple (aP, bP) au lieu du couple $(P, 2P)$.

Démonstration. Soient $c, c' \in \mathcal{M}_n$, avec $c \neq c'$. Posons, $Q_1 = \chi_{a,b}(c)P$ et $Q_2 = \chi_{a,b}(c')P$. Les points Q_1 et Q_2 sont calculés respectivement par les chaînes c et c', avec l'algorithme 41 en partant du couple (aP, bP). On a $Q_1 = Q_2$ si et seulement si $\chi_{a,b}(c) \equiv \chi_{a,b}(c') \bmod d$.
Avec les deux premières conditions du corollaire, la proposition 5.2 nous assure que $\chi_{a,b}(c) \neq \chi_{a,b}(c')$. On a $\chi_{a,b}(c) = (a, b)\mu(c)$ et $\chi_{a,b}(c') = (a, b)\mu(c')$. Donc, avec la troisième condition du corollaire, l'équation 5.1 implique que $\chi_{a,b}(c) < d$ et $\chi_{a,b}(c') < d$. D'où, $Q_1 \neq Q_2$. □

Ce corollaire permet de dégager deux exemples intéressants.

Exemple 5.1. Soient $E(\mathbb{F}_p)$ une courbe elliptique et $P \in E(\mathbb{F}_p)$ un point d'ordre $d > F_{2n+4}$. Posons, $a = F_{n+2}$ et $b = F_{n+3}$. On a $aF_{n+1} + bF_{n+2} = F_{2n+4}$. De plus, on a $aF_{n+2} + bF_{n+1} < aF_{n+1} + bF_{n+2}$, car $aF_{n+1} + bF_{n+2} - (aF_{n+2} + bF_{n+1}) = F_nF_{n+1} > 0$. Les trois conditions du corollaire 5.1 étant satisfaites, on en déduit qu'en partant du couple $(F_{n+2}P, F_{n+3}P)$, l'algorithme 41 permet, avec l'ensemble $\mathcal{M}_n$, de calculer 2^n points différents. Cet exemple correspond à la méthode 1 dans [HLM+10], où les auteurs proposent de pré-calculer les points $F_{n+2}P$ et $F_{n+3}P$ tels qu'ils partagent la même coordonnée Z.

Exemple 5.2. Soient $E(\mathbb{F}_p)$ une courbe elliptique et $P \in E(\mathbb{F}_p)$ un point d'ordre d. Les conditions du corollaire 5.1 sont satisfaites en prenant $a = 1$, $b > F_{n+2}$ et $d > F_{n+1} + bF_{n+2}$. Dans ce cas, l'algorithme 41 permet de calculer 2^n points différents en partant du couple (P, bP). Cette approche requiert le calcul de P et bP avec la même coordonnée Z. Pouvoir effectuer efficacement ce calcul, pour un point P qui n'est pas nécessairement fixé, permettrait de généraliser l'approche de l'exemple 5.1. Dans la section suivante, nous voyons comment le faire à l'aide des courbes munies d'un endomorphisme efficace.

5.2 Multiplication scalaire avec les EAC sur des courbes munies d'un endomorphisme efficace

Reprenons les éléments de la remarque 4.1 du chapitre précédent. Nous supposons donc que $\#E(\mathbb{F}_p) = m \times h$, où m est un nombre premier et $h < m$. Soit ϕ un endomorphisme de $E(\mathbb{F}_p)$ sur $\mathbb{F}_p$. Comme expliqué dans cette remarque, il existe $\lambda \in [0, m-1]$, tel que pour tout $P \in E(\mathbb{F}_p)$ d'ordre m, $\phi(P) = \lambda P$. Si l'endomorphisme ϕ est facilement calculable, l'exemple 5.2 devient très intéressant, car on pourrait prendre $b = \lambda$, et donc $(P, bP) = (P, \phi(P))$. Cependant,

rien ne garantit que les conditions du corollaire 5.1 seront satisfaites. Dans la sous-section qui suit, nous donnons des résultats d'injectivité pour le couple $(P, \phi(P))$. Puis, dans la sous-section d'après, nous présentons une modification de l'algorithme 41.

5.2.1 Consistance de la multiplication scalaire sur les courbes munies d'un endomorphisme efficace

Soit $P \in E(\mathbb{F}_p)$ un point d'ordre m. Pour la suite, nous supposons que le polynôme caractéristique f de l'endomorphisme ϕ est tel que :

$$f(X) = X^2 + rX + s.$$

Dans [SCQ02] (section 2.1), la proposition suivante est donnée.

Proposition 5.3. *[SCQ02] Soit $(k_1, k_2) \in \mathbb{Z}^2 \setminus \{(0,0)\}$. Si $k_1 + k_2\lambda \equiv 0 \bmod m$, alors :*

$$\max(|k_1|, |k_2|) \geqslant \sqrt{\frac{m}{1 + |r| + s}}.$$

À partir de cette proposition, on déduit le résultat d'injectivité qui suit.

Proposition 5.4. *Soit $n \in \mathbb{N} \setminus \{0\}$. Supposons que l'ordre m du point P soit tel que :*

$$m > F_{n+2}^2 (1 + |r| + s).$$

Alors, en partant du couple $(P, \phi(P))$, l'algorithme 41 permet de calculer 2^n points différents, avec l'ensemble $\mathcal{M}_n$.

Démonstration. En partant du couple $(P, \phi(P))$ et une EAC $c = (c_1, \ldots, c_n) \in \mathcal{M}_n$, l'algorithme 41 calcule $k_1 P + k_2 \lambda P$, avec $\mu(c) = \binom{k_1}{k_2} = \prod_{i=1}^{n} S_{c_i} \binom{1}{1}$.
Soient $(c, c') \in \mathcal{M}_n^2$, telles que c et c' calculent le même point à partir du couple $(P, \phi(P))$. Alors, $k_1 + k_2\lambda \equiv k_1' + k_2'\lambda \bmod m$, avec $\mu(c') = \binom{k_1'}{k_2'}$.
On en déduit donc que :

$$(k_1 - k_1') + (k_2 - k_2')\lambda \equiv 0 \bmod m.$$

D'après l'équation 5.1, on sait que les composantes des vecteurs $\mu(c)$ et $\mu(c')$ sont inférieures ou égales à F_{n+2}. Ainsi, on a :

$$|k_i - k_i'| < F_{n+2} < \sqrt{\frac{m}{1 + |r| + s}} \text{ pour } i \in \{1, 2\}.$$

Si $(k_1 - k_1', k_2 - k_2') \neq (0,0)$, alors d'après la proposition 5.3, on aurait :

$$\max(|k_1 - k_1'|, |k_2 - k_2'|) \geqslant \sqrt{\frac{m}{1 + |r| + s}},$$

ce qui serait absurde. Donc, $(k_1 - k_1', k_2 - k_2') = (0,0)$, i.e. $\mu(c) = \mu(c')$.
La proposition 5.1 permet donc de conclure que $c = c'$. □

Avec la proposition ci-dessus, la nouvelle approche pour effectuer la multiplication scalaire consiste à générer aléatoirement un EAC $c \in \mathcal{M}_n$ et ensuite exécuter l'algorithme 41 en partant du couple $(P, \phi(P))$. En partant du point P, on obtient ainsi une application qui associe à une EAC aléatoire de taille n un point aléatoire du groupe $\langle P \rangle$. On a :

$$F_n = \frac{\gamma^n - \overline{\gamma}^n}{\sqrt{5}}, \text{ avec } \gamma = \frac{1+\sqrt{5}}{2} \text{ et } \overline{\gamma} = \frac{1-\sqrt{5}}{2}.$$

Donc, pour satisfaire la condition de la proposition 5.4, il suffit que :

$$m > \gamma^{(2n+4)} \frac{1 + |r| + s}{5}.$$

On a : $\log_2(\gamma^{(2n+4)} \frac{1+|r|+s}{5}) \approx 2n \log_2(\gamma) \in [1.388n, 1.389n]$. Par conséquent, il est nécessaire de choisir le corps de base $\mathbb{F}_p$ tel que $\log_2(p) > 1.389n$ pour assurer l'injectivité avec des EAC de taille n. Car, d'après le théorème de Hasse (théorème 4.1), p et $E(\mathbb{F}_p)$ sont du même ordre de grandeur. La table 5.1 donne quelques exemples de tailles pour des niveaux de sécurité donnés.

Niveau de sécurité	96	128	192
Taille du corps de base (en bits)	269	358	536

TABLE 5.1 – Taille de corps requis par niveau de sécurité, lorsque l'endomorphisme ϕ satisfait $\phi^2 + r\phi + s = 0$, avec $(r, s) \in \{(0, 1), (1, 1), (-1, 2)\}$.

5.2.2 Nouvelle méthode de multiplication scalaire avec les chaînes d'additions euclidiennes

Nous commençons par présenter une légère modification de l'algorithme `ZADDU` (algorithme 34) présenté dans la section 4.1.3.3. Cette modification, appelée `ZADDb` (algorithme 44), a comme paramètre un bit b. Elle modifie directement deux points P et Q tels que $Z_P = Z_Q$, en remplaçant le couple (P, Q) par :

- le couple $(P, P + Q)$, si $b = 1$, avec $Z_P = Z_{P+Q}$,
- le couple $(Q, P + Q)$, si $b = 0$, avec $Z_Q = Z_{P+Q}$.

Soit $c \in \mathcal{M}_n$. Supposons que le point de base P soit tel que $P = (X_0, Y_0) \sim (X_0, Y_0, 1)$. Soit $Q \in E(\mathbb{F}_p)$, tel que $Q = \phi(P) = (X_1, Y_1) \sim (X_1, Y_1, 1)$. Avec `ZADDb`, la multiplication scalaire peut être faite très simplement sans avoir besoin de point intermédiaire. En effet, on peut observer que l'algorithme 45 qui réalise cette opération ne fait que des appels à `ZADDb`, avec les bits de c comme paramètres. Reprenons les notations utilisées dans le chapitre précédent pour le coût des opérations de doublement et d'addition de points. Le coût de l'algorithme 45 est :

$$C_{algo\ 45} = (n + 1)(5\mathcal{M} + 2\mathcal{S}). \tag{5.2}$$

Algorithme 44 `ZADDb`

Entrée(s) : $b \in \{0, 1\}$

Sortie : Met à jour X_0, Y_0, X_1, Y_1 et Z, tels que : (X_0, Y_0, Z) et (X_1, Y_1, Z) représentent respectivement P et $P + Q$ si $b = 1$, ou Q et $P + Q$ si $b = 0$.

1: $C \leftarrow X_{1-b} - X_b$
2: $Z \leftarrow Z.C$
3: $C \leftarrow C^2$
4: $W \leftarrow X_b.C$
5: $X_0 \leftarrow X_{1-b}.C$
6: $C \leftarrow Y_{1-b} - Y_b$
7: $X_1 \leftarrow C^2 - W - X_0$
8: $W \leftarrow X_0 - W$
9: $Y_0 \leftarrow Y_{1-b}.W$
10: $W \leftarrow X_0 - X_1$
11: $Y_1 \leftarrow C.W - Y_1$

Algorithme 45 EAC-Mult, *Multiplication scalaire avec une EAC*

Entrée(s) : $c = (c_1, \ldots, c_n) \in \mathcal{M}_n$ et $P, Q \in E(\mathbb{F}_p)$, tels que : $Q = \phi(P) = \lambda P$, avec $Z_P = Z_Q = 1$.

Sortie : $Q = \chi_{1,\lambda}(c)P$

1: **for** $i = 1$ **to** n **do**
2: ZADDb(c_i)
3: **end for**
4: ZADDb(1)
5: **return** Q

5.3 Analyse et implémentation

L'algorithme 45 est régulier et utilise des courbes munies d'un endomorphisme efficace. L'élément de comparaison le plus pertinent est donc la méthode GLV régulière. Les comparaisons seront effectuées pour deux types de courbes : les courbes de Weierstrass courtes et les courbes "twisted" Edwards.

Pour illustrer la pertinence de l'approche présentée ici, nous considérons le contexte des objets mobiles, où la quantité de ressources est en général limitée. Nous avons effectué plusieurs implémentations, dont les codes sources et résultats d'exécutions sont disponibles sur `GitHub` :

`https://github.com/eacElliptic`.

Les caractéristiques détaillées des plates-formes (un smartphone Android et un ordinateur x64) utilisées pour nos tests, ainsi que les différents outils auxiliaires sont listés dans l'annexe E.1.

On notera $\mathcal{M}_n$ le coût d'une multiplication de deux entiers de n bits modulo un nombre premier de n bits. On notera également $\mathcal{S}_n$ le coût du carré d'un

entier de n bits modulo un nombre premier de n bits. La méthode GLV ainsi que celle présentée plus haut requièrent le calcul de $\phi(P)$, le coût de ce calcul ne sera donc pas pris en compte dans les comparaisons.

D'après la discussion effectuée dans la section 5.2.1, pour garantir un niveau de sécurité de $\ell/2$ bits en utilisant l'algorithme 5.2, il est nécessaire que le corps de base de la courbe soit de taille 1.4ℓ (au moins). Ce choix assure que cet algorithme permettra de calculer 2^ℓ points différents avec l'ensemble $\mathcal{M}_\ell$, en partant du couple $(P, \phi(P))$. Pour la méthode GLV régulière, il faut un corps de taille ℓ au moins.

Dans la suite de cette section, nous supposerons donc un corps de base de taille ℓ bits pour la méthode GLV (i.e. $\boldsymbol{\lceil \log_2(p) \rceil = \ell}$) et un corps de base de taille $t = 1.4\ell$ pour la méthode basée sur les EAC (i.e. $\boldsymbol{t = \lceil \log_2(p) \rceil = 1.4\ell}$).

5.3.1 Comparaison à la méthode SGLV pour des courbes de Weierstrass courtes

Dans la section 4.2.3.1, nous avons vu deux exemples particulièrement intéressants d'endomorphismes efficaces, où le calcul de $\phi(P)$ se fait avec seulement une multiplication dans $\mathbb{F}_p$. Dans cette partie, nous nous focalisons sur l'exemple 4.6 où l'équation de la courbe $E(\mathbb{F}_p)$ est telle que :

$$E(\mathbb{F}_p) : y^2 = x^3 + b.$$

Le paramètre a de la courbe est donc égal à zéro. Le tableau 5.2 rappelle le coût de la méthode GLV régulière (algorithme 43, désignée par W-SGLV) et celui de la méthode basée sur les EAC (algorithme 45, désignée par EAC-Mult). Le tableau 5.3 donne quelques exemples de coûts pour certains niveaux de sécurité.

Méthode	Coût
EAC-Mult	$(\ell+1)(5\mathcal{M}_t + 2\mathcal{S}_t)$
W-SGLV	$\ell/2 \times (9\mathcal{M}_\ell + 9\mathcal{S}_\ell) + (7\mathcal{M}_\ell + 4\mathcal{S}_\ell)$

TABLE 5.2 – Coûts de certaines méthodes de multiplication scalaire, avec $\ell/2$ bits comme niveau de sécurité.

Méthode	96 bits de sécurité	128 bits de sécurité	192 bits de sécurité
EAC-Mult	$965\mathcal{M}_{269} + 386\mathcal{S}_{269}$	$1285\mathcal{M}_{358} + 514\mathcal{S}_{358}$	$1925\mathcal{M}_{536} + 770\mathcal{S}_{536}$
W-SGLV	$864\mathcal{M}_{192} + 864\mathcal{S}_{192}$	$1152\mathcal{M}_{256} + 1152\mathcal{S}_{256}$	$1728\mathcal{M}_{384} + 1728\mathcal{S}_{384}$

TABLE 5.3 – Exemples de coûts pour des niveaux de sécurité de 96, 128 et 192 bits.

5.3.1.1 Comparaison des coûts théoriques

Supposons un contexte où une multiplication modulaire coûte autant qu'un carré modulaire (i.e. $\mathcal{M} = \mathcal{S}$). Le tableau 5.2 montre que l'algorithme `EAC-Mult` pourrait être plus performant que l'algorithme `SGLV` dans certains cas. En effet, d'après ce tableau, pour un niveau de sécurité de ℓ' bits, une multiplication scalaire avec l'algorithme `EAC-Mult` requiert $14\ell' + 7$ multiplications modulaires d'entiers de taille t bits (avec $t \geqslant 2.8\ell'$). La même opération avec l'algorithme `SGLV` nécessite $18\ell' + 11$ multiplications modulaires d'entiers de taille $2\ell'$ bits. Ainsi, `EAC-Mult` devrait être plus performant que `SGLV` dès que :

$$\mathcal{M}_t < \frac{18\ell'}{14\ell'}\mathcal{M}_{2\ell'} . \tag{5.3}$$

Ce qui implique que $\mathcal{M}_t < 1.29\mathcal{M}_{2\ell'}$.

5.3.1.2 De la théorie à la pratique

Supposons que $t \simeq 2.8\ell'$. D'un point de vue théorique, le ratio $\mathcal{M}_t/\mathcal{M}_{2\ell'}$ devrait être proche de $(1.4)^2$ puisque la multiplication modulaire a un coût quadratique. Ainsi, la condition de l'équation 5.3 ne devrait pas être vérifiée de façon générale. Nous verrons dans ce qui suit que la condition de cette équation peut être satisfaite dans plusieurs cas en pratique. Nous montrons en particulier que c'est le cas lorsque certaines bibliothèques multi-précisions sont utilisées. Notons que :

- la bibliothèque cryptographique populaire OpenSSL repose sur la sous-bibliothèque BIGNUM,
- la bibliothèque de communication sécurisée GnuTLS incluse dans le système d'exploitation Synology Diskstation repose sur la bibliothèque arithmétique GNU Multiple Precision,
- la bibliothèque cryptographique Spongy Castle fournie dans le système d'exploitation Android repose sur la bibliothèque BigInteger de Java.

D'un point de vue pratique, la façon dont la bibliothèque multi-précision gère les grands nombres entiers ainsi que la quantité réelle de mémoire utilisée pour stocker ces nombres doivent être prises en compte. D'une manière générale, chaque bibliothèque multi-précision fournit une procédure de multiplication qui effectue deux taches : elle exécute le calcul réel de la multiplication et effectue également quelques opérations supplémentaires nécessaires pour gérer les grands nombres entiers impliqués dans ce calcul. Notons $\tilde{M}_t$ le coût d'une telle procédure pour deux entiers de taille t bits. La méthode `EAC-Mult` sera plus efficace que la méthode `SGLV` dès que

$$\tilde{M}_t < 1.29\tilde{M}_{2\ell'} . \tag{5.4}$$

Pour identifier des cas où cette inégalité est vérifiée, nous avons fait plusieurs tests pour les multiplications modulaires avec des entiers de tailles 192, 269 ($\simeq 192 \times 1.4$), 256, 358 ($\simeq 256 \times 1.4$) et 384, 538 ($\simeq 384 \times 1.4$) bits. Pour ces

tests, nous avons utilisé la bibliothèque BigInteger de Java fournie dans le kit de développement logiciel (SDK) d'Android, la bibliothèque BigInteger de Java fournie par Oracle dans Java SE pour les plates-formes x64 et la bibliothèque arithmétique GNU Multiple Precision.

Le tableau 5.4 donne les ratios du temps d'exécution pour 2^{21} multiplications modulaires entre des entiers de taille $1.4t$ bits et des entiers de taille t bits. Les cas qui vérifient l'équation 5.4 (et donc favorables à l'algorithme `EAC-Mult`) apparaissent en gras. Le rapport entre les entiers de 538 bits et les entiers de 384 bits est d'environ $1,9$ pour la procédure de multiplication sur notre plate-forme Android. Ceci est dû au fait que la bibliothèque BigInteger gère différemment les entiers de taille inférieure à 512 et les entiers de taille supérieure ou égale à 512 bits.

Dans le tableau 5.4, $\tau_\times$, $\tau_\%$ et τ_* désignent respectivement le ratio de la multiplication des entiers, le ratio de la réduction modulaire et le ratio de la multiplication modulaire.

t (bits)	$\simeq 1.4t$	$\tau_\times$	$\tau_\%$	τ_*
192	269	1,04095	1,05315	**1,04927**
256	358	1,00435	1,08456	**1,05934**
384	538	1,91747	1,12702	1,36432
BigInteger d'Android				
t (bits)	$\simeq 1.4t$	$\tau_\times$	$\tau_\%$	τ_*
192	269	1,75618	1,64688	1,66882
256	358	1,83208	1,80546	1,81148
384	538	1,75453	1,89138	1,85883
BigInteger de Java SE				
t (bits)	$\simeq 1.4t$	$\tau_\times$	$\tau_\%$	τ_*
192	269	1.15566	1.32298	**1.25854**
256	358	1.17041	1.32226	**1.26592**
384	538	1.37065	1.55186	1.48633
Gnu MP				

TABLE 5.4 – Ratios entre les temps d'exécution des opérations sur des entiers de taille $1.4t$ et ceux des opérations sur des entiers de taille t bits.

Depuis 2014, le système d'exploitation Android utilise Spongy Castle une version réduite de la bibliothèque Bouncy Castle pour le langage Java. Cette dernière fournit une API de cryptographie légère. Spongy castle utilise la classe BigInteger pour les opérations sur les grands entiers. Cette bibliothèque fait également partie de Java SE fournie par Oracle, mais elle diffère de la version fournie dans Android pour les calculs arithmétiques de bas niveau. La classe BigInteger de Java SE est écrite en Java pur, tandis que celle de Spongy Castle (utilisée par Android) fait des appels à la bibliothèque BIGNUM qui est écrite en C et utilisée dans OpenSSL.

Aucune des bibliothèques que nous avons utilisées ne fournit une méthode spécifique pour le carré modulaire. Ce qui s'inscrit bien dans le contexte de notre

étude théorique, avec une multiplication modulaire qui coûte autant qu'un carré modulaire.

5.3.1.3 Résultats obtenus

Pour atteindre le même niveau de sécurité que la méthode `SGLV`, la conclusion de la section 5.2.1 suggère qu'on travaille avec des entiers qui sont environ 1.4 fois plus gros. On peut observer dans le tableau 5.4 que, par rapport à une multiplication modulaire d'entiers de taille t bits, une multiplication modulaire d'entiers de taille $1.4t$ bits est :

- environ 1.05 fois plus lente sur Android, pour $t \in \{192, 256\}$,
- environ 1.26 fois plus lente sur une plate-forme x64 avec Gnu MP, pour $t \in \{192, 256\}$,
- au moins $1,7$ fois plus lente sur une plate-forme x6 avec Java SE, pour $t \in \{192, 256, 384\}$.

On est donc loin du ratio théorique qui est $(1.4)^2$. Dans ce qui suit, nous expliquons brièvement pourquoi.

Lors de l'utilisation de bibliothèques de multi-précision, deux facteurs déterminent le coût d'une multiplication modulaire : l'arithmétique (dont le coût augmente de façon quadratique) et la gestion de mémoire (dont le coût augmente linéairement). Pour comprendre le coût total d'une telle opération, il faut considérer deux cas : l'arithmétique et et la gestion de mémoire sont implémentées dans le même langage ou chacune d'entre elles est implémentée dans un langage spécifique. Dans le premier cas, le coût dépend fortement du coût relatif du calcul et de l'allocation de la mémoire dans le langage correspondant. Dans le second cas, les vitesses relatives des langages de programmation utilisées peuvent avoir plus d'impact sur les performances globales que sur le coût des opérations elles-mêmes. Par exemple, dans la bibliothèque Java BigInteger fournie par Android, l'arithmétique est effectuée en utilisant une bibliothèque C (BIGNUM). Bien que cette arithmétique soit quadratique, Java est beaucoup plus lent que le C. Ainsi, la gestion de la mémoire dans ce cas est l'opération la plus coûteuse. Pour ce qui est de la classe Big Integer de Java fournie par Oracle pour une plate-forme x64, l'arithmétique et la mémoire sont gérées en Java. Pour les tailles de données manipulées, la gestion de la mémoire est plus performante que la multiplication. Pour Gnu MP, l'arithmétique est plus performante que la gestion de la mémoire pour les tailles de clé utilisées dans le cadre de la cryptographie sur les courbes elliptiques.

Comme illustration, nous avons utilisé le profileur inclus dans le kit de développement de logiciel pour Android afin d'obtenir l'anatomie d'une multiplication modulaire entre deux entiers de 256 bits en utilisant la classe BigInteger (voir Figure 5.1). Avec notre implémentation, nous avons effectué le calcul de 2^{17} multiplications modulaires de deux entiers aléatoires de 256 bits, modulo un entier premier aléatoire de 256 bits. Pour effectuer une telle multiplication, la méthode utilisée fait appel aux méthodes *multiply* et *mod* de la classe BigInteger qui, à leurs tours, invoquent d'autres méthodes internes. Pour chacune d'elles, le ratio du temps d'exécution de chaque appel ainsi que le ratio du temps

d'exécution pour exécuter les instructions de la méthode elle-même (définie par "Self" dans la figure) sont donnés.

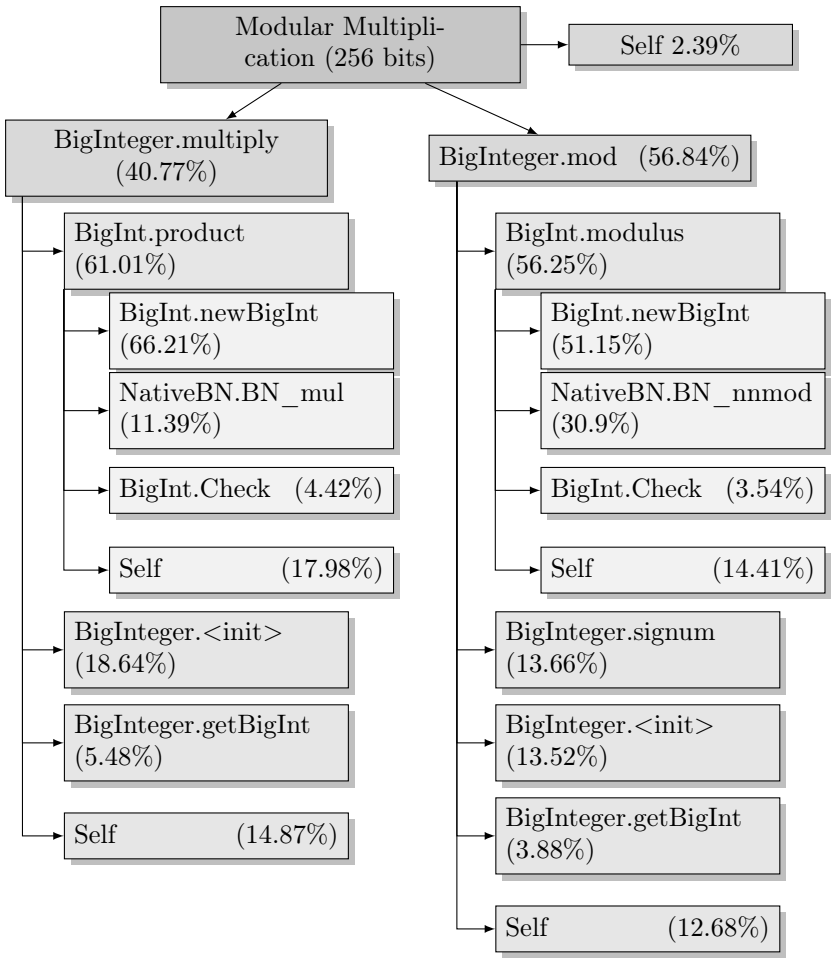


FIGURE 5.1 – Anatomie, obtenue à partir du profiler d'Android SDK, de la multiplication modulaire pour des entiers de 256 bits, en utilisant la classe BigInteger de Java fournie avec le système d'exploitation Android.

Il s'avère que la multiplication (*NativeBN.BNBN_mul*) et la réduction modulaire réelles(*NativeBN.BN_nnmod*) représentent respectivement environ 2,8% et 9,9% de l'ensemble du processus. Le reste du temps est passé entre l'allocation de la mémoire (*BigInt.newBigInt*) et la gestion des objets Java utilisés pour stocker les grands entiers ; voir la figure 5.2 où nous avons isolé le temps passé pour l'allocation et l'exécution réelle du temps utilisé pour les autres traitements.

Lorsque la taille des entiers est comprise entre 192 et 384 bits, le temps consacré au calcul effectif d'une multiplication modulaire prend au maximum 14% du temps total d'exécution. Cela s'étend jusqu'à 20% au maximum pour les entiers de 512 à 717 bits. Ainsi, pour des taille des clés cryptographiques, la plus grande partie du temps est passé entre les allocations de mémoire et la gestion des objets Java impliqués dans le processus de calcul. D'après nos résultats expérimentaux, ce temps est presque similaire pour les entiers de t

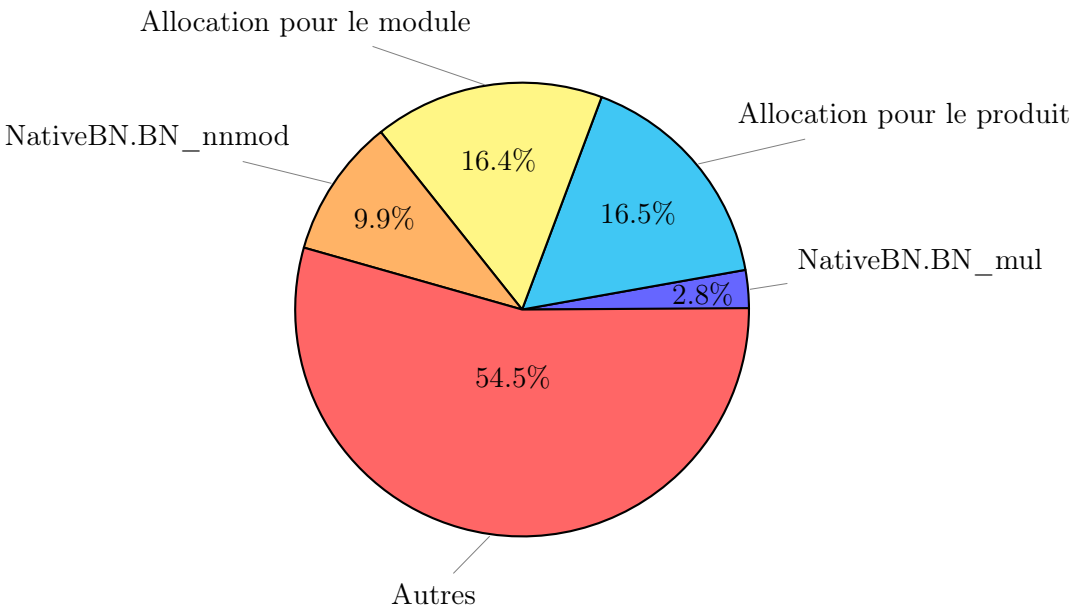


FIGURE 5.2 – Répartition du temps d'exécution des différentes opérations impliquées dans la multiplication modulaire d'entiers 256 bits sur une plate-forme Android

bits et $1.4t$ bits (voir le tableau 5.5). Ceci explique la différence importante entre le ratio théorique de $(1, 4)^2$ et le ratio observé.

t (bits)	$\simeq 1.4t$	Ratio de temps d'exécution pour la gestion de la mémoire et des objets
192	269	1,00564
256	358	0,99838
384	538	1,00485

TABLE 5.5 – Rapport du temps d'exécution pour l'allocation et la gestion de la mémoire entre les objets BigInteger de t bits et les objets BigInteger de $1.4t$ bits.

En utilisant les profileurs de GNU C et de Netbeans pour Java, nous avons réalisé une étude similaire pour obtenir l'anatomie de la multiplication modulaire pour une plate-forme x64 (voir les figures E.1 et E.2 en annexe E.2). Il s'avère que pour une implémentation Java, on ne peut pas espérer un algorithme compétitif. En effet, le calcul réel de la multiplication est écrit en Java pur et son temps d'exécution est une partie non-négligeable de la procédure `multiply`.

En résumé, les résultats du tableau 5.4 montrent qu'il existe des contextes pratiques où le temps d'exécution d'une multiplication modulaire entre les entiers utilisés dans la méthode de multiplication scalaire basée EAC satisfait l'équation 5.4.

5.3.1.4 Implémentations pratiques

Pour illustrer ce qui a été dit plus haut, nous avons implémenté sur un appareil Android et sur une plate-forme x64 (utilisant Gnu MP) les méthodes `SGLV` et `EAC-Mult`.

On peut remarquer que l'algorithme de multiplication scalaire basé sur les EAC n'a besoin que de quelques lignes de code (voir les algorithmes 44 et 45). Tous les points sont en coordonnées jacobiennes et seule la formule d'addition `ZADDb` doit être implémentée. Pour la méthode `SGLV`, deux étapes préliminaires doivent être implémentées : une étape de pré-calcul qui divise k en k_1 et k_2 (ligne 1, algorithme 43) et un processus d'encodage pour convertir k_1 et k_2 dans la représentation GLV-SAC (ligne 6, algorithme 43). En outre, il est nécessaire d'implémenter le formule d'addition mixte (coordonnées affines-jacobiennes), ainsi qu'une formule de doublement.

Nous avons implémenté la méthode `SGLV` pour la courbe $E(\mathbb{F}_p) : y^2 = x^3 + 5$, avec $p = 2^{256} - 1539$, et notre schéma basé sur les EAC pour la courbe $E(\mathbb{F}_p) : y^2 = x^3 + 17$, avec $p = 2^{358} - 36855$. Ces deux ensembles de paramètres garantissent un niveau de sécurité de 128 bits (voir tableau 5.1). L'endomorphisme ϕ correspondant satisfait $\phi^2 + \phi + 1 = 0$. Il s'agit de l'application $(x, y) \mapsto (\beta x, y)$, où β est un élément d'ordre 3.

Pour comparer ces méthodes, nous avons utilisé la moyenne des temps d'exécution de 2^{17} de multiplications scalaires. Le tableau 5.6 donne les ratios correspondants. Dans ce tableau, W-SGLV fait référence à la méthode `SGLV` pour des courbes de Weierstrass. Les cas favorables à la méthode `EAC-Mult` apparaissent en gras. Comme mentionné plus haut, le calcul de kP avec la méthode `SGLV` nécessite la décomposition de k en deux entiers k_1 et k_2 plus petits. Cette étape est souvent négligée dans le coût du calcul de kP. Cependant, dans certains contextes, k et P peuvent tous deux changer et donc cette étape peut ralentir l'exécution de la méthode `SGLV`.

Au lieu de choisir aléatoirement l'entier k, une autre approche consiste à générer aléatoirement k_1 et k_2, comme suggéré et justifié dans [CHS14]. Notons que dans ce cas, la recomposition de k à partir de k_1 et k_2 n'est plus injective. Dans le tableau 5.6, nous avons fait des comparaisons avec les deux approches pour l'algorithme `SGLV` : l'une avec l'encodage de k, et l'autre sans encodage.

Dans ce tableau, on peut observer que sur la plate-forme Android, la méthode `EAC-Mult` est significativement plus performante que la méthode `SGLV`. En effet, notre schéma est au moins 25% plus rapide que la version régulière de GLV. Pour la plate-forme x64, notre temps d'exécution est de 2% inférieur à celui de la méthode `SGLV`. Le gain avec l'approche basée sur les EAC est encore plus significatif si on ne s'intéresse qu'aux systèmes utilisant seulement la coordonnées x, voir [CHS14].

5.3.2 Comparaison à la méthode SGLV pour des courbes "twisted" Edwards

Ted127-glv4 [FHLS15] et FourQ [CL15] sont les deux algorithmes de multiplication scalaire les plus efficaces (et sûrs). Ils s'appuient sur l'algorithme

Méthodes — Plateforme	Android
EAC-Mult /W-SGLV (sans encodage)	**0.75**
EAC-Mult / W-SGLV (avec encodage)	**0.73**

Méthodes — Plateforme	Gnu MP, x64
EAC-Mult / W-SGLV (sans encodage)	**0.98**
EAC-Mult / W-SGLV (avec encodage)	**0.96**

Méthodes — Plateforme	Android (x-only)
EAC-Mult / W-SGLV (sans encodage)	**0.69**
EAC-Mult / W-SGLV (avec encodage)	**0.67**

Méthodes — Plateforme	Gnu MP, x64 (x-only)
EAC-Mult / W-SGLV (sans encodage)	**0.88**
EAC-Mult / W-SGLV (avec encodage)	**0.86**

TABLE 5.6 – Ratio entre le temps d'exécution de la méthode `EAC-Mult` et celui de la méthode `SGLV`, pour un niveau de sécurité de 128 bits, sur différentes plateformes.

GLV à quatre dimensions et tirent parti de deux endomorphismes efficaces ainsi que d'un système de coordonnées étendues sur les courbes "twisted" Edwards [HWCD08]. Ce système de coordonnées fournit les formules d'addition de points les plus rapides pour des courbes basées sur des corps de grandes caractéristiques. Cependant, certains points à pré-calculer doivent être stockés pour des opérations de lecture lors de la multiplication scalaire. Par exemple, 512 octets de mémoire sont nécessaires pour stocker 8 points dans Ted127-glv4.

Une comparaison brute de notre approche avec ces méthodes ne serait pas pertinente. En effet, d'une part, il est certain que les méthodes utilisant deux endomorphismes sont plus rapides. D'autre part, notre contexte d'implémentation est très différent (arithmétique sur $\mathbb{F}_p$, faible consommation mémoire et courbes avec un endomorphisme). Cependant, il semble équitable de profiter du système de coordonnées des courbes "twisted" Edwards pour comparer notre approche à la méthode `SGLV` sur les courbes munies d'un endomorphisme, qui correspond à notre contexte.

5.3.2.1 Comparaisons aux coûts théoriques

Dans le système de coordonnées "twisted" Edwards étendu, un point est représenté par quatre coordonnées (X, Y, T, Z) avec $T = XY/Z$. En utilisant l'algorithme 43, chaque doublement de point est suivi d'une addition de points. Les meilleures performances pour cette séquence d'opérations sont obtenues en combinant les coordonnées "twisted" Edwards standard avec les coordonnées "twisted" Edwards étendues (comme expliqué dans [HWCD08]). Cela conduit à un coût de $10\mathcal{M}_\ell + 4\mathcal{S}_\ell$ par bit du scalaire, pour un niveau de sécurité de $\ell/2$ bits (voir les tableaux 5.7 et 5.8).

Méthode	Coût
EAC-Mult	$(\ell+1)(5\mathcal{M}_t + 2\mathcal{S}_t)$
TED-SGLV	$\ell/2 \times (10\mathcal{M}_\ell + 4\mathcal{S}_\ell)$

TABLE 5.7 – Analyse du coût des méthodes `EAC-Mult` et `TED-SGLV`, pour un niveau de sécurité de $\ell/2$ bits ($t \simeq 1.4\ell$).

Méthode	96 bits de sécurité	128 bits de sécurité	192 bits de sécurité
EAC-Mult	$965\mathcal{M}_{269} + 386\mathcal{S}_{269}$	$1285\mathcal{M}_{358} + 514\mathcal{S}_{358}$	$1925\mathcal{M}_{536} + 770\mathcal{S}_{536}$
TED-SGLV	$960\mathcal{M}_{192} + 384\mathcal{S}_{192}$	$1280\mathcal{M}_{256} + 512\mathcal{S}_{256}$	$1920\mathcal{M}_{384} + 768\mathcal{S}_{384}$

TABLE 5.8 – Coût théorique de la multiplication scalaire pour un niveau de sécurité donné.

Il ressort de cette approche théorique que la méthode avec les EAC ne peut pas faire mieux que la méthode `SGLV` en utilisant le système de coordonnées étendu sur des courbes "twisted" Edwards. Cependant, les résultats expérimentaux (voir tableau 5.9) révèlent, une fois de plus, qu'il faut prendre en compte tous les calculs afin d'évaluer le coût réel d'un algorithme multiplication scalaire. Les résultats du tableau 5.9 ont été obtenus en implémentant la méthode `SGLV` sur la courbe "twisted" Edwards : $E(\mathbb{F}_p) : -x^2 + y^2 = 1 + x^2y^2$, avec $p = 2^{256} - 43443$. L'ordre de la courbe est de $8.h$, où h est un nombre premier. L'endomorphisme ϕ correspondant satisfait $\phi^2 + 1 = 0$. Il correspond à l'application $(x, y) \mapsto (\alpha x, 1/y)$, où α est un élément d'ordre 4 [LWG+15]. Les paramètres pour la méthode `EAC-Mult` sont ceux de la section 5.3.1.4. Afin de comparer ces deux méthodes, nous avons effectué une moyenne de 2^{17} multiplications scalaires.

Penchons nous sur les résultats obtenus sur la plateforme Android. D'une part, dans le tableau 5.4, on peut observer que le coût d'une multiplication modulaire d'entiers de 358 bits est à peine supérieure à celui d'une multiplication modulaire d'entiers de 256 bits. D'autre part, la procédure `ZADDb` comporte précisément 7 multiplications modulaires (en supposant le même coût pour l'élévation au carré et la multiplication) et 7 des additions modulaires. L'addition en coordonnées mixtes pour les courbes "twisted" Edwards [HWCD08] coûte précisément 14 multiplications modulaires, 14 additions modulaires, 1 multiplication par 2 et en moyenne le calcul d'un opposé modulaire :

1. un doublement de point nécessite 8 multiplications modulaires, 6 additions modulaires et une multiplication par la constante de courbe a, qui dans notre cas est égale à -1,
2. une addition de points nécessite 6 multiplications modulaires, 8 additions modulaires et 2 multiplications par 2 [HWCD08],
3. une fois sur deux, l'opposé de l'un des deux points affines P ou $P + \phi(P)$ doit être calculé.

La multiplication par 2 peut être réalisée en utilisant l'une des méthodes

Méthodes — Plateforme	Android
EAC-Mult / TED-SGLV (sans encodage)	**0.95**
EAC-Mult / TED-SGLV (avec encodage)	**0.93**

Méthodes — Plateforme	Gnu MP, x64
EAC-Mult / TED-SGLV (sans encodage)	1.41
EAC-Mult / TED-SGLV (avec encodage)	1.39

Méthodes — Plateforme	Android (x-only)
EAC-Mult / TED-SGLV (sans encodage)	**0.84**
EAC-Mult / TED-SGLV (avec encodage)	**0.82**

Méthodes — Plateforme	Gnu MP, x64 (x-only)
EAC-Mult / TED-SGLV (sans encodage)	1.21
EAC-Mult / TED-SGLV (avec encodage)	1.19

TABLE 5.9 – Ratio entre le temps d'exécution de la méthode `EAC-Mult` et celui de la méthode `TED-SGLV`, pour un niveau de sécurité de 128 bits, sur différentes plateformes.

`BigInteger.shiftleft()` ou `BigInteger.add()`. Les résultats expérimentaux ont montré que les temps d'exécutions de ces deux méthodes sont presque équivalents. Ces deux méthodes allouent un nouvel objet de type `BigInteger`, et comme nous l'avons déjà expliqué dans la section 5.3.1.3, la plus importante partie du temps d'exécution de ces opérations provient l'allocation mémoire. De même, le calcul d'un opposé modulaire peut être fait en utilisant soit la méthode `BigInteger.neg()`, soit la méthode `BigInteger.subtract()`. Pour les mêmes raisons, ces deux opérations sont presque équivalentes.

Le tableau 5.10 résume le coût des méthodes `EAC-Mult` et `TED-SGLV`, pour un niveau de sécurité de $\ell/2$ bits (D_ℓ correspond à la multiplication par 2 d'un entier de ℓ bits et N_ℓ l'opposé modulaire d'un entier de ℓ bits). On peut déduire de ce tableau que si le coût de $\ell(3D_\ell/2 + N_\ell)$ est supérieur au coût de $7\mathcal{M}_t + 7\mathcal{A}_t$, alors la méthode `EAC-Mult` devient plus efficace que la méthode `TED-SGLV`. En d'autres termes, nous devons comparer le temps d'allocation de 14 objets `BigInteger` utilisés pour calculer $7\mathcal{M}_t+7\mathcal{A}_t$, avec le temps d'allocation de $5\ell/2$ objets `BigInteger` utilisés pour le calcul $\ell(3D_\ell/2 + N_\ell)$.

Méthode	Coût
EAC-Mult	$(\ell + 1)(7\mathcal{M}_t + 7\mathcal{A}_t)$
TED-SGLV	$\ell/2 \times (14\mathcal{M}_\ell + 14\mathcal{A}_\ell + 3D_\ell + 2N_\ell)$

TABLE 5.10 – Analyse du coût des méthodes `EAC-Mult` et `TED-SGLV`, pour un niveau de sécurité de $\ell/2$ bits ($t \simeq 1.4\ell$).

Soit $T_{\text{EAC}-\text{Mult}}$ le temps d'exécution de la méthode `EAC-Mult`. En supposant que le temps d'exécution d'une multiplication (ou d'une addition) modulaire

d'entiers de 256 bits est à peu près le même que celui d'entiers de 358 bits, le coût de la méthode `TED-SGLV` devient :

$$T_{\text{EAC}-\text{Mult}} + \delta(\frac{5 \times 256}{2} - 14),$$

où δ est le temps d'exécution de l'allocation de la mémoire.
Ces allocations sont faites par des appels à la méthode `BigInt.newBigInt()`. Nos expériences ont montré que son temps d'exécution est d'environ 50 microsecondes, ce qui donne un ratio suivant :

$$\frac{T_{\text{EAC}-\text{Mult}}}{T_{\text{EAC}-\text{Mult}} + 31300}.$$

En effectuant 2^{17} multiplications scalaires, nous avons obtenu un temps d'exécution moyen de 693943 microsecondes pour $T_{\text{EAC}-\text{Mult}}$, ce qui donne un ratio d'environ 0.96, résultat qui est conforme à la ligne 1 du tableau 5.9. Notons qu'une même analyse ne peut être faite pour la plate-forme x64, car la différence entre les temps d'exécution des multiplications modulaires est trop importante (voir tableau 5.4).

En résumé, sur une plate-forme Android, la méthode de multiplication scalaire basée sur les EAC est plus intéressante que la méthode `SGLV` (avec un endomorphisme), même en utilisant le système de coordonnées étendu des courbes "twisted" Edwards. Avec les systèmes utilisant la coordonnée x uniquement, la comparaison devient encore plus favorable.

Sur une plate-forme x64, la méthode `EAC-Mult` est plus rapide que la méthode `SGLV`, mais plus lente que la méthode `TED-SGLV`. Cependant, cette dernière nécessite plus de mémoire (voir section 5.3.4). Ainsi, notre méthode fournit un compromis raisonnable entre le temps d'exécution et le stockage des données.

5.3.3 Sécurité

La méthode `EAC-Mult` est basée sur l'algorithme `ZADDb`, qui est à temps constant. Ainsi, elle est intrinsèquement sûre contre les attaques de type SPA. En outre, l'algorithme `ZADDb` est composé d'une séquence fixe d'instructions (un mélange de multiplications et de soustractions dans le corps associé à la courbe) qui ne dépend pas du scalaire k. Ce qui protège la méthode `EAC-Mult` contre plusieurs types d'attaques sur le cache d'instructions, voir [ABG10, YF14].

Cependant, des précautions supplémentaires doivent être prises afin de résister aux attaques décrites dans [BH09, BVdPSY14]. En effet, si l'on considère la table construite à partir des points à pré-calculer, chaque bit de la clé conduit à une recherche dans cette table pour sélectionner un de ces points. L'indice correspondant est un bit du scalaire qui peut être révélé grâce à une trace de timing du cache. Puisque l'indice sélectionné est lié à la clé, l'attaquant peut déterminer la valeur de la clé. Notons que la méthode `SGLV` est également vulnérable à ces attaques.

Comme contre-mesure, Käsper [KÏ2] propose de parcourir toute la table pour chaque opération de recherche dans cette table et d'utiliser certaines fonctions pour calculer l'indice correct. Cela signifie que la table entière est chargée dans le cache. Dans la procédure `ZADDb`, on a besoin d'une table de deux points, et à chaque itération, ces deux points sont utilisés. Par conséquent, leurs adresses doivent toujours être chargées dans le cache. Cependant, comme on peut le voir dans la procédure `ZADDb`, certaines opérations dépendent de l'adresse du point utilisé. Par exemple, à la ligne 4, le CPU chargera X_0 ou X_1 pour effectuer la multiplication $X_b.C$. Un attaquant pourrait remplir le cache avec des valeurs pour observer les différences de timing selon l'endroit et le nombre de fois que les opérandes X_0 et X_1 sont utilisés. Dans notre contexte, la contre-mesure décrite dans [KÏ2] peut être simplifiée, car il nous faut lire seulement deux points. Nous proposons l'algorithme 46 qui permute de manière sûre deux points P et Q, selon la valeur du bit b. L'algorithme 47 est une adaptation de l'algorithme `EAC-Mult` pour prendre en compte cette permutation, ce qui permet de se protéger contre ces attaques sur le cache.

Cette contre-mesure ne peut être appliquée telle quelle pour les méthodes `SGLV` et `TED-SGLV`, même si deux points sont pré-calculés pour ces méthodes. En effet, les points P et $P + \phi(P)$ peuvent être sélectionnés en toute sécurité, cependant un fois sur deux, le point $-P$ ou $-(P + \phi(P))$ doit être calculé, ce qui peut entraîner des fuites. Pour éviter cela, les quatre points P, $P + \phi(P)$, $-P$ et $-(P + \phi(P))$ doivent être pré-calculés et stockés. Ensuite, la recherche dans la table doit être implémentée comme suggérée dans [KÏ2].

Algorithme 46 `SafePerm`

Entrée(s) : $P = (X_0, Y_0, Z)$, $Q = (X_1, Y_1, Z)$ et $bit \in \{0, 1\}$
Sortie : Permute de manière sûre P et Q si bit est égal à 0.

1: mask $\leftarrow \neg(-bit)$ $\quad \#(0 \quad \text{ou} \quad -1 \rightarrow 0xFFF...FF)$
2: $X_0 \leftarrow X_0 \oplus X_1$
3: $X_1 \leftarrow (\text{mask} \,\&\, X_0) \oplus X_1$
4: $X_0 \leftarrow X_0 \oplus X_1$
5: $Y_0 \leftarrow Y_0 \oplus Y_1$
6: $Y_1 \leftarrow (\text{mask} \,\&\, Y_0) \oplus Y_1$
7: $Y_0 \leftarrow Y_0 \oplus Y_1$

5.3.4 Consommation mémoire

Le tableau 5.11 résume (pour un niveau de sécurité de 128 bits) le nombre de registres nécessaires pour stocker les coordonnées des différents points impliqués dans le calcul de kP ainsi que le nombre de registres auxiliaires utilisés pour les formules d'addition et de doublement (voir algorithme 44 et [BLa]).

Notons que pour l'algorithme `EAC-Mult`, les deux points initiaux en entrée sont représentés en coordonnées jacobiennes et partagent une coordonnée commune Z. De plus, puisque ces points sont utilisés pour stocker les résultats intermédiaires ainsi que les coordonnées du point calculé (algorithmes 45

Algorithme 47 EAC-Mult, *Multiplication scalaire avec une EAC*

Entrée(s) : $c = (c_1, \ldots, c_n) \in \mathcal{M}_n$ et $P, Q \in E(\mathbb{F}_p)$, tels que : $Q = \phi(P) = \lambda P$, avec $Z_P = Z_Q = 1$.
Sortie : $Q = \chi_{1,\lambda}(c)P$

```
1: for i = 1 to n do
2:     SafePerm(P, Q, c_i)
3:     ZADDb(1)
4: end for
5: ZADDb(1)
6: return Q
```

et 47), l'implémentation de la procédure `ZADDb` ne requiert que deux registres supplémentaires.

Pour la méthode `SGLV`, avec une courbe de Weierstrass, les deux points en entrée sont en coordonnées affines, tandis que les points intermédiaires et le point calculé sont en coordonnées jacobiennes. Pour la méthode `TED-SGLV`, les deux points en entrée sont en coordonnées affines étendues $(x, y, xy, 1)$, et le point intermédiaire Q est en coordonnées standards ou étendues sur la courbe "twisted" Edwards. Dans les deux cas, les résultats intermédiaires ne peuvent pas être stockés dans les coordonnées des points en entrée, car l'algorithme 43 utilise ces points initiaux à chaque itération de leur boucle principale. Quatre (respectivement deux) registres supplémentaires sont nécessaires pour la multiplication scalaire avec la méthode `SGLV` (respectivement `TED-SGLV`).

Méthode	EAC-Mult	EAC-Mult (x-only)
Taille des entiers	358	358
Nombre de registres pour les points	5	4
Registres auxiliaires	2	2
Quantité mémoire (bits)	2506	2148
Quantité mémoire (octets)	314	269
Quantité mémoire (32 bits)	79	68
Quantité mémoire (64 bits)	40	34

Méthode	SGLV	TED-SGLV
Taille des entiers	256	256
Nombre de registres pour les points	7	10
Registres auxiliaires	4	2
Quantité mémoire (bits)	2816	3072
Quantité mémoire (octets)	352	384
Quantité mémoire (32 bits)	88	96
Quantité mémoire (64 bits)	44	48

TABLE 5.11 – Consommation mémoire en bits/octets/mots de 32 bits/mots de 64 bits, pour un niveau de sécurité de 128 bits.

En résumé, par rapport aux méthodes utilisant au moins un endomorphisme, la méthode `EAC-Mult` consomme moins de mémoire. Elle est donc adaptée pour

les environnements à mémoire limitée.

5.4 Conclusion et perspectives

Dans ce chapitre, nous avons présenté un contexte mathématique qui permet d'utiliser les chaînes d'additions euclidiennes pour effectuer la multiplication scalaire sur une courbe munie d'un endomorphisme efficace. Nous avons proposé un algorithme efficace et sûr qui peut être implémenté très facilement. La méthode proposée est moins performante que celles basées sur les courbes possédant deux endomorphismes efficaces (telles que Ted127-glv4 et FourQ), cependant elle est (à notre connaissance) celle qui nécessite la plus petite quantité de mémoire parmi toutes les méthodes de multiplication scalaire basées sur des endomorphismes.

Nos résultats expérimentaux ont montré que, sur une plate-forme Android et pour un niveau de sécurité de 128 bits, la méthode présentée dans ce chapitre fournit de meilleures performances que la version régulière de la célèbre méthode `GLV` (avec un endomorphisme), même lorsque le système de coordonnées étendues sur les courbes "twisted" Edwards (qui est reconnu pour son efficacité) est utilisé.

À niveau de sécurité équivalent, la méthode `EAC-Mult` nécessite des entiers de taille 1.4 fois plus grande que celle requise pour la méthode `SGLV`. Comme perspective, il serait intéressant de se pencher sur une éventuelle diminution de ce facteur. Cela aurait pour conséquence de rendre la méthode présentée dans ce chapitre encore plus intéressante autant pour la performance que la consommation mémoire.

Chapitre 6

Détection de fautes lors de la multiplication scalaire

Sommaire

6.1 Introduction

Dans [PCRS09], Pontarelli et al. abordent le problème de la détection d'erreurs lors de la multiplication scalaire, afin d'éviter l'injection de fautes dans le matériel qui exécute l'algorithme cryptographique sous-jacent. Cette question a déjà été traitée dans plusieurs articles, dont [FRM+08, BMM00, CJ05, BOS06]. Les auteurs de [PCRS09] considèrent un modèle de faute où le résultat d'une opération élémentaire effectuée dans $\mathbb{F}_p$ (le corps de base de la courbe) peut être changé de manière arbitraire. Pour détecter ce type de faute, ils proposent une modification de l'algorithme de multiplication scalaire basé sur les chaînes d'additions euclidiennes (EAC), proposé par Meloni dans [Mel07]. Nous renvoyons le lecteur vers la section 4.2.2.2 où un rappel sur cet algorithme est effectué. Pour rappel, cet algorithme de multiplication scalaire est sûr contre les attaques de type SPA. Des implémentations matérielles sont décrites dans [BCM+07, PVCTM15].

Dans [PCRS09], les auteurs utilisent le système de coordonnées affine pour représenter les points sur les courbes. Ce qui implique le calcul d'une inversion modulaire pour obtenir la somme de deux points. Dans ce chapitre, nous présentons une amélioration de l'algorithme proposé par ces auteurs. Cette amé-

lioration porte à la fois sur les performances, la sûreté ainsi que la capacité de détection de fautes lors de la multiplication scalaire. L'idée principale est de combiner le système de coordonnées jacobien avec les EAC, pour ensuite tirer parti de l'efficace formule d'addition proposée par Goundar et al. dans [GJM+11]. Nous verrons que la méthode présentée est sûre contre un modèle d'attaquant plus avancé et qu'elle est également sûre contre les attaques sur le cache d'instructions et de données. Par rappel, nous avons abordé ces attaques dans la section 5.3.3.

Ce chapitre est organisé comme suit. Nous commençons par présenter la méthode de détection de Pontarelli et al.[PCRS09]. Ensuite, nous présentons une amélioration de leur méthode. Puis, nous effectuons une analyse de notre proposition, suivie d'une comparaison avec la méthode proposée dans [PCRS09]. Enfin, nous terminons par une conclusion et quelques perspectives.

Soient $E(\mathbb{F}_p)$ une courbe elliptique définie sur $\mathbb{F}_p$ et $P \in E(\mathbb{F}_p)$ un point de cette courbe.

6.2 La méthode de Pontarelli et al.

Cette section porte sur la méthode proposée par Pontarelli et al.[PCRS09] pour la détection de fautes lors de la multiplication scalaire en utilisant les EAC. Les auteurs proposent une méthode pour détecter les erreurs qui pourraient être induites par une faute sur des résultats intermédiaires lors de la multiplication scalaire. Plus précisément, il s'agit d'une faute qui pourrait changer de manière arbitraire le résultat d'une opération arithmétique effectuée dans le corps de base $\mathbb{F}_p$ de la courbe. Ce modèle de faute fait abstraction de la manière dont les opérations arithmétiques sont implémentées dans le corps $\mathbb{F}_p$. Nous verrons que les contre-mesures proposées sont également indépendantes de l'implémentation de l'arithmétique dans $\mathbb{F}_p$. Ce qui fournit une grande marge de manoeuvre pour d'éventuelles optimisations ou contre-mesures.

6.2.1 Principe et algorithme

Pour faire face au problème de détection d'erreurs lors de la multiplication scalaire kP, les auteurs suggèrent d'utiliser une EAC c qui calcule le k scalaire. Avec le couple $(P, 2P)$ et l'EAC c, le point kP peut être calculé assez simplement à partir de l'algorithme 41 (voir section 4.2.2.2). Nous rappelons le principe de cet algorithme avec l'exemple qui suit.

Exemple 6.1. Soit $k = 23$. Une EAC calculant k est donné par $c = (10110)$, i.e. $\chi(c) = 23$. Soit $(U_1, U_2) = (P, 2P)$. En lisant c de la gauche vers la droite, on calcule dans l'ordre : $(P, 2P) \xrightarrow{1} (P, 3P) \xrightarrow{0} (3P, 4P) \xrightarrow{1} (3P, 7P) \xrightarrow{1} (3P, 10P) \xrightarrow{0} (10P, 13P)$ et enfin $23P = 10P + 13P$. Remarquons qu'à chaque étape, selon le bit c_i, la couple courant (U_1, U_2) est mise à jour avec $(U_1, U_1 + U_2)$ ou $(U_2, U_1 + U_2)$.

Notons $U_1(i-1)$ et $U_2(i-1)$ les deux points du couple (U_1, U_2), lorsque le bit c_{i-1} de l'EAC a été lu. L'idée principale dans [PCRS09] pour faire face au problème de la détection d'erreurs vient du fait qu'à l'étape i, la valeur $U_1(i-1)+U_2(i-1)$ est toujours calculée et stockée dans $U_2(i)$, i.e. $U_2(i) \leftarrow U_1(i-1)+U_2(i-1)$. En outre, on peut observer que :

- si $c_i = 1$, $U_2(i-1)$ est écarté, et $U_1(i) \leftarrow U_1(i-1)$. Donc $U_2(i) - U_1(i) = U_2(i-1)$.
- si $c_i = 0$, $U_1(i-1)$ est écarté et $U_1(i) \leftarrow U_2(i-1)$. Donc $U_2(i) - U_1(i) = U_1(i-1)$.

Notons $U_C(i)$ la valeur écartée. Au début de l'étape $i+1$, la différence $U_D(i+1) = U_2(i) - U_1(i)$ doit toujours être égale à $U_C(i)$. Pour faire face au problème de détection d'erreurs, un triplet $(U_C(i), U_1(i), U_2(i))$ est utilisé. Au début de l'étape $i+1$, on vérifie que $U_D(i+1) = U_C(i)$, sinon une erreur est générée. Si l'égalité est satisfaite, $U_C(i+1)$ est mis à jour en fonction du bit c_{i+1}. L'ensemble du processus est initialisé à l'aide du triplet $(U_C(0) = P, U_1(0) = P, U_2(0) = 2P)$.

L'algorithme 48 décrit cette méthode. Notons qu'il y a quelques différences avec l'algorithme 2 donné dans [PCRS09]. En effet, il semble que la séquence d'instructions donnée ne corresponde pas à l'idée décrite dans l'article. Dans [PCRS09], l'algorithme commence par $U_1 = 2P$, $U_2 = P$ et $U_C = P$. À la première itération, on a $U_1 = 3P$, puis $U_D = 2P$ et la ligne 4 vérifie si $U_D \neq U_C$. Ce qui est toujours le cas, donc une erreur est toujours détectée. Le problème principal est que U_1 a été mis à jour avant le calcul de U_D. Ceci a également un impact sur les lignes 8 et 10 de l'algorithme. Pour que l'algorithme 2 [PCRS09] soit correct, les mises à jour faites dans ces lignes devraient utiliser la version non modifiée de U_1.

6.2.2 Coût et sûreté

Dans [PCRS09], les points sont représentés dans le système de coordonnées affine. Le calcul de $U_1 + U_2$ nécessite, dans $\mathbb{F}_p$, une inversion ($\mathcal{I}$), deux multiplications ($\mathcal{M}$), une élévation au carré ($\mathcal{S}$) et six additions ($\mathcal{A}$) : $\mathcal{I} + 2\mathcal{M} + \mathcal{S} + 6\mathcal{A}$. Pour calculer $U_1 - U_2$, les auteurs font remarquer que certaines valeurs intermédiaires calculées pour $U_1 + U_2$ peuvent être réutilisées, économisant ainsi une inversion modulaire et deux additions. Le coût total du calcul de $U_1 + U_2$ et $U_1 - U_2$ est donc :

$$\mathcal{I} + 4\mathcal{M} + 2\mathcal{S} + 10\mathcal{A}.$$

Notons que ce coût correspond également à celui de l'algorithme 48, par bit de l'EAC c.

Dans le chapitre précédent, nous avons expliqué qu'il faut éviter les dépendances entre la séquence des instructions et la clé pour se protéger des attaques sur le cache d'instructions. De même, il faut éviter les dépendances entre la lecture des données et la clé pour se protéger des attaques sur le cache de données. La méthode présentée plus haut est sûre contre les attaques de type SPA. Cependant, elle est vulnérable à ces attaques sur le cache. En effet, les instructions "*if*"

Algorithme 48 *Multiplication scalaire avec détection d'erreurs*

Entrée(s) : P, $2P$ et une EAC c de longueur l
Sortie : $(Q = \chi(c)P, 0)$ s'il n'y a pas eu d'erreur ou $(0, 1)$ sinon

1: $(U_C, U_1, U_2) \leftarrow (P, P, 2P)$
2: **for** $i = 1$ **to** l **do**
3: $\quad U_D \leftarrow U_2 - U_1$
4: $\quad$ **if** $U_D \neq U_C$ **then**
5: $\quad\quad$ **return** $(Q = 0, \text{error} = 1)$
6: $\quad$ **end if**
7: $\quad$ **if** $c_i = 1$ **then**
8: $\quad\quad U_1 \leftarrow U_1$
9: $\quad\quad U_C \leftarrow U_2$
10: $\quad\quad U_2 \leftarrow U_1 + U_C = U_1 + U_2$
11: $\quad$ **else**
12: $\quad\quad U_C \leftarrow U_1$
13: $\quad\quad U_1 \leftarrow U_2$
14: $\quad\quad U_2 \leftarrow U_1 + U_C = U_1 + U_2$
15: $\quad$ **end if**
16: **end for**
17: $U_D \leftarrow U_2 - U_1$
18: **if** $U_D \neq U_C$ **then**
19: $\quad$ **return** $(Q = 0, \text{error} = 1)$
20: **end if**
21: $U_C \leftarrow U_2$, $U_2 \leftarrow U_1 + U_C = U_1 + U_2$
22: $Q \leftarrow U_2$
23: $\#\{$l'étape suivante vérifie le dernier résultat de U_2 (i.e. $Q)\}$
24: $U_D \leftarrow U_2 - U_1$
25: **if** $U_D \neq U_C$ **then**
26: $\quad$ **return** $(Q = 0, \text{error} = 1)$
27: **end if**
28: **return** $(Q, \text{error} = 0)$

et "*else*" dans l'algorithme 48 (lignes 7 et 11) dépendent de l'élément c_i de la clé. Pour plus de détails sur ces attaques, voir [ABG10, BVdPSY14, YF14, BH09].

Enfin, il faut également éviter les branchements conditionnels dépendant de la clé, même pour des branches équivalentes, à cause de la prédiction de branche utilisée dans les CPU modernes [OKS06]. Raison supplémentaire pour éviter les "*if*" et "*else*" (liés à la clé) présents dans cet algorithme.

6.3 La nouvelle proposition

Dans cette section, nous présentons une amélioration de la méthode décrite plus haut, pour la détection de faute lors de la multiplication scalaire, en utilisant les EAC. Ensuite, nous expliquons pourquoi celle-ci est sûre contre les attaques sur le cache (d'instructions et de données) et comment elle peut détecter une erreur qui s'est produite lors du calcul des résultats intermédiaires. Notre algorithme utilise l'algorithme `ZADDC` de Goundar et al. [GJM+11], pré-

senté dans le chapitre 4. Nous avons toutefois effectué quelques modifications pour nos besoins. Nous appelons donc notre version `mZADDC` (algorithme 49).

6.3.1 Principe et algorithmes

Pour la suite, nous nous focaliserons sur le système de coordonnées jacobien. Soient $P, Q \in E(\mathbb{F}_p)$ deux points. Dans ce qui suit, $\overline{P}$ et $\overline{Q}$ désignerons respectivement des représentants de (i.e. des points équivalents à) P et Q, qui partagent la même coordonnée Z. Donc, $P \sim \overline{P}$ et $Q \sim \overline{Q}$, avec $Z_{\overline{P}} = Z_{\overline{Q}}$.

Algorithme 49 `mZADDC`

Entrée(s) : $C = (X_C, Y_C, Z_C)$, $P = (X_P, Y_P, Z)$, et $Q = (X_Q, Y_Q, Z)$.
Sortie : Met à jour $(C, P, Q) \leftarrow (\overline{Q}, \overline{P}, \overline{P+Q})$ et retourne 0 si et seulement si $\overline{Q-P} = \overline{C}$.

1: $Z \leftarrow Z.(X_Q - X_P)$ #{calcule de $Z_{\overline{Q-P}} = Z_{\overline{P+Q}} = Z_{\overline{P}} = Z_{\overline{Q}}$}
2: $Z_C \leftarrow Z_C.(X_Q - X_P)$ #{calcule de $Z_{\overline{C}}$}
3: $A \leftarrow (X_Q - X_P)^2$
4: $X_P \leftarrow X_P.A$
5: $X_Q \leftarrow X_Q.A$ #{calcule de $X_{\overline{Q}}$}
6: $X_C \leftarrow X_C.A$ #{calcule de $X_{\overline{C}}$}
7: $B \leftarrow Y_Q - Y_P$
8: $E \leftarrow Y_Q + Y_P$
9: $F \leftarrow X_Q - X_P$
10: $Y_P \leftarrow Y_P.F$
11: $Y_Q \leftarrow Y_Q.F$ #{calcule de $Y_{\overline{Q}}$}
12: $Y_C \leftarrow Y_C.F$ #{calcule de $Y_{\overline{C}}$}
13: $Dx \leftarrow E^2 - X_P - X_Q$ #{calcule de $X_{\overline{Q-P}}$}
14: $Dy \leftarrow E.(X_P - Dx) - Y_P$ #{calcule de $Y_{\overline{Q-P}}$}
15: # {l'instruction suivante teste si $\overline{Q-P} = \overline{C}$}
16: $check \leftarrow (Dx \oplus X_C) \vee (Dy \oplus Y_C) \vee (Z \oplus Z_C)$
17: $X_C \leftarrow X_Q$
18: $Y_C \leftarrow Y_Q$
19: $Z_C \leftarrow Z$
20: $X_Q \leftarrow B^2 - X_P - X_Q$ #{calcule de $X_{\overline{P+Q}}$}
21: $Y_Q \leftarrow B.(X_P - X_Q) - Y_P$ #{calcule de $Y_{\overline{P+Q}}$}
22: **return** *check*

Cet algorithme prend en entrée le triplet (C, P, Q), tel que :

$$P = (X_P, Y_P, Z), Q = (X_Q, Y_Q, Z), C = (X_C, Y_C, Z_C)\,.$$

À l'étape i, `mZADDC` met à jour ses paramètres $P(i-1)$, $Q(i-1)$ et $C(i-1)$ comme suit :

- $P(i)$ est un représentant de $P(i-1)$, i.e. $P(i) = \overline{P(i-1)}$,
- $C(i)$ est un représentant de $Q(i-1)$, i.e. $C(i) = \overline{Q(i-1)}$,
- $Q(i)$ est un représentant de $Q(i-1) + P(i-1)$, i.e. $Q(i) = \overline{Q(i-1) + P(i-1)}$.

Le point $Q(i)$ et la différence $\overline{Q(i-1) - P(i-1)}$ sont calculés en utilisant les formules de `ZADDC`, de plus `mZADDC` vérifie si $\overline{Q(i-1) - P(i-1)} = \overline{C(i-1)}$. Cet algorithme retourne 0 si aucune erreur n'est survenue à l'étape précédente (i.e. $\overline{Q(i-1) - P(i-1)} = \overline{C(i-1)}$) ou un nombre non nul sinon. En effet, à la ligne 16 (de l'algorithme 49) si aucune erreur ne s'est produite, on doit avoir $Dx = X_C$, $Dy = Y_C$ et $Z = Z_C$, et donc $check = 0$.

Notons que dans `mZADDC`, on doit calculer les représentants de Q et C (qui ne sont pas donnés par `ZADDC`). Nous discuterons plus loin du coût supplémentaire de ces opérations.

Soit c une EAC calculant un entier k. À l'étape i de la multiplication scalaire, nous avons un triplet (C, P, Q) et le bit courant c_i est lu :

- si $c_i = 1$, (C, P, Q) doit être mis à jour avec $(\overline{Q}, \overline{P}, \overline{P+Q})$,
- si $c_i = 0$, (C, P, Q) doit être mis à jour avec $(\overline{P}, \overline{Q}, \overline{P+Q})$.

En d'autres termes, la méthode `mZADDC` doit être appelée avec (C, P, Q) ou (C, Q, P). Ce qui conduit à un branchement conditionnel qui dépend de l'EAC c. Comme mentionné plus haut, un tel branchement conditionnel est vulnérable aux attaques sur le cache (de données et d'instructions). Afin d'éviter cette vulnérabilité, nous utilisons l'astuce suivante. Nous appelons toujours `mZADDC` avec (C, P, Q) comme entrées. À partir de la sortie $(\overline{Q}, \overline{P}, \overline{P+Q})$ obtenue, nous appliquons une permutation sûre sur $(\overline{Q}, \overline{P})$ selon la valeur c_i ; nous échangeons $\overline{Q}$ et $\overline{P}$ si $c_i = 0$. Par sûr, il faut comprendre que la permutation est en temps constant, la séquence d'instructions est toujours la même et que les entrées sont toujours chargées dans le même ordre quelle que soit la valeur de c_i. Cette permutation est réalisée avec l'algorithme 46 (ci-dessous). Notons que cet algorithme est déjà présenté dans le chapitre précédent. Ce type de contre-mesure est suggéré et utilisé dans [KÏ12, Bru15].

Algorithme 46 `SafePerm`

Entrée(s) : $P = (X_0, Y_0, Z)$, $Q = (X_1, Y_1, Z)$ et $bit \in \{0, 1\}$
Sortie : Permute de manière sûre P et Q, si bit est égal à 0.

1: mask $\leftarrow \neg(-bit)$ $\quad \#(0 \quad \text{ou} \quad -1 \rightarrow 0xFFF...FF)$
2: $X_0 \leftarrow X_0 \oplus X_1$
3: $X_1 \leftarrow (\text{mask} \,\&\, X_0) \oplus X_1$
4: $X_0 \leftarrow X_0 \oplus X_1$
5: $Y_0 \leftarrow Y_0 \oplus Y_1$
6: $Y_1 \leftarrow (\text{mask} \,\&\, Y_0) \oplus Y_1$
7: $Y_0 \leftarrow Y_0 \oplus Y_1$

Remarquons que le coût de cette contre-mesure est négligeable par rapport à celui des opérations effectuées dans $\mathbb{F}_p$ lors de la multiplication scalaire, car elle ne nécessite que des opérations logiques.

L'algorithme 50 décrit l'ensemble de la procédure, en utilisant les notations de [PCRS09].

Algorithme 50 *Multiplication scalaire avec détection d'erreurs*

Entrée(s) : P, $2P$ tels que $Z_P = Z_{2P}$ et une EAC c de longueur l
Sortie : $(Q = \chi(c)P, 0)$ s'il n'y a pas eu d'erreur ou $(0, 1)$ sinon

1: $(U_C, U_1, U_2) \leftarrow (P, P, 2P)$
2: $check \leftarrow 1$
3: **for** $i = 1$ **to** l **do**
4: $\quad check \leftarrow$ `mZADDC`(U_C, U_1, U_2)
5: $\quad$ **if** $check \neq 0$ **then**
6: $\quad\quad$ **return** $(Q = 0, \text{error} = 1)$;
7: $\quad$ **end if**
8: $\quad$ `SafePerm`(U_1, U_C, c_i)
9: **end for**
10: $check \leftarrow$ `mZADDC`(U_C, U_1, U_2)
11: **if** $check \neq 0$ **then**
12: $\quad$ **return** $(Q = 0, \text{error} = 1)$;
13: **end if**
14: $Q \leftarrow U_2$
15: # {l'étape suivante vérifie le dernier résultat de U_2 (i.e. Q)}.
16: $check \leftarrow$ `mZADDC`(U_C, U_1, U_2)
17: **if** $check \neq 0$ **then**
18: $\quad$ **return** $(Q = 0, \text{error} = 1)$
19: **end if**
20: **return** $(Q, \text{error} = 0)$

6.3.2 Analyse de la solution proposée

Cette section porte sur les performances et la sûreté de la méthode décrite plus haut. Nous effectuons également une comparaison avec la méthode de Pontarelli et al.

6.3.2.1 Sûreté

Pour une longueur d'EAC fixée, on peut voir que l'algorithme 50 exécute toujours les mêmes instructions dans le même ordre et aussi que les données (P, $2P$, U_C, U_1, U_2) sont toujours chargées dans le même ordre (grâce à l'algorithme 46) indépendamment du bit courant de l'EAC. Par conséquent, la méthode proposée est sûre contre les attaques du cache (de données et des instructions).

Rappelons que le modèle de faute décrit dans [PCRS09] permet d'injecter une seule faute afin de modifier de manière arbitraire le résultat d'une opération élémentaire effectuée dans le corps de base $\mathbb{F}_p$. Ainsi, le calcul de $U_1 + U_2$ et $U_2 - U_1$ ainsi que le calcul des représentations équivalentes des points (dans notre cas) peuvent être modifiés.

Notons qu'une erreur dans le calcul du représentant $\overline{U}_C$ de U_C sera détectée dans l'itération **courante**, car elle fera immédiatement que $\overline{U_2 - U_1} \neq \overline{U}_C$. De la même manière, il est facile de voir que toute erreur faite lors du calcul des représentants $\overline{U}_1$ ou $\overline{U}_2$, sera détectée à l'itération **suivante** lors de la vérification de la relation $\overline{U_2 - U_1} = \overline{U}_C$. Ensuite, lors du calcul de $\overline{U_1 + U_2}$, deux

types d'erreurs doivent être considérés :

1. l'erreur envoie $\overline{U_1 + U_2}$ sur un autre point de la courbe elliptique ; ainsi, il existe un point E de la courbe tel que le point calculé est $\tilde{U}_2 = \overline{U_1 + U_2} + E$. À cette étape, nous avons $U_C = \overline{U}_2$. Dans l'itération **suivante**, nous calculons $\overline{\tilde{U}_2 - \overline{U}_1} = \overline{\overline{U}_2 + E} \neq \overline{U}_C$, rendant $check \neq 0$.
2. l'erreur envoie $\overline{U_1 + U_2}$ sur un point qui n'est pas de la courbe. Dans l'itération **suivante**, $\overline{U_1 + U_2} - \overline{U_1}$ ne peut pas être égal à $\overline{U}_C$ (cela reste vrai pour tout représentant de U_C), sinon $\overline{U_1 + U_2} = \overline{U_1} + \overline{U}_C$ devrait être un point de la courbe car il est la somme de deux points qui sont sur la courbe.

Dans [AKS12], les auteurs considèrent un attaquant plus fort qui peut refléter la même erreur E sur différentes valeurs. Un tel attaquant peut contourner la contre-mesure proposée par Pontarelli et al. dans [PCRS09]. En effet, supposons que lors du calcul de la somme de deux points P et Q, un attaquant soit capable d'injecter une faute afin que le résultat soit $P + Q + E$, pour un "point erreur" E. Cet attaquant pourrait contourner la détection d'erreurs décrite dans [PCRS09] de la manière suivante :

1. À l'itération i, lors du calcul de $U_1(i-1) + U_2(i-1)$ (ligne 10 ou 14 de l'algorithme 48), l'attaquant injecte sa faute et le résultat calculé est :

$$U_2(i) = U_1(i-1) + U_2(i-1) + E.$$

   Notons que :

   - si $c_i = 0$, alors $U_C(i) = U_1(i-1)$ et $U_1(i) = U_2(i-1)$,
   - si $c_i = 1$, alors $U_C(i) = U_2(i-1)$ et $U_1(i) = U_1(i-1)$.

2. À l'itération $i+1$, lors du calcul de $U_D = U_2(i) - U_1(i)$, l'attaquant utilise $-E$ au lieu de E et exécute sa méthodologie d'injection de faute pour changer U_D par $U_D - E$. On obtient alors :

$$\begin{aligned} U_D - E &= U_1(i-1) + U_2(i-1) + E - U_1(i) - E \\ &= U_1(i-1) + U_2(i-1) - U_1(i) \\ &= U_C(i). \end{aligned}$$

   L'erreur n'est donc pas détectée.

Un attaquant avec les mêmes capacités ne peut pas contourner la méthode que nous proposons. En effet, à l'itération i nous avons un triplet (U_C, U_1, U_2). Ce triplet est mis à jour pour obtenir $(\overline{U_2}, \overline{U_1}, \overline{U_1 + U_2})$. Supposons que l'attaquant injecte sa faute, alors le triplet devient $(\overline{U_2}, \overline{U_1}, \overline{U_1 + U_2} + E)$. Notons $(U'_2, U'_1, U'_2 + U'_1 + E)$ le nouveau triplet. Dans l'itération suivante, on teste si $\overline{U'_2 + U'_1 + E} - \overline{U'_1} = \overline{U'_2}$. Pour que cette égalité soit vraie, l'attaquant doit injecter une faute lors du calcul de la différence afin d'ajouter $\overline{-E}$. Ainsi, il ne peut pas réutiliser le "point erreur" E et doit calculer un représentant équivalent avec le même coefficient utilisé pour calculer les représentants de U'_1 et U'_2. Ceci est impossible, car dans [AKS12], on suppose que l'attaquant ne peut pas observer les données existantes sur le circuit pendant l'injection de faute et qu'il ne peut

pas non plus observer les données afin de choisir un vecteur d'erreurs approprié. En outre, dans la section 4 de [AKS12] où le modèle d'attaquant est défini, les auteurs supposent que, lorsqu'une erreur est détectée, alors le matériel qui effectue la multiplication scalaire est désactivé ou réinitialise les informations secrètes. Ainsi, l'attaquant ne peut pas essayer plusieurs vecteurs d'erreurs différents pour retrouver le secret. Il n'a donc qu'une seule chance pour réussir à injecter des fautes. Notre méthode est par conséquent sûre contre cette attaque. Notons qu'un tel attaquant est très puissant. En effet, si l'on peut admettre que le résultat d'une opération dans le corps de base puisse être modifié, il semble plus compliqué d'injecter des fautes pour que le résultat corresponde à une addition de points valable sur une courbe.

Pour finir, il est clair que l'algorithme 50 est, tout comme l'algorithme 48, sûr contre les attaques de type SPA.

6.3.2.2 Performances et comparaison

Dans cette partie, nous ne considérons pas le coût de l'addition, qui est négligeable par rapport celui des autres opérations qui sont la multiplication, le carré et l'inversion modulaires.

Dans la section 6.3.1, nous avons mentionné que `mZADDC` utilise la procédure `ZADDC` pour calculer $\overline{P+Q}$ et $\overline{Q-P}$. Le coût total de la méthode `ZADDC` est de $6\mathcal{M}+3\mathcal{S}$. Dans `mZADDC`, nous devons aussi calculer les représentants des entrées C et Q (le représentant de P est déjà calculé avec `ZADDC`). Le calcul de $\overline{C}$ et $\overline{Q}$ peut être effectué en utilisant seulement quatre multiplications. En effet, dans `ZADDC`, les résultats intermédiaires nous donnent déjà les coordonnées X et Z de $\overline{Q}$. Ainsi, nous n'avons plus qu'à calculer la coordonnée Y de $\overline{Q}$ et les trois coordonnées de $\overline{C}$ (lignes 2, 5, 6, 11 et 12 de l'algorithme 49). Le tableau 6.1 donne le coût de notre méthode et celui de la méthode proposée dans [PCRS09], pour chaque bit de l'EAC.

Méthode de Pontarelli et al. [PCRS09]	Notre méthode
$\mathcal{I}+4\mathcal{M}+2\mathcal{S}$	$10\mathcal{M}+3\mathcal{S}$

TABLE 6.1 – Coût par bit des méthodes de multiplication scalaire basées sur les EAC, avec détection d'erreurs.

On peut observer que notre méthode devient plus rapide dès que le coût d'une inversion est supérieur au coût d'un carré et six multiplications, dans le corps de base $\mathbb{F}_p$. À titre d'exemple, nous avons considéré une plate-forme x64 (Core i5-6500 CPU, 3.20GHz, noyau Linux 4.4.0-59), et avons utilisé le benchmark fourni dans la bibliothèque cryptographique MIRACL [Mir], afin d'obtenir le ratio entre le coût d'une inversion et le coût d'une multiplication pour différentes tailles d'entiers. Avec cette bibliothèque, le coût du carré modulaire et de la multiplication modulaire sont presque les mêmes ; nous avons donc considéré que $\mathcal{S}=\mathcal{M}$. Le tableau 6.2 liste les ratios obtenus, donne une estimation du coût de la méthode de Pontarelli et al., du notre (qui ne change pas) et donne

également une estimation du ratio entre le coût de notre méthode et celui de la méthode de Pontarelli et al.. Par exemple, on peut observer dans ce tableau que le temps d'exécution de notre méthode devrait être à peu près 45% de celui de [PCRS09], pour des entiers de taille 384 bits.

Taille des entiers (en bits)	Ratio $\mathcal{I}$ /$\mathcal{M}$	Coût alg. 48 (alg. 2, [PCRS09])	Coût alg. 50 (notre méthode)	Ratio : alg. 50/alg. 48
192	25	31$\mathcal{M}$	13$\mathcal{M}$	42%
256	26	32$\mathcal{M}$	13$\mathcal{M}$	41%
384	23	29$\mathcal{M}$	13$\mathcal{M}$	45%
512	19	25$\mathcal{M}$	13$\mathcal{M}$	52%

TABLE 6.2 – Estimation et comparaison du coût par bit des méthodes de multiplication scalaire, à l'aide de la bibliothèque MIRACL.

6.4 Conclusion et perspectives

Dans ce chapitre, nous avons présenté une version plus sûre et plus rapide de la méthode décrite dans [PCRS09], pour la multiplication scalaire avec détection de faute. En plus d'être sûr contre les attaques de type SPA, l'algorithme présenté dans ce chapitre est résistant aux attaques sur le cache d'instructions et de données. En outre, cet algorithme résiste à des scénarios d'attaques qui contournent le système de détection de fautes proposé dans [PCRS09].

Comme perspective, il serait intéressant d'étudier plus en détail la résistance de la méthode proposée face au modèle d'attaquant décrit dans [AKS12]. En effet, nous avons montré que cette méthode résiste à certaines attaques qui seraient commises par cet attaquant. Cependant, notre preuve ne s'étend pas à toutes les possibilités de ce dernier.

Conclusion générale

Dans cette thèse, nous nous sommes intéressés à deux éléments actuellement incontournables de la cryptographie à clé publique, qui sont l'arithmétique modulaire avec de grands entiers et l'arithmétique des courbes elliptiques.

Concernant l'arithmétique modulaire, nous avons dans le chapitre 1 commencé par faire un tour d'horizon des méthodes pour effectuer de manière efficace les opérations principales qui sont l'addition, la multiplication, l'inversion ainsi que l'exponentiation modulaires. Dans cet état de l'art, nous avons mis l'accent sur la résistance de ces méthodes aux attaques par canaux auxiliaires. En outre, nous nous sommes étendus sur le système de représentation modulaire adapté (AMNS) proposé par Bajard et al. en 2004. Dans ce système, les éléments sont représentés sous forme de polynômes. De cette particularité découlent plusieurs avantages pour les performances et la sûreté des opérations arithmétiques dans ce système. Une autre particularité de ce système est qu'il est redondant. Cela offre plusieurs possibilités pour la protection des opérations cryptographiques, à travers par exemple la randomisation des opérations arithmétiques.

Après l'état de l'art, nous avons dans le chapitre 2 donné dans un premier temps un ensemble complet d'algorithmes pour effectuer, entre autres, les opérations élémentaires de l'arithmétique modulaire ainsi que toutes opérations de conversions indispensables. Ensuite, nous avons présenté un processus de génération d'AMNS pour effectuer de manière efficace ces opérations avec les algorithmes présentés auparavant. Nos résultats d'implémentation ont par ailleurs montré que l'AMNS permet d'effectuer la multiplication modulaire (et de façon triviale l'addition modulaire) plus efficacement que les bibliothèques populaires telles que OpenSSL et Gnu MP ; cela, sans utiliser le potentiel de parallelisation offert par l'AMNS.

Le chapitre 3 traitait de la randomisation des opérations arithmétiques avec l'AMNS. L'objectif était, d'un coté de tirer parti de l'existence de plusieurs AMNS pour un même nombre premier pour randomiser le processus de conversion vers ce système, et d'un autre coté d'exploiter la redondance de l'AMNS pour randomiser à la fois le processus de conversion et la multiplication modulaire dans ce système. Comme application, nous avons vu comment utiliser ces randomisations pour appliquer les contre-mesures suggérées dans la littérature pour protéger la multiplication scalaire sur les courbes elliptiques des attaques de type DPA. En outre, de l'analyse que nous avons effectuée ensuite, il est ressorti que la randomisation avec l'AMNS protège naturellement

la multiplication scalaire contre des attaques pour lesquelles il est nécessaire de prendre des mesures supplémentaires lorsque la représentation binaire classique est utilisée pour représenter les coordonnées des points. Nous avons également comparé différentes stratégies de randomisation de la multiplication scalaire sur un exemple de courbe.

Pour ce qui est de l'arithmétique des courbes elliptiques, nous avons dans le chapitre 4 commencé par un état de l'art dans lequel un ensemble de formules d'addition et de méthodes pour la multiplication scalaire a été présenté. Nous avons en particulier abordé l'utilisation des chaînes d'additions euclidiennes pour effectuer la multiplication scalaire et aussi rappelé la méthode GLV qui permet d'effectuer efficacement la même opération.

Dans le chapitre 5, nous avons présenté le travail de généralisation de l'utilisation des chaînes d'additions euclidiennes pour effectuer efficacement la multiplication scalaire, lorsque le point de base n'est pas fixé. Cette généralisation a été faite en utilisant les courbes munies d'un endomorphisme efficace. Nos résultats expérimentaux ont montré que, pour un niveau de sécurité de 128 bits, la méthode proposée offre de meilleures performances que la version régulière de la méthode GLV, sur les plates-formes Android et x64 avec Gnu MP, pour des courbes de Weierstrass courtes. Pour les courbes "twisted" Edwards, nous avons également obtenu que la méthode proposée est plus intéressante que la version régulière de la méthode GLV, pour un niveau de sécurité de 128 bits sur les plates-formes Android.

Enfin, le chapitre 6 portait sur la détection de fautes lors de la multiplication scalaire. Nous avons présenté une amélioration de la méthode proposée par Pontarelli et al. en 2009. Cette amélioration porte à la fois sur les performances et la sûreté de la multiplication scalaire. Plus précisément, l'algorithme proposé résiste à un attaquant plus fort que celui considéré par Pontarelli et al.. En outre, selon l'estimation du ratio de l'inversion modulaire sur la multiplication modulaire (fourni par la bibliothèque cryptographique MIRACL), le temps d'exécution de l'algorithme que nous proposons varie entre 41 et 52% de celui de l'algorithme proposé par Pontarelli et al., pour des tailles d'entiers entre 192 et 512 bits.

Les travaux présentés dans cette thèse offrent plusieurs perspectives. À la fin de chacun des chapitres (2, 3, 5 et 6) de contributions, nous en avons présenté quelques-unes. Pour les AMNS par exemple, il serait intéressant de se pencher sur des implémentations qui tireraient parti du potentiel de parallelisation de ce système, car comme nous l'avons expliqué dans les chapitres 1 et 2, ce système offre plusieurs possibilités de parallelisation à plusieurs niveaux.

Pour l'arithmétique des courbes elliptiques, il serait intéressant d'une part de se pencher sur une éventuelle réduction de la taille minimale requise pour assurer la consistance de la méthode proposée dans le chapitre 5. D'autre part, une étude plus approfondie de la résistance de l'algorithme proposé dans le chapitre 6 face au modèle d'attaquant décrit par Akdemir et al. dans [AKS12] est envisagée.

Annexe

Annexe A

Quelques notions

Sommaire

A.1 Notions sur le résultant de polynômes

Soit $\mathcal{A}$ un ensemble. Lorsque $\mathcal{A}$ n'est pas un corps commutatif, les résultats essentiels de l'arithmétique (tels que l'identité de Bézout, le lemme d'Euclide et le théorème fondamental de l'arithmétique) ne sont pas toujours vérifiés dans $\mathcal{A}[X]$. Dans ce cas, savoir par exemple si deux polynômes A et B ont des facteurs communs (ou racines communes) peut s'avérer très difficile. La notion de résultant fournit des outils assez simples pour répondre à ces questions, lorsque $\mathcal{A}$ est un anneau commutatif. Dans cette section, nous donnons les résultats qui nous serons utiles sur le résultant. Pour plus d'informations sur ce dernier, nous renvoyons le lecteur vers [Mis93, Woo16]. Pour la suite de cette section, nous supposerons que $\mathcal{A}$ est un anneau commutatif.

Avant de présenter la définition du résultant, nous rappelons ce qu'est la matrice de Sylvester de deux polynômes. Soit $A, B \in \mathcal{A}[X]$ deux polynômes, tels que : $A(X) = a_0 + a_1X + \cdots + a_nX^n$ et $B(X) = b_0 + b_1X + \cdots + b_mX^m$. La matrice de Sylvester de A et B, notée $\mathcal{S}_{A,B}$, est la matrice $(n+m) \times (n+m)$ définie comme suit :

$$\mathcal{S}_{A,B} = \begin{pmatrix} a_n & 0 & \dots & 0 & b_m & 0 & \dots & 0 \\ a_{n-1} & a_n & \ddots & \vdots & \vdots & b_m & \ddots & \vdots \\ \vdots & a_{n-1} & \ddots & 0 & \vdots & & \ddots & 0 \\ \vdots & \vdots & \ddots & a_n & b_1 & & & b_m \\ a_0 & & & a_{n-1} & b_0 & \ddots & \vdots & \vdots \\ 0 & \ddots & & \vdots & 0 & \ddots & b_1 & \vdots \\ \vdots & \ddots & a_0 & \vdots & \vdots & \ddots & b_0 & b_1 \\ 0 & \dots & 0 & a_0 & 0 & \dots & 0 & b_0 \end{pmatrix}$$

Note : On retrouve souvent une définition de la matrice de Sylvester qui est la transposée de $\mathcal{S}_{A,B}$ ci-dessus. Cependant, une matrice et sa transposée ayant le même déterminant, les deux conventions s'appliquent à la définition A.1.

Définition A.1 (Résultant). [Mis93, Déf. 7.2.2, p. 227]
Soit $A, B \in \mathcal{A}[X]$ deux polynômes. Le résultant de A et B, noté Res(A,B), est le déterminant de leur matrice de Sylvester. Ainsi, c'est un élément de $\mathcal{A}$.

Comme mentionné plus haut, l'identité de Bézout n'est pas toujours vérifiée dans $\mathcal{A}[X]$. Cependant, on a la proposition A.1 qui est une généralisation de l'identité de Bézout à tout anneau commutatif.

Proposition A.1. *[Mis93, Lemma 7.2.1, p. 228]*
Soit $A, B \in \mathcal{A}[X]$ deux polynômes non nuls, tels que : $\deg(A) + \deg(B) \geqslant 1$. *Il existe $U, V \in \mathcal{A}[X]$ tels que :* $A(X)U(X) + B(X)V(X) = Res(A, B)$, *avec* $\deg(U) < \deg(B)$ *et* $\deg(V) < \deg(A)$.

Si de plus $\mathcal{A}$ est intègre, alors on a les résultats suivants, qu'on retrouve dans la littérature. Ces résultats sont équivalents.

Propriété A.1. Soit $\mathcal{A}$ un anneau intègre et $A, B \in \mathcal{A}[X]$.

1. $\text{Res}(A, B) = 0$ si et seulement si A et B ont un facteur en commun.
2. $\text{Res}(A, B) = 0$ si et seulement si A et B ont une racine commune dans une clôture algébrique contenant les coefficients de A et B.

Nous terminons cette section en rappelant que l'ensemble $\mathbb{Z}$ des entiers relatifs est un anneau intègre et donc les résultats ci-dessus lui sont applicables.

A.2 Notions sur les réseaux euclidiens

Dans cette section, nous donnons les notions et propriétés essentielles sur les réseaux euclidiens. Tous les éléments présentés ici peuvent être retrouvés dans la littérature avec plus ou moins de détails. En outre, nous nous limitons à ceux utiles pour l'étude du MNS. Pour une présentation plus complète des réseaux euclidiens, nous renvoyons le lecteur vers [Coh93, Geo13, KF18], entre autres.

A.2.1 Notions générales

Définition A.2. Un réseau euclidien $\mathcal{L}$ est un sous groupe additif discret non vide de $\mathbb{R}^n$. Plus précisément, c'est l'ensemble des combinaisons linéaires entières de d vecteurs $\boldsymbol{b}_i$ linéairement indépendants de $\mathbb{R}^n$, avec $d \leqslant n$:

$$\mathcal{L} = \mathbb{Z}\boldsymbol{b}_0 + \cdots + \mathbb{Z}\boldsymbol{b}_{d-1} = \{\alpha_0\boldsymbol{b}_0 + \cdots + \alpha_{d-1}\boldsymbol{b}_{d-1} : \alpha_i \in \mathbb{Z}\}.$$

d est la dimension du réseau $\mathcal{L}$ et le tuple $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{d-1})$ est une de ses *bases*. Le nombre de bases d'un réseau euclidien est infini et toutes ces bases ont la même dimension. Lorsque $d = n$, on dit que $\mathcal{L}$ est un réseau *total*.

Remarque A.1. Dans cette thèse, nous ne considérerons que des réseaux totaux. Aussi, nous écrirons la base d'un réseau sous forme matricielle. Cette matrice sera appelée la *matrice génératrice* du réseau. Chacune de ses lignes représentera les coordonnées d'un des vecteurs de la base du réseau. Ainsi, si $\mathcal{L}$ est un réseau de dimension n, alors sa matrice génératrice $\boldsymbol{B}$ sera une matrice de n lignes et n colonnes, i.e. $\boldsymbol{B} \in \mathbb{R}^{n\times n}$. Enfin, un élément du réseau sera appelé indifféremment *point* ou *vecteur*.

Exemple A.1. Dans $\mathbb{R}^2$, les vecteurs $\boldsymbol{b}_0 = (2, 1)$ et $\boldsymbol{b}_1 = (1, 3)$ sont linéairement indépendants. Donc, $\mathcal{L} = \mathbb{Z}\boldsymbol{b}_0 + \mathbb{Z}\boldsymbol{b}_1$ est un réseau *total*, de dimension 2, dont une base est $\boldsymbol{B} = (\boldsymbol{b}_0, \boldsymbol{b}_1) = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$. Ce réseau est schématisé par les points bleus de la figure A.1.

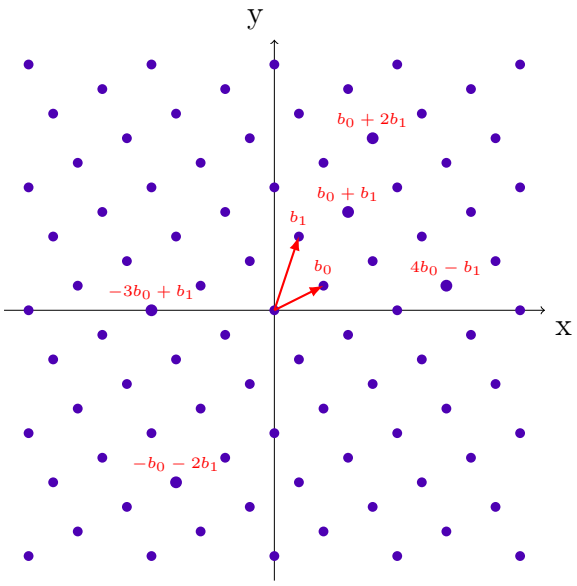


FIGURE A.1 – Le réseau $\mathcal{L}$ de $\mathbb{R}^2$ défini par les vecteurs $(2, 1)$ et $(1, 3)$.

Remarque A.2. Si $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ est une base du réseau $\mathcal{L}$, alors $\boldsymbol{B}$ est aussi une base de $\mathbb{R}^n$, car les vecteurs $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}$ sont linéairement indépendants. Plus précisément, $\mathbb{R}^n$ est le $\mathbb{R}$-espace vectoriel engendré par $\boldsymbol{B}$:

$$\mathbb{R}^n = \mathbb{R}\boldsymbol{b}_0 + \cdots + \mathbb{R}\boldsymbol{b}_{n-1} = \{\beta_0\boldsymbol{b}_0 + \cdots + \beta_{n-1}\boldsymbol{b}_{n-1} : \beta_i \in \mathbb{R}\}.$$

La définition qui suit précise la notion de congruence de points modulo le réseau.

Définition A.3. Soit $\mathcal{L}$ un réseau total de dimension n et $a, b \in \mathbb{R}^n$. On dira que le point a est congru au point b modulo le réseau $\mathcal{L}$ s'il existe $\nu \in \mathcal{L}$ tel que : $a = b + \nu$. On notera alors : $a \overset{\mathcal{L}}{\equiv} b$.

Sachant qu'un réseau possède une infinité de bases et donc une infinité de matrices génératrices, on peut se demander quand deux matrices génératrices distinctes génèrent le même réseau. Pour répondre à cette question, il est nécessaire de rappeler la définition de la matrice de Gram ainsi que certaines de ses propriétés.

Définition A.4. Soit $x_0, \ldots, x_{n-1}$ des vecteurs de $\mathbb{R}^n$. La matrice de Gram des vecteurs $x_0, \ldots, x_{n-1}$, noté $\boldsymbol{M_G}(x_0, \ldots, x_{n-1})$, est définie comme suit :

$$\boldsymbol{M_G}(x_0, \ldots, x_{n-1}) = \begin{pmatrix} \langle x_0|x_0\rangle & \langle x_0|x_1\rangle & \ldots & \langle x_0|x_{n-1}\rangle \\ \langle x_1|x_0\rangle & \langle x_1|x_1\rangle & \ldots & \langle x_1|x_{n-1}\rangle \\ \vdots & \vdots & & \vdots \\ \langle x_{n-1}|x_0\rangle & \langle x_{n-1}|x_1\rangle & \ldots & \langle x_{n-1}|x_{n-1}\rangle \end{pmatrix}$$

où $\langle x_i|x_j\rangle$ est le produit scalaire des vecteurs x_i et x_j. On appelle *déterminant de Gram* le déterminant de cette matrice et on note : $G(x_0, \ldots, x_{n-1}) = \det(\boldsymbol{M_G}(x_0, \ldots, x_{n-1}))$.

Propriété A.2. Soit $x_0, \ldots, x_{n-1}$ des vecteurs de $\mathbb{R}^n$. Soit $\boldsymbol{X} \in \mathbb{R}^{n\times n}$ la matrice correspondant aux vecteurs x_i comme expliqué dans la remarque A.1. On a alors les propriétés suivantes :

1. $\boldsymbol{M_G}(x_0, \ldots, x_{n-1}) = \boldsymbol{X}^t.\boldsymbol{X}$, où $\boldsymbol{X}^t$ est la matrice transposée de $\boldsymbol{X}$.
2. $\text{rang}(\boldsymbol{M_G}(x_0, \ldots, x_{n-1})) = \text{rang}(x_0, \ldots, x_{n-1})$.

Nous rappelons que le rang d'une matrice est le plus grand nombre de vecteurs lignes (ou colonnes) linéairement indépendants de cette matrice. De même, le rang de la famille de vecteurs $(x_0, \ldots, x_{n-1})$ est la dimension du sous-espace vectoriel engendré par cette famille. Comme $\boldsymbol{M_G}(x_0, \ldots, x_{n-1}) = \boldsymbol{X}^t.\boldsymbol{X}$, on a : $G(x_0, \ldots, x_{n-1}) = \det(\boldsymbol{X}^t.\boldsymbol{X}) = \det(\boldsymbol{X})^2$. Donc, $G(x_0, \ldots, x_{n-1}) \geqslant 0$.

De la propriété A.2, on déduit également la proposition A.2 utile pour la preuve de la relation entre les matrices génératrices d'un même réseau.

Proposition A.2. *Soit $x_0, \ldots, x_{n-1}$ des vecteurs de $\mathbb{R}^n$. Ces vecteurs sont linéairement indépendants si et seulement si $G(x_0, \ldots, x_{n-1}) > 0$.*

Démonstration. Si les vecteurs $x_0, \ldots, x_{n-1}$ sont linéairement indépendants, alors $\text{rang}(x_0, \ldots, x_{n-1}) = n$. Ce qui implique que la matrice $\boldsymbol{X}$ est inversible, i.e. $\det(\boldsymbol{X}) \neq 0$. Or, $G(x_0, \ldots, x_{n-1}) = \det(\boldsymbol{X})^2$. D'où, $G(x_0, \ldots, x_{n-1}) > 0$.

Si $G(x_0, \ldots, x_{n-1}) = \det(\boldsymbol{X})^2 > 0$, alors $\det(\boldsymbol{X}) \neq 0$. Donc, $\boldsymbol{X}$ est inversible. Donc, $\text{rang}(x_0, \ldots, x_{n-1}) = \text{rang}(\boldsymbol{X}) = n$; i.e. les vecteurs $x_0, \ldots, x_{n-1}$ sont linéairement indépendants. □

Le théorème qui suit donne la relation entre deux matrices génératrices d'un même réseau. Nous rappelons qu'une matrice $\boldsymbol{U} \in \mathbb{Z}^{n\times n}$ est dite unimodulaire si $\det(\boldsymbol{U}) \in \{1, -1\}$. De plus, si $\boldsymbol{U}$ est unimodulaire, alors son inverse $\boldsymbol{U}^{-1}$ appartient également à $\mathbb{Z}^{n\times n}$ et $\boldsymbol{U}^{-1}$ est aussi unimodulaire.

Théorème A.1. *Deux matrices génératrices $\boldsymbol{B}, \boldsymbol{B'} \in \mathbb{R}^{n\times n}$ génèrent le même réseau $\mathcal{L}$ si et seulement si il existe une matrice unimodulaire $\boldsymbol{U} \in \mathbb{Z}^{n\times n}$ telle que : $\boldsymbol{B'} = \boldsymbol{U}.\boldsymbol{B}$.*

Démonstration. Supposons que $\boldsymbol{B}$ et $\boldsymbol{B'}$ génèrent le même réseau. Alors, il existe deux matrices $\boldsymbol{U}, \boldsymbol{U'} \in \mathbb{Z}^{n\times n}$ telles que $\boldsymbol{B'} = \boldsymbol{U}.\boldsymbol{B}$ et $\boldsymbol{B} = \boldsymbol{U'}.\boldsymbol{B'}$; chaque ligne i de $\boldsymbol{U}$ multipliée par $\boldsymbol{B}$ génère le vecteur $\boldsymbol{b}'_i$ de $\boldsymbol{B'}$; idem pour $\boldsymbol{U'}$. Ainsi, $\boldsymbol{B} = \boldsymbol{U'}.(\boldsymbol{U}.\boldsymbol{B}) = (\boldsymbol{U'}.\boldsymbol{U}).\boldsymbol{B}$. Par conséquent, $\boldsymbol{B}\boldsymbol{B}^t = (\boldsymbol{U'}.\boldsymbol{U}).\boldsymbol{B}\boldsymbol{B}^t$. On a $\det(\boldsymbol{B}\boldsymbol{B}^t) = \det(\boldsymbol{B}^t\boldsymbol{B})$. Les vecteurs de $\boldsymbol{B}$ sont linéairement indépendants, donc d'après la proposition A.2, $\det(\boldsymbol{B}^t\boldsymbol{B}) > 0$. Donc, la matrice $\boldsymbol{B}\boldsymbol{B}^t$ est inversible. Par conséquent, $\boldsymbol{B}\boldsymbol{B}^t = (\boldsymbol{U'}.\boldsymbol{U}).\boldsymbol{B}\boldsymbol{B}^t$ implique que $(\boldsymbol{U'}.\boldsymbol{U})$ est la matrice identité ; i.e. $\det(\boldsymbol{U'}.\boldsymbol{U}) = \det(\boldsymbol{U'}).\det(\boldsymbol{U}) = 1$. Or $\boldsymbol{U}, \boldsymbol{U'} \in \mathbb{Z}^{n\times n}$, donc $\det(\boldsymbol{U}) \in \{1, -1\}$ et $\det(\boldsymbol{U'}) \in \{1, -1\}$. D'où, $\boldsymbol{U}$ et $\boldsymbol{U'}$ sont unimodulaires.

Supposons maintenant qu'il existe une matrice unimodulaire $\boldsymbol{U} \in \mathbb{Z}^{n\times n}$ telle que : $\boldsymbol{B'} = \boldsymbol{U}.\boldsymbol{B}$ (et donc $\boldsymbol{B} = \boldsymbol{U}^{-1}.\boldsymbol{B'}$). Soit $\mathcal{L}$ et $\mathcal{L}'$ les réseaux générés respectivement par $\boldsymbol{B}$ et $\boldsymbol{B'}$. Soit $\omega \in \mathcal{L}$, alors il existe $c \in \mathbb{Z}^n$ tel que $\omega = c.\boldsymbol{B}$. Donc, $\omega = c.\boldsymbol{U}^{-1}.\boldsymbol{B'}$. On a : $c.\boldsymbol{U}^{-1} \in \mathbb{Z}^n$, donc $\omega \in \mathcal{L}'$. Par conséquent, $\mathcal{L} \subseteq \mathcal{L}'$. De la même manière, on montre que $\mathcal{L}' \subseteq \mathcal{L}$. On obtient donc au final que $\mathcal{L}' = \mathcal{L}$, i.e. $\boldsymbol{B}$ et $\boldsymbol{B'}$ génèrent le même réseau. □

Le théorème précédent montre que deux bases d'un réseau sont liées par une matrice unimodulaire. D'après ce théorème, si $\boldsymbol{B}$ est une base et $\boldsymbol{U}$ une matrice unimodulaire, alors $\boldsymbol{U}.\boldsymbol{B}$ est aussi une base du réseau. Ainsi, pour tester si deux bases engendrent le même réseau, il suffit de calculer la matrice de passage d'une base à l'autre et de vérifier si cette dernière est unimodulaire.

A.2.2 Régions fondamentales du réseau

Un réseau euclidien possède des zones particulières appelées *régions fondamentales*. Ces zones possèdent une propriété essentielle que nous verrons ici.

Soit $\mathcal{K} \subseteq \mathbb{R}^n$ et $v \in \mathbb{R}^n$. On notera $v + \mathcal{K}$ l'ensemble correspondant aux translations de tous les éléments de $\mathcal{K}$ par le vecteur v. Formellement, cela signifie que :

$$v + \mathcal{K} = \{v + y, \text{ avec } y \in \mathcal{K}\}.$$

Définition A.5. Soit $\mathcal{L}$ un réseau total de dimension n. Un ensemble $\mathcal{R} \subset \mathbb{R}^n$ est une région fondamentale du réseau $\mathcal{L}$ s'il satisfait les conditions suivantes :

1. $\mathbb{R}^n = \bigcup_{\eta\in\mathcal{L}} (\eta + \mathcal{R})$.
2. Pour tout $\eta_1, \eta_2 \in \mathcal{L}$, avec $\eta_1 \neq \eta_2$, on a : $(\eta_1 + \mathcal{R}) \cap (\eta_2 + \mathcal{R}) = \emptyset$.

En d'autres termes, les translations d'une région fondamentale $\mathcal{R}$ par les points du réseau $\mathcal{L}$ forment une partition de $\mathbb{R}^n$.

Une conséquence triviale de cette définition est le lemme ci-dessous.

Lemme A.1. *Une région fondamentale $\mathcal{R}$ ne peut pas contenir deux points y_1 et y_2 dont la différence est un point non nul du réseau.*

Démonstration. Soit $\mathcal{R}$ une région fondamentale et $y_1, y_2 \in \mathcal{R}$. Supposons que $y_1 - y_2 = \eta \in \mathcal{L}$, avec $\eta \neq 0$. On a $y_1 = \eta + y_2$. Donc, $y_1 \in \eta + \mathcal{R}$ or $y_1 \in \mathcal{R} = 0 + \mathcal{R}$. Par conséquent, $y_1 \in (0 + \mathcal{R}) \cap (\eta + \mathcal{R})$. Ceci est en contradiction avec le second point de la définition A.5 puisque $\eta \neq 0$. □

Le théorème qui suit nous sera utile pour définir des bornes sur les éléments de l'AMNS.

Théorème A.2. *Soit $\mathcal{L}$ un réseau total de dimension n et $\mathcal{R}$ une région fondamentale de $\mathcal{L}$. Tout vecteur $v \in \mathbb{R}^n$ est congru modulo $\mathcal{L}$ à un unique vecteur y de $\mathcal{R}$:*

$$\forall v \in \mathbb{R}^n,\ \exists!\ y \in \mathcal{R} : v \overset{\mathcal{L}}{\equiv} y.$$

Démonstration. Soit $v \in \mathbb{R}^n$. D'après la définition A.5, il existe un point $\eta \in \mathcal{L}$ tel que $v \in \eta + \mathcal{R}$. Soit $y = v - \eta$. On a alors : $y \in \mathcal{R}$ et $v \overset{\mathcal{L}}{\equiv} y$; i.e. v est congru à y modulo $\mathcal{L}$, cf. définition A.3.

Montrons que y est unique. Soit $x \in \mathcal{R}$ tel que $v \overset{\mathcal{L}}{\equiv} x$. Donc, il existe $\mu \in \mathcal{L}$ tel que $x = v - \mu$. Ainsi, $v = y + \eta = x + \mu$. Donc, $y - x = \mu - \eta \in \mathcal{L}$. D'après le lemme A.1, $\mu - \eta = 0$, car $x, y \in \mathcal{R}$. D'où, $x = y$. □

A.2.3 Domaine fondamental du réseau

Toute base $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ d'un réseau euclidien $\mathcal{L}$ définit une région fondamentale particulière appelée le *domaine fondamental*. Ce domaine possède des propriétés intéressantes que nous verrons ici.

Définition A.6. Soit $\mathcal{L}$ un réseau total et $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ une base de $\mathcal{L}$. Le domaine fondamental $\mathcal{H}$ du réseau $\mathcal{L}$, par rapport à la base $\boldsymbol{B}$, est défini comme suit :

$$\mathcal{H} = \{y \in \mathbb{R}^n, \text{ tel que : } y = \sum_{i=0}^{n-1} y_i \boldsymbol{b}_i,\ 0 \leqslant y_i < 1\}$$

Du point de vue géométrique, le domaine fondamental $\mathcal{H}$ correspond au parallélotope formé par les vecteurs $\boldsymbol{b}_i$ de la base $\boldsymbol{B}$.

Exemple A.2. Le domaine fondamental, par rapport à la base $\boldsymbol{B} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$, du réseau $\mathcal{L}$ de l'exemple A.1 est schématisé par le parallélogramme vert de la figure A.2.

Le domaine fondamental permet de définir un invariant associé au réseau. Il s'agit du déterminant du réseau aussi appelé covolume du réseau.

Définition A.7. Soit $\mathcal{L}$ un réseau total et $\boldsymbol{B}$ une base de $\mathcal{L}$. Le déterminant $\det(L)$ de $\mathcal{L}$ est le volume de son domaine fondamental. On a :

$$\det(\mathcal{L}) = \sqrt{G(b_0, \ldots, b_{n-1})} = |\det(\boldsymbol{B})|.$$

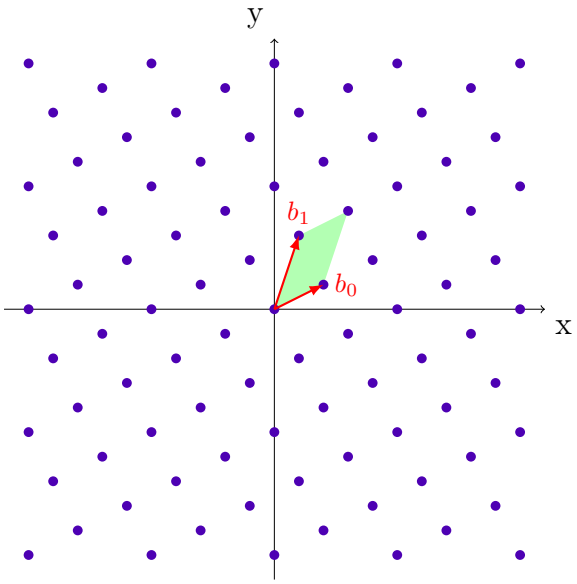


FIGURE A.2 – Le domaine fondamental du réseau $\mathcal{L}$.

Ce déterminant est invariant quelque soit la base $\boldsymbol{B}$ utilisée et tous les domaines fondamentaux ont le même volume. En effet, si $\boldsymbol{B}$ et $\boldsymbol{B}'$ sont deux bases de $\mathcal{L}$. D'après la théorème A.1, il existe une matrice unimodulaire $\boldsymbol{U} \in \mathbb{Z}^{n\times n}$ telle que $\boldsymbol{B}' = \boldsymbol{U}.\boldsymbol{B}$. Or $\det(\boldsymbol{U}) \in \{1, -1\}$, donc $|\det(\boldsymbol{B}')| = |\det(\boldsymbol{B})|$.

Exemple A.3. Reprenons le réseau $\mathcal{L}$ de l'exemple A.2 dont $\boldsymbol{B} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$ est une base. Le déterminant $\det(\mathcal{L})$ du réseau $\mathcal{L}$ est le volume du parallélogramme de la figure A.2. Comme ce réseau est total, on a :

$$\det(\mathcal{L}) = \left|\det \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}\right| = |6 - 1| = 5.$$

D'après la proposition A.3 qui suit, le domaine fondamental d'un réseau est bien une région fondamentale de ce réseau.

Proposition A.3. *Soit $\mathcal{L}$ un réseau total de dimension n. Soit $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ une base de $\mathcal{L}$ et $\mathcal{H}$ le domaine fondamental associé. Alors, $\mathcal{H}$ est une région fondamentale de $\mathcal{L}$.*

Démonstration. Soit $v \in \mathbb{R}^n$. D'après la remarque A.2, il existe un unique tuple $(\alpha_0, \ldots, \alpha_{n-1}) \in \mathbb{R}^n$ tel que $v = \alpha_0\boldsymbol{b}_0 + \cdots + \alpha_{n-1}\boldsymbol{b}_{n-1}$. Soit $\eta = \lfloor\alpha_0\rfloor\boldsymbol{b}_0 + \cdots + \lfloor\alpha_{n-1}\rfloor\boldsymbol{b}_{n-1}$. On a : $\eta \in \mathcal{L}$ et $v \in (\eta + \mathcal{H})$. Donc, la première condition de la définition A.5 est satisfaite.

Soit $\mu \in \mathcal{L}$ tel que $v \in (\mu + \mathcal{H})$, avec $\mu = \beta_0\boldsymbol{b}_0 + \cdots + \beta_{n-1}\boldsymbol{b}_{n-1}$. Supposons que $\mu \neq \eta$. Alors, il existe une position i telle que $\beta_i \neq \lfloor v_i\rfloor$. Comme $v \in (\eta + \mathcal{H}) \cap (\mu + \mathcal{H})$, alors $v_i \in (\beta_i + [0, 1[) \cap (\lfloor v_i\rfloor + [0, 1[)$. Ce qui est absurde, car $\beta_i \neq \lfloor v_i\rfloor$ implique que $(\beta_i + [0, 1[) \cap (\lfloor v_i\rfloor + [0, 1[) = \emptyset$, puisque $\beta_i \in \mathbb{N}$. Donc, nécessairement $\mu = \eta$. Ainsi, la seconde condition de la définition A.5 est satisfaite. Par conséquent, $\mathcal{H}$ est une région fondamentale de $\mathcal{L}$. □

La proposition A.3 implique que les translations du domaine fondamental $\mathcal{H}$ par les points du réseau $\mathcal{L}$ forment une partition de $\mathbb{R}^n$. Du point de vue géométrique, cela signifie que le parallélotope formé par le domaine fondamental $\mathcal{H}$ peut être utilisé pour effectuer un recouvrement, avec intersection vide, de $\mathbb{R}^n$. Sur la figure A.3 qui reprend le réseau de l'exemple A.2, nous donnons des exemples de translations par des points du réseau du domaine fondamental. Il est facile de voir que ces translations peuvent être utilisées pour effectuer une partition du plan $\mathbb{R}^2$.

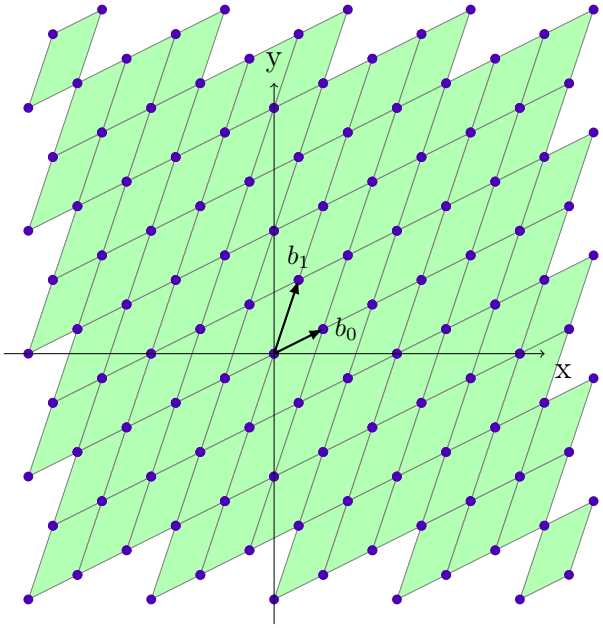


FIGURE A.3 – Exemple de translations du domaine fondamental du réseau $\mathcal{L}$.

Une autre région fondamentale assez proche du domaine fondamental est l'ensemble $\mathcal{H}'$ défini comme suit :

$$\mathcal{H}' = \{t \in \mathbb{R}^n, \text{ tel que : } t = \sum_{i=0}^{n-1} t_i \boldsymbol{b}_i, -\tfrac{1}{2} \leqslant t_i < \tfrac{1}{2}\}.$$

D'après la proposition A.4 qui suit, $\mathcal{H}'$ est effectivement une région fondamentale du réseau.

Proposition A.4. *Soit $\mathcal{L}$ un réseau total de dimension n. Soit $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ une base de $\mathcal{L}$. Alors, $\mathcal{H}'$ est une région fondamentale de $\mathcal{L}$.*

Démonstration. La preuve est similaire à celle de la proposition A.3.

Soit $v \in \mathbb{R}^n$. D'après la remarque A.2, il existe un unique tuple $(\alpha_0, \ldots, \alpha_{n-1}) \in \mathbb{R}^n$ tel que $v = \alpha_0 \boldsymbol{b}_0 + \cdots + \alpha_{n-1}\boldsymbol{b}_{n-1}$. Soit $\eta = \lfloor \alpha_0 \rceil \boldsymbol{b}_0 + \cdots + \lfloor \alpha_{n-1} \rceil \boldsymbol{b}_{n-1}$. On a : $\eta \in \mathcal{L}$ et $v \in (\eta + \mathcal{H}')$. Donc, la première condition de la définition A.5 est satisfaite.

Soit $\mu \in \mathcal{L}$ tel que $v \in (\mu + \mathcal{H}')$, avec $\mu = \beta_0 \boldsymbol{b}_0 + \cdots + \beta_{n-1}\boldsymbol{b}_{n-1}$. Supposons que $\mu \neq \eta$. Alors, il existe une position i telle que $\beta_i \neq \lfloor v_i \rceil$. Comme $v \in (\eta + \mathcal{H}') \cap (\mu + \mathcal{H}')$, alors $v_i \in (\beta_i + [-\frac{1}{2}, \frac{1}{2}[) \cap (\lfloor v_i \rfloor + [-\frac{1}{2}, \frac{1}{2}[)$. Ce qui est

absurde, car $\beta_i \neq \lfloor v_i \rceil$ implique que $(\beta_i + [-\frac{1}{2}, \frac{1}{2}[) \cap (\lfloor v_i \rfloor + [-\frac{1}{2}, \frac{1}{2}[) = \emptyset$, puisque $\beta_i \in \mathbb{N}$. Donc, nécessairement $\mu = \eta$. Ainsi, la seconde condition de la définition A.5 est satisfaite. Par conséquent, $\mathcal{H}'$ est une région fondamentale de $\mathcal{L}$. □

Remarque A.3. Avec les définitions de $\mathcal{H}$ et $\mathcal{H}'$, on a les propriétés suivantes :

- Si $y \in \mathcal{H}$, alors $\|y\|_\infty < \|\boldsymbol{B}\|_1$.
- Si $t \in \mathcal{H}'$, alors $\|t\|_\infty \leqslant \frac{1}{2}\|\boldsymbol{B}\|_1$.

Ces propriétés sont triviales, car $\boldsymbol{B}$ est la matrice dont les lignes correspondent aux vecteurs $\boldsymbol{b}_i$ qui constituent une base du réseau. Ainsi, par rapport à la réduction dans le domaine fondamental $\mathcal{H}$, la réduction dans la région $\mathcal{H}'$ permet de gagner un facteur $1/2$ sur les coefficients des vecteurs réduits. Nous verrons que ce gain est important pour l'AMNS.

A.2.4 Orthogonalisation de Gram-Schmidt

Le procédé d'orthogonalisation de Gram-Schmidt est un algorithme permettant de construire une famille orthogonale de vecteurs à partir d'une famille libre de vecteurs dans un espace euclidien. La base orthogonale obtenue possède plusieurs propriétés intéressantes. Elle est par exemple utile pour mesurer l'orthogonalité d'une base et aussi utilisée pour plusieurs algorithmes de réduction de réseau tels que l'algorithme LLL [LLL82] et l'algorithme de Babaï [Bab86]. Ici, nous ne présentons que l'essentiel du procédé d'orthogonalisation de Gram-Schmidt. Pour une présentation plus détaillée, nous renvoyons le lecteur à des travaux comme la thèse de Mariya Georgieva [Geo13].

Définition A.8 (Orthogonalisation de Gram-Schmidt).
Soit $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1})$ une famille de vecteurs linéairement indépendants de $\mathbb{R}^n$. La base orthogonalisée de Gram-Schmidt de $\boldsymbol{B}$ est la famille de vecteurs linéairement indépendants $\boldsymbol{B}^* = (\boldsymbol{b}_0^*, \ldots, \boldsymbol{b}_{n-1}^*)$ définie par :

$$\begin{cases} \boldsymbol{b}_0^* = \boldsymbol{b}_0 \\ \boldsymbol{b}_i^* = \boldsymbol{b}_i - \sum_{j=0}^{i-1} m_{i,j}\boldsymbol{b}_j^* \quad \text{pour} \quad i \geqslant 1 \text{ , avec } \quad m_{i,j} = \dfrac{\left\langle \boldsymbol{b}_i \middle| \boldsymbol{b}_j^* \right\rangle}{\left\| \boldsymbol{b}_j^* \right\|^2} \end{cases}$$

Les vecteurs de $\boldsymbol{B}^*$ sont orthogonaux ; i.e. $\left\langle b_i^* \middle| b_j^* \right\rangle = 0, \forall\ i \neq j$. De plus, chaque vecteur $\boldsymbol{b}_i^*$ de $\boldsymbol{B}^*$ est la projection orthogonale de $\boldsymbol{b}_i$ orthogonalement au sous-espace vectoriel engendré par $(\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{i-1})$. Ce sous-espace est le même que celui engendré par les vecteurs orthogonaux $(\boldsymbol{b}_0^*, \ldots, \boldsymbol{b}_{i-1}^*)$. La norme $\|\boldsymbol{b}_i^*\|$ est la distance entre $\boldsymbol{b}_i$ et ce sous-espace.

La matrice de passage $\mathcal{P}$ de $\boldsymbol{B}$ à $\boldsymbol{B}^*$ ($\boldsymbol{B} = \mathcal{P}.\boldsymbol{B}^*$) est la suivante :

$$\mathcal{P} = \begin{array}{c} \\ \boldsymbol{b}_0 \\ \boldsymbol{b}_1 \\ \vdots \\ \boldsymbol{b}_{n-1} \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{b}_0^* & \boldsymbol{b}_1^* & \ldots & \boldsymbol{b}_{n-1}^* \end{array} \\ \begin{pmatrix} 1 & 0 & \ldots & 0 \\ m_{1,0} & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n-1,0} & m_{n-1,1} & \ldots & 1 \end{pmatrix} \end{array} \tag{A.1}$$

On peut remarquer que $\mathcal{P}$ est une matrice triangulaire avec seulement des 1 sur la diagonale. Les bases $\boldsymbol{B}$ et $\boldsymbol{B}^*$ n'engendrent pas forcément le même réseau, puisque qu'on n'a pas forcément $\mathcal{P} \in \mathbb{Z}^{n\times n}$ et unitaire (cf. théorème A.1).

La base $\boldsymbol{B}^*$ donne une information importante sur les vecteurs du réseau engendré par $\boldsymbol{B}$.

Proposition A.5. *[Geo13, Proposition 1.5, p. 13]*
Soit $\mathcal{L}$ le réseau engendré par $\boldsymbol{B}$ et $\boldsymbol{B}^*$ la base orthogonalisée de Gram-Schmidt de $\boldsymbol{B}$. Si $v \in \mathcal{L} \setminus \{0\}$, alors :

$$\|v\| \geqslant \min_{0\leqslant i<n} \|\boldsymbol{b}_i^*\|.$$

Puisque c'est la norme infinie qui nous intéresse, nous en déduisons le corollaire suivant.

Corollaire A.1. *Soit $\mathcal{L}$ le réseau engendré par $\boldsymbol{B}$ et $\boldsymbol{B}^*$ la base orthogonalisée de Gram-Schmidt de $\boldsymbol{B}$. Si $v \in \mathcal{L} \setminus \{0\}$, alors :*

$$\|v\|_\infty \geqslant n^{-1/2} \min_{0\leqslant i<n} \|\boldsymbol{b}_i^*\|.$$

Démonstration. On a : $\|v\| = \sqrt{\sum_{i=0}^{n-1} v_i^2} \leqslant \sqrt{\sum_{i=0}^{n-1} \|v\|_\infty^2} = \sqrt{n\|v\|_\infty^2} = \sqrt{n}\|v\|_\infty$.
Donc, d'après la proposition A.5, on a : $\sqrt{n}\|v\|_\infty \geqslant \min_{0\leqslant i<n} \|\boldsymbol{b}_i^*\|$. □

A.2.5 Problèmes fondamentaux : *CVP* et *SVP*

Les réseaux euclidiens ont de nombreuses applications en cryptographie qui sont basées sur deux problèmes fondamentaux de la théorie des réseaux. Il s'agit du Closest Vector Problem (*CVP*) et du Shortest Vector Problem (*SVP*). Nous rappelons ces problèmes ci-dessous pour la norme infinie.

Soit $\mathcal{L}$ un réseau total de dimension n et $\boldsymbol{B} \in \mathbb{R}^{n\times n}$ une base de $\mathcal{L}$.

Problème A.1 (CVP_∞, Closest Vector Problem)**.** *Étant donné $t \in \mathbb{R}^n$, trouver (à partir de la base $\boldsymbol{B}$) le vecteur $\eta \in \mathcal{L}$ le plus proche de t :*

$$\forall v \in \mathcal{L},\ 0 \leqslant \|t-\eta\|_\infty \leqslant \|t-v\|_\infty.$$

Problème A.2 (SVP_∞, Shortest Vector Problem)**.** *À partir de la base $\boldsymbol{B}$, trouver un plus court vecteur η non nul du réseau $\mathcal{L}$:*

$$\forall v \in \mathcal{L},\ 0 < \|\eta\|_\infty \leqslant \|v\|_\infty.$$

Ces problèmes existent pour chaque norme. Dans cette thèse, nous nous intéressons aux variantes avec la norme infinie, car c'est cette norme qui nous sera utile pour le MNS. Dans [vEB81], van Emde Boas a prouvé que les problèmes CVP_∞ et SVP_∞ sont *NP-difficiles*. Les problèmes d'approximation associés (α-CVP_∞ et α-SVP_∞) ont donc été proposés pour relâcher leurs difficultés.

Soit $\alpha \in \mathbb{R}^+$ le facteur d'approximation (ou facteur de relâchement de la difficulté du problème).

Problème A.3 (α-CVP$_\infty$, Approximate Closest Vector Problem)**.** *Étant donné $t \in \mathbb{R}^n$, trouver (à partir de la base $\boldsymbol{B}$) un vecteur $\eta \in \mathcal{L}$ tel que :*

$$\forall v \in \mathcal{L},\ 0 \leqslant \|t - \eta\|_\infty \leqslant \alpha \|t - v\|_\infty.$$

Problème A.4 (α-SVP$_\infty$, Approximate Shortest Vector Problem)**.** *À partir de la base $\boldsymbol{B}$, trouver un vecteur η non nul du réseau $\mathcal{L}$:*

$$\forall v \in \mathcal{L},\ 0 < \|\eta\|_\infty \leqslant \alpha \|v\|_\infty.$$

Il est évident que si $\alpha = 1$, alors on retrouve les problèmes initiaux. Dans [GG00], Goldreichet et Goldwasser ont montré que ces problèmes d'approximation ne sont plus *NP-difficiles* lorsque le facteur d'approximation est supérieur ou égal $n/\mathcal{O}(\log n)$. Il existe plusieurs algorithmes pour résoudre ces problèmes d'approximation. L'un des plus célèbres est l'algorithme *LLL*, proposé par Lenstra et al. en 1982 [LLL82]. Cet algorithme permet de réduire la base d'un réseau.

Le théorème suivant donne un lien intéressant entre les problèmes α-CVP$_\infty$ et α-SVP$_\infty$.

Théorème A.3. *[GMSS99] Pour tout $\alpha \geqslant 1$, il existe une réduction efficace du problème α-SVP$_\infty$ au problème α-CVP$_\infty$.*

Remarque A.4.

1. L'approche triviale qui consiste à prendre $t = 0$ dans le problème α-CVP$_\infty$ pour avoir le problème α-SVP$_\infty$ ne marche pas, car on obtient 0 comme solution alors que 0 ne peut pas être une solution du problème α-SVP$_\infty$.
2. Dans [GMSS99], le théorème A.3 est prouvé pour toutes les normes.

Nous terminons cette sous-section par deux théorèmes fondamentaux qui donnent certaines propriétés du réseau. Le premier est le théorème de Minkowski [Min10] qui est un résultat essentiel de la théorie des réseaux euclidiens. Il donne une majoration de la longueur du plus court vecteur (*SVP*) du réseau. Ce vecteur est appelé *vecteur de Minkowski*. Nous donnons ici la variante pour la norme infinie, car c'est cette norme qui nous intéresse pour le MNS. Le second est le théorème de Hadamard [Had93] qui majore le déterminant d'une matrice donnée.

Théorème A.4 (**Minkowski**)**.** *Pour tout réseau total $\mathcal{L}$ de dimension n, il existe (au moins) un vecteur non nul $\eta \in \mathcal{L}$ tel que :*

$$\|\eta\|_\infty \leqslant \det(\mathcal{L})^{1/n}.$$

Théorème A.5 (**Inégalité de Hadamard**)**.** *Soit $\boldsymbol{B} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}) \in \mathbb{R}^{n \times n}$ une matrice. Alors :*

$$|\det(\boldsymbol{B})| \leqslant \prod_{i=0}^{n-1} \|\boldsymbol{b}_i\|.$$

Ce théorème est souvent présenté avec les vecteurs colonnes de la matrice, cependant la formulation ci-dessus est correcte, car une matrice et sa transposée ont le même déterminant. Aussi, on peut remarquer que ce théorème est défini pour la norme euclidienne. La norme infinie étant celle qui nous intéresse dans cette thèse, nous déduisons le corollaire A.2 de l'inégalité de Hadamard.

Corollaire A.2. *Soit* $\boldsymbol{B} = (\boldsymbol{b}_0, \dots, \boldsymbol{b}_{n-1}) \in \mathbb{R}^{n \times n}$ *une matrice. Alors :*

$$|\det(\boldsymbol{B})| \leqslant n^{n/2} \prod_{i=0}^{n-1} \|\boldsymbol{b}_i\|_\infty.$$

Démonstration. Soit $x = (x_0, \dots, x_{n-1}) \in \mathbb{R}^n$ un vecteur. On a : $\|x\| \leqslant \sqrt{n}\|x\|_\infty$. Donc, d'après l'inégalité de Hadamard, on a : $|\det(\boldsymbol{B})| \leqslant \prod_{i=0}^{n-1} (\sqrt{n}\|\boldsymbol{b}_i\|_\infty) = n^{n/2} \prod_{i=0}^{n-1} \|\boldsymbol{b}_i\|_\infty$. □

Annexe B

Compléments AMNS

Sommaire

B.1 Exemples d'AMNS pour différents nombres premiers

Dans cette section, nous donnons quelques exemples des AMNS que nous avons utilisés dans la section 2.5.3.1 pour nos expériences numériques. Tous ces AMNS ont le paramètre commun $\phi = 2^{64}$.

B.1.1 AMNS 1 : nombre premier de 192 bits.

- $p = 5503107796700669008431628908030675378946606110449411028761$
- $n = 4$
- $\lambda = -1$
- $\rho = 2^{51}$

- $\gamma = 3008346391693464928129778370348362265007331585099724367125$
- $E(X) = X^4 + 1$
- $M(X) = 82725675895109.X^3 - 160806122828248.X^2 + 63451193288485.X + 217157238130147$
- $M'(Y) = 7938559283596582735.Y^3 + 10329516294078499608.Y^2 + 7611637825038858703.Y + 13753522462301038407$

B.1.2 AMNS 2 : nombre premier de 224 bits.

- $p = 2448789317456695321926503645816464860071668447461277912765493983234 7$
- $n = 4$
- $\lambda = -2$
- $\rho = 2^{60}$
- $\gamma = 1058766467803258370980853016731601280032508431025030999956462375319 9$
- $E(X) = X^4 + 2$
- $M(X) = -29874833826956302.X^3 - 63383301401180582.X^2 - 11021753097552510.X - 18930956794395489$
- $M'(Y) = 9029559292635237954.Y^3 + 16182033252524253094.Y^2 + 6507598739445091122.Y + 15987366406211865945$

B.1.3 AMNS 3 : nombre premier de 256 bits.

- $p = 6512486640432429376312695394456675821144756895360350259736042433889657181155 7$
- $n = 5$
- $\lambda = 2$
- $\rho = 2^{55}$
- $\gamma = 3000384688032706924753198367365059320590240653806160428765880297419067083290 3$
- $E(X) = X^5 - 2$
- $M(X) = -139870059579771.X^4 - 1231122361135433.X^3 - 440957204555448.X^2 - 711730804792483.X + 1006443731177701$
- $M'(Y) = 7692632765118308294.Y^4 + 2198077546609165490.Y^3 + 6662003598150341841.Y^2 + 3720615719839163431.Y + 14734666259801548803$

B.1.4 AMNS 4 : nombre premier de 384 bits.

- $p = 3752726234810159453229085266711067023375303679320110399994304820129448651477997149984958201248002232949654181570695 1$

- $n = 7$
- $\lambda = 2$
- $\rho = 2^{59}$
- $\gamma = 25514254664435372606538349049214338253238188801036740616997407330438465621206923541821730620745447972948043891797700$
- $E(X) = X^7 - 2$
- $M(X) = 1222685297250892 1.X^6 + 11075924203076518.X^5 + 11863295143647713.X^4 + 5129356447896743.X^3 - 10885530247933181.X^2 + 11056475883179821.X - 18694471201508887$
- $M'(Y) = 3954277807273553420.Y^6 + 10514836889721811421.Y^5 + 5916546089089550476.Y^4 + 5861328222335664584.Y^3 + 8162128042527127834.Y^2 + 14759006294108824103.Y + 5650374102301417619$

B.1.5 AMNS 5 : nombre premier de 521 bits.

- $p = 459237834050574759307809382083508906733823834391388239579063047328893748621225655851264734984648219957529151148447421026207551747263684708797814818787055101 1$
- $n = 10$
- $\lambda = -2$
- $\rho = 2^{57}$
- $\gamma = 80339601974968296761822203174318892861266128211543834765971352228751926661947142283556978415995672899872268211633219114727990756227262481722132759965569937 7$
- $E(X) = X^{10} + 2$
- $M(X) = -1078823401354540.X^9 - 828300062112966.X^8 + 1061275849892818.X^7 - 1713147644876124.X^6 + 2571098790027282.X^5 - 1844957467998610.X^4 - 2699132360489056.X^3 - 877975421296440.X^2 + 1098509906391716.X + 897494113030397$
- $M'(Y) = 13446898917409989580.Y^9 + 5607572832843259142.Y^8 + 12156202058438924 66.Y^7 + 1842361179046800940.Y^6 + 10542100343558431874.Y^5 + 9181300598519214998.Y^4 + 7576094996452613280.Y^3 + 8573294372331940552.Y^2 + 13627575613084226676.Y + 14404640412319668243$

B.2 Exemples d'AMNS pour un même nombre premier

Dans la section 3.2, nous avons dit que l'existence de plusieurs AMNS pour un premier donné pouvait être utilisée pour randomiser les données d'entrée des protocoles cryptographiques. Ici, nous donnons trois exemples d'AMNS pour le premier $p = 2^{255} + 95$. Nous donnons également des représentations de trois éléments de $\mathbb{Z}/p\mathbb{Z}$ dans ces AMNS.

B.2.1 Les AMNS

Paramètres communs :

- $p = 2^{255} + 95$
- $\phi = 2^{64}$
- $n = 5$

B.2.1.1 AMNS 1.

- $\lambda = 2$
- $\rho = 2^{55}$
- $\gamma = 335028200566670017646622376576528083394786409404492936410996$
  19619291855395228
- $E(X) = X^5 - 2$
- $M(X) = 715680693526224.X^4 + 1042849028688725.X^3 -$
  $338087635434513.X^2 + 1238358587894820.X - 759363865979019$
- $M'(Y) = 8093936307502676601.Y^4 + 17649382591052594653.Y^3 +$
  $3866541394866451269.Y^2 + 12820261562051249844.Y + 17886731541140503075$

B.2.1.2 AMNS 2.

- $\lambda = 4$
- $\rho = 2^{56}$
- $\gamma = 360478466771822813146219070005712676739470629568854269494859$
  58237492277889644
- $E(X) = X^5 - 4$
- $M(X) = -997607846912686.X^4 + 1477438063702052.X^3 +$
  $943954124913102.X^2 - 1075612900869508.X - 307830523302197$
- $M'(Y) = 4628341613311959542.Y^4 + 13300162570116404036.Y^3 +$
  $3873519015384157966.Y^2 + 11866652964575599708.Y + 8052778708212021597$

B.2.1.3 AMNS 3.

- $\lambda = -3$
- $\rho = 2^{56}$
- $\gamma = 139074767475114509958139097380422877895175597525207187006496$
  51466455445912896
- $E(X) = X^5 + 3$
- $M(X) = -502897611187533.X^4 + 521078262072700.X^3 -$
  $1070958868484884.X^2 - 1470958101631976.X + 123832154353812$
- $M'(Y) = 549993428560787174 0.Y^4 + 8021025144503673160.Y^3 +$
  $5585149715501652324.Y^2 + 1266882257394783817.Y + 9384265636655253228$

B.2.2 Les représentations

Ici, nous donnons des représentations pour trois éléments de $\mathbb{Z}/p\mathbb{Z}$ dans les AMNS précédents. Soit w_1, w_2 et w_3 trois éléments de $\mathbb{Z}/p\mathbb{Z}$, tels que :

$w_1 = 2951177285814272445928127214983673891185810234426885844969284000073708015432 75$

$w_2 = 1326217869117615251684443766509213516051872945961093022252131205762821750634 8$

$w_3 = 3702001850384560851236366845740202833463789842701785782050885174266480715994 7$

B.2.2.1 Représentations de w_1

Des représentations de w_1 dans les AMNS ci-dessus sont :

- Dans l'AMNS 1 : $520179504630254.X^4 + 2483680444228898.X^3 + 992959768422742.X^2 + 2454072605528359.X + 2214075165553364$
- Dans l'AMNS 2 : $214119749491882.X^4 - 1626856045966503.X^3 + 1000845216293185.X^2 + 2318684701223152.X + 150215708719938$
- Dans l'AMNS 3 : $-253859761891926.X^4 + 482270132383076.X^3 - 2551560170490732.X^2 + 1057816326891127.X + 3882014404812070$

B.2.2.2 Représentations de w_2

Des représentations de w_2 dans les AMNS ci-dessus sont :

- Dans l'AMNS 1 : $1644391645160988.X^4 + 1980533612040528.X^3 + 2839623315645136.X^2 + 3082877134426208.X + 3859532966377745$
- Dans l'AMNS 2 : $782279937218913.X^4 - 2297888948134019.X^3 + 3935682791715208.X^2 + 200386964533500.X - 2261121392769328$
- Dans l'AMNS 3 : $-731871078916383.X^4 + 256482061549324.X^3 - 1935920218550724.X^2 + 1484924365305733.X + 5599754567559631$

B.2.2.3 Représentations de w_3

Des représentations de w_3 dans les AMNS ci-dessus sont :

- Dans l'AMNS 1 : $682692009798521.X^4 + 1563056668458880.X^3 + 2309298993601887.X^2 + 1242182206810759.X + 3215972273653199$
- Dans l'AMNS 2 : $982082644103246.X^4 - 1080024756737285.X^3 + 2831405080124323.X^2 - 896298424331140.X - 2926520686037898$
- Dans l'AMNS 3 : $-1656588634212511.X^4 - 139487939495395.X^3 - 1049690755289191.X^2 - 579902126343386.X + 4424592325274872$

B.3 Structures des grands entiers dans les bibliothèques GNU MP et OpenSSL

Dans cette section, nous donnons les structures des grands entiers dans GNU MP et OpenSSL. Ce sont ces structures que nous avons utilisées pour calculer les consommations mémoire dans le tableau 2.5.

B.3.1 Le type mpz_t dans GNU MP

```
typedef struct
{
  int _mp_alloc;   /* Number of *limbs* allocated and
                   pointed to by the '_mp_d' field. */
  int _mp_size;    /* abs(_mp_size) is the number of
                   limbs the last field points to.
                   If _mp_size is negative this
                   is a negative number. */
  mp_limb_t *_mp_d;   /* Pointer to the limbs. */
} __mpz_struct;
```

Dans le type mpz_t de GNU MP, il y a 2 entiers de type *int* et un tableau de type *mp_limb_t*. Sur l'ordinateur que nous avons utilisé pour nos tests (voir la section 2.5.3), le type *int* est de taille 32 bits et le type *mp_limb_t* 64 bits. Dans le calcul de la consommation mémoire, nous avons considéré les 2 entiers de type *int* comme un entier de 64 bits. Pour plus de détails, voir : https://gmplib.org/manual/Integer-Internals.html.

B.3.2 Le type bignum_st dans OpenSSL

```
struct bignum_st
{
   BN_ULONG *d; /* Pointer to an array of 'BN_BITS2'
                 bit chunks. */
   int top;     /* Index of last used d +1. */
/* The next are internal book keeping for bn_expand. */
   int dmax;    /* Size of the d array. */
   int neg;     /* one if the number is negative */
   int flags;
};
```

Dans le type bignum_st d'OpenSSL, il y a 4 entiers de type *int* et un tableau de type *BN_ULONG*, qui fait 64 bits de large (sur notre ordinateur). Dans le calcul de la consommation de mémoire, nous avons considéré les 4 entiers de type *int* comme deux entiers de 64 bits. Pour plus de détails, voir : https://linux.die.net/man/3/bn_internal_internal.

Annexe C

Exemples d'AMNS pour la randomisation

Sommaire

Cette annexe présente les paramètres des AMNS que nous avons utilisés dans les exemples du chapitre 3. Nous ne donnons pas la valeur du paramètre $M' = -M^{-1} \bmod (E, \phi)$ qui peut être calculée très simplement.

C.1 AMNS utilisés pour l'étude de redondance

Nous donnons ici la liste des AMNS utilisés dans la section 3.3.6.1. Ces AMNS ont été générés pour un nombre premier p de 256 bits, avec le paramètre n égal à 5 ou 6, suivant la stratégie d'implémentation logicielle présentée dans la remarque 2.6 du chapitre 2, en prenant $k = 64$.

L'ensemble de ces AMNS ont en commun les paramètres suivants :

- $p = 877986631880235280730309946384803882269167525514844705484330134094228264005 53$
- $\phi = 2^{64}$

C.1.1 AMNS 1

Cet AMNS ne permet aucune randomisation ($z = 0$).

- $z = 0$
- $\delta = 4$
- $n = 5$
- $\lambda = 2$
- $\rho = 2^{55}$
- $\gamma = 6996496533990134066287446405440342961450345031525964723645983945827907683147 2$
- $E(X) = X^5 - 2$
- $M(X) = 788313845896709.X^4 - 260400327879084.X^3 - 694419019569738.X^2 - 1395236943350850.X - 1116653473956573$

C.1.2 AMNS 2

Cet AMNS permet la conversion randomisée.

- $z = 43$
- $\delta = 0$
- $n = 5$
- $\lambda = 2$
- $\rho = 2^{59}$
- $\gamma = 6996496533990134066287446405440342961450345031525964723645983945827907683147 2$
- $E(X) = X^5 - 2$
- $M(X) = 788313845896709.X^4 - 260400327879084.X^3 - 694419019569738.X^2 - 1395236943350850.X - 1116653473956573$

C.1.3 AMNS 3

Cet AMNS permet la conversion randomisée.

- $z = 7$
- $\delta = 1$
- $n = 5$
- $\lambda = 2$
- $\rho = 2^{57}$
- $\gamma = 6996496533990134066287446405440342961450345031525964723645983945827907683147 2$
- $E(X) = X^5 - 2$
- $M(X) = 788313845896709.X^4 - 260400327879084.X^3 - 694419019569738.X^2 - 1395236943350850.X - 1116653473956573$

C.1.4 AMNS 4

Cet AMNS permet la conversion et la multiplication randomisées.

- $z = 1$
- $\delta = 0$
- $n = 5$
- $\lambda = 2$
- $\rho = 2^{56}$
- $\gamma = 69964965339901340662874464054403429614503450315259647236459839458279076831472$
- $E(X) = X^5 - 2$
- $M(X) = 788313845896709.X^4 - 260400327879084.X^3 - 694419019569738.X^2 - 1395236943350850.X - 1116653473956573$

C.1.5 AMNS 5

Cet AMNS ne permet aucune randomisation ($z = 0$).

- $z = 0$
- $\delta = 63$
- $n = 6$
- $\lambda = 3$
- $\rho = 2^{47}$
- $\gamma = 74559519895263714729394476906346823361024584835510614720523372584286779999702$
- $E(X) = X^6 - 3$
- $M(X) = 2673074756597.X^5 + 2725992072390.X^4 - 788062307520.X^3 - 3906633120091.X^2 + 2493237548159.X + 1764737145532$

C.1.6 AMNS 6

Cet AMNS permet la conversion randomisée.

- $z = 45$
- $\delta = 15$
- $n = 6$
- $\lambda = 3$
- $\rho = 2^{51}$
- $\gamma = 70543026063874968083330835012161440042437135154375017765226926371162653011399$
- $E(X) = X^6 - 3$
- $M(X) = 1565749579740.X^5 + 123136832359.X^4 - 1697278502061.X^3 + 2943623922815.X^2 - 838163884223.X + 2587093525133$

C.1.7 AMNS 7

Cet AMNS permet la conversion et la multiplication randomisées.

- $z = 46$
- $\delta = 0$
- $n = 6$
- $\lambda = 3$
- $\rho = 2^{52}$
- $\gamma = 705430260638749680833308350121614400424371351543750177$
  65226926371162653011399
- $E(X) = X^6 - 3$
- $M(X) = 1565749579740.X^5 + 123136832359.X^4 - 1697278502061.X^3 +$
  $2943623922815.X^2 - 838163884223.X + 2587093525133$

C.1.8 AMNS 8

Cet AMNS permet la conversion et la multiplication randomisées.

- $z = 10$
- $\delta = 4$
- $n = 6$
- $\lambda = 3$
- $\rho = 2^{50}$
- $\gamma = 705430260638749680833308350121614400424371351543750177$
  65226926371162653011399
- $E(X) = X^6 - 3$
- $M(X) = 1565749579740.X^5 + 123136832359.X^4 - 1697278502061.X^3 +$
  $2943623922815.X^2 - 838163884223.X + 2587093525133$

C.2 AMNS utilisés pour la randomisation de l'ECSM

Nous donnons ici les paramètres des AMNS utilisés pour la randomisation de l'ECSM dans la section 3.4.3. Ces AMNS ont été générés suivant la stratégie d'implémentation logicielle présentée dans la remarque 2.6 du chapitre 2, en prenant $k = 64$.

L'ensemble de ces AMNS ont en commun les paramètres suivants :

- $p = 2^{256} - 1539$
- $\phi = 2^{64}$

C.2.1 AMNS pour $\delta = 0$

C.2.1.1 AMNS 1

Cet AMNS a été utilisé pour la stratégie **S0**. Donc, aucune randomisation.

- $z = 0$
- $n = 5$
- $\lambda = 2$
- $\rho = 2^{55}$
- $\gamma = 548825236014721906489008526481141955934565587611407805$
  5982745049293038111292
- $E(X) = X^5 - 2$
- $M(X) = -445306306252361.X^4 + 1128639516641756.X^3 +$
  $1487567958265786.X^2 + 387584554680825.X + 405557347974121$

C.2.1.2 AMNS 2

Cet AMNS a été utilisé pour la stratégie **S1**. Il permet la conversion randomisée uniquement.

- $z = 6$
- $n = 5$
- $\lambda = -2$
- $\rho = 2^{57}$
- $\gamma = 1103038368771689763586808997438764882939243287895264859834748389585862009152710 5$
- $E(X) = X^5 + 2$
- $M(X) = 445306306252361.X^4 + 1128639516641756.X^3 -$
  $1487567958265786.X^2 + 387584554680825.X - 405557347974121$

C.2.1.3 AMNS 3

Cet AMNS a été utilisé pour les stratégies **S2**, **S3** et **S4**. Il permet la conversion et la multiplication randomisées.

- $z = 2$
- $n = 5$
- $\lambda = 3$
- $\rho = 2^{57}$
- $\gamma = 5554113387019498646625156435366879661472614678220737713694129449958586530722 6$
- $E(X) = X^5 - 3$
- $M(X) = -1503862990184956.X^4 - 760877823754781.X^3 +$
  $1195093941117297.X^2 - 1724260301352681.X + 655798597090430$

C.2.2 AMNS pour $\delta = 5$

C.2.2.1 AMNS 4

Cet AMNS a été utilisé pour la stratégie **S0**. Donc, aucune randomisation.

- $z = 0$
- $n = 6$
- $\lambda = -4$
- $\rho = 2^{48}$
- $\gamma = 627250360721988538580076038086157973421939300460003077$
  61413395459424990168884
- $E(X) = X^6 + 4$
- $M(X) = -4206607882342.X^5 + 4378314204044.X^4 + 2969591951538.X^3 + 2136971407166.X^2 + 1054234256357.X + 4063330874669$

C.2.2.2 AMNS 5

Cet AMNS a été utilisé pour la stratégie **S1**. Il permet la conversion randomisée uniquement.

- $z = 6$
- $n = 6$
- $\lambda = -4$
- $\rho = 2^{50}$
- $\gamma = 530636610281690412136588526038791952914366092329280251$
  14591173620883154748686
- $E(X) = X^6 + 4$
- $M(X) = 1713922966852.X^5 - 4019109313310.X^4 - 3528849197116.X^3 - 872361268682.X^2 - 2313924362292.X - 2243091601121$

C.2.2.3 AMNS 6

Cet AMNS a été utilisé pour les stratégies **S2**, **S3** et **S4**. Il permet la conversion et la multiplication randomisées.

- $z = 6$
- $n = 6$
- $\lambda = -4$
- $\rho = 2^{50}$
- $\gamma = 106130714193286382779222233803951305802512663852568281$
  392635362169371294218199
- $E(X) = X^6 + 4$
- $M(X) = 3619394862984.X^5 + 854256361656.X^4 - 3561257883749.X^3 - 3828204969225.X^2 + 1095767185424.X - 958314944803$

Annexe D

Randomisation avec la méthode de Babaï

Sommaire

Cette annexe présente brièvement la randomisation de la conversion du binaire vers l'AMNS ainsi que la multiplication modulaire dans l'AMNS, lorsque l'algorithme de Babaï (algorithme 16) est la méthode de réduction interne utilisée. Les algorithmes et bornes présentés dans cette annexe ont été publiés dans [DDEM+19]. Ici, nous ne présentons pas les preuves des résultats qui énoncent ces bornes. L'utilisation de la méthode de Babaï pour la réduction interne dans le PMNS (qui inclut l'AMNS, voir section 1.3.3) a été approfondie par Jérémy Marrez. Les bornes présentées ici, leurs preuves et bien d'autres résultats devraient apparaître dans sa thèse. Nous renvoyons donc le lecteur vers cette dernière, si besoin. Pour rappel, la méthode de Babaï a été présentée dans la section 1.3.5.2 du chapitre 1.

Soit $p \geqslant 3$ un nombre premier. Dans cette annexe, on considérera l'AMNS $\mathcal{B} = (p, n, \gamma, \rho, E)$, avec $E(X) = X^n - \lambda$. Comme dans le chapitre 3, la randomisation se fera à l'aide des polynômes de randomisation. L'objectif est le même. Il s'agit de garantir que chaque élément de $\mathbb{Z}/p\mathbb{Z}$ ait au moins $(2\,z+1)^n$ représentations distincts dans l'AMNS. Ensuite, les polynômes de randomisation permettront d'atteindre ces représentations de manière aléatoire. La fonction **RandPoly** du même chapitre sera donc utilisée ici également.

Soit $D = \{D_i, 1 \leq i \leq n\}$ une base réduite du réseau euclidien associé à l'AMNS $\mathcal{B}$, obtenue avec l'algorithme LLL. Soit $\widetilde{D} = \{\widetilde{D_i}, 1 \leq i \leq n\}$ la base de Gram-Schmidt de D. La randomisation se fera avec deux polynômes. Un polynôme de masquage V pour randomiser les opérations intermédiaires, et un polynôme de randomisation Z pour randomiser la sortie des opérations. Ces polynômes seront générés aléatoirement grâce à la fonction **RandPoly**.

D.1 Randomisation du processus de conversion

Cette section présente l'algorithme de conversion randomisée ainsi que la borne sur le paramètre ρ de l'AMNS pour garantir la consistance des opérations.

L'algorithme 51 est la conversion randomisée du binaire à l'AMNS. Cet algorithme nécessite les représentations *exactes* P_i de ρ^i dans $\mathcal{B}$. Dans la section 2.3.4 du chapitre 2, nous avons expliqué comment calculer ces représentations à l'aide de l'algorithme 23.

Algorithme 51 *Conversion randomisée du binaire à l'AMNS : Babaï*

Entrée(s) : $a \in \mathbb{Z}/p\mathbb{Z}$, $\mathcal{B} = (p, n, \gamma, \rho, E)$, les polynômes P_i, les bases D et $\widetilde{D}$, et $z, v \in \mathbb{N}$
Sortie : A, tel que $A(\gamma) \equiv a \pmod{p}$

1: $V \leftarrow$ **RandPoly**(v)
2: $Z \leftarrow$ **RandPoly**(z)
3: $b \leftarrow (a_{n-1}, ..., a_0)_\rho$ # la décomposition en base ρ de a
4: $T \leftarrow \sum_{i=0}^{n-1} a_i P_i$
5: $A \leftarrow T + \sum_{i=0}^{n-1} v_i D_{i+1}$
6: **for** $i = 1$ to n **do**
7: $\quad c \leftarrow \lfloor < A, \widetilde{D}_{n-i+1} > / \|\widetilde{D}_{n-i+1}\|^2 \rceil + z_{n-i}$
8: $\quad A \leftarrow A - c \times D_{n-i+1}$
9: **end for**
10: retourner A

Théorème D.1. *Soit $a \in \mathbb{Z}/p\mathbb{Z}$ l'entier à convertir. Si le paramètre ρ de l'AMNS $\mathcal{B}$ est tel que :*

$$\rho \geqslant \left(\tfrac{1}{2} + z\right) \left(2^{\frac{3n-1}{2}} p^{1/n}\right)$$

alors, l'algorithme 51 permet de calculer $(2\,z+1)^n$ représentations distinctes de a, toutes appartenant à $\mathcal{B}$.

D.2 Randomisation de la multiplication modulaire

L'algorithme 52 est la multiplication modulaire randomisée. Le théorème donne la borne sur le paramètre ρ de l'AMNS pour garantir la consistance des opérations.

Théorème D.2. *Soit $A, B \in \mathcal{B}$. Si le paramètre ρ de l'AMNS $\mathcal{B}$ est tel que :*

$$\rho \geqslant \left(\tfrac{1}{2} + z\right) \left(2^{\frac{3n-1}{2}} p^{1/n}\right)$$

alors, avec A et B comme entrées, l'algorithme 52 permet de calculer $(2\,z+1)^n$ représentations distinctes de $A(\gamma)B(\gamma) \pmod{p}$, toutes appartenant à $\mathcal{B}$.

Algorithme 52 *Multiplication randomisée dans l'AMNS : Babaï*

Entrée(s) : $A, B \in \mathcal{B}$, $\mathcal{B} = (p,\ n,\ \gamma,\ \rho, E)$, les bases D et $\widetilde{D}$, et $z, v \in \mathbb{N}$
Sortie : R, tel que $R(\gamma) = A(\gamma)B(\gamma) \mod p$

1: $V \leftarrow$ **RandPoly**(v)
2: $Z \leftarrow$ **RandPoly**(z)
3: $J \leftarrow \sum_{i=0}^{n-1} v_i\, D_{i+1}$
4: $B' \leftarrow B + J$
5: $R \leftarrow A \times B' \mod E$
6: **for** $i = 1$ **to** n **do**
7: $\quad c \leftarrow \lfloor < R, \widetilde{D}_{n-i+1} > / \|\widetilde{D}_{n-i+1}\|^2 \rceil + z_{n-i}$
8: $\quad R \leftarrow R - c \times D_{n-i+1}$
9: **end for**
10: retourner R

D.3 Coûts des algorithmes

Cette section traite du coût des algorithmes présentés plus haut. On suppose une architecture de k bits. Nous reprenons les éléments de la section 3.3.5 du chapitre 3. On suppose que $\rho < 2^k$ et $\lambda = \pm 2^i + \varepsilon 2^j$, avec $\varepsilon \in \{-1, 0, 1\}$. L'addition de deux mots de k bits est notée $\mathcal{A}$, le produit de deux mots de k bits est noté $\mathcal{M}$. Pour les opérations de décalage, $\mathcal{S}_l^i$ et $\mathcal{S}_r^i$ sont respectivement un décalage à gauche et un décalage à droite de i bits. De plus, si x et y sont chacun le produit de deux mots machines, alors l'opération $x + y$ coûte au maximum $3\mathcal{A}$. Le coût d'un appel à la fonction **RandPoly** est noté $\mathcal{R}$. Comme dans la section 1.3.5.5, on note $\mathcal{D}$ le coût d'une division et $\mathcal{C}$ le coût de la fonction $\lfloor \cdot \rceil$.

Aussi, on suppose que ρ est une puissance de deux (la décomposition en base ρ pour la conversion se fait alors très simplement) et que les éléments des bases D et $\widetilde{D}$ sont tels que $\|D_i\|_\infty, \|\widetilde{D}_i\|_\infty < \rho$.

Pour la conversion randomisée (algorithme 51), nous rappelons que ρ peut toujours être pris comme une puissance de deux. Dans ce cas, la décomposition en base ρ (ligne 3 de l'algorithme) se fait très simplement. Nous ignorons son coût ici.

Coût total randomisation	$2\mathcal{R} + n^2\mathcal{M} + (n^2 + 2n)\mathcal{A}$
Étape 1 (ligne 4)	$n^2\mathcal{M} + (3n^2 - 3n)\mathcal{A}$
Réduction interne	$n\mathcal{D} + 2n^2\mathcal{M} + (6n^2 - 3n)\mathcal{A} + n\mathcal{C}$
Coût total	$2\mathcal{R} + n\mathcal{D} + 4n^2\mathcal{M} + (10n^2 - 4n)\mathcal{A} + n\mathcal{C}$

TABLE D.1 – Coût théorique de la conversion randomisée (alg. 51).

Coût total randomisation	$2\mathcal{R} + n^2\mathcal{M} + (n^2 + 2n)\mathcal{A}$
Multiplication polynomiale	$n^2\mathcal{M} + (3n^2 - 6n + 3)\mathcal{A}$
Réduction externe	$3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^i$ $+ \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$
Réduction interne	$n\mathcal{D} + 2n^2\mathcal{M} + (6n^2 - 3n)\mathcal{A} + n\mathcal{C}$
Coût total	$2\mathcal{R} + n\mathcal{D} + 4n^2\mathcal{M} + (10n^2 - 4n)\mathcal{A} + n\mathcal{C} + (n-1)\mathcal{S}_l^i$ $+ \lvert\varepsilon\rvert\big(3(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^j\big)$

TABLE D.2 – Coût théorique de la multiplication randomisée (alg. 52), avec $E(X) = X^n - \lambda$, où $\lambda = \pm 2^i + \varepsilon 2^j$, $\varepsilon \in \{-1, 0, 1\}$.

Annexe E

Compléments ECSM

Sommaire

E.1 Caractéristiques des plates-formes de tests

Tous nos codes sources et les résultats collectés sont disponibles sur GitHub :
`https://github.com/eacElliptic`.

Les caractéristiques des plates-formes que nous avons utilisées pour nos benchmarks sont les suivantes :

- Plateforme Android : Téléphone Wiko Cik Peax 2, avec processeur MediaTek MT6589 (4 cœurs ARM Cortex-A7, 1,21 GHz), version Android : 4.1.2, API Level 16.
- Plateforme Java : Intel Core I5-4210U 4 cœurs 1.7Ghz, technologie Broadwell, JDK 1.7.0_79, Ubuntu 14.04 LTS.
- Plateforme C : Intel Core I5-4210U 4 cœurs 1.7Ghz, technologie Broadwell, gcc 5.2.1, gmp 6.1.0, Ubuntu 14.04 LTS.

La technologie Intel Turbo Boost a été désactivée sur la plate-forme x64 afin que la fréquence du processeur soit constante. Pour collecter les différents résultats d'exécution, nous avons utilisé les outils suivants :

- Plateforme Android : les méthodes `startMethodTracing` et `stopMethodTracing` de la classe `Debug` pour générer des traces, et la méthode `System.currentTimeMillis()` pour mesurer le temps d'exécution,
- Plateforme Java : le profileur fourni avec Netbeans IDE (v. 8.1) et la méthode `System.currentTimeMillis()`,

- Plateforme C : l'appel système `clock.gettime()` avec l'option `CLOCK_PROCESS_CPUTIME_ID`, et `taskset` pour lier notre processus actif à un seul processeur.

E.2 Anatomie de la multiplication modulaire

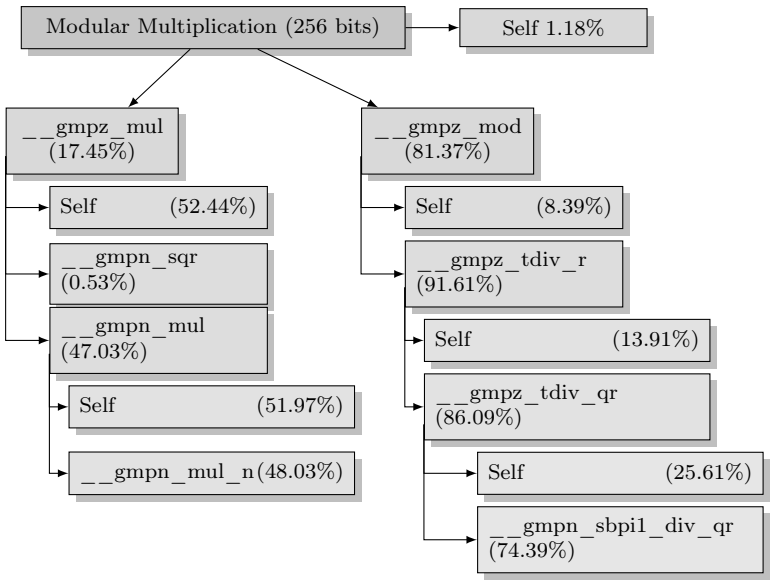


FIGURE E.1 – Anatomie d'une fonction C effectuant une multiplication modulaire sur des entiers de 256 bits, en utilisant Gnu MP (obtenu à partir de gprof).

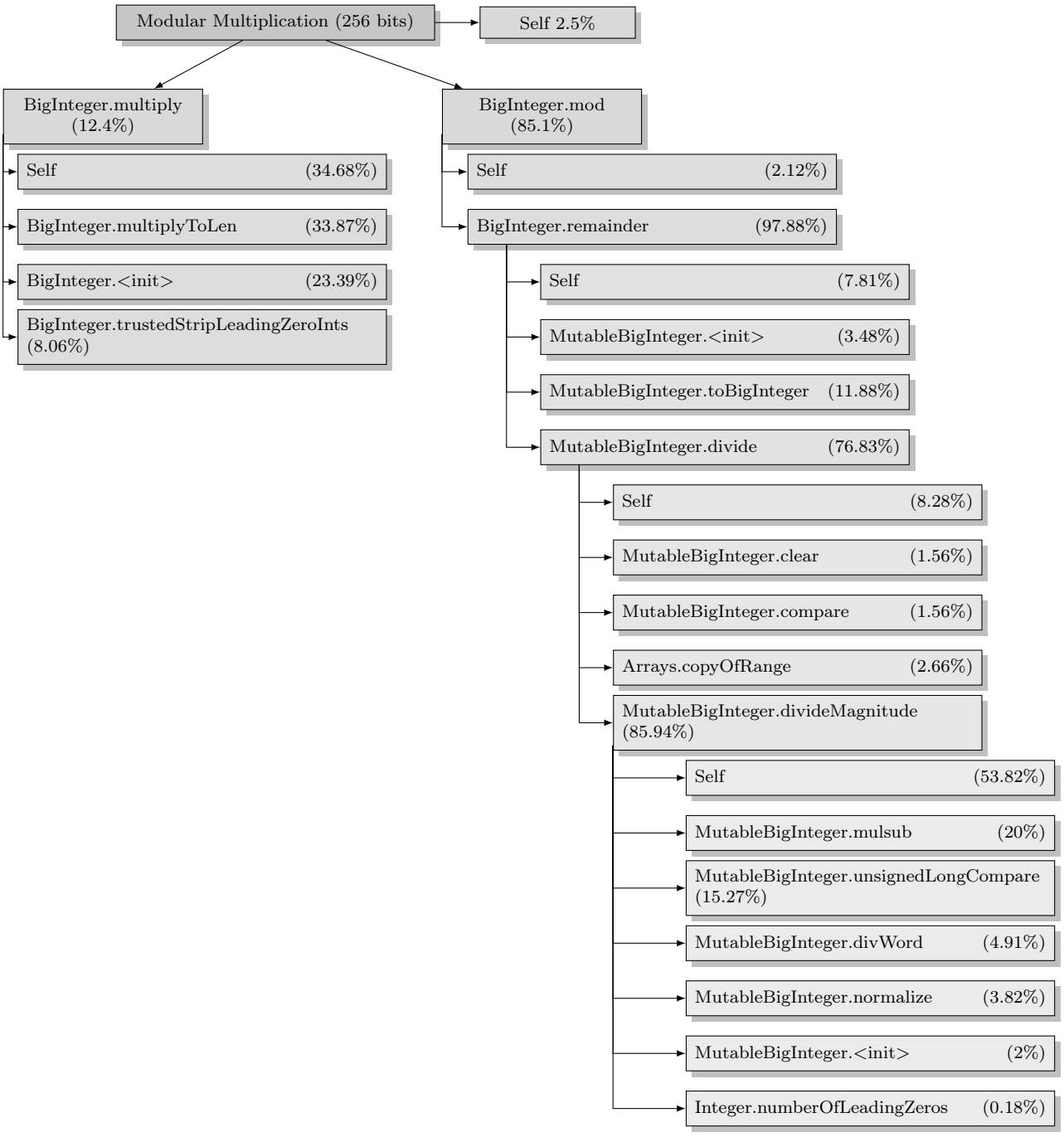


FIGURE E.2 – Anatomie d'une méthode Java effectuant une multiplication modulaire sur des entiers de 256 bits, à l'aide de la classe BigInteger de Java, sur une plate-forme x64 (obtenue de Netbeans profiler).

Bibliographie

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[ACL19] Rodrigo Abarzúa, Claudio Valencia Cordero, and Julio López. Survey for performance & security problems of passive side-channel attacks countermeasures in ECC. *IACR Cryptology ePrint Archive*, 2019 :10, 2019.

[AFT+08] Frederic Amiel, Benoit Feix, Michael Tunstall, Claire Whelan, and William P Marnane. Distinguishing multiplications from squaring operations. In *International Workshop on Selected Areas in Cryptography*, pages 346–360. Springer, 2008.

[AKS12] K. D. Akdemir, D. Karakoyunlu, and B. Sunar. Non-linear error detection for elliptic curve cryptosystems. *IET Information Security*, 6(1) :28–40, March 2012.

[Bab86] László Babai. On lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6 :1–13, 1986.

[Bal14] Marco Baldi. *QC-LDPC Code-Based Cryptography.* SpringerBriefs in Electrical and Computer Engineering. Springer International Publishing, 2014.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer.

[BBJ+08] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In *International Conference on Cryptology in Africa*, pages 389–405. Springer, 2008.

[BCM+07] Andrew Byrne, Francis Crowe, William Peter Marnane, Nicolas Meloni, Arnaud Tisserand, and Emanuel Popovici. Spa resistant elliptic curve cryptosystem using addition chains. *International Journal of High Performance Systems Architecture*, 1(2) :133–142, 2007.

[BDK98] J-C Bajard, L-S Didier, and Peter Kornerup. An rns montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7) :766–776, 1998.

[BH09] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 667–684. Springer, 2009.

[BIP04] J.-C. Bajard, Laurent Imbert, and Thomas Plantard. Modular number systems : Beyond the mersenne family. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada*, pages 159–169, 2004.

[BIP05] J.-C. Bajard, Laurent Imbert, and Thomas Plantard. Arithmetic operations in the polynomial modular number system. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17) 2005, Cape Cod, MA, USA*, pages 206–213, 2005.
Extended (complete) version available at : https://hal-lirmm.ccsd.cnrs.fr/lirmm-00109201/document.

[BJP+15] Aurélie Bauer, Eliane Jaulmes, Emmanuel Prouff, Jean-René Reinhard, and Justine Wild. Horizontal collision correlation attack on elliptic curves. *Cryptography and Communications*, 7(1) :91–119, 2015.

[BLa] D. J. Bernstein and T. Lange. Explicit-Formulas Database. https://www.hyperelliptic.org/EFD/.

[BLb] D. J. Bernstein and T. Lange. Explicit-formulas database : Jacobian coordinates for short weierstrass curves. http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian.html.

[BLc] D. J. Bernstein and T. Lange. Explicit-formulas database : Projective coordinates for short weierstrass curves. http://hyperelliptic.org/EFD/g1p/auto-shortw-projective.html.

[BLd] D. J. Bernstein and T. Lange. Explicit-formulas database : Short weierstrass curves. http://hyperelliptic.org/EFD/g1p/auto-shortw.html.

[BLe] D. J. Bernstein and T. Lange. Safecurves. http://safecurves.cr.yp.to/.

[BLf] D. J. Bernstein and T. Lange. Safecurves : Fields. http://safecurves.cr.yp.to/field.html.

[Bla83] George R Blakely. A computer algorithm for calculating the product ab modulo m. *IEEE Transactions on Computers*, 100(5) :497–500, 1983.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In *Annual International Cryptology Conference*, pages 131–146. Springer, 2000.

[BMP05] Jean-Claude Bajard, Nicolas Meloni, and Thomas Plantard. Efficient rns bases for cryptography. In *IMACS'05 : World Congress : Scientific ComputationApplied Mathematics and Simulation*, 2005.

[BOS06] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. Sign change fault attacks on elliptic curve cryptosystems. In *Fault Diagnosis and Tolerance in Cryptography*, pages 36–52. Springer, 2006.

[Bru15] Billy Bob Brumley. Faster software for fast endomorphisms. In *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015*, pages 127–140, 2015.

[BSS+16] Vijay Bahadur, David Selvakumar, PM Sobha, et al. Reconfigurable side channel attack resistant true random number generator. In *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, pages 1–6. IEEE, 2016.

[BVdPSY14] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92. Springer, 2014.

[BZ10] Richard P. Brent and Paul Zimmermann. *Modular arithmetic and the FFT*, page 47–78. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010.

[CFG+11] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Square always exponentiation. In *International Conference on Cryptology in India*, pages 40–57. Springer, 2011.

[CH03] Jaewook Chung and Anwar Hasan. More generalized mersenne numbers. In *International Workshop on Selected Areas in Cryptography*, pages 335–347. Springer, 2003.

[CHS14] C. Costello, H. Hisil, and B. Smith. Faster compact diffie–hellman : endomorphisms on the x-line. In *Advances in Cryptology–EUROCRYPT 2014*, pages 183–200. Springer, 2014.

[CJ05] Mathieu Ciet and Marc Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, codes and cryptography*, 36(1) :33–43, 2005.

[CL15] Craig Costello and Patrick Longa. Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the mersenne prime. In *Advances in cryptology – ASIACRYPT 2015. 21st international conference on the theory and application of cryptology and information security, Auckland, New Zealand, November 29 – December 3, 2015. Proceedings. Part I*, pages 214–235. Berlin : Springer, 2015.

[Coh93] Henri Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1993.

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International Workshop*

on Cryptographic Hardware and Embedded Systems, pages 292–302. Springer, 1999.

[Cra92] Richard E Crandall. Method and apparatus for public key exchange in a cryptographic system, October 27 1992. US Patent 5,159,632.

[DDEM+19] Laurent-Stéphane Didier, Fangan Yssouf Dosso, Nadia El Mrabet, Jérémy Marrez, and Pascal Véron. Randomization of arithmetic over polynomial modular number system. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 199–206, June 2019.

[DDV20] Laurent-Stéphane Didier, Fangan Yssouf Dosso, and Pascal Véron. Efficient modular operations using the Adapted Modular Number System. *Journal of Cryptographic Engineering*, pages 1–23, 2020.

[DGH+16] Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. Improving the big mac attack on elliptic curve cryptography. In *The New Codebreakers*, pages 374–386. Springer, 2016.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6) :644–654, 1976.

[DHMV18] Yssouf Dosso, Fabien Herbaut, Nicolas Méloni, and Pascal Véron. Euclidean addition chains scalar multiplication on curves with efficient endomorphism. *Journal of Cryptographic Engineering*, 8(4) :351–367, 2018.

[Did98] Laurent-Stéphane Didier. *Division et multiplication modulaire en représentation modulaire.* PhD thesis, Université de Provence, 1998.

[DK91] Stephen R. Dussé and Burton S. Kaliski. A cryptographic library for the motorola dsp56000. In Ivan Bjerre Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, pages 230–244, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[DKJ05] William Dupuy and Sébastien Kunz-Jacques. Resistance of randomized projective coordinates against power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–14. Springer, 2005.

[DV17] Fangan-Yssouf Dosso and Pascal Véron. Cache timing attacks countermeasures and error detection in euclidean addition chains based scalar multiplication algorithm for elliptic curves. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 163–168. IEEE, 2017.

[EG12] Nadia El Mrabet and Nicolas Gama. Efficient multiplication over extension fields. In *WAIFI*, volume 7369 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2012.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4) :469–472, 1985.

[EN09] Nadia El Mrabet and Christophe Nègre. Finite field multiplication combining AMNS and DFT approach for pairing cryptography. In *ACISP*, volume 5594 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 2009.

[FHLS15] Armando Faz-Hernández, Patrick Longa, and Ana H Sánchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on glv–gls curves (extended version). *Journal of Cryptographic Engineering*, 5(1) :31–52, 2015.

[FIP99] PUB FIPS. 46-3. data encryption standard (des). *National Institute of Standards and Technology*, 25(10) :1–22, 1999.

[FIP01] PUB FIPS. 197 : Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441) :0311, 2001.

[FRM+08] Julien Francq, Jean-Baptiste Rigaud, Pascal Manet, Assia Tria, and Arnaud Tisserand. Error detection for borrow-save adders dedicated to ECC unit. In *FDTC 2008, Washington, DC, USA, 10 August 2008*, 2008.

[FV12] Junfeng Fan and Ingrid Verbauwhede. An updated survey on secure ecc implementations : Attacks, countermeasures and cost. In *Cryptography and Security : From Theory to Applications*, pages 265–282. Springer, 2012.

[Ga] Torbjörn Granlund and al. GNU multiple precision arithmetic library 6.1.2. https://gmplib.org/.

[Gar59] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EL 8(6) :140–147, 1959.

[Geo13] Mariya Georgieva. *Analyse probabiliste de la réduction des réseaux euclidiens cryptographiques*. PhD thesis, Université de Caen, 2013.

[GG00] Oded Goldreich and Shafi Goldwasser. On the limits of nonapproximability of lattice problems. *Journal of Computer and System Sciences*, 60(3) :540–563, 2000.

[GH98] J. Von Zur Gathen and S. Hartlieb. Factoring modular polynomials. *Journal of Symbolic Computation*, 26(5) :583 – 606, 1998.

[GI13] A. Guillevic and S. Ionica. Four-dimensional GLV via the weil restriction. In *Advances in Cryptology - ASIACRYPT 2013*, pages 79–96, 2013.

[Gio04] Pascal Giorgi. *Arithmétique et algorithmique en algebre linéaire exacte pour la bibliotheque LinBox*. PhD thesis, Ecole Normale Supérieure Lyon, 2004.

[GJM+11] Raveen R Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on weierstraß elliptic

curves from co-z arithmetic. *Journal of cryptographic engineering*, 1(2) :161, 2011.

[GLS09] S. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 518–535. Springer Berlin Heidelberg, 2009.

[GLV01] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology — CRYPTO*, volume 2139 of *LNCS*, pages 190–200. Springer, 2001.

[GMSS99] O. Goldreich, D. Micciancio, S. Safra, and J.-P. Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. *Information Processing Letters*, 71(2) :55 – 61, 1999.

[Gor98] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1) :129 – 146, 1998.

[Gou03] L. Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In *Public Key Cryptography*, volume 2567 of *LNCS*, pages 199–210. Springer, 2003.

[Had93] Jacques Hadamard. Résolution d'une question relative aux déterminants. *Bull. des sciences math.*, 2 :240–246, 1893.

[Hia00] Ahmad A Hiasat. New efficient structure for a modular multiplier for rns. *IEEE Transactions on Computers*, 49(2) :170–174, 2000.

[HLM$^+$10] Fabien Herbaut, Pierre-Yvan Liardet, Nicolas Méloni, Yannick Téglia, and Pascal Véron. Random euclidean addition chain generation and its application to point multiplication. In *International Conference on Cryptology in India*, pages 238–261. Springer, 2010.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[Hor19] William George Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 109 :308–335, 1819.

[HV10] Fabien Herbaut and Pascal Véron. A public key cryptosystem based upon euclidean addition chains. In *International Conference on Sequences and Their Applications*, pages 284–297. Springer, 2010.

[HWCD08] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 326–343. Springer, 2008.

[Joh99] Anna M. Johnston. A generalized qth root algorithm. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 929–930. Society for Industrial and Applied Mathematics, 1999.

[JT01] Marc Joye and Christophe Tymen. Protections against differential analysis for elliptic curve cryptography — an algebraic approach —. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 377–390. Springer, Berlin, Heidelberg, 2001.

[JY02] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 291–302. Springer, 2002.

[KÏ2] Emilia Käsper. Fast elliptic curve cryptography in openssl. In *Proceedings of the 2011 International Conference on Financial Cryptography and Data Security*, FC'11, pages 27–39. Springer-Verlag, 2012.

[KAK96] C Kaya Koc, Tolga Acar, and Burton S Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE micro*, 16(3) :26–33, 1996.

[KF18] Frank R. Kschischang and Chen Feng. An introduction to lattices and their applications in communications, 2018. http://shannon.engr.tamu.edu/wp-content/uploads/sites/138/2018/05/talk.pdf.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[Knu97] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[KO63] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[KO90] J K. Omura. A public key cell design for smart card chips. *Information Technology - IT*, 01 1990.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177) :203–209, 1987.

[Koc96a] P. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *CRYPTO*, LNCS, pages 104–113. Springer, 1996.

[Koc96b] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4) :515–534, Dec 1982.

[LS14] Patrick Longa and Francesco Sica. Four-dimensional Gallant–Lambert–Vanstone scalar multiplication. *Journal of Cryptology*, 27(2) :248–283, 2014.

[LWG+15] Zhe Liu, Husen Wang, Johann Großschädl, Zhi Hu, and Ingrid Verbauwhede. Vlsi implementation of double-base scalar multiplication on a twisted edwards curve with an efficiently computable endomorphism. *IACR Cryptology ePrint Archive*, 2015 :421, 2015.

[MDS99] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Power analysis attacks of modular exponentiation in smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 144–157. Springer, 1999.

[Mel07] Nicolas Meloni. New point addition formulae for ecc applications. In *International Workshop on the Arithmetic of Finite Fields*, pages 189–201. Springer, 2007.

[Mil85] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[Min10] H. Minkowski. *Geometrie der Zahlen*. Number vol. 2 in Geometrie der Zahlen. B.G. Teubner, 1910.

[Mir] Miracl. Miracl cryptographic library. https://libraries.docs.miracl.com/.

[Mis93] Bhubaneswar Mishra. *Algorithmic Algebra*. Springer-Verlag, Berlin, Heidelberg, 1993.

[Möl01] Bodo Möller. Algorithms for multi-exponentiation. In *International Workshop on Selected Areas in Cryptography*, pages 165–180. Springer, 2001.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170) :519–521, 1985.

[Mon87] P. Montgomery. Speeding the Pollard and elliptic curve method of factorization. In *Mathematics of computation*, pages 243–264. Springer, 1987.

[NIS19] NIST. Cryptographic standards and guidelines, 2019. https://csrc.nist.gov/Projects/Cryptographic-Standards-and-Guidelines.

[NP08] Christophe Negre and Thomas Plantard. Efficient modular arithmetic in adapted modular number system using lagrange representation. In *Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia*, pages 463–477, 2008.

[NP17] Christophe Negre and Thomas Plantard. Efficient regular modular exponentiation using multiplicative half-size splitting. *Journal of Cryptographic Engineering*, 7(3) :245–253, Sep 2017.

[OKS06] Acıiçmez Onur, Çetin Kaya Koc, and Jean-Pierre Seifer. Predicting secret keys via branch prediction. In *Proceedings of CT-RSA'07*, CT-RSA'07, pages 225–242. Springer-Verlag, 2006.

[Pag02] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002(169), 2002.

[Pag03] Daniel Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1) :30–44, 2003.

[PCRS09] Salvatore Pontarelli, Gian-Carlo Cardarilli, Marco Re, and Adelio Salsano. Error detection in addition chain based ecc point multiplication. In *2009 15th IEEE International On-Line Testing Symposium*, pages 192–194. IEEE, 2009.

[PJKL02] Young-Ho Park, Sangtae Jeong, Chang Han Kim, and Jongin Lim. An alternate decomposition of an integer for faster point multiplication on certain elliptic curves. In *International Workshop on Public Key Cryptography*, pages 323–334. Springer, 2002.

[Pla05] Thomas Plantard. *Arithmétique modulaire pour la cryptographie*. PhD thesis, Montpellier 2 University, France, 2005.

[PQ12] Christophe Petit and Jean-Jacques Quisquater. On polynomial systems arising from a weil descent. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 451–466. Springer, 2012.

[Pro] The OpenSSL Project. Openssl. https://www.openssl.org/.

[PVCTM15] Julien Proy, Nicolas Veyrat-Charvillon, Arnaud Tisserand, and Nicolas Méloni. Full hardware implementation of short addition chains recoding for ecc scalar multiplication. In *Compas : Conférence d'informatique en Parallélisme, Architecture et Système*, 2015.

[Rei60] George W Reitwiesner. Binary arithmetic. In *Advances in computers*, volume 1, pages 231–308. Elsevier, 1960.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[Sa] Victor Shoup and al. Ntl : A library for doing number theory. https://www.shoup.net/ntl/.

[Sa18] William Stein and al. Sagemath, 2018. http://www.sagemath.org/index.html.

[SCQ02] Francesco Sica, Mathieu Ciet, and Jean-Jacques Quisquater. Analysis of the gallant-lambert-vanstone method based on efficient endomorphisms : Elliptic and hyperelliptic curves. In *International Workshop on Selected Areas in Cryptography*, pages 21–36. Springer, 2002.

[SD16] Noah Stephens-Davidowitz. Lecture 5 : Cvp and babai's algorithm, 2016. http://www.noahsd.com/mini_lattices/05__babai.pdf.

[SEC14] SEC. Standard for efficient cryptography, 2014. https://www.secg.org/.

[SF10] Jerome Solinas and David E. Fu. Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2. RFC 5903, June 2010.

[Smi13] B. Smith. Families of fast elliptic curves from ıı-curves. In *Advances in Cryptology - ASIACRYPT 2013*, pages 61–78, 2013.

[SMKLM01] Yen Sung-Ming, Seungjoo Kim, Seongan Lim, and SangJae Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In *International Conference on Information Security and Cryptology*, pages 414–427. Springer, 2001.

[Sol99] Jerome A Solinas. *Generalized mersenne numbers*. Citeseer, 1999.

[SOP08] N. P. Smart, E. Oswald, and D. Page. Randomised representations. *IET Information Security*, 2(2) :19–27, June 2008.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4) :281–292, 1971.

[Tak92] Naofumi Takagi. A radix-4 modular multiplication hardware algorithm for modular exponentiation. *IEEE Transactions on Computers*, 41(8) :949–956, 1992.

[Tay81] F Taylor. Large moduli multipliers for signal processing. *IEEE Transactions on circuits and systems*, 28(7) :731–736, 1981.

[TJ10] Michael Tunstall and Marc Joye. Coordinate blinding over large prime fields. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 443–455. Springer, 2010.

[vEB81] P. van Emde-Boas. *Another NP-complete partition problem and the complexity of computing short vectors in a lattice*. Report. Department of Mathematics. University of Amsterdam. Department, Univ., 1981.

[Wal04] Colin D Walter. Simple power analysis of unified code for ecc double and add. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 191–204. Springer, 2004.

[Woo16] Henry Woody. Polynomial resultants, 2016. http://buzzard.ups.edu/courses/2016spring/projects/woody-resultants-ups-434-2016.pdf.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload : A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[YJ00] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on computers*, 49(9) :967–970, 2000.

[YK75] Andrew C Yao and Donald E Knuth. Analysis of the subtractive algorithm for greatest common divisors. *Proceedings of the National Academy of Sciences*, 72(12) :4720–4722, 1975.

Fangan Yssouf DOSSO
Laboratoire IMATH, Université de Toulon

Contribution de l'arithmétique des ordinateurs aux implémentations résistantes aux attaques par canaux auxiliaires

Résumé

Cette thèse porte sur deux éléments actuellement incontournables de la cryptographie à clé publique, qui sont l'arithmétique modulaire avec de grands entiers et la multiplication scalaire sur les courbes elliptiques (ECSM). Pour le premier, nous nous intéressons au système de représentation modulaire adapté (AMNS), qui fut introduit par Bajard et al. en 2004. C'est un système de représentation de restes modulaires dans lequel les éléments sont des polynômes. Nous montrons d'une part que ce système permet d'effectuer l'arithmétique modulaire de façon efficace et d'autre part comment l'utiliser pour la randomisation de cette arithmétique afin de protéger l'implémentation des protocoles cryptographiques contre certaines attaques par canaux auxiliaires. Pour l'ECSM, nous abordons l'utilisation des chaînes d'additions euclidiennes (EAC) pour tirer parti de la formule d'addition de points efficace proposée par Méloni en 2007. L'objectif est d'une part de généraliser au cas d'un point de base quelconque l'utilisation des EAC pour effectuer la multiplication scalaire ; cela, grâce aux courbes munies d'un endomorphisme efficace. D'autre part, nous proposons un algorithme pour effectuer la multiplication scalaire avec les EAC, qui permet la détection de fautes qui seraient commises par un attaquant que nous détaillons.

Mots clés : Arithmétique modulaire, Cryptographie sur les courbes elliptiques, Contre-mesures aux attaques par canaux auxiliaires, Système de représentation modulaire adapté, Randomisation de l'arithmétique modulaire, Réseaux euclidiens, Multiplication scalaire sur courbe elliptique, Chaînes d'additions euclidiennes, Détection de fautes.

Computer arithmetic contribution to side channel attacks resistant implementations

Abstract

This thesis focuses on two currently unavoidable elements of public key cryptography, namely modular arithmetic over large integers and elliptic curve scalar multiplication (ECSM). For the first one, we are interested in the Adapted Modular Number System (AMNS), which was introduced by Bajard et al. in 2004. In this system of representation, the elements are polynomials. We show that this system allows to perform modular arithmetic efficiently. We also explain how AMNS can be used to randomize modular arithmetic, in order to protect cryptographic protocols implementations against some side channel attacks. For the ECSM, we discuss the use of Euclidean Addition Chains (EAC) in order to take advantage of the efficient point addition formula proposed by Meloni in 2007. The goal is to first generalize to any base point the use of EAC for ECSM ; this is achieved through curves with one efficient endomorphism. Secondly, we propose an algorithm for scalar multiplication using EAC, which allows error detection that would be done by an attacker we detail.

Keywords : Modular arithmetic, Elliptic curve cryptography, Side-channel countermeasures, Adapted modular number system, Modular arithmetic randomization, Euclidean lattices, Elliptic curve scalar multiplication, Euclidean addition chains, Fault detection.