ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Jérémy Métairie

préparée à l'unité de recherche IRISA (UMR6074)
(Institut de recherche en informatique et systèmes aléatoires)
École Nationale Supérieure des Sciences Appliquées et de Technologie (ENSSAT) - Équipe CAIRN

Contributions aux opérateurs arithmétiques $\mathbf{GF}(2^m)$ et leurs applications à la cryptographie sur courbes elliptiques.

Thèse soutenue à Lannion le 19 mai 2016

devant le jury composé de :

Laurent-Stéphane DIDIER
Professeur à l'Université de Toulon / rapporteur

Lilian BOSSUET
Maître de conférence HDR à l'Université de Saint-Étienne / rapporteur

Christophe NÈGRE
Maître de conférence à l'Université de Perpignan / examinateur

Caroline FONTAINE
Chargée de recherche au CNRS, Brest / examinateur

Arnaud TISSERAND
Directeur de recherche au CNRS, IRISA, Lannion / directeur de thèse

Emmanuel CASSEAU
Professeur des Universités, ENSSAT-Université de Rennes 1 / co-directeur de thèse

Remerciements

Je remercie mes directeurs Arnaud et Emmanuel de m'avoir permis de réaliser cette thèse au sein de l'équipe CAIRN à Lannion. Je les remercie aussi pour leurs conseils mais également pour leur aide précieuse concernant la « dernière ligne droite » ; la préparation de ma soutenance n'en a été que facilitée.

Je remercie l'ANR ; mon travail a été intégralement financé par le projet ANR Pavois *.

Je remercie l'ensemble du jury d'avoir accepté l'invitation et d'avoir fait le déplacement, dans les Côtes-d'Armor, dans un contexte social particulier. Je remercie notamment Laurent-Stéphane Didier et Lilian Bossuet d'avoir été rapporteurs. Leurs retours ont été très positifs, j'en ai été ravi. Je remercie Caroline Fontaine et Christophe Nègre pour leurs questions pertinentes et leur calme apparent.

Je remercie les « gens » de CAIRN. Les visages ont changé durant ces trois ans : je ne parle pas seulement des rides qui tiraillent nos portraits. Je parle ainsi des allées et venues du personnel, du renouvellement. Cela a été l'occasion de rencontrer des profils variés, des personnalités d'horizons différents, de cultures différentes. Un *melting-pot* riche et très lié au monde de la recherche, qui ne se soucie guère des frontières. Je remercie la bonne humeur de CAIRN qui m'a souvent remonté le moral dans les moments de doute.

Une pensée pour les « gens » de Rennes (et d'ailleurs) qui, si je ne les ai pas vus souvent, ont contribué, de par leur présence numérique, à rendre ce travail de thèse plus digeste.

Je remercie mes parents : sans leur soutien logistique et financier, tout cela n'aurait pas été possible. Merci pour leur compréhension : les longues études ne s'imposaient pas pour les « petits gens » que nous sommes. Merci.

Je remercie aussi Marion qui, par je ne sais quel obscur stratagème, est parvenue à me supporter. Chapeau.

∗. plus de détails accessibles sur http://pavois.irisa.fr/ à l'heure où j'écris ces lignes.

Table des matières

1 Introduction **1**

2 Quelques mots sur la cryptographie **5**
2.1 La problématique . . . 6
Algorithme de Diffie-Hellman . . . 6
Le problème du Logarithme Discret . . . 7
Algorithme RSA . . . 8
Algorithme Elgamal . . . 9
2.2 Courbes Elliptiques . . . 10
Définitions . . . 10
Types de Coordonnées . . . 14
Multiplication Scalaire . . . 16
2.3 Et les FPGAs ? . . . 23
2.4 Chiffres d'implémentations de multiplieurs sur divers FPGAs. . . . 25

3 « Petit » multiplieur/inverseur (PISO) [†] en base normale dans GF(2^m) **28**
3.1 Introduction . . . 28
3.2 État de l'art . . . 30
3.2.1 Le corps fini GF(2^m) . . . 30
3.2.2 Représentation des éléments d'un corps fini GF(2^m) . . . 31
3.2.3 Additions, Traces et Multiplications en base normale GF(2^m) . . . 31
3.2.4 Algorithmes d'Inversion dans GF(2^m) . . . 38
3.3 Solution proposée . . . 41
3.3.1 Décaler avec des blocs-mémoires . . . 42
3.3.2 Exploitation des formes $A^{2^k} \times A$ . . . 44
3.3.3 Multiplication et inversion en base normale permutée . . . 44
3.3.4 Estimation du coût . . . 47
3.4 Détails de l'implémentation sur FPGA et résultats . . . 49
3.5 Conclusion . . . 54

4 Crypto-processeur intégrant une unité de *Halving* **55**
4.1 Opération de *Halving* . . . 55
4.2 Opérateur de résolution de $\lambda^2 + \lambda = c$ . . . 58
4.3 Additionneur en base normale dans GF(2^m) . . . 61
4.4 Racine carrée . . . 61
4.5 Parallélisme *halve-and-add* et *double-and-add* . . . 62

†. Parallel-In Serial-Out.

4.6 Architecture proposée . . . 65
4.7 Performances . . . 69
4.8 Évaluation de la sécurité physique de notre crypto-processeur. . . . 73
4.8.1 La génération des *templates* . . . 73
4.8.2 Exploitation des *templates* . . . 76
4.8.3 Des idées pour réduire la vulnérabilité du crypto-processeur . . . 77
4.8.4 Génération de l'aléa . . . 79
4.8.5 Résultats des attaques par Templates . . . 80
4.9 Conclusion . . . 85

5 RNS dans GF(2^m) 86
5.1 Réduction modulaire en RNS . . . 86
5.1.1 Algorithme de Montgomery . . . 86
5.1.2 Multiplication de Mastrovito . . . 88
5.1.3 Théorème chinois des restes . . . 91
5.2 Φ-RNS . . . 95
5.2.1 Quelques notions mathématiques . . . 95
5.2.2 Architecture proposée . . . 101
5.2.3 Résultats d'implémentation et comparaisons . . . 105
5.2.4 Racine carrée en RNS . . . 110
5.2.5 Inversion en RNS . . . 114
5.3 Conclusion . . . 118

6 Conclusion 120

A Multiplications en Base Normale sur CPU 122
A.1 État de l'art . . . 123
A.2 Parallélisme de la Multiplication en Base Normale . . . 127
A.3 Parallélisme dans l'Inversion en Base Normale . . . 129
A.4 Conclusion . . . 131

Bibliographie 133

Chapitre 1

Introduction

La sécurité informatique est un marché en pleine expansion, en témoignent les chiffres d'une étude réalisée en 2015 [1]. Il va, d'après certaines estimations, croître d'un montant de 77 milliards de dollars en 2015 à plus de 170 en 2020, soit une croissance annuelle d'environ 17%. Alors qu'elle n'était qu'un sujet secondaire au début des années 2000, la sécurité est devenue un point névralgique important, notamment avec l'émergence du « *cloud* » et de fait, de l'apparition de la problématique de la vie privée. Les révélations d'Edward Snowden en décembre 2013 sur les activités d'espionnage de la NSA (*National Security Agency*) ont d'ailleurs amplifié le phénomène. La protection des données privées s'est rapidement propulsée au rang de *réel argument* de vente auprès des principales sociétés d'informatique (*Apple*, *Google*, *Microsoft*, etc). Le domaine de la sécurité informatique recouvre de nombreuses aires de recherche (qui s'imbriquent parfois) : la cryptographie [2], la complexité algorithmique [3], la sécurité des réseaux [4] etc.

Dans cette thèse, nous nous intéresserons plus spécifiquement à la cryptographie, qui est souvent référée comme « la science du secret ». La préoccupation de communiquer sans que son message ne puisse être lu par des parties adverses a depuis très longtemps été présente dans l'esprit des êtres humains. Le chiffrement de César (-100 avant J.C.) en est un exemple manifeste. Certes, à l'époque, le chiffrement était très rudimentaire. Dans le cas présent, il s'agissait d'un décalage constant des lettres (A devient D, B devient E, ainsi de suite). Le message **ATTAQUE CE SOIR** devenait alors **DWWDTXH FH VRLU**, incompréhensible pour celui qui ne connaissait pas le système d'encodage. D'après les principes de Kerckhoffs [5], la sécurité d'un système ne doit pas dépendre de la connaissance de l'algorithme de chiffrement exclusivement. Chaque système de chiffrement sera public mais prendra en entrée deux paramètres : la donnée à chiffrer (secrète) et une **clef** (qui sera elle aussi, confidentielle). Voir figure 1.1. Un exemple d'algorithme

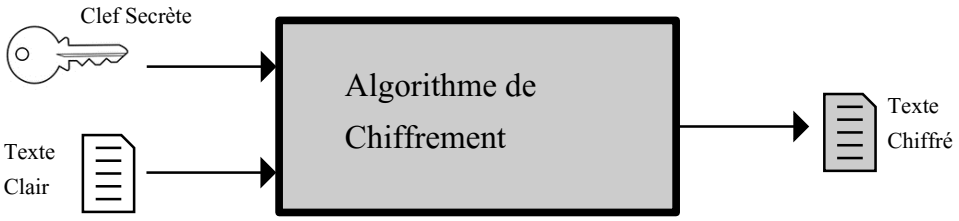


FIGURE 1.1: Chiffrement d'un texte à l'aide d'une clef secrète et d'un algorithme de chiffrement.

serait le décalage de lettre que nous venons de voir avec une clef qui quantifiera ce dit décalage. Par exemple, l'algorithme de César est un algorithme à décalage avec une clef de chiffrement 3 (nous décalons les lettres de trois dans l'ordre alphabétique).

Évidemment, aujourd'hui ces méthodes seraient beaucoup trop vulnérables, d'autant plus avec la puissance des ordinateurs dont nous disposons. En 2015, les systèmes de chiffrement reposent sur des problèmes mathématiques réputés difficiles que même les machines les plus puissantes ne peuvent résoudre en un « temps raisonnable ». La notion de « temps raisonnable » est occasionnellement discutée. Il parait aberrant de fournir une grande sécurité à des informations qui seront publiques dans les mois à venir. Cette sécurité liée au « temps raisonnable » est, en particulier, liée à la longueur de la clef (et aussi de l'algorithme employé, nous y reviendrons). Cette longueur de clef a d'ailleurs été réglementée par la loi en France, la cryptographie étant considérée, jusqu'en 1996, comme une « arme de guerre » [6].

Lors du chiffrement d'un message, de nombreux calculs sont effectués. Typiquement, pour la cryptographie dite « asymétrique », beaucoup d'entre eux sont basés sur du calcul modulaire, c'est à dire modulo un nombre premier ou polynôme irréductible. La puissance de traitement requise pour un chiffrement dépend du type d'algorithme utilisé, de la longueur de la clef. Les cartes à puce fournissent suffisamment de capacité calculatoire pour chiffrer (et déchiffrer) : autant dire qu'il y a eu, dans l'idée de chiffrement, un paradigme supplémentaire, qui est venu se rattacher, par l'expérience quotidienne, à la cryptographie. Il faut que les calculs ne soient pas trop « compliqués » pour qu'ils soient réalisables sur des puces, des systèmes embarqués, etc. Bref, la contrainte est aujourd'hui de concevoir des dispositifs sûrs et rapides. Ce que nous ignorions jusqu'en 1995 [7] est qu'une puce mal protégée peut laisser fuir des informations secrètes. La consommation d'énergie, le temps d'exécution d'un programme peuvent varier dans le temps ; ces variations sont en partie corrélées avec le secret. Des informations auxiliaires permettent d'attaquer, physiquement, un système alors que les algorithmes exécutés au sein du circuit ont été éprouvés et considérés comme théoriquement sûrs. Le concept

«d'attaques par canaux cachés» (*side-channel attacks* en anglais) était né. Immanquablement, les scientifiques ont cherché à combler cette faiblesse à travers un remaniement des algorithmes cryptographiques les plus courants. Le but était de réduire au possible la corrélation qu'il existe entre information secrète et activité du circuit. Les nouvelles cibles sont *a priori* maintenant protégées (dans une moindre mesure) contre les attaques par canaux cachés (ou auxiliaires). Ces protections ne doivent, cependant, pas impacter les performances de manière significative.

Dans cette thèse, nous nous sommes intéressés aux plates-formes FPGA (*Field Programmable Gates Array*), qui sont des circuits reconfigurables. Ils peuvent être programmés pour effectuer une tâche spécifique. Cette flexibilité, qui, liée au parallélisme intrinsèque dont dispose cette technologie, permet d'intégrer des architectures spécialisées pour le calcul cryptographique (et généralement plus rapide qu'un *CPU* et surtout moins énergivore). Dans cette thèse, nous nous sommes principalement intéressés aux chiffrements sur courbes elliptiques définies sur $\mathrm{GF}(2^m)$. Il est vrai que le corps $\mathrm{GF}(2^m)$ tend à être délaissé. C'est également vrai pour les corps en petites caractéristiques $(3, 5, 7, \ldots)$. Des chercheurs ont en effet réussi à trouver des attaques en complexité sous-exponentielle (à la place d'exponentielle) [8]. Les corps en petites caractéristiques constituent une arithmétique « simple » qui permet des implémentations rapides. Elle semble, dans un même temps, être l'une des sources de cette nouvelle vulnérabilité. Malgré tout, ces corps restent attrayants pour les circuits reprogrammables et les systèmes embarqués.

Le mémoire est organisé de la façon suivante : nous présenterons en préambule, dans le chapitre 2, les problèmes sous-jacents de la cryptographie au sens large. Nous étudierons le principe de la cryptographie « asymétrique » et introduirons brièvement la cryptographie sur courbes elliptiques. Nous nous sommes ensuite (chapitre 3) intéressés aux bases normales et à l'opération d'inversion dans cette représentation (qui peut être récurrente dans le cadre d'un algorithme type « échelle de Montgomery combinée au *halving* » [9], nous y reviendrons). Nous introduirons la notion de « base normale permutée » qui nous autorise une accélération de l'inversion. Nous avons implémenté un processeur-cryptographique complet (sur FPGA) dans le chapitre 4, opérant sur courbes elliptiques définies sur $\mathrm{GF}(2^m)$. Nous voulions observer l'effet du parallélisme sur l'accélération qu'il peut procurer mais aussi étudier sa propriété comme moyen de contre-mesure face aux canaux cachés. Nous montrerons que le parallélisme seul des calculs ne peut nous prémunir des attaques dites par *templates* [10]. Enfin, dans le chapitre 5, nous étudierons la pertinence de l'utilisation de la représentation **RNS** (*Residue Number System*) [11] dans $\mathrm{GF}(2^m)$. Nous proposerons un nouvel algorithme de multiplication modulaire basé

sur l'algorithme de réduction de Montgomery [12].

En annexe, nous toucherons quelques mots sur la multiplication et l'inversion en base normale sur CPU(Annexe A).

Chapitre 2

Quelques mots sur la cryptographie

La cryptographie est un domaine dont l'objectif principal est de « protéger » l'information, de la rendre inintelligible à celles ou ceux à qui elle n'est pas destinée. La cryptographie repose sur des algorithmes solides qui s'appuient eux-même sur des problèmes mathématiques réputés difficiles (logarithme discret, factorisation des grands nombres, etc).

L'histoire de la cryptographie est passionnante, sa pratique remonterait à l'Antiqué sous des formes évidemment plus rudimentaires qu'aujourd'hui. Sans retracer les méandres de son évolution, notons tout de même que son utilisation a connu un véritable boom peu avant la seconde guerre mondiale par le développement d'une machine à « chiffrer » : Enigma [2]. Le récit a été d'ailleurs popularisé en 2015 par le film americo-britannique « *Imitation game* » dans lequel Alain Turing, aujourd'hui considéré comme l'un des pères fondateurs de l'informatique, parvient à casser le chiffrement allemand à travers un processus d'automatisation des calculs.

Bien qu'il soit complexe, sur papier, d'attaquer ces systèmes de protection, l'implantation matérielle ou logicielle, si elle est négligée, peut apporter à des entités malveillantes des renseignements complémentaires (temps d'exécution, consommation d'énergie, etc) : on parle de canaux cachés ou de canaux auxiliaires. De par ces faiblesses, un attaquant peut parvenir à récupérer des informations secrètes (telle que la clef de chiffrement).

L'enjeu de cette thèse est double : il faut d'un côté, pouvoir concevoir des opérateurs arithmétiques rapides et peu gourmands en silicium, de l'autre côté, pouvoir concevoir des architectures robustes qui limitent la « fuite » du secret. Ces contre-mesures ne devront pas pénaliser, dans la mesure du possible, les performances du circuit cryptographique.

2.1 La problématique

En cryptographie, parmi les labeurs les plus difficiles à résoudre, a été celui concernant l'échange de clefs secrètes. Jusque dans les années 1970, il fallait alors que deux protagonistes, souhaitant échanger sur un réseau non sécurisé, se soient auparavant concertés sur le jeu de clefs qu'ils allaient utiliser pour chiffrer leurs communications. La clef permettant de chiffrer était la même que celle qui permettait de déchiffrer. Durant la seconde guerre mondiale, les « nazis » utilisaient un livre de codes, un recueil de clefs, qui à chaque jour de l'année attribuait une clef de chiffrement. Cela rendait la communication secrète possible entre différents escadrons partageant le même livre, la même clef du jour. Bien évidemment, si l'ouvrage tombait entre des mains ennemis, c'est tout un pan de la sécurité qui était mis à vif et il fallait, dès lors, changer l'intégralité du corpus de clefs. La communauté scientifique se doutait de la possibilité de communiquer dans un environnement hostile sans qu'Alice et Bob * ne se soient préalablement rencontrés. L'analogie était la suivante : Alice veut envoyer un message à Bob. Alice enferme son message dans une boîte qu'elle verrouillera à l'aide de son propre cadenas (dont elle seule détient la clef). Elle envoie cette boîte à Bob qui va, à son tour, ajouter son verrou sur la boîte. La boîte est donc protégée par deux cadenas : celui d'Alice et celui de Bob. Bob renvoie le colis à Alice qui retirera son cadenas avant de le retourner de nouveau à Bob. Bob n'a plus qu'à ouvrir la boîte et lire le message qu'Alice désirait lui envoyer. Durant ces différents échanges, le paquet ne s'est pas un seul instant retrouvé sans protection ! C'était là, la preuve, certes très intuitive, que l'échange d'informations secrètes était possible sans contact physique antérieur à la transmission de données sensibles. C'est en 1976 que Diffie et Hellman [13] proposent un schéma qui assure à deux entités distinctes la création d'une quantité commune sans que quiconque observant le réseau ne puisse en déterminer la véritable teneur. La métaphore souvent usitée pour expliquer la technique est celle du mélange de peintures. Alice possède son propre tube de gouache d'une certaine couleur. La couleur est une information secrète qu'elle gardera pour elle. De même pour Bob, qui ne révélera pas la couleur de sa peinture. Au milieu d'eux, deux pots de peinture de couleur verte, accessibles aussi bien à Alice qu'à Bob, mais aussi à tous les membres du réseau. Il s'agit là d'une information **publique**. Alice prendra l'un d'eux et y ajoutera un peu de sa gouache (la quantité ajoutée est fixe). Bob fera de même avec le deuxième pot de peinture verte encore inutilisé. Il y a donc deux mélanges qu'Alice et Bob s'échangeront publiquement. Bob prendra le pot d'Alice et Alice, celui de Bob. Bob et Alice verseront dans les deux récipients leurs couleurs respectives. Ils devraient individuellement obtenir la même couleur, leur **secret commun**, qu'ils partagent tous les deux. Une personne qui aurait eu accès aux premiers mélanges d'Alice et

∗. Des personnages récurrents de la cryptographie.

Bob ne parviendrait pas à extraire facilement les couleurs choisies par les deux compères ni même deviner, là encore, facilement, le secret commun.
Si nous tentons de formaliser le problème, nous obtenons la procédure qui suit :

- Alice ajoute sa peinture P_A au pot vert V : $P_A + V$
- Bob ajoute sa peinture P_B au pot vert V : $P_B + V$
- Alice ajoute sa peinture au mélange de Bob : $P_B + (P_A + V)$
- Bob ajoute sa peinture au mélange d'Alice : $P_A + (P_B + V)$

Évidemment, il faut que $P_B + (P_A + V) = P_A + (P_B + V)$ mais aussi qu'il soit difficile, connaissant $(P_A + V)$ ou $(P_B + V)$ de remonter à P_A ou P_B. Dans les faits, il serait tout de même peu pratique d'utiliser des pots de peinture ... Il est employé des groupes mathématiques comme le groupe multiplicatif des entiers modulo un nombre premier ou bien, comme nous l'avons fait dans cette thèse, le groupe additif des courbes elliptiques (Nous y reviendrons dans 2.2). Pour le groupe multiplicatif des entiers modulo un (grand) nombre premier p dénoté $(\mathrm{GF}(p), \times)$ ou $(\mathbb{F}_p, \times)$, le protocole de Diffie-Hellman s'écrierait de cette façon :

- Alice ajoute sa peinture P_A au pot vert V : $C_A = V^{P_A}$
- Bob ajoute sa peinture P_B au pot vert V : $C_B = V^{P_B}$
- Alice ajoute sa peinture au mélange de Bob : $(V^{P_B})^{P_A}$
- Bob ajoute sa peinture au mélange d'Alice : $(V^{P_A})^{P_B}$

où $V \in \mathrm{GF}(p)$ et P_A, P_B sont des nombres entiers compris entre 0 et $p-1$. Ici, si un attaquant souhaite retrouver la valeur secrète $C_{AB} = (V^{P_A})^{P_B} = (V^{P_B})^{P_A} = V^{P_A \times P_B}$, il peut par exemple s'intéresser à la première valeur circulant en clair sur le réseau, c'est à dire $C_A = V^{P_A}$. S'il résout l'équation $V^x = C_A$, il lui suffira d'injecter x dans la seconde valeur observée sur le réseau $C_B = V^{P_B}$ en calculant la valeur $C_B^x = C_{AB}$. Résoudre l'équation $V^x = C_A$ est le problème dit du **logarithme discret** de base V (souvent référé comme **DLP** : *Discrete Logarithm Problem* pour les anglophones). L'une des solutions naïves est de tester pour chaque élément x compris entre 0 et $p-1$ si $V^x = C_A$. Cela dit, p doit être très grand (plusieurs centaines de bits) et ce test exhaustif semble hors d'atteinte, pour nos machines actuelles (2015), dont on estime le domaine du calculable aux alentours d'une centaine de bits. Il existe des algorithmes plus élaborés, comme la méthode rho de Pollard [14], l'algorithme « pas de bébé, pas de géant » [15] qui autorisent de passer d'une complexité en $\mathcal{O}(p)$ pour une attaque naïve à une complexité en $\mathcal{O}(p^{1/2})$. Malgré l'intérêt de ces attaques, elles ne mettent pas à mal la résistance du DLP si p, ou d'une manière générale, l'ordre du groupe, est grand. Qu'en est-il des autres groupes ? Les alternatives sont nombreuses et nous pourrions stipuler que tout groupe serait trivialement candidat. Cette affirmation est clairement fausse, il serait typiquement peu sécurisant de considérer le groupe additif des entiers modulo un

nombre premier p. Trouver une solution x à $V^x = C_A$ dans ce groupe additif reviendrait à trouver x tel que $\underbrace{V + V + \ldots + V}_{x \text{ fois}} = x \times V = C_A$. Dans ce contexte précis, V est un élément de $(\mathrm{GF}(p), +)$ et est donc inversible : ici $x = V^{-1} \times C_A$. Il faut donc que le groupe choisi à des desseins cryptographiques vérifie un certain nombre de propriétés ou plutôt une absence de structure arithmétique remarquable afin d'assurer la robustesse du *DLP*. Deux années après la proposition de Diffie et Hellman fut publié le premier algorithme cryptographique à clef publique, conçu par deux mathématiciens américains et un mathématicien israélien : Ron **R**ivest, Adi **S**hamir et Leonard **A**dleman. L'échange de Diffie-Hellman n'est pas en soi un algorithme de chiffrement, il s'agit d'un protocole qui permet à deux entités de concevoir un secret commun, pouvant servir de clefs, a postériori, à destination d'algorithmes cryptographiques symétriques. L'algorithme **RSA** [16], nommé d'après les initiales des noms des trois auteurs permet de chiffrer des données directement sans avoir eu recours à une série d'échanges avec l'interlocuteur. Nous allons de nouveau user d'une simple métaphore pour expliquer le principe du RSA dans les grandes lignes. Bob met à disposition des cadenas ouverts, identiques les uns aux autres. Lui seul détient la clef ouvrant l'ensemble de ces cadenas. Si Alice souhaite lui envoyer un message secret, elle dispose son courrier dans une boîte et la verrouille avec l'un des cadenas de Bob. Elle envoie le colis à Bob qui pourra donc lire les mots d'Alice puisqu'il sera assurément en mesure d'ouvrir le cadenas. Le schéma ici proposé est encore bien plus simple que celui de Diffie-Hellman et dicte les codes d'un nouveau paradigme : le chiffrement à clef publique. La clef pour chiffrer (le cadenas de Bob, appelé **clef publique**) et la clef pour déchiffrer (la clef de Bob, appelée **clef privée**) ne sont pas les mêmes !

Le fonctionnement formel de l'algorithme RSA est présenté dans l'algorithme 1. La difficulté d'un éventuel attaquant est de savoir factoriser rapidement le nombre n, produit de deux nombres premiers p et q. S'il y parvient, alors il lui sera facile de calculer $\Phi(n) = (p-1) \times (q-1)$ et l'inverse $e^{-1} \bmod \Phi(n) = d$ qui est, dans notre cas, la clef *secrète* de Bob. Il existe cependant des algorithmes très efficaces de factorisation, comme la crible quadratique [17] s'exécutant en $\approx \mathcal{O}(2.718^{\log(n)^{1/2} \times \log(\log(n))^{1/2}})$: il s'agit là d'un algorithme jouissant d'une complexité sous-exponentielle. Nonobstant cela, factoriser des grands nombres est toujours considéré comme difficile (cela sous-entendra par contre l'utilisation de clefs plus grandes que celles destinées pour les algorithmes symétriques). Les algorithmes modernes de cribles sont généralement basés sur la recherche d'entiers partageant des propriétés arithmétiques, comme $x^2 = y^2 \bmod n \Rightarrow \gcd(x-y, n)$ divise n. Nous pouvons citer la factorisation en 2009 d'un nombre de 768 bits [18]. Une attaque exhaustive sur $m < n$ (trouver m' tel que $(m')^d = c$) paraît là encore inatteignable, n pouvant tenir sur plusieurs centaines de bits. Pour être exact, la sécurité du RSA provient du problème prétendu difficile dit « problème RSA ». Sa définition est

directement issue du déroulement de l'algorithme RSA. Étant donnés un entier $n = pq$, e premier avec $\Phi(n)$ et un entier naturel c, trouver m' compris entre 0 et $n-1$ tel que $(m')^e = c \bmod n$. La factorisation de n répond à cette question, tout comme la recherche exhaustive sur le message m' mais il n'est pas impossible que le problème RSA possède une solution qui lui est propre : la question reste ouverte. En 1985, le schéma de Elgamal [19], schéma très utilisé de nos jours, est proposé en tant qu'algorithme de chiffrement asymétrique, à clefs publiques, basé sur le protocole de Diffie-Hellman. Le principe est de « dissimuler » le message qu'Alice souhaite envoyer à Bob par le biais d'un secret commun. Il est décrit dans Algo. 2.

Algorithme 1 : Algorithme RSA [16].

Données : p, q deux nombres premiers gardés secrets par Bob, $n = pq$, $\Phi(n) = (p-1)(q-1)$, e un nombre premier à $\Phi(n)$.
Données : La clef publique de Bob (n, e), la clef secrète de Bob d telle que $d \times e = 1 \bmod \Phi(n)$.
Données : Alice veut envoyer un message m à Bob, message qui peut être considéré comme un nombre compris entre 0 et $n-1$.
Chiffrement :
1) Alice calcule $c = m^e \bmod n$ grâce à la clef publique e de Bob.
2) Alice envoie le message c à Bob.
Déchiffrement :
1) Bob récupère le message c et calcule l'exponentiation $c^d \bmod n = m$ avec sa clef privée d.

Le chiffrement par clefs publiques répond également au problème d'authentification ; s'assurer que la personne avec qui nous communiquons est bien celle qu'elle prétend être. Cela est fait par l'intermédiaire de défis. Pour ce faire, il suffit d'envoyer un message m que l'initiateur de la conversation, Alice, chiffrera avec la clef publique de Bob $c' = m^e \bmod n$. Si Bob est bien celui qu'il dit être, il sera capable de déchiffrer c' et de retourner à Alice le message m.

Avant d'aborder les courbes elliptiques, nous présentons un tableau récapitulatif des tailles de clefs conseillées par l'ANSSI [†] afin que le lecteur puisse réaliser le fossé qu'il y a entre cryptographie asymétrique et cryptographie symétrique.

†. Agence Nationale de la Sécurité des Systèmes d'Information.

Algorithme 2 : Algorithme de chiffrement à clef publique d'Elgamal [19].

Génération de la clef de Bob :
1) Bob choisit un groupe cyclique G (typiquement $(\mathrm{GF}(p), \times)$ ou bien un groupe défini sur une courbe elliptique), d'élément générateur g.
2) Bob choisit un nombre aléatoire x compris entre 0 et l'ordre du groupe G noté $|G|$.
3) Bob calcule $h = g^x$.
4) Bob publie sa clef publique qui est le triplet (G, g, h).
Alice chiffre son message m :
1) Alice choisit un nombre aléatoire y et calcule $c_1 = g^y$.
2) Alice calcule le secret commun $s = h^y$.
3) Alice attribue à son message m une valeur m' appartenant à G.
4) Alice calcule $c_2 = m' \times s$.
5) Alice envoie à Bob le couple (c_1, c_2).
Bob déchiffre le message d'Alice (c_1, c_2) :
1) Bob calcule le secret commun $s = c_1^x$.
2) Bob calcul $m' = c_2 \times s^{-1}$, le résultat est correct puisque
$c_2 \times s^{-1} = m' \times s \times s^{-1} = m'$.

TABLE 2.1: Tailles des clefs recommandées par l'ANSSI (2014–2020)

Types de chiffrement	Taille des clefs
Symétrique (AES)	100
RSA	2048
DLP-EC	200
DLP-FF	200

Dans le tableau Tab. 2.1, **DLP-EC** renvoie au problème du logarithme discret dans les courbes elliptiques et **DLP-FF** renvoie au problème du logarithme discret dans le groupe $(\mathrm{GF}(p), \times)$. L'intérêt des courbes elliptiques réside dans la taille des éléments manipulés. Même si la sécurité est assurée par 200 bits pour les deux protocoles DLP-EC et DLP-FF, les éléments que la machine traite sont représentés par 200 bits dans le premier cas (DLP-EC) contre ... 2048 dans le second (DLP-FF) ! Les opérations sur courbes elliptiques sont un peu plus couteuses que des opérations dans $(\mathrm{GF}(p), \times)$ (pour une taille des éléments égale), reste que le caractère quadratique de la multiplication rend la cryptographie sur courbe elliptique pérenne.

2.2 Courbes Elliptiques

Nous rappellerons très brièvement dans cette section la notion de courbes elliptiques, le lecteur intéressé par le sujet est invité à se référer au livre [20]. Une courbe elliptique $\mathbb{E}$

dans les réels est définie comme un ensemble de points de l'espace $(x, y) \in \mathbb{R}^2$ respectant une équation (de Weierstrass) du type :

$$y^2 = x^3 + a \times x + b.$$

Si nous réécrivons l'équation précédente comme $P(x, y) = y^2 - x^3 - a \times x - b = 0$, cet ensemble doit également posséder la propriété de **lissité** : il ne doit exister aucun point (x, y) de $\mathbb{E}$ tel que

$$\frac{\partial P(x,y)}{\partial x} = 0 \quad \text{et} \quad \frac{\partial P(x,y)}{\partial y} = 0.$$

Cette exigence permet d'éviter l'existence de singularités ; points auxquels il est impossible de définir une tangente à la courbe, ce dont nous aurons besoin pour spécifier la loi de groupe sur $\mathbb{E}$. Deux exemples d'ensembles sont donnés à la figure 2.1.
Nous munissons ces points d'une loi de groupe $+$ et nous y rajoutons un point à l'infini $\mathcal{O}$, élément neutre de l'opération $+$.

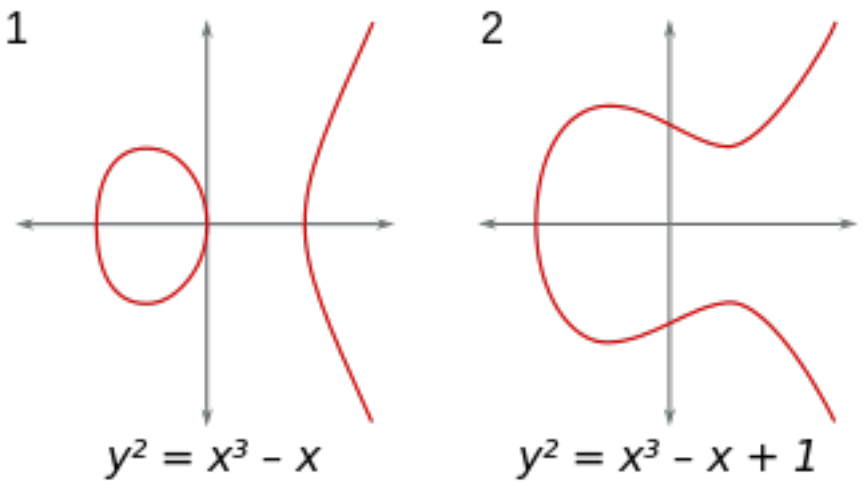


FIGURE 2.1: Exemples de courbes elliptiques définies sur $\mathbb{R}^2$.

- Élément neutre : Si $P = (x_1, y_1) \in \mathbb{E}$ alors $P + \mathcal{O} = P = (x_1, y_1)$
- Opposé d'un point : Il existe pour tout $P = (x_1, y_1) \in \mathbb{E}$ un autre élément dans $\mathbb{E}$ nommé $(-P)$ tel que $P + (-P) = \mathcal{O}$. Cet élément vaut $(-P) = (x_1, -y_1)$.
- Addition de points : Si $P = (x_1, y_1) \in \mathbb{E} \neq Q = (x_2, y_2) \in \mathbb{E}$ alors $P+Q = (x_3, y_3) \in \mathbb{E}$ avec $x_3 = \lambda^2 - x_1 - x_2$, $y_3 = \lambda(x_1 - x_3) - y_1$ et $\lambda = \frac{(y_2-y_1)}{(x_2-x_1)}$
- Doublement de points : Si $P = (x_1, y_1) \in \mathbb{E}$ alors $2P = P + P = (x_3, y_3) \in \mathbb{E}$ avec $x_3 = \lambda^2 - 2x_1$, $y_3 = \lambda(x_1 - x_3) - y_1$ et $\lambda = \frac{(3x_1^2+a)}{(2y_1)}$

Cette loi de groupe est associative $(P + Q) + R = P + (Q + R) = P + Q + R$. Elle est de plus commutative $P + Q = Q + P$.
Ces opérations peuvent s'interpréter géométriquement. Si $P \neq Q$, alors l'addition de ces deux points est le point qui est le symétrique (par rapport à l'axe des abscisses) du

point d'intersection de la droite passant par P et Q et de la courbe elle même. De façon complémentaire, si $P = Q$ alors le doublement de P est le point qui est le symétrique (par rapport à l'axe des abscisses) du point d'intersection de la droite tangente à P et de la courbe $\mathbb{E}$. Une explication visuelle est donnée à la figure 2.2.

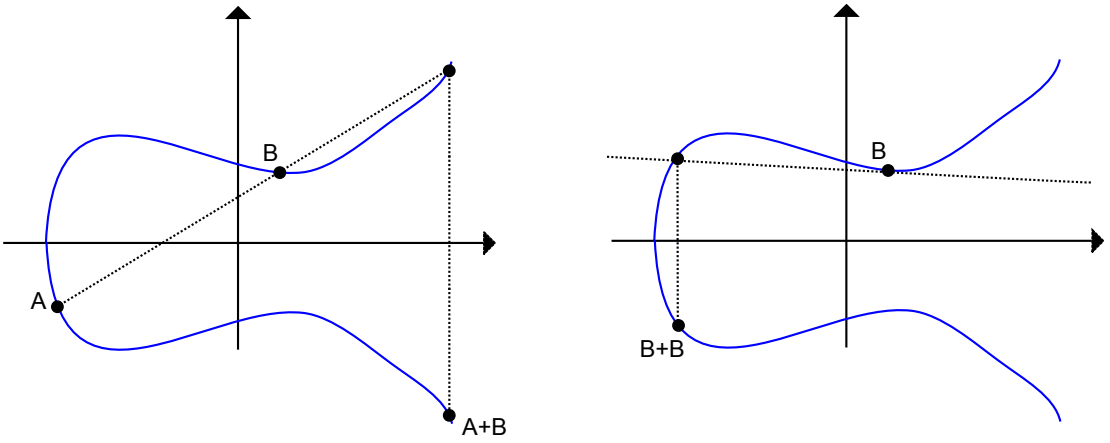


FIGURE 2.2: Loi de groupe sur $\mathbb{E}$.

Bien que nous ayons défini ces courbes sur $\mathbb{R}$, il est tout à fait possible de choisir un autre corps et notamment, un corps fini. Typiquement, nous pourrions choisir le corps premier GF(p) ou bien les extensions du corps fini GF(2). Les formules d'addition et de doublement vont changer (dépendant de la caractéristique du corps) face à ce que nous avons présenté jusqu'à maintenant. Il est également plus difficile de représenter géométriquement la loi de groupe, c'est pourquoi il est généralement présenté, en premier, les courbes elliptiques sur $\mathbb{R}^2$. Un exemple de courbe sur GF(191) est donné dans la figure 2.2. Il s'agit, là encore, d'un exemple pédagogique, la taille du corps GF(191) ne permet pas d'assurer la sécurité du *DLP-EC*. Nous remarquons que tous les points de la courbe de l'exemple de la figure 2.2 sont bel et bien dans un carré de taille 191 par 191 unités. Dans cette thèse, nous nous sommes principalement intéressés aux courbes elliptiques définies sur le corps GF(2^m). Il est, là aussi, délicat de représenter les points de ces ensembles dans un plan bien qu'une injection entre GF(2^m) et $\mathbb{Z}$ soit plausible et envisageable. Les opérations sur GF(2^m) sont plus légères, moins gourmandes en temps et en silicium (si la surface est une contrainte) : cela est en particulier lié à une propriété fondamentale : il n'y a pas de propagation de retenues lors d'addition de deux éléments appartenant à ce groupe contrairement à GF(p). Nous noterons $\mathbb{E}$(GF(2^m)) l'ensemble des courbes elliptiques définies sur le corps GF(2^m) et $\mathbb{E}$(GF(p)) l'ensemble des courbes elliptiques définies sur GF(p).

Sommairement, les corps GF(2^m) seront au centre de cette thèse, ils feront l'objet de deux chapitres 3 et 5 dédiés, dans lesquels nous parlerons à la fois de l'état de l'art et de nos contributions. L'arithmétique dans GF(2^m) regorge d'astuces et il serait peu viable d'en faire la description dans ce chapitre. Toutefois, disons que GF(2^m) est l'arithmétique

des polynômes à coefficients dans GF(2) modulo un polynôme irréductible f. Le corps fini GF(p) est de façon similaire le corps des entiers modulo un nombre premier mais ne fera pas l'objet de notre attention dans le présent manuscrit.

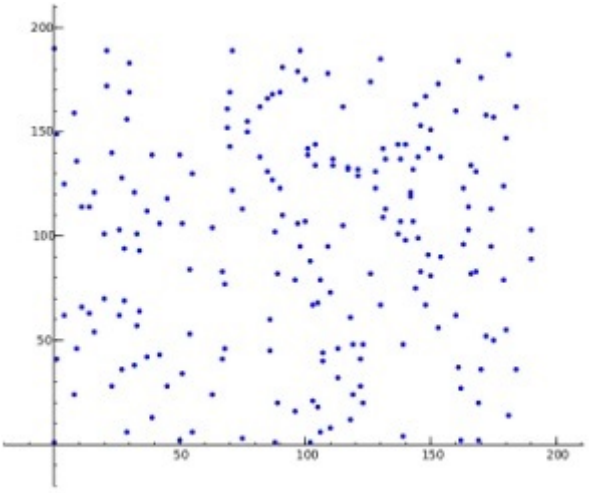


FIGURE 2.3: Courbe elliptique $x^2 + xy = x^3 + x^2 + 1$ définie sur GF(191).

À partir des opérations de doublement et d'addition, il est possible de définir la procédure de multiplication scalaire : à un point P et à un entier k nous attribuons la valeur $[k]P = \underbrace{P + P + P + \ldots + P}_{\text{k fois}}$. Cette tâche n'est jamais réalisée naïvement (en additionnant P, $(k-1)$ fois à lui même, cela serait couteux – en fait, cela serait même impossible sur nos machines actuelles, étant donnée la taille des clefs cryptographiques). L'une des résolutions est d'utiliser un schéma de type Hörner : si $k = k_0 + 2 \times k_1 + 2^2 \times k_2 + \ldots + k_{m-1} \times 2^{m-1} = (k_0, k_1, \ldots, k_{m-1})$ alors $[k]P = 2(2(\ldots 2(k_{m-1}P) + k_{m-2}P) + \ldots) + k_1P) + k_0P$. Ce schéma d'Hörner est accompli à l'aide de l'algorithme Algo. 3 ci-dessous.

Algorithme 3 : Algorithme *Double-and-Add* pour la multiplication scalaire ECC [20].
Données : P un point de la courbe $\mathbb{E}$, $k = (k_0, k_1, \ldots, k_{m-1})$ un entier
Résultat : $[k]P$

```
Q ← O
pour i de 0 à m − 1 faire
    Q ← 2 × Q
    si k_{m−1−i} = 1 alors
        Q ← Q + P
retourner Q
```

Une multiplication scalaire $[k]P$ implique une succession de doublements et d'additions de points au niveau de la courbe $\mathbb{E}$. Chaque addition ou doublement de point implique,

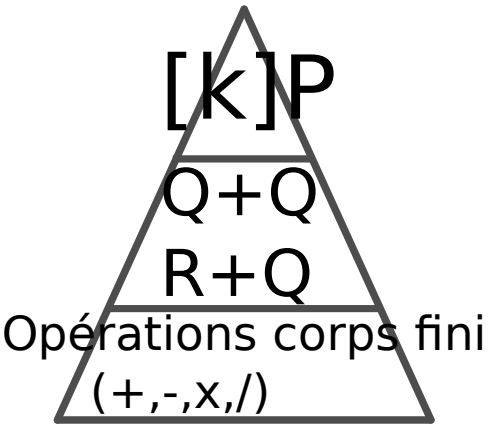


FIGURE 2.4: Imbrication des opérations.

quant à elle, une succession d'opérations (multiplications, divisions, additions et soustractions) au niveau du corps fini (cette idée est représentée par ce schéma pyramidale 2.4).

Dans $\mathbb{E}(\mathrm{GF}(2^m))$, si $P = (x_1, y_1)$ et $Q = (x_2, y_2)$, le calcul de $P + P$ se fait par le biais de la formule tirée de [20]

$$\begin{cases} \lambda = x_1 + y_1/x_1 \\ x_3 = \lambda^2 + \lambda + a \\ y_3 = \lambda \times (x_1 + x_3) + x_3 + y_1 \end{cases} \tag{2.1}$$

et le calcul de $P + Q$ par :

$$\begin{cases} \lambda = (y_1 + y_2)/(x_1 + x_2) \\ x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 = \lambda \times (x_1 + x_3) + x_3 + y_1 \end{cases} \tag{2.2}$$

La division est l'opération la plus couteuse dans un corps fini, on estime généralement qu'elle vaut entre 10 et 100 multiplications (au bas mot, selon l'algorithme employé, la caractéristique du corps mais aussi la taille même des éléments, en bits). Jusqu'ici, lorsque nous souhaitions ajouter un point à un autre, ou bien le doubler, il était nécessaire de calculer une inversion. Il existe différentes représentations des points des courbes elliptiques afin de réduire le nombre d'inversions à sa plus simple expression. Les plus populaires sont sans doute les coordonnées projectives [21] et les coordonnées Jacobiennes [22]. Concernant les coordonnées projectives, le principe est de rajouter à un point $P = (x, y)$ une troisième coordonnée z qui va « accumuler » les inversions, comme nous pourrions

le faire dans les nombres rationnels. Il est plus facile de calculer $\frac{a_0}{b_0} \times \frac{a_1}{b_1} \times \ldots \times \frac{a_d}{b_d}$ comme $\frac{a_0 \times a_1 \times \ldots \times a_d}{b_0 \times b_1 \times \ldots \times b_d}$ (nous multiplions tous les numérateurs ensemble, tous les dénominateurs ensemble puis effectuons une division finale si nous désirons avoir une représentation décimale de ce rationnel) plutôt que comme

$$\frac{((\frac{\frac{a_0}{b_0} \times a_1}{b_1}) \times \ldots) \times a_d}{b_d}$$

où multiplications et divisions s'enchainent. Si nous reprenons les opérations $P + Q$ de l'équation 2.2, il est tout à fait possible de « retarder » l'inversion, en réécrivant, par exemple, la coordonnée x_3, de façon à faire apparaitre un dénominateur commun à chacun des termes :

$$x_3 = \frac{(y_1 + y_2)^2 + (x_1 + x_2)(x_1 + x_2)(y_1 + 2) + (x_1 + x_2)^2 x_1 + (x_1 + x_2)^2 x_2 + (x_1 + x_2)^2 a}{(x_1 + x_2)^2}$$

Ce terme commun (ici $(x_1 + x_2)^2$) peut également apparaitre dans le calcul de la coordonnée y_3. Plutôt que d'effectuer la division dans chacune des coordonnées x_3 et y_3, nous stockons ce dénominateur commun en z_3. Il faut évidemment réécrire les formules d'addition et de doublement pour prendre en considération l'ajout de cette coordonnée redondante. À titre d'exemple, le doublement de points $R = (x_1, y_1)$ en coordonnées projectives est opéré de la façon suivante :

$$\begin{cases} A = x_1^2 \quad B = A + y_1 \times z_1 \quad C = x_1 z_1 \\ D = C^2 \quad E = (B^2 + B \times C + a \times D) \\ x_3 = C \times E \quad y_3 = (B + C) \times E + A^2 \times C \quad z_3 = C \times D \end{cases} \tag{2.3}$$

Pour retrouver les coordonnées « classiques » en (x, y) (dites affines), il suffit de mettre à l'échelle en évaluant $x = x_3/z_3$ et $y = y_3/z_3$. À noter que l'opération inverse qui consiste, à partir de coordonnées affines, d'extraire les coordonnées projectives est triviale : en effet, si $P = (x, y)$ en affine, nous pouvons écrire $P = (x, y, 1)$ en projectif ! De cette observation, il parait évident qu'un point P représenté en coordonnées projectives possède plusieurs représentations : $P = (c \times x, c \times y, c)$ où $c \in \mathrm{GF}(2^m)$ non nul parfois appelé facteur d'échelle. Vous pourriez remarquer que le calcul de doublement issu de l'équation 2.3 contient plus de multiplications que le doublement issu de l'équation 2.1. C'est effectivement le cas puisqu'il n'y a, dans le système affine, qu'un carré et deux multiplications contre huit multiplications et quatre carrés en système projectif. Cependant, le système projectif permet de faire fi de la division, qui rappelons le, peut couter entre

10 et 100 multiplications selon le corps fini choisi. Les coordonnées Jacobiennes sont une autre façon de représenter un point P de la courbe $\mathbb{E}$. La philosophie est essentiellement la même que la méthode précédente, puisqu'il s'agira fondamentalement de rajouter des composantes redondantes au point P pour tenter de s'affranchir de l'inversion. Tandis que nous avions une relation du type $x = x_3/z_3$ et $y = y_3/z_3$ entre coordonnées affines et coordonnées projectives, le lien ici est différent puisque nous avons alors $x = x_3/z_3^2$ et $y = y_3/z_3^3$. Les coordonnées Jacobiennes sont particulièrement intéressantes puisqu'il ne faudra pour un doublement que cinq multiplications et cinq carrés. En fait, il existe une multitude de systèmes de représentation comme le système de Lopez-Dahab, les coordonnées lambda (...), certains de ces systèmes présentant des intérêts particuliers en termes de coût de calcul. Il faut aussi considérer la façon avec laquelle est écrit l'ensemble de ces formules ; une disposition particulière permet de gagner un carré, une multiplication. Typiquement, pour la formule de doublement de l'équation 2.3 qui compte huit multiplications et quatre carrés, il est tout a fait possible de réarranger les calculs de manière à obtenir sept multiplications et trois carrés [21]. Pour conclure sur cet aspect, nous pouvons aussi utiliser des systèmes de coordonnées mixtes [23] dans lesquels, à titre d'illustration, des points en coordonnées projectives peuvent être ajoutés à des points en coordonnées affines : cela permet de simplifier un certain nombre de calculs et de soulager le crypto-processeur. Cette astuce est extrêmement utile dans un algorithme de type *double-and-add* dans lequel est ajouté à chaque tour de la boucle principale (ou pas, selon la valeur de k_i) le point $P = (x, y)$ à $Q = (x_q, y_q, z_q)$.

Une formule attribuée au mathématicien allemand Helmut Hasse permet d'estimer, dans le cas d'un groupe fini, le nombre de points N que la courbe elliptique $\mathbb{E}$ contient (nous parlons de cardinal de la courbe que nous noterons $|\mathbb{E}(\mathrm{GF}(2^m))|$ ou $|\mathbb{E}(\mathrm{GF}(p))|$) :

$$|N - (q + 1)| < 2\sqrt{q}$$

où $q = 2^m$ dans le cas $\mathbb{E}(\mathrm{GF}(2^m))$ ou $q = p$ dans le cas $\mathbb{E}(\mathrm{GF}(p))$. Cela nous dit, en substance, que le cardinal de ces courbes est sensiblement le même que le cardinal du corps fini sur lesquels elles ont été définies. Pour des questions de sécurité, il faut que le cardinal de la courbe elliptique choisie possède un certains nombres de propriétés ; il faut en particulier éviter que $|\mathbb{E}(\mathrm{GF}(q))| = q$. Si tel était le cas, il existerait un isomorphisme entre cette courbe et le groupe $(\mathrm{GF}(q),+)$, groupe dans lequel il est facile de casser le logarithme discret comme nous l'avons vu auparavant [24].

Nous trouvons quantité de méthodes pour calculer $k[P]$, certaines reposent sur le précalcul de valeurs particulières comme la méthode 2^t-ère, comme un dérivé de cette dernière avec la fenêtre glissante (*sliding window*) et enfin, comme l'échelle de Montgomery (*Montgomery ladder*) [9]. Cette liste n'est évidemment pas exhaustive et nous

allons nous pencher très succinctement sur chacune des méthodes évoquées ici.
La méthode 2^t-ère est une généralisation de la méthode binaire *double-and-add* de l'algorithme 3 où elles coïncident quand $t = 1$. La méthode 2^t-ère requiert de pré-calculer les quantités $P, 2P, 3P, \ldots, (2^t - 1)P$. L'algorithme est détaillé en Algo. 4. L'idée sous-jacente va s'illustrer à travers l'exemple suivant. Si $k = (0, 1, 1, 1, 0, 0, 1, 0, 1)$ (des bits de poids faibles aux bits de poids forts) une méthode 2^3-ère se déroulera comme suit : La première chose à faire est de réécrire k dans le « sens » inverse, en plaçant les bits de poids forts en tête de la représentation. Nous noterons cela par la présence de l'indice b sur l'écriture de k. Ici $k = (\underline{1, 0, 1}, \underline{0, 0, 1}, \underline{1, 1, 0})_b$. Par la suite, l'algorithme se déroule en calculant :

- $h \leftarrow (\underline{1, 0, 1})_b \; P$
- $h \leftarrow (\underline{1, 0, 1}, 0, 0, 0)_b \; P$ $\qquad (h \leftarrow h \times 2^3)$
- $h \leftarrow (\underline{1, 0, 1}, \underline{0, 0, 1})_b \; P$ $\qquad (\mathbf{h \leftarrow h + g_{(0,0,1)}})$
- $h \leftarrow (\underline{1, 0, 1}, \underline{0, 0, 1}, 0, 0, 0)_b \; P$ $\qquad (h \leftarrow h \times 2^3)$
- $h \leftarrow (\underline{1, 0, 1}, \underline{0, 0, 1}, \underline{1, 1, 0})_b \; P$ $\qquad (\mathbf{h \leftarrow h + g_{(1,1,0)}})$
- $[k]P \leftarrow h$

L'astuce est ici de travailler bloc de bits par bloc de bits (ici des blocs de 3 bits) plutôt que bit à bit pour réduire le nombre de multiplications. Plus la taille des blocs est grande, plus la méthode sera efficace, au détriment, cela dit, de nombreux pré-calculs. Ici, il n'y a que 2 additions (en gras, en ignorant les doublements successifs) contre 8 sur une méthode *double-and-add* . Cet algorithme peut être modifié et légèrement optimisé. L'idée même de la fenêtre glissante est d'ignorer une série de 0 qui pourrait apparaître dans le scalaire k considéré. Dans la méthode 2^t-ère, la position dans le scalaire k du bloc courant de t bits est toujours un multiple de t. Dans l'exemple précédent, nous considérons premièrement le bloc $\underline{(1, 1, 1)}$ qui recouvre les bits en position **0** à 2 puis les bits en position **3** à 5 ($\underline{(0, 0, 1)}$) et enfin, pour terminer, les bits aux positions **6** à 8 ($\underline{(1, 1, 0)}$). Les nombres en gras pourraient être vus comme les coordonnées du bloc dans le scalaire k. Nous observons bien que tous ces nombres sont des multiples de 3. Le concept même de la fenêtre glissante est de lever cette contrainte, le bloc de t bits, ou plutôt fenêtre, pourra « glisser » le long du scalaire. Si le premier bit du bloc courant est un 0 alors nous effectuons une opération du type $h \leftarrow h \times 2$ et la position de la fenêtre se voit incrémentée de 1. Aussi, la taille même de la fenêtre n'est pas fixe : nous ignorons également tous les bits de poids faibles égaux à 0 du bloc courant. Ainsi, avec $t = 3$ l'algorithme traitera $\underline{(1, 1, 0)}$ comme $\underline{(1, 1)}$ (la taille de la fenêtre est *donc* réduite). Si nous reprenons l'exemple précédent avec une méthode de fenêtre glissante, les opérations se dérouleront dans l'ordre suivant :

- $h \leftarrow (\underline{1, 0, 1})_b \; P$
- $h \leftarrow (\underline{1, 0, 1}, 0)_b \; P$ $\qquad (h \leftarrow h \times 2)$

- $h \leftarrow (\underline{1,0,1},0,0)_b\ P$ $\quad (h \leftarrow h \times 2)$
- $h \leftarrow (\underline{1,0,1},0,0,\underline{1,1,1})_b\ P$ $\quad (\mathbf{h} \leftarrow \mathbf{h} + \mathbf{g_{(1,1,1)}})$
- $h \leftarrow (\underline{1,0,1},0,0,\underline{1,1,1},0)P_b$ $\quad (\mathbf{h} \leftarrow \mathbf{2} \times \mathbf{h})$
- $[k]P \leftarrow h$

Nous pouvons remarquer que cette version ne requiert qu'une addition de points contrairement aux deux de la version 2^3-ère. De plus, puisque les bits 0 de poids faibles sont ignorés, il n'est plus nécessaire de calculer l'ensemble $P, 2P, \ldots, (2^t-1)P$ mais seulement les valeurs « impaires » de cet ensemble : $P, 3P, 5P, \ldots, (2^t-1)P$.

L'échelle de Montgomery Algo. 5 est un autre algorithme permettant de calculer $[k]P$. Ce qui est remarquable avec cette méthode de Montgomery est qu'elle s'effectue en temps constant (le temps ne dépend pas du nombre de bit à 1 de la clef k) et permet de contrer les attaques dites par analyse de temps (ou *timing attack* en anglais) mais aussi les attaques par canaux cachés, de type **SPA** [25] (*Simple Power Analysis*). Les méthodes d'attaques par *timing* sont simplement basées sur l'analyse du temps d'exécution d'un algorithme cryptographique. Précisément, le temps d'exécution d'algorithmes de type *double-and-add* (voir Algo. 3) dépend en partie du poids de Hamming du scalaire k : le calcul de la clef $(1,1,1,\ldots,1)$ sera plus chronophage que le calcul de la clef $(0,0,0,\ldots,0)$. L'une des attaques par *timing* sur courbes elliptiques est la conséquence de cette différence de traitement face à la nature de la clef k. Un algorithme de type *double-and-add* est d'autant plus vulnérable que le déroulement du calcul ne dépend que des bits de la clef k et que chaque bit de k est traité indépendamment des autres. Il est tout à fait possible de créer un modèle probabiliste qui assure que les hypothèses sur les d premiers bits de clef k sont vraisemblablement correctes et de, par indépendance des probabilités, déterminer le $(d+1)$-ème bit de k. Cette attaque est détaillée dans l'article [7], qui si elle ne s'attarde que sur RSA, est tout a fait adaptable aux courbes elliptiques et à l'algorithme *double-and-add*. Les attaques **SPA** sont plus complexes à mettre en œuvre puisqu'elles nécessitent de pouvoir mesurer la consommation d'énergie de l'appareil cryptographique en cours de fonctionnement : il est, de fait, possible de réaliser une courbe de la consommation électrique de l'appareil en fonction du temps et de dissocier, sur cette dernière, les opérations de doublement et d'addition dans l'algorithme *double-and-add* (la consommation de l'appareil dépend notamment des opérations effectuées en son cœur). L'attaque SPA, si elle requiert du matériel particulier (comme une sonde de mesure de courant ainsi qu'un oscilloscope performant) est l'attaque par canaux auxiliaires (ou cachés) la plus rudimentaire. Certains dispositifs cryptographiques sont effectivement protégés pour résister à cette approche, il sera donc indispensable pour un attaquant d'élaborer des stratégies plus évoluées, comme *les attaques différentielles* (**DPA** : *Differential Power Analysis*) [26] et les attaques **templates** [10]. Nous y reviendrons plus tard dans le manuscrit dans la section 4.

Algorithme 4 : Multiplication scalaire $k[P]$ avec la méthode 2^t-ère [20].

Données : P un point de la courbe $\mathbb{E}$, $k = (k_0, k_1, \ldots, k_{m-1})$ un entier
Résultat : $[k]P$
Précalculs : $P, 2P, 3P, P4, 5P \ldots, (2^t - 1)P$. Ici, k est representé par bloc en le réécrivant $k = d_0 + (2^t) \times d_1 + (2^{2t}) \times d_2 + \ldots + (2^{2l})d_l$ où $d_i = (k_{i\times t} + 2 \times k_{i\times t+1} + 2^2 \times k_{i\times t+2} + \ldots + 2^{t-1} \times k_{i\times t+t-1})$.

```
1 Q ← O
2 pour i de m − 1 à 0 par pas de −t faire
3 |   Q ← 2^t × Q
4 |   si d_i > 0 alors
5 |   |   Q ← Q + (d_i × P)        La valeur soulignée est pré-calculée.
6 retourner Q
```

Dans l'échelle de Montgomery Algo. 5, à chaque bit de clef (qu'il vaille 0 ou 1) est réalisée la même séquence d'opérations (doublement et addition de points), seules les adresses peuvent changer d'un tour de boucle à l'autre selon la valeur courante du bit k_i. Cette propriété est partagée avec la methode *double-and-add always* (voir algorithme 6) qui lance à chaque bit du scalaire k doublement et addition de points. L'addition ne sera retenue que si le bit courant de la clef est égal à 1, sa présence systématique permet d'offusquer le calcul utile à la multiplication scalaire $[k]P$. L'échelle de Montgomery possède cependant une relation entre les différentes quantités manipulées au cours de son exécution : il existe un invariant tel qu'à chaque itération de la boucle principale $R_1 - R_0 = P$. Ce rapport permet en particulier de vérifier le bon déroulement du calcul et de s'assurer qu'aucune erreur ne s'est glissée en cours de parcours (une façon de se protéger contre l'injection de fautes).

Coron présente dans son article [27] trois manières de prévenir les vulnérabilités rattachées aux canaux cachés. La première est de rendre aléatoire la clef k en rajoutant à cette dernière un multiple de l'ordre du groupe ; cela permet notamment d'avoir une trace d'exécution différente à chaque multiplication scalaire et ainsi de limiter le nombre de traces exploitables par l'attaquant. L'une des faiblesses dont tire parti l'attaque [7] est le fait de pouvoir recueillir de l'information des multiplications scalaires $[k]P_0, [k]P_1, \ldots [k]P_d$. L'idée est ici de « cacher » k et de l'écrire comme $k_i = k + n_i \times \mathcal{E}$ où $\mathcal{E}$ est l'ordre du groupe et n_i un nombre aléatoire (d'une vingtaine de bits en pratique). Plutôt que de calculer $[k]P_0, [k]P_1, [k]P_2, \ldots [k]P_d$ comme précédemment seront lancés $[k_0]P_0, [k_1]P_1, \ldots [k_d]P_d$. Nous nous assurons trivialement que $[k_i]P = [k]P$. Une autre méthode d'offuscation toujours proposée par Coron est de confondre le point P_i avec un autre point, de manière à ce que l'attaquant n'ait aucun contrôle sur ce premier. Le concept est de rajouter à P_i un autre point aléatoire R_i, l'algorithme calculera $[k](P_i + R_i)$ ainsi que $[k]R_i$ pour finalement retourner $[k](P_i + R_i) - [k]R_i$. Enfin, sa dernière proposition est de profiter du fait qu'un point ait plusieurs représentations en

coordonnées projectives $P = (c \times x, c \times y, c) = (d \times x, d \times y, d)$ $(c \neq 0, d \neq 0)$. Plutôt que de rajouter un point à P pour l'offusquer, Coron rend aléatoire ses composantes en générant un nombre aléatoire c, calcule, pour $P = (x, y, 1)$, sa valeur mise à l'échelle par c, c'est à dire $P = (c \times x, c \times y, c)$. Combinées, ces différentes approches devraient permettre de protéger le circuit face à des attaques par canaux cachés peu évoluées. Les méthodes de Coron ne sont qu'un aperçu, de nombreux moyens sont employables, comme par exemple, des méthodes inspirées de la thèse de Danuta Pamula [28] dans laquelle elle concevait un multiplieur à consommation constante (il est alors difficile de déterminer des motifs dans la trace de consommation du crypto-processeur). Nous pouvons aussi recoder k en base-double, c'est à dire sous la forme $\sum_{i=0}^{j} a_{(i,j)} \times 2^i 3^j$. L'intérêt de cette représentation est qu'il existe de multiples façons d'encoder un scalaire k et ainsi de rendre aléatoire son écriture. Thomas Chabrier explorait cette piste dans sa thèse [29].

Algorithme 5 : Échelle de Montgomery pour le calcul de $[k]P$ [9].

Données : P un point de la courbe $\mathbb{E}$, $k = (k_0, k_1, \ldots, k_{m-1})$ un entier
Résultat : $[k]P$

$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
pour i **de** m **à** 0 **faire**
  si $k_i = 0$ **alors**
    $R_1 \leftarrow R_0 + R_1$
    $R_0 \leftarrow 2 \times R_0$
  sinon
    $R_0 \leftarrow R_0 + R_1$
    $R_1 \leftarrow 2 \times R_1$
retourner R_0

Algorithme 6 : *Double-and-Add Always* pour le calcul de $[k]P$ [27].

Données : P un point de la courbe $\mathbb{E}$, $k = (k_0, k_1, \ldots, k_{m-1})$ un entier
Résultat : $[k]P$

$Q \leftarrow \mathcal{O}$
pour i **de** 0 **à** $m - 1$ **faire**
  $Q_0 \leftarrow 2 \times Q$
  $Q_1 \leftarrow Q + P$
  $Q \leftarrow Q_{k_{m-1-i}}$
retourner Q

Le recodage de clef est un domaine important. Le recodage permet de se protéger de certaines attaques par canaux auxiliaires mais aussi, dans une certaine mesure, d'accélérer les calculs (minimiser le nombre d'additions de points ou de doublements de points). Nous toucherons quelques mots sur le recodage du scalaire sans pouvoir être complet. Nous introduirons le concept de **représentations redondantes du scalaire**. L'une des formes les plus élémentaires est la forme **NAF** : *Non-Adjacent-Form*. Dans cette

représentation, il n'y aura dans l'écriture binaire de ce scalaire aucune série de termes non nuls consécutifs. Un bit 0 viendra forcément s'intercaler entre les chiffres non nulles. Plutôt que d'encoder le scalaire k sous la forme $k_0 + k_1 \times 2 + k_2 \times 2^2 + \ldots + k_{m-1} \times 2^{m-1}$ avec $k_i \in \{0, 1\}$ nous autorisons les k_i à appartenir à un ensemble un peu plus grand $k_i \in \{-1, 0, 1\}$. Typiquement le nombre 7 en binaire peut être stocké comme $(1, 1, 1)_b$, comme $(1, 0, 0, -1)_b$ ou encore comme $(1, -1, 0, 0, -1)_b$ etc. Parmi toutes ces représentations, seule $(1, 0, 0, -1)_b$ est une forme NAF, les deux autres formes contiennent des termes non nul contigus. D'ailleurs, cette condition assure *l'unicité* de l'écriture NAF. En moyenne, nous ne ferons pas la démonstration dans ce manuscrit, il n'y a que 33% de termes non nuls dans cette écriture. C'est relativement intéressant puisque dans une méthode de *double-and-add* traditionnelle, il y a une addition de points à chaque bit du scalaire k égal non nul. L'algorithme *double-and-add* dans le cas d'une représentation NAF est le même que l'algorithme original (voir Algo. 7), si ce n'est qu'il faut parfois enlever P plutôt que de le rajouter. Or, calculer l'opposé de P est une opération peu couteuse (dans $\mathbb{E}(\mathrm{GF}(2^m))$, l'opposé d'un point $P = (x, y)$ est $-P = (x, x + y)$).

Algorithme 7 : Algorithme NAF du *Double-and-Add* pour la multiplication scalaire ECC [20].

Données : P un point de la courbe $\mathbb{E}$, $k = (k_0, k_1, \ldots, k_{m-1})$ un entier
Résultat : $[k]P$

```
Q ← O
pour i de 0 à m − 1 faire
    Q ← 2 × Q
    si k_{m−1−i} = 1 alors
        Q ← Q + P
    si k_{m−1−i} = −1 alors
        Q ← Q + (−P)
retourner Q
```

La méthode NAF peut se généraliser en w-NAF où l'ensemble auquel appartiennent les k_i est élargi : $k_i \in \{-(2^w - 1), -(2^w - 2), \ldots, 0, \ldots, (2^w - 2), (2^w - 1)\}$. Augmenter la taille de cet ensemble permet de réduire davantage le nombre d'éléments non nuls dans la représentation w-NAF du scalaire k : en moyenne, seuls $1/(w + 1)$ bits de k seront non nuls. Elle impliquera, par contre, des pré-calculs, lors du calcul de $[k]P$. Typiquement, il faudra stocker par avance les valeurs $P, 3P, 5P \ldots, (2^w - 1)P$ (il n'est pas nécessaire de stocker les valeurs « paires » de cet ensemble, les 0 du scalaire seront traités différemment comme sur l'algorithme de fenêtre glissante que nous avons vu précédemment). L'algorithme *double-and-add* adapté au w-NAF est présenté dans Algo. 8. Dans cet algorithme, Signe(k_j) est simplement le signe (1 ou -1) de k_j.

Enfin, il est possible d'utiliser des **chaines d'additions** [30] pour effectuer une multiplication scalaire $[k]P$. Une chaine d'additions est une suite d'entiers tel que tout élément

Algorithme 8 : Algorithme w-NAF du *Double-and-Add* pour la multiplication scalaire ECC [20].

Données : P un point de la courbe $\mathbb{E}$, $k = (k_0, k_1, \ldots, k_{m-1})$ un entier
Résultat : $[k]P$
Précalculs : $P, 3P, 5P \ldots, (2^w - 1)P$. Ici, k est representé en w-NAF, i est le nombre de digit de k dans cette représentation.

1 $Q \leftarrow \mathcal{O}$
2 **pour** $j = i - 1$ **à** 0 **faire**
3 $\quad Q \leftarrow 2 \times Q$
4 $\quad$ **si** $k_j \neq 0$ **alors**
5 $\quad\quad Q \leftarrow Q + \text{Signe}(k_j) \times \underline{(|k_j| \times P)}$ *La valeur soulignée est pré-calculée.*
6 **retourner** Q

(hormis le premier) de la séquence est la somme de deux autres éléments le précédant. Les suites $(U_0) = (1, 2, 3, 6, 7, 14)$ et $(U_1) = (1, 2, 3, 4, 7, 11, 14)$ sont deux exemples de chaines d'additions. En effet, pour la chaine (U_0), nous avons bien $2 = 1 + 1$, $3 = 2 + 1$, $6 = 3 + 3$, $7 = 6 + 1$ et $14 = 7 + 7$. Le lecteur est invité à vérifier que la chaine (U_1) est pareillement une chaine d'additions. Cette notion de chaines d'additions autorise à ordonnancer la multiplication scalaire $[k]P$ avec diverses possibilités. Si nous reprenons les chaines (U_0) et (U_1), nous avons la possibilité d'organiser les calculs de $[14]P$ des deux façons suivantes :

$$\begin{array}{lcll} Q_0 & \leftarrow & 2 \times P & \\ Q_1 & \leftarrow & Q_1 + Q_0 & = 3 \times P \\ Q_1 & \leftarrow & 2 \times Q_1 & = 6 \times P \\ Q_1 & \leftarrow & Q_1 + Q_0 & = 7 \times P \\ Q & \leftarrow & 2 \times Q_1 & = 14 \times P \end{array} \qquad \left| \qquad \begin{array}{lcll} Q_0 & \leftarrow & 2 \times P & \\ Q_1 & \leftarrow & Q_1 + Q_0 & = 3 \times P \\ Q_2 & \leftarrow & 2 \times Q_0 & = 4 \times P \\ Q_0 & \leftarrow & Q_1 + Q_0 & = 7 \times P \\ Q_2 & \leftarrow & Q_0 + Q_2 & = 11 \times P \\ Q & \leftarrow & Q_1 + Q_2 & = 14 \times P \end{array} \right.$$

Nous pouvons constater que deux séquences d'opérations différentes nous conduisent au même résultat $14 \times P$. Il est possible de générer des chaines d'additions à la volée et ainsi, de rendre aléatoire le déroulement de la multiplication scalaire [31]. Cela permet notamment de se prémunir des attaques SPA. Deux chaines d'additions ayant en commun leurs derniers termes ne sont pas forcément de mêmes longueurs ou de mêmes natures : il faudra plus d'opérations et plus de registres (variables) intermédiaires pour calculer $14 \times P$ avec (U_1) qu'avec (U_0).

2.3 Et les FPGAs ?

L'un des buts de cette thèse est d'implémenter des solutions de chiffrements sur des plates-formes matérielles, d'en « mesurer » leurs faiblesses (face à des attaques par canaux auxiliaires), d'en « mesurer » leurs performances, en termes de temps d'exécution, mais aussi en termes de surface silicium occupée (ou un équivalent). Nous avons choisi d'implémenter nos propositions sur des circuits **FPGA**.

Un **FPGA** (*Field Programmable Gate Array*) est un circuit re-programmable dont la fonctionnalité est entièrement déterminée par un fichier de configuration (*bitstream*), généralement généré à partir d'un langage de description matériel (HDL : *Hardware Description Language*). Aujourd'hui, deux langages sont principalement usités : VHDL (VHSIC[‡] *Hardware Description Language*) plus populaire en Europe que son homologue Verilog, davantage employé sur le continent américain. Le circuit FPGA, du moins, concernant celui produit par le constructeur *Xilinx*, est constitué d'une grille sur laquelle se trouve des éléments logiques configurables (CLB : *Configurable Logic Block*) et des éléments de routage permettant de relier ces premiers entre eux. Ces CLB contiennent généralement un petit nombre de *slices* (voir figure 2.5). Comme illustration, il y a, dans un CLB d'un FPGA d'entrée de gamme Xilinx Spartan-6, deux slices. Les slices ne sont, sur les dernières générations de FPGA, pas homogènes. Xilinx propose par exemple sur ses dernières cartes des « *slices M* » (M comme *Memory*, Mémoire) spécialisées dans le stockage de petites quantités de bits. Au sein d'une même slice se trouvent un petit nombre de LUTs (*Look-Up Tables*), une ligne de *fast carry* (pour faciliter l'implémentation d'additionneurs) et des bascules, etc. Les LUTs prennent généralement 4 ou 6 bits en entrée, la sortie (1 ou 2 bits) dépend du fichier de configuration. Nous pouvons voir la LUT comme une petite mémoire à laquelle on accède via 4 ou 6 bits d'adresse et dont la sortie occupe 1 ou 2 bits. Une seule LUT permet de créer une fonction logique combinatoire élémentaire qui reliée à d'autres LUTs offre la possibilité de concevoir des fonctions logiques combinatoires plus évoluées. La sortie de ces LUTs est directement liée à des bascules, qui permettent de stocker le résultat intermédiaire si l'utilisateur souhaite augmenter la cadence de son circuit et ainsi augmenter la fréquence de fonctionnement de l'architecture. Similairement aux LUTs, les *Full-Adders* peuvent être placés les uns à la suite des autres pour obtenir un additionneur complet sur le nombre de bits désiré par l'utilisateur. Les circuits FPGA diposent des blocs spéciaux comme les **BRAMs** (plusieurs centaines sur les plus gros modèles des principaux constructeurs Xilinx et Altera) et **DSPs** (plus d'un millier). Les BRAMs (*Block-RAMs*) sont des blocs mémoires pouvant stocker de larges quantités de données. Sur les modèles les plus récents de FPGA, ces blocs peuvent contenir jusqu'à 32 768 bits. L'utilisateur a

‡. *Very High Speed Integrated Circuit.*

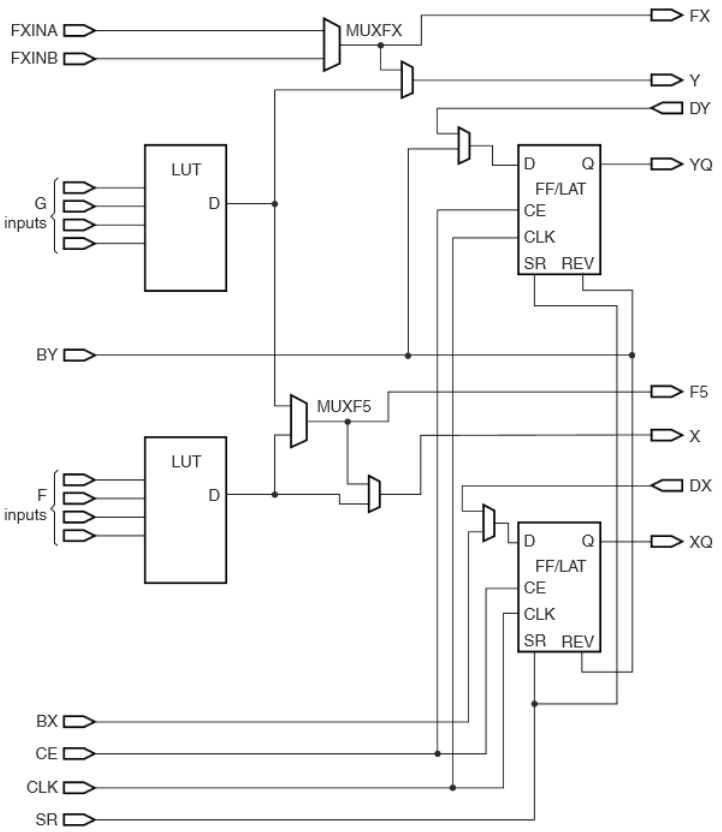


FIGURE 2.5: Schéma d'une *slice* issue d'un FPGA Virtex-II [32].

aussi la possibilité de les configurer en « *ram* double-port ». Dans cette configuration, le bloc dispose de deux ports d'écriture/lecture indépendants, comme si nous accédions à deux mémoires distinctes partageant les mêmes données. Les DSPs sont, quant à eux, des blocs spécialement conçus pour le traitement du signal (Digital Signal Processing) : ils supportent, entre autres, le calcul de FFT (transformée de Fourrier rapide), des multiplications sur de larges opérandes (25 bits pour l'une, 18 bits pour l'autre). Ces DSPs sont notamment très utiles pour le calcul scientifique, mais dans le contexte de cette thèse, étant donnée la nature du corps auquel nous nous intéressons ($\mathrm{GF}(2^m)$), il ne serait pas commode de s'en servir.

Les circuits FPGA modernes bénéficient d'une technologie de gravure avancée (les Virtex 7 de Xilinx profitent d'une finesse de 28nm) qui fait d'eux une véritable alternative face aux ASIC (*Application-Specific Integrated Circuit*, des circuits figés qui ne réalisent que la fonction pour laquelle ils ont été conçus et produits). Bien que les FPGAs soient plus énergivores que des ASICs, le coût de développement sur cette plate-forme est réduit : inutile de prendre en compte les spécificités de l'ASIC (contraignantes) lors de

la phase de programmation. En dehors du développement, la gravure elle-même est couteuse (elle implique, dans le cas d'une gravure par lithographie, la conception de masques pouvant couter des millions d'euros). Si le FPGA est moins efficace, c'est en partie à cause de la portion du circuit dédiée au routage et à la configuration du circuit ; partie qui occupe plus de 90% de la surface [33]. À noter aussi que toutes les ressources du FPGA ne seront pas forcément accessibles : l'algorithme de routage fera de son mieux pour aboutir à un haut taux d'utilisation des CLBs en évitant, alors, de perdre des zones logiques. Typiquement, il se peut qu'une *slice* se retrouve isolée sur la matrice de routage et ne puisse être connectée à d'autres. De plus, le FPGA a une cadence de fonctionnement généralement inférieure à celle d'un circuit ASIC atteignant le GHz (voir davantage) tandis que le FPGA se cantonne à quelques centaines de MHz. Malgré tout, la possibilité de coder une architecture à *grain fin*, la capacité du FPGA à accueillir des circuits massivement parallèles comblent, avec son faible coût, les défauts inhérents à cette technologie. Quand les volumes souhaités sont faibles, le FPGA semble être une vraie option, un vrai compromis entre programmation purement logicielle et conception purement matérielle (sur ASIC). C'est pourquoi nous avons d'ailleurs choisi d'utiliser cette technologie pour implémenter nos différentes architectures matérielles.

2.4 Chiffres d'implémentations de multiplieurs sur divers FPGAs.

Nous fournissons dans cette section un tableau 2.2 à partir duquel le lecteur pourra se référer tout au long de ce manuscrit. C'est aussi l'occasion de donner un ordre de grandeur (en surface et en temps) que peut représenter un circuit placé sur FPGA. Il s'agit d'une compilation de différentes architectures de multiplieurs dans GF(2^m) implémentées sur différents FPGA de la gamme Xilinx. Nous y indiquons la fréquence (en MHz) atteinte par ces implémentations, le temps nécessaire pour le calcul d'une seule multiplication ainsi qu'une estimation de la surface (en slices, LUTs, FFs [§] quand ces informations sont disponibles). Ce qu'il faut comprendre est que dans toutes les solutions ici rapportées, tout est une question de compromis temps/surface. Nous pouvons concevoir des solutions couteuses en surface mais avec un débit de calculs très élevé. À l'inverse, nous pouvons choisir des architectures à la fois très petites mais très lentes. Évidemment, il existe une métrique permettant de caractériser l'efficacité d'une solution donnée : le produit temps surface (PTS). Le PTS est simplement le produit entre la « surface » (qui peut être exprimée en mm^2, en nombre de *slices*, etc) et le temps de calcul requis pour

§. FF : Flip-Flop : Bascule.

	Base	Multiplieur *type, base, etc*	FPGA	Slices (LUTs, FFs)	Freq. (MHz)	Mult. μs
[34]	PB	Folded ($m = 233$) Pipelined ($m = 233$)	$V2$	466 (933,699) 108461 (216923,108578)	168 167	2.8 0.012
[35]	PB	Classical ($m = 233$) Karatsuba ($m = 233$)	$V2$	18648 (37296,37522) 5873 (11746,13941)	77 90	3.03 2.6
	NB	Massey-Omura ($m = 233$) Sunar-Koc ($m = 233$)		18428 (36857,8543) 22717 (45435,41942)	62 93	3.7 2.5
[28]	PB	Matrice-Vector ($m = 163$)	$V6$	$-$ (1050, $-$)	520	0.62
[36]	RNS	RNS-Hybride	$S3$	2752($-$, $-$)	5	18.6
[28]	PB	Mastrovito	$V6$	$-$ (3760, $-$)	295	0.25
[37]	PB	Digit-Serial ($m = 163, g = 1$) Digit-Serial ($m = 163, g = 11$) Digit-Serial ($m = 571, g = 1$) Digit-Serial ($m = 571, g = 24$)	$V5$	252 (504, 500) 589 (1179, 495) 4025 (1731, 1727) 5175 (8051, 1720)	561 423 540 335	0.29 0.035 1.056 0.067
[38]	PB	Comb. ($m = 163, i = 32$) Comb. ($m = 233, i = 32$) Comb. ($m = 283, i = 16$) Comb. ($m = 409, i = 32$) Comb. ($m = 571, i = 32$)	$V6$	$-$(2668, $-$) $-$(3809, $-$) $-$(2893, $-$) $-$(6677, $-$) $-$(9315, $-$)	268 390 423 384 381	0.06 0.15 0.08 0.13 0.18
[39]	PB	($m = 120$) ($m = 240$) ($m = 480$) ($m = 720$) ($m = 1080$)	VE	603($-$, $-$) 1211($-$, $-$) 2426($-$, $-$) 3641($-$, $-$) 5485($-$, $-$)	88 57 59 56 54	1.36 4.14 8.1 12.7 19.8
[40]	PB	Karatsuba ($m = 191$)	VE	6265($-$, $-$)	$-$	0.045
[41]	PB	($m = 191$) ($m = 210$) ($m = 239$)	VE VE $V2$	334($-$, $-$) 385($-$, $-$) 415($-$, $-$)	$-$ $-$ $-$	2.21 2.47 2.42
[42]	NB	($m = 130$) ($m = 138$) ($m = 148$)	VE	5587($-$, $-$) 6266($-$, $-$) 7167($-$, $-$)	$-$ $-$ $-$	36.9 39.9 36.3
[43]	NB	Hybride ($m = 163, d = 11$) Hybride ($m = 163, d = 55$)	$V4$	1691(3365, 326) 9323(16715, 326)	208 149	0.072 0.020

TABLE 2.2: Implémentations de multiplieurs dans GF(2^m) sur divers FPGAs.

effectuer l'opération considérée. Plus le PTS est petit, plus l'architecture se montre efficace.

Les notations $V6, V5, V2$ signifient respectivement Virtex-6, Virtex-5 et Virtex-2. La notation VE est utilisée faire référence aux Virtex-E, un FPGA à faible consommation. Enfin, $S3$ correspond au Spartan-3.

La colonne « Base » permet de déterminer le type de base utilisé lors de la multiplication :

– PB (*Polynomial Basis*) : Base polynomiale : $(1, \alpha, \alpha^2, \ldots, \alpha^{m-1})$ nous y reviendrons dans la section 5,

– NB (*Normal Basis*) : Base Normale : $(\beta, \beta^2, \beta^4, \ldots, \beta^{2^{m-1}})$ nous y reviendrons dans la section 3,

– RNS : Base RNS. Il s'agit d'une représentation modulaire des polynômes, nous y reviendrons dans la section 5.

Nous expliciterons plus spécifiquement l'ensemble des notions ici abordées tout au long de ce document. Notons qu'il existe dernièrement un regain d'intérêt pour les bases redondantes [44]. Il s'agit d'injecter le corps GF(2^m) dans un anneau de type $\frac{\mathrm{GF}(2)[X]}{(X^n+1)}$ (avec $n = k \times m + 1$ premier) dans lequel la réduction est très peu couteuse. Il existe, en

fait, une relation claire entre base normale et base redondante. Nous n'étudierons pas cette représentation dans cette thèse mais il nous semblait important de signaler son existence.

Chapitre 3

≪ Petit ≫ multiplieur/inverseur (PISO) * en base normale dans GF(2^m)

3.1 Introduction

Pour effectuer une multiplication scalaire sur les courbes elliptiques définies sur un corps fini binaire, la méthode du *double-and-add* ou ses nombreux dérivés sont les plus usités. Dans [45], une approche alternative a été suggérée, celle du *halve-and-add*. L'algorithme est sensiblement le même, mais au lieu de doubler P (voir Algo. 9 ligne 3) à chaque tour de boucle, P est ≪ divisé ≫ par deux. Cette division fait paradoxalement disparaitre l'inversion au niveau corps GF(2^m) qu'il était nécessaire d'effectuer lors d'un doublement de point (dans un système de coordonnées affines). Nous reviendrons plus précisément sur l'algorithme de *halve-and-add* [45] dans le chapitre 4.
Pour avoir une solution en partie protégée contre des attaques **SPA**, l'algorithme de Montgomery Algo. 5 est généralement préféré à l'algorithme *double-and-add* . Nous pouvons aussi adapter l'algorithme de Montgomery à la méthode de *halving*. L'échelle de Montgomery adaptée au *halving* nécessite d'ailleurs que les points de la courbe elliptique E soient représentés en coordonnées affines. Il y a dès lors, au cours d'une multiplication scalaire ECC davantage d'inversions que dans d'autres systèmes de coordonnées (comme les coordonnées projectives [20, Sec. 3.2]). Dans le cadre d'une approche de Montgomery basée sur du *halving*, la base normale apporte de par ses carrés faciles à calculer (ce sont des décalages circulaires) et de par un calcul de trace Tr simple un avantage non négligeable.

∗. Parallel-In Serial-Out.

Algorithme 9 : Algorithme *Halve-and-Add* pour la multiplication scalaire ECC [45].

Données : P un point de la courbe $\mathbb{E}$, $\underline{k} = (\underline{k}_0, \underline{k}_1, \ldots, \underline{k}_{m-1})$ un entier, avec $\underline{k} = k \times 2^{-l} \mod N$ où N est l'ordre du point P et l le nombre de bits minimal nécessaire pour stocker N

Résultat : $[\underline{k}]P$

```
Q ← O
pour i de 0 à m − 1 faire
    Q ← Q/2
    si k_{m−1−i} = 1 alors
        Q ← Q + P
retourner Q
```

Nous pouvons combiner les méthodes de Montgomery en *double* et en *halve* (voir [45]) en coupant la clef k en deux sous-parties (k_0 et k_1). De cette façon, nous avons la possibilité de lancer parallèlement un algorithme de Montgomery en *double* sur k_0 et un algorithme de Montgomery en *halve* sur k_1. Il convient ensuite d'additionner les deux points Q_0 et Q_1 respectivement obtenus afin d'obtenir le point $[k]P$. Des travaux récents [46] étudient l'impact d'une telle démarche sur CPU. Un des autres intérêts qui semble émerger est la protection naturellement portée par le calcul parallèle (de deux algorithmes indépendants) face aux attaques par canaux cachés.

Dans ce chapitre de thèse, nous présentons une unité qui combine multiplieur et inverseur (MIU : *Multiplier-Inverser Unit*) opérant en base normale, notamment destinée aux implantations de type « *halving* ». Nous nous focalisons sur des petites solutions, nécessitant moins de surface mais étant nécessairement plus lentes. En l'occurrence, notre solution aura un intérêt pour les hauts niveaux de sécurité seulement (au moins de 571 bits ECC) dont l'utilisation, sans doute plus rare, ne justifie pas des surfaces extravagantes.

Dans une première partie, nous rappellerons quelques faits sur les corps finis GF(2^m), rappellerons aussi brièvement l'état de l'art des multiplieurs en base normale, mentionnerons les deux principales approches pour l'inversion. En second lieu, nous présenterons en détails la solution que nous proposons et nous exposerons en troisième partie nos résultats d'implémentation sur FPGA (Spartan-6 LX75T et Virtex-4 LX100) avant de conclure.

3.2 État de l'art

3.2.1 Le corps fini GF(2^m)

Les corps finis GF(2^m) peuvent être vus comme l'arithmétique des polynômes à coefficients dans GF(2) (noté GF(2)[x]) modulo un polynôme irréductible, c'est à dire un polynôme qui ne peut s'écrire sous la forme d'un produit d'autres polynômes (différents de 1). Typiquement, $f = x^3 + x + 1$ est irréductible (nous pouvons le remarquer, dans le cas d'un polynôme de degré 3, car celui-ci ne possède aucune racine dans GF(2)). Si nous souhaitons calculer $A \times B \bmod f$ où $A = x^2$ et $B = x + 1$, nous effectuons les calculs suivants :

$$\begin{aligned} A \times B \bmod (x^3 + x + 1) &= (x^2) \times (x + 1) \bmod (x^3 + x + 1) \\ &= x^3 + x^2 \bmod (x^3 + x + 1) \\ &= x^2 + x + 1 + 1 \times (x^3 + x + 1) \\ &= x^2 + x + 1 \bmod (x^3 + x + 1) \end{aligned}$$

Le reste de la division de $A \times B$ par $x^3 + x + 1$ est égal à $x^2 + x + 1$.
Il faut plusieurs critères pour obtenir mathématiquement un corps, il faut notamment que, si $A, B, C \in \mathrm{GF}(2^m)$:

- $A + (B + C) = (A + B) + C$ **(Associativité de l'addition)**
- $A \times (B \times C) = (A \times B) \times C$ **(Associativité de la multiplication)**
- $A \times (B + C) = A \times B + A \times C$ **(Distributivité)**
- Il existe un élément neutre de l'addition 0 tel que $A + 0 = 0 + A = A$ **(Existence de l'élément neutre de l'addition)**
- Il existe un élément neutre (non nul, différent de 0) de la multiplication 1 tel que $A \times 1 = 1 \times A = A$ **(Existence de l'élément neutre de la multiplication)**
- Il existe pour tout $A \in \mathrm{GF}(2^m)$ un opposé $(-A)$ tel que $A + (-A) = 0$ **(Existence de l'opposé de l'addition)**
- Il existe pour tout $A \in \mathrm{GF}(2^m)$ non nul un inverse (A^{-1}) tel que $A \times A^{-1} = A^{-1} \times A = 1$ **(Existence de l'opposé de la multiplication)**

L'existence de l'opposé de A de l'addition est triviale dans GF(2^m) puisque dans ce cas $-A = A$. Par exemple $(x^2 + 1) + (x^2 + 1) = 0$. L'existence de l'inverse est un peu plus complexe, elle repose sur la relation de Bézout qui stipule que si A et f sont premiers entre eux (ce qui sera toujours le cas si on choisit A de degré inférieur à f, puisque f est supposé ici irréductible), il existera deux éléments u et v de GF(2)[x] tels que

$$u \times A + v \times f = 1,$$

autrement dit, il existe $u \in \mathrm{GF}(2)[x]$ tel que $u \times A = 1 \bmod f$.

3.2.2 Représentation des éléments d'un corps fini GF(2^m)

Il existe pour les extensions du corps fini binaire deux représentations populaires : *les bases normales* et *les bases polynomiales*. En base normale, $A \in \mathrm{GF}(2^m)$ est représenté comme $\sum_{i=0}^{m-1} a_i \beta^{2^i}$ où les coefficients a_i appartiennent à GF(2) et β est un élément particulier générateur du corps. Pour simplifier l'écriture, nous utiliserons aussi la notation $A = (a_0, a_1, \ldots, a_{m-1})$ pour désigner la représentation vectorielle des coefficients de A. En base normale, mettre un élément au carré revient à effectuer un décalage circulaire vers la droite sur le vecteur de ses coefficients. Ainsi, si $A = (a_0, a_1, \ldots, a_{m-1})$ alors $A^2 = (a_{m-1}, a_0, \ldots, a_{m-2})$. Les racines carrées se calculent dès lors en faisant également un décalage, mais cette fois-ci vers la gauche : $\sqrt{A} = (a_1, a_2, \ldots, a_0)$. La multiplication de deux éléments entre eux est basée sur du calcul matriciel, comme nous allons le voir dans la session 3.2.3 suivante. Nous utiliserons dans le reste de ce chapitre ces bases qui nous apporteront, de par leurs carrés faciles à réaliser, un avantage clef.
En base polynomiale, A est représenté comme $\sum_{i=0}^{m-1} a'_i \times x^i$ où les coefficients a'_i appartiennent à GF(2). Les additions et multiplications sont des opérations polynomiales le tout modulo un polynôme irréductible $f \in \mathrm{GF}(2)[x]$.
Il existe d'autres bases spécifiques pour les corps finis binaires comme les bases de Dicksons [47] ou bien encore, les bases duales [48].

3.2.3 Additions, Traces et Multiplications en base normale GF(2^m)

Pour l'addition, il n'y a pas de propagation de retenues entre les différents coefficients des éléments à additionner. Cela signifie qu'il suffit, pour chaque couple de bits a_i et b_i issu des opérandes $A, B \in \mathrm{GF}(2^m)$, d'une simple porte `XOR`. Si $C = A + B$, alors $c_i = a_i + b_i$. Ces additions peuvent être, en pratique, faites en parallèle.
Le calcul de la trace Tr est relativement simple à effectuer en base normale. La trace Tr d'un élément A de GF(2^m) est définie comme

$$\mathrm{Tr}(A) = A + A^2 + A^{2^2} + A^{2^3} + \ldots + A^{2^{m-1}}.$$

Si nous écrivons $A = (a_0, a_1, a_{,2}, \ldots, a_{m-1})$, nous remarquons que Tr(A) vaut :

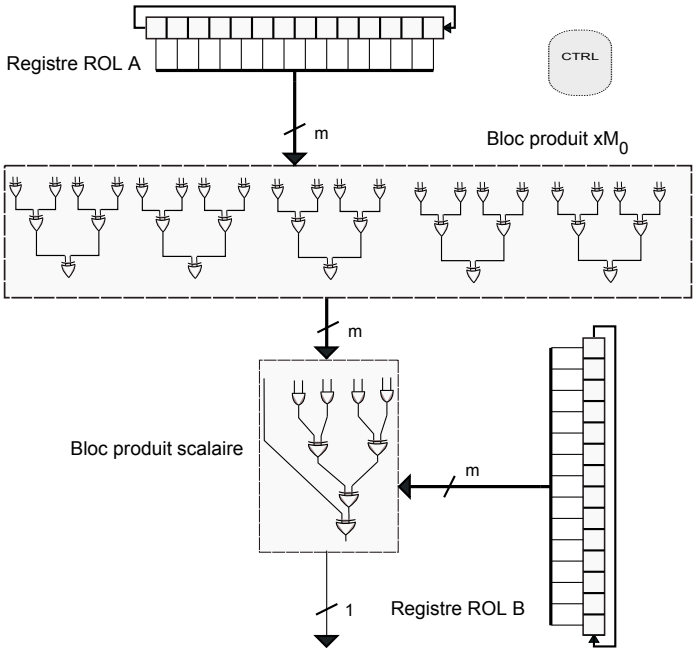


FIGURE 3.1: Architecture du multiplieur de Massey Omura.

$$\begin{array}{l} (a_0, a_1, \ldots, a_{m-1}) \\ +(a_{m-1}, a_0, \ldots, a_{m-2}) \\ +(a_{m-2}, a_{m-1}, \ldots, a_{m-3}) \\ + \quad \vdots \\ +(a_1, a_2, \ldots, a_0) \\ \hline (\lambda, \lambda, \ldots, \lambda) \end{array}$$

avec $\lambda = \sum_{i=0}^{m-1} a_i$. Autrement dit, calculer la trace d'un élément revient à sommer ses coefficients dans GF(2). Ce calcul permet aussi de montrer que $\mathrm{Tr}(A) \in \mathrm{GF}(2)$. En effet, $\mathrm{Tr}(A) = (0, 0, \ldots, 0) = 0$ ou $\mathrm{Tr}(A) = (1, 1, \ldots, 1) = 1$.

En base normale, les schémas de multiplications sont basés sur du calcul matriciel. Cela engendre des multiplieurs plus gros et plus gourmands en surface silicium que ceux dédiés aux bases polynomiales. Massey et Omura (MO) ont proposé en 1986 une méthode (voir Algo. 10) qui a donné lieu à un brevet (voir [49]). Leur approche permet une extrême régularité et ne requiert que peu d'instructions de contrôle (une seule boucle dans l'algorithme 10, qui ne comporte elle-même que très peu d'opérations). La matrice M_0 de l'algorithme 10 est une matrice binaire, elle est exclusivement composée de 0 et de 1. Elle est de plus, symétrique. La notation $\texttt{ROL}(x, n)$ ($\texttt{ROR}$) représente le décalage circulaire de n-bit vers la gauche (droite) du vecteur x. La notation $P[i]$ correspond au i-ème bit de P. L'architecture matérielle est décrite au travers de la figure 3.1. Une esquisse de la démonstration du fonctionnement de l'algorithme est proposée en Dem. 5. Un multiplieur de Massey-Omura, dont la sortie est en série requiert un bloc de multiplication par M_0

(noté $\times M_0$, composé d'arbres de `XOR`), deux registres `ROL` de m bits et un bloc consacré au produit scalaire. Un dernier registre `ROL` de m bits dans lequel nous stockerions les bits produits séquentiellement peut être employé dans le cas où nous souhaiterions avoir une sortie parallèle.

Démonstration. Voici une esquisse de la démonstration de l'algorithme de multiplication de Massey-Omura. Soit F la fonction qui, à deux éléments A et B dans GF(2^m) représentés en base normale, associe le produit $A \times B$ dont le résultat est lui aussi exprimé en base normale.

$$F(A, B) = A \times B = P = (p_0, p_1, \dots p_{m-1}).$$

Soit Φ la fonction qui, à un élément $A \in \text{GF}(2^m) = (a_0, a_1, \dots, a_{m-1})$ en base normale, attribue la valeur a_0, le premier bit de A. Nous obtenons alors que $\Phi(F(A, B)) = p_0$, mais aussi que $\Phi(F(A^2, B^2)) = p_{m-1}$, que $\Phi(F(A^{2^2}, B^{2^2})) = p_{m-2}$ et que plus généralement $\Phi(F(A^{2^k}, B^{2^k})) = p_{m-k}$. En somme, il suffit de la connaissance seule de la fonction $\Phi(F(A, B))$ pour calculer séquentiellement l'ensemble des bits du produit $A \times B$ en évaluant consécutivement $\Phi(F(A^{2^k}, B^{2^k}))$ pour k compris entre 0 et $m - 1$. Il s'avère que $\Phi(F(A, B)) = A \times M_0 \times B^T$. □

Nous avons utilisé dans ce travail les bases normales gaussiennes (GNB : *Gaussian Normal Basis*, voir [50]). Ces bases fournissent une matrice M_0 très creuse : cela pondère la complexité matérielle du bloc $\times M_0$. Plus M_0 est creuse, moins il y a de "1" et plus le bloc $\times M_0$ est petit. Le nombre de "1" est donné par $C_m \leq t \times m - t + 1$ où t est le type du corps fini. Le type t est le plus petit entier tel que $p = m \times t + 1$ soit premier et tel que $\gcd(\frac{tm}{k}, m) = 1$ où k est l'ordre multiplicatif de 2 mod p. Le nombre de portes `XOR` dans le bloc $\times M_0$ est $C_m - m$. Le tableau 3.1 fournit C_m ainsi qu'une estimation de la place occupée (en LUTs sur Spartan-6) par $\times M_0$ pour les corps conseillés par le NIST [51].

Le multiplieur original de Massey-Omura génère un bit du produit par cycle d'horloge (un bit pour chaque tour de boucle). Il est possible, en implémentant un deuxième bloc $\times M_0$, prenant en entrée les registres A et B décalés de 1 bit vers la gauche, d'obtenir

Algorithme 10 : Multiplication de Massey-Omura [49].

Données : A, B dans GF(2^m) en base normale
Résultat : $P = A \times B$

```
1 P ← 0
2 pour i de 0 à m − 1 faire
3 |   P[0] ← A × M0 × B^T
4 |   A ← ROL(A, 1) ; B ← ROL(B, 1) ; P ← ROL(P, 1)
5 retourner P
```

un bit supplémentaire du produit. Déployer parallèlement deux blocs $\times M_0$ nous permet donc d'aller « deux fois plus vite ». Par exemple, dans le corps GF(2^{233}), la multiplication avec deux blocs ne requiert que $117 = \lceil 233/2 \rceil$ cycles d'horloge. Cette approche est assurément généralisable et nous pouvons considérer des multipieurs avec d blocs $\times M_0$. De toute évidence, l'ajout de ce bloc supplémentaire n'est pas gratuit et peut coûter très cher en silicium (ce qui est particulièrement vrai quand le type t du corps est grand). Dans [52], les auteurs parviennent à exploiter les redondances naturellement présentes dans les d copies parallèles du bloc $\times M_0$. Le nombre de portes `XOR` de l'architecture est borné par $d \cdot \big(t \cdot (m - (d+1)/2) + (d-1)/2\big)$. Prenons pour exemple le corps GF(2^7), la matrice de Massey-Omura (en base normal gaussienne) est la suivante :

$$M_0 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Fondamentalement, si nous voulons obtenir les bits p_i et p_{i+1} du produit à chaque cycle d'horloge, nous avons à calculer les valeurs

$$p_i = \texttt{ROL}(A, i) \times M_0 \times \texttt{ROL}(B, i)^T$$

et

$$p_{i+1} = \texttt{ROL}(A, i+1) \times M_0 \times \texttt{ROL}(B, i+1)^T$$

qui peuvent se réécrire comme

$$p_i = \texttt{ROL}(A, i) \times M_0 \times \texttt{ROL}(B, i)^T$$

et

$$p_{i+1} = \texttt{ROL}(A, i) \times M_0^{(1)} \times \texttt{ROL}(B, i)^T$$

TABLE 3.1: Évaluation FPGA de la complexité matérielle du bloc $\times M_0$ (sur Spartan-6).

m / t	163 / 4	233 / 2	283 / 6	409 / 4	571 /10
C_m / #LUT	645 / 159	465 / 232	1677 / 282	1629 / 408	5637 / 1128

avec

$$M_0^{(1)} = \begin{pmatrix} \underline{1} & \underline{0} & \underline{1} & \underline{0} & \underline{0} & \underline{1} & \underline{1} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Ici la matrice $M_0^{(1)}$ est construite à partir de la matrice M_0 : les lignes et les colonnes de M_0 ont été respectivement descendues d'une ligne vers le bas et d'une colonne vers la droite (de façon circulaire, c'est à dire que la dernière ligne, en bas, se retrouve au sommet de la matrice et que la dernière colonne, à droite, se retrouve en première position). Nous pouvons nous apercevoir que les lignes ici soulignées dans M_0 et $M_0^{(1)}$ sont identiques. Ce phénomène apparaitra pour chaque nouveau bloc matriciel $\times M_0$ ajouté à l'architecture. Ajouter un nouveau bloc peut dévoiler des lignes identiques, ce qui permet de factoriser le calcul. Si nous dénommons la ligne soulignée par ℓ_1, nous pouvons, lors de l'évaluation de $M_0 \times \texttt{ROL}(B,i)^T$ et $M_0^{(1)} \times \texttt{ROL}(B,i)^T$ décomposer le calcul de manière à faire apparaitre dans chacune des expressions la quantité $\ell_1 \times \texttt{ROL}(B,i)^T$. Ainsi en ayant plusieurs blocs de multiplication, il n'est pas toujours utile de dédoubler chacune des lignes de la matrice en matériel étant donné que certaines d'entre elles seront communes. Nommons $M_0^{(i)}$ la matrice M_0 dont les lignes et les colonnes ont été respectivement circulairement descendues d'une ligne vers le bas et d'une colonne vers la droite (à la manière avec laquelle $M_0^{(1)}$ a été définie). Reyhani-Masoleh a proposé dans [52] (un schéma est proposé figure 3.2) d'utiliser cette propriété pour réduire la redondance présente intrinsèquement dans les multiplieurs. L'architecture PISO [†] proposée se décompose en trois couches : la première couche est la couche que l'auteur nomme **BTX** (*binary tree of* `XOR`) dans laquelle les d multiplications vecteur-matrice $M_0^{(0)} \times \texttt{ROL}(B,i)^T$, $M_0^{(1)} \times \texttt{ROL}(B,i)^T$, ..., $M_0^{(d-1)} \times \texttt{ROL}(B,i)^T$ sont effectuées simultanément. Cette couche, dispose de la « factorisation » observée précédemment, dans le but de réduire la complexité matérielle de la solution. La seconde couche est un bus (**v-bus**, il ne s'agit ici que de routage sans porte logique) qui relie correctement les signaux issus du **BTX** à la dernière couche **IP-Array** (IP : Inner-Product), quant à elle chargée du calcul des produits scalaires, produisant ainsi les bits $p_i, p_{i+1}, \ldots, p_{i+d-1}$ du produit $P = A \times B$. À noter que les registres `ROL` A et B se décalent de d bits à chaque cycle d'horloge.

Des multiplieurs dont la sortie est entièrement parallèle ont été proposés dans [53, 54]. Le produit P est une accumulation des produits partiels et n'est disponible qu'après

†. Parallel-Input/Parallel-Output.

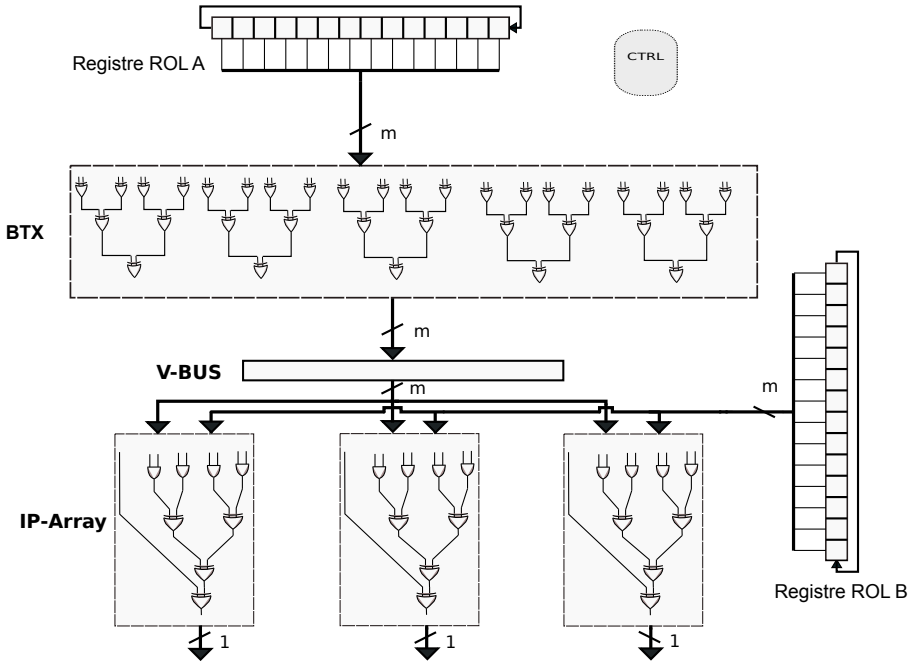
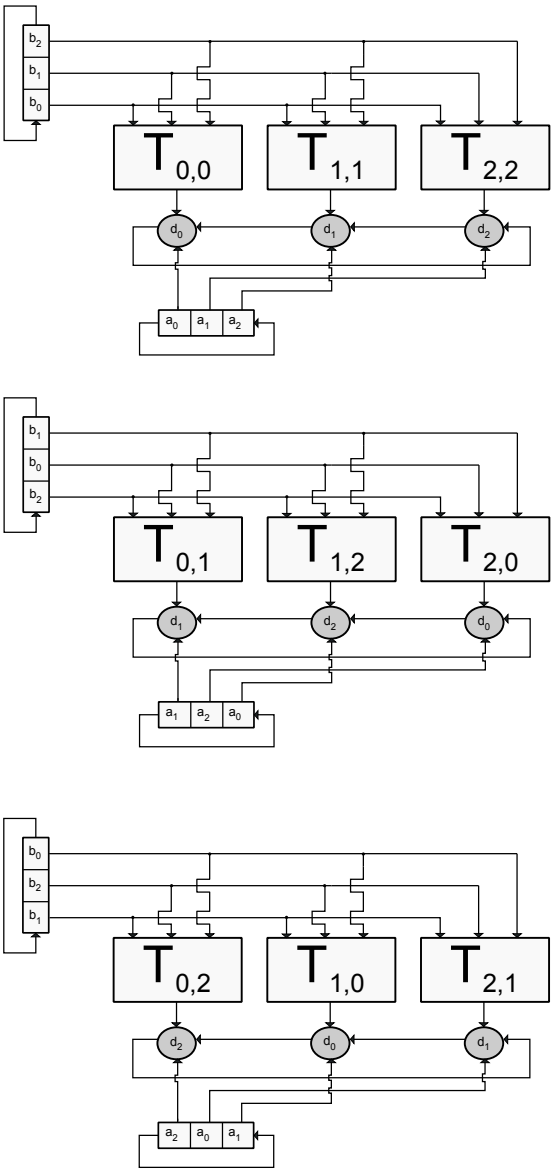


FIGURE 3.2: Architecture du multiplieur PISO de Reyhani-Masoleh [52] avec $d = 3$.

m cycles d'horloge. Le concept suggéré par Agnew et al. est de construire en m cycles d'horloge chacun des bits p_i du produit $P = A \times B$. Reprenons la formule suivante

$$p_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} \times b_{j+k} \times M_0[i,j] = \sum_{i=0}^{m-1} a_{i+k} \times \sum_{j=0}^{m-1} b_{j+k} \times M_0[i,j] = \sum_{i=0}^{m-1} a_{i+k} \times T_{i,k} \tag{3.1}$$

avec $T_{i,k} = \sum_{j=0}^{m-1} b_{j+k} \times M_0[i,j]$ (les indices sont réduits modulo m). L'architecture [54] calcule à chaque cycle d'horloge i la quantité $T_{k,k+i}$ pour tous les $0 \leq k < m$. Les résultats partiels seront accumulés dans un registre temporaire $D = (d_0, d_1, \ldots, d_{m-1})$ de m bits qui contiendra après m cycles d'horloge le résultat escompté P. Par exemple, si nous nous intéressons au premier bit d_0 de ce registre D, il contiendra au premier cycle d'horloge la valeur $a_0 \times T_{0,0}$ puis la valeur de $a_0 \times T_{0,0} + a_{m-1} \times T_{m-1,0}$ au second cycle (...) et enfin $d_0 = p_0 = \sum_{i=0}^{m-1} a_i \times T_{i,0}$ au dernier cycle m de la multiplication. Évidemment, l'exemple que nous donnons ne s'attarde que sur un bit, mais il faut garder à l'esprit que ce calcul est fait en **parallèle** sur chacun des d_i mais avec un ordre différent dans la sommation. Typiquement, pour le bit d_1, au premier cycle d'horloge, d_1 contiendra la valeur $a_2 \times T_{1,1}$ puis $a_1 \times T_{0,1} + a_2 \times T_{1,1}$ (...) et finalement, toujours après m cycles d'horloge, la valeur $d_1 = p_1 = \sum_{i=0}^{m-1} a_{i+1} \times T_{i,1}$. Pour être précis, D est un registre à décalage circulaire. À chaque coup d'horloge, les bits « circulant dans l'opérateur », recevront les contributions des $T_{i,k}$ correspondants. Un exemple du déroulement d'un calcul est donné dans la figure 3.3 sur GF(2^3). Remarquons que les blocs $T_{i,k}$ contiennent



Cycle d'horloge 0 :
$d_0 \leftarrow T_{0,0} \times a_0$
$d_1 \leftarrow T_{1,1} \times a_2$
$d_2 \leftarrow T_{2,2} \times a_1$

Cycle d'horloge 1 :
$d_1 \leftarrow T_{0,1} \times a_1 + T_{1,1} \times a_2$
$d_2 \leftarrow T_{1,2} \times a_0 + T_{2,2} \times a_1$
$d_0 \leftarrow T_{2,0} \times a_2 + T_{0,0} \times a_0$

Cycle d'horloge 2 :
$d_2 \leftarrow T_{0,2} \times a_2 + T_{1,2} \times a_0 + T_{2,2} \times a_1$
$d_0 \leftarrow T_{1,0} \times a_1 + T_{2,0} \times a_2 + T_{0,0} \times a_0$
$d_1 \leftarrow T_{2,1} \times a_0 + T_{0,1} \times a_1 + T_{1,1} \times a_2$

Finalement, lors de cette dernière étape, nous obtenons $\mathbf{d_0 = p_0}, \mathbf{d_1 = p_1}$ et $\mathbf{d_2 = p_2}$.

FIGURE 3.3: Architecture d'Agnew [54] et déroulement du calcul à travers un exemple

évidemment la même logique d'une étape à l'autre. L'évolution des indices de ces blocs est liée à la modification de l'entrée, reliée au registre à décalage circulaire B. La partie chargée de l'accumulation (les cercles gris dans la figure 3.3), que nous appellerons cellule d_i, est décrite dans la figure 3.4. La cellule est composé d'une bascule, d'une porte `XOR` et d'une porte `AND`.

L'architecture de la figure 3.3 possède la même complexité matérielle et temporelle que le multiplieur de Massey-Omura ; néanmoins le chemin critique est plus petit (le calcul matriciel est « coupé » en plusieurs cycles) ce qui permet d'atteindre des fréquences de fonctionnement plus élevées.

Les lignes communes (présentes dans les blocs $\times M_0$) qui apparaissent dans l'architecture

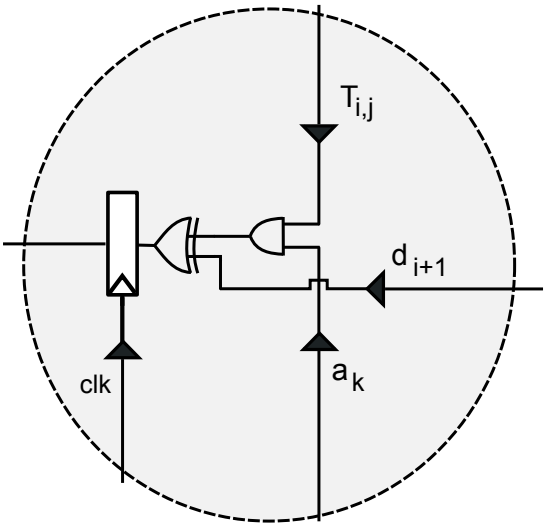


FIGURE 3.4: Schéma logique d'une cellule « d_i » [54].

de Reyhani-Masoleh [52] peuvent être astucieusement utilisées dans un multiplieur à sortie parallèle. Il suffit de remarquer que ces lignes communes sont finalement une façon de dire qu'il existe, pour chaque ligne ℓ_i de la matrice M_0, un cycle d'horloge $j < m$ et une autre ligne k pour lesquels $\texttt{ROL}(A, j) \times \ell_i \times \texttt{ROL}(B, j)^T = \texttt{ROL}(A, 0) \times \ell_k \times \texttt{ROL}(B, 0)^T$. Cette observation, nous permettant de coupler les lignes ℓ_i et ℓ_k deux à deux, donne l'opportunité de décroitre grandement la complexité du circuit en divisant par deux des portions de logiques.

Dans [43], un multiplieur à entrée série et sortie parallèle est présenté. Les entrées sont des sous-mots de w bits, le résultat est disponible en parallèle après $\lceil m/w \rceil$ cycles d'horloge. Les auteurs ont tiré profit de la redondance implicite pour aboutir à une architecture presque deux fois plus petite que des multiplieurs en sortie série. En d'autres mots, l'aspect « sortie série » amène une complexité matérielle non négligeable. Néanmoins, la sortie série peut être exploitée (à condition que l'ordre de sortie des bits s'y prête, ce qui est le cas ici) en lançant parallèlement des calculs annexes dès qu'un bit du produit est retourné, comme cela a été proposé en [55].

3.2.4 Algorithmes d'Inversion dans GF(2^m)

Il y a deux méthodes principales pour inverser un élément dans GF(2^m) : *l'algorithme d'Euclide* et l'utilisation du *petit théorème de Fermat*. Jusqu'à maintenant, les implantations de l'inversion reposant sur l'algorithme d'Euclide [56] sont rarement utilisées en matériel. En effet, une telle stratégie sous-entend dédier une partie du circuit à l'inversion, ce qui, si elle est une opération rare, n'a pas de sens. L'inversion basée sur le

petit théorème de Fermat n'est qu'une exponentiation et s'appuie sur le multiplieur déjà présent sur au sein du circuit (les multiplications sont nombreuses dans une système de chiffrement ECC). Nous factorisons, en quelque sorte, le coût en silicium. Le petit théorème de Fermat (**FLT** : *Fermat's Little Theorem*) affirme que $A^{2^m-1} = 1$ et en conséquence que $A^{-1} = A^{2^m-2}$. Inverser un élément consiste alors à calculer A^{2^m-2}, ce n'est donc qu'une exponentiation, une succession de multiplications. Pour ce faire, nous pourrions utiliser l'algorithme *square-and-multiply* (voir Algo. 11) mais il ne tiendrait pas compte de la spécificité de l'exposant $2^m - 2$. Itoh et Tsujii ont proposé en 1988 dans [57] une façon très efficace d'aborder le problème d'exponentiation. Ils notent que $2^m - 2 = (111\ldots110)_2$: la représentation binaire de l'exposant est une succession de $m-1$ bits à 1 suivie d'un 0 (des poids forts vers les poids faibles). Dans une telle situation, un algorithme de type *square-and-multiply* appliquerait $m-1$ mises au carré et $m-2$ multiplications. Itoh et Tsujii réduisent cette complexité à $\mathcal{O}(\log_2(m-1))$ multiplications. Leur approche exige toujours $m-1$ carrés mais de par la nature de la base normale, ces opérations sont quasiment gratuites et négligeables. L'exemple du tableau 3.3 permet de comprendre comment Itoh et Tsujii y sont parvenus : le poids de Hamming de l'exposant des T_i (produits temporaires) peut doubler d'une étape à l'autre, comme entre $T_2 = A^{(111)_2}$ et $T_3 = A^{(111111)_2}$. Nous comprenons ainsi le caractère logarithmique de leur approche. Informellement, elle consiste à ajouter à l'exposant d'un T_i autant de 0 que souhaité en procédant à des carrés consécutifs et à venir, enfin, « combler » ces 0 avec des 1 en venant multiplier cette nouvelle quantité par un autre T_j. Il reste maintenant à savoir comment établir cette suite de (T_i) construite pour obtenir en dernier lieu A^{-1} : cela repose sur la notion de chaînes d'additions. Une chaîne d'additions (U) est une suite finie d'additions où tous les opérandes sont choisis parmi des valeurs déjà calculées. Mathématiquement, chaque terme U_i de la suite (U) est la somme de deux éléments U_j et U_k avec $j < i$ et $k < i$. Un exemple est donné dans le tableau 3.4 dans lequel nous voyons apparaitre une suite (V). Cette suite (V) dans laquelle chaque terme est composé d'un couple d'entier (a_i, b_i) est la suite associée à (U). Si $U_i = U_j + U_k$ alors $V_i = (j, k)$. La définition d'une telle suite facilite la formalisation de l'algorithme d'Itoh-Tsujii. Nous souhaitons enfin que le dernier terme U_n de la chaîne (U) soit égal à $m-1$. Notons $\beta_k = A^{2^k-1}$. La relation qui existe entre β_k et (U) est mise en exergue par les égalités suivantes :

$$\beta_{U_i} = \beta_{U_k+U_j} = (\beta_{U_k})^{2^{U_j}} \times \beta_{U_j} = (\beta_{U_j})^{2^{U_k}} \times \beta_{U_k} \tag{3.2}$$

$$\beta_{U_n}^2 = \beta_{m-1}^2 = A^{-1} \tag{3.3}$$

Autrement dit, nous pouvons « atteindre » β_{U_n} par un enchainement de calculs de type $\beta_{U_k+U_j}$. L'algorithme d'Itoh-Tsujii est décrit dans Algo. 12. Plus la chaîne d'additions

(U) est courte, moins il y aura de multiplications dans l'exponentiation d'Itoh-Tsujii. Trouver une chaîne courte dont le dernier terme U_n vaut $m-1$ est un problème réputé difficile (voir par exemple [58, Section 4.6.3]).

Algorithme 12 : Algorithme d'Itoh-Tsujii [57].

Données : Un élément non nul $\alpha \in \mathrm{GF}(2^m)$, (U_n) une chaîne d'additions de longueur n et sa suite associée $(V_n) = ((a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n))$

Résultat : $\alpha^{-1} \in \mathrm{GF}(2^m)$

1 **Initialisation** : $\beta_0 \leftarrow \alpha$

2 **pour** i ***de*** 1 ***à*** n **faire**

3 $\quad \beta_{U_i} \leftarrow \beta_{U_{b_i}}^{2^{U_{a_i}}} \times \beta_{U_{a_i}}$

4 **retourner** $\beta_{U_n}^2$

Algorithme 11 : *Square-and-Multiply* [20].

Données : $A \in \mathrm{GF}(2^m)$, un exposant $e = (e_0, e_1, \ldots, e_{t-1})_2$ sur t bits

Résultat : $P = A^e$

1 $P \leftarrow 1$

2 **pour** i **de** 0 **à** $t-1$ **faire**

3 $\quad P \leftarrow P^2$

4 $\quad$ **si** $(e_{t-1-i} = 1)$ **alors**

5 $\quad\quad P \leftarrow P \times A$

6 **retourner** P

$T_0 = A^{(1)_2}$	$u_0 = 1$
$T_1 = T_0^2 \times T_0 = A^{(10)_2} \times A^{(1)_2} = A^{(11)_2}$	$u_1 = u_0 + u_0$
$T_2 = T_1^2 \times T_0 = A^{(110)_2} \times A^{(1)_2} = A^{(111)_2}$	$u_2 = u_1 + u_0$
$T_3 = T_2^{2^3} \times T_2 = A^{(111000)_2} \times A^{(111)_2} = A^{(111111)_2}$	$u_3 = u_2 + u_2$
$A^{-1} = T_3^2 = A^{(1111110)_2}$	

TABLE 3.2: Les étapes pour l'inversion A de $\mathrm{GF}(2^{10})$ basée sur l'algorithme d'Itoh-Tsujii avec la chaîne d'additions correspondante.

$T_0 = A^{(1)_2}$
$T_1 = T_0^2 \times T_0 = A^{(10)_2} \times A^{(1)_2} = A^{(11)_2}$
$T_2 = T_1^{2^2} \times T_1 = A^{(1100)_2} \times A^{(11)_2} = A^{(1111)_2}$
$T_3 = T_2^{2^4} \times T_2 = A^{(11110000)_2} \times A^{(1111)_2} = A^{(11111111)_2}$
$T_4 = T_3^2 \times T_0 = A^{(1111111110)_2} \times A^{(1)_2} = A^{(111111111)_2}$
$T_5 = T_4^2 = A^{2^{10}-2} = A^{(1\ 111\ 111\ 110)_2}$

TABLE 3.3: Les étapes pour l'inversion A de $\mathrm{GF}(2^7)$ basée sur l'algorithme d'Itoh-Tsujii.

Termes U_i	Termes associés V_i	Valeurs effectives calculées
U_0		1
$U_1 = U_0 + U_0$	$V_1 = (0, 0)$	2
$U_2 = U_1 + U_1$	$V_2 = (1, 1)$	4
$U_3 = U_2 + U_0$	$V_3 = (2, 0)$	5
$U_4 = U_2 + U_2$	$V_4 = (2, 2)$	8
$U_5 = U_4 + U_3$	$V_5 = (4, 3)$	13

TABLE 3.4: Une chaîne d'additions pour atteindre $U_n = 13$ avec $n = 5$ additions.

Récemment dans [55], un nouvel algorithme basé sur la méthode d'Itoh-Tsujii a été proposé. Les auteurs utilisent le multiplieur hybride issu de [43] qui réalise l'opération $A \times B \times C$ en $\lceil m/w \rceil + 1$ cycles d'horloge où w est la taille des sous-mots adoptée pour l'ensemble de l'architecture. Fin 2014, dans [59], il est remarqué que certains calculs de la séquence d'Itoh-Tsujii peuvent être parallélisés, utilisant pour cela un second multiplieur dédié. Leurs implémentations montrent que leur solution se montre aussi véloce (en nombre de cycles d'horloge à surface quasiment équivalente) tout en étant plus petite que l'architecture suggérée dans [55].

3.3 Solution proposée

Dans une multiplication scalaire ECC basée sur une méthode de *Montgomery* adaptée au *halving*, une inversion (niveau corps fini) est nécessaire à chaque bit de clef égal à 1. Dans un système de représentation de type NAF (*Non-Adjacent Form*), la densité de 1 est de un tiers. Par exemple, sur une clef aléatoire de 33 bits, il y a en moyenne 11 bits égaux à 1. Le problème reste qu'une unité d'inversion spécifique nous conduit à une sous-utilisation du silicium occupé par le circuit. Des multiplications sont quant à elles opérées à chaque bit de clef, ce qui rend un tel opérateur (multiplieur) indispensable. Nous proposons dès lors d'utiliser ce multiplieur pour effectuer l'inversion (reposant sur une exponentiation) et ainsi minimiser le cout en surface. L'unité arithmétique que nous proposons sera capable d'inverser un élément mais aussi d'en multiplier deux entre eux : nous la dénoterons MIU (pour *Multiplication-Inversion Unit*). Pour ce faire, nous avons utilisé l'algorithme d'Itoh-Tsujii qui ne requiert, pour l'inversion, qu'une succession de multiplications et de carrés (qui ne sont, rappelons le, que des décalages circulaires en base normale). Ainsi, réaliser une inversion est finalement réduit à assurer le bon déroulement de la séquence d'Itoh-Tsujii. Si nous regardons de nouveau l'algorithme d'Itoh-Tsujii, nous pouvons remarquer que des registres temporaires de taille m sont employés : leur nombre dépend de la nature de la chaîne d'addition utilisée. En utilisant des chaînes binaires, ce nombre est réduit à deux (pour U_0 et U_{i-1}, nous ne pouvons

aller en deçà). En d'autres termes, réaliser une MIU concentre l'emploi d'un registre à décalage (ROL), de registres temporaires, d'un multiplieur et enfin, d'un contrôleur pour orchestrer le tout. Bien que l'idée première fût de ne pas dédier une unité à l'inversion, nous avons malgré tout grossi (modérément) l'architecture de notre MIU afin d'accélérer certaines multiplications. Ces multiplications particulières apparaissent pendant l'algorithme d'exponentiation d'Itoh-Tsujii. Pour ces termes multiplicatifs particuliers (SMP : *Specific Multiplication Pattern*), nous sommes capables de produire deux bits du produit par cycle d'horloge. Malgré tout, nous ne parvenons pas à obtenir un facteur deux en terme de vitesse pour ces SMPs : en effet, certains bits p_i seront générés deux fois. Cette redondance nous empêche de gagner davantage sur le débit moyen en sortie d'un multiplieur SMP. La modification apportée à l'algorithme de Massey-Omura pour prendre en considération les SMPs nous a conduit à l'introduction d'une nouvelle base : les éléments sont maintenant encodées en base normale permutée (PNB : *Permuted Normal Basis*). Les coefficients d'un élément en base normale ne sont plus « rangés » dans l'ordre naturel mais subissent une permutation. Les raisons et la définition de la base permutée (PNB) seront explicitées plus tard dans le manuscrit.

3.3.1 Décaler avec des blocs-mémoires

Durant l'exponentiation d'Itoh-Tsujii, les produits intermédiaires doivent subir des mises au carré successives. Cela implique de devoir décaler (circulairement) les opérandes d'un un nombre de bits définis par la chaîne d'addition choisie par l'utilisateur. Par exemple, si nous reprenons la chaîne d'addition de l'exemple 3.4 (c'est à dire $U = (1, 2, 4, 5, 8, 13)$), les décalages sont les suivants : $1, 3$. Pour des chaînes plus grandes, le nombre de décalages différents à prendre en compte croît de façon logarithmique. Par exemple, pour $m = 571$, il est nécessaire de considérer une dizaine de valeurs de décalage différentes. Si nous souhaitions implanter sur circuit un opérateur effectuant ces décalages en un cycle d'horloge, la surface requise explose. Par exemple, pour le corps GF(2^{571}), une telle fonctionnalité synthétisée sur Spartan 6 occupe 3425 LUTs tandis qu'à titre de comparaison un multiplieur de Massey-Omura complet ne requiert que 2246 LUTs. Nous remplaçons ces unités de décalage par un bloc-mémoire (BRAM). Plutôt que stocker chaque bit du (ou des) produit(s) intermédiaire(s) dans un (ou des) registre(s) de taille m, nous les dupliquons w fois dans un bloc mémoire (BRAM) en nous appuyant sur le schéma suivant : $[p_0, p_1, \ldots, p_{w-1}]$ sera stocké à l'adresse @$=0$, $[p_1, p_2, \ldots, p_w]$ à l'adresse @$=1$, $[p_2, p_3, \ldots, p_{w+1}]$, …, $[p_{m-1}, p_0, \ldots, p_{w-2}]$ à l'adresse @ $= m - 1$. Les blocs-mémoire (BRAMs) dans les FPGAs actuels sont assez gros pour supporter les $m \cdot w$ bits requis pour une telle méthode. Avec un bus de largeur $w = 32$, 7456 bits sont requis pour $m = 233$ et 18272 bits pour $m = 571$. Dans un Spartan 6 (qui est un produit d'entrée de

gamme), des blocs de 18Kb sont disponibles. Ainsi, pour $m=571$, nous en avons besoin de deux. Un exemple est donné dans la figure 3.5.

Le multiplieur de Massey-Omura est un opérateur séquentiel qui produit à chaque cycle d'horloge un bit du produit. Nous utilisons un petit registre de w bits afin de stocker les w bits consécutifs ($[p_i, p_{i+1}, \ldots, p_{i+w-1}]$). Le contenu de ce registre est envoyé au bloc-mémoire à chaque cycle d'horloge (il y a donc une latence de w cycles, le temps que ce registre tampon se remplisse). En conséquence, le nombre de cycles d'horloge nécessaires pour calculer chaque produit de la séquence d'exponentiation est de fait $m + w$.

Les mises au carré sont réalisées en lisant les mots composants l'élément permuté à l'adresse $(i + \alpha w) \bmod m$ pour $\alpha \in \{0, 1, 2, \ldots, \lfloor m/w \rfloor\}$, où i est la valeur du décalage. Cette méthode semble équivalente à une stratégie plus naïve, qui consisterait à décaler de 1 bit autant de fois que nécessaire (et en autant de cycles d'horloge) le registre en question. Cette stratégie nous permettra cependant d'avoir un contrôle plus aisé à implanter et une plus grande flexibilité. Cette méthode rentre aussi dans la philosophie selon laquelle le transfert de données entre unités fonctionnelles doit se faire sur des bus de tailles modérées.

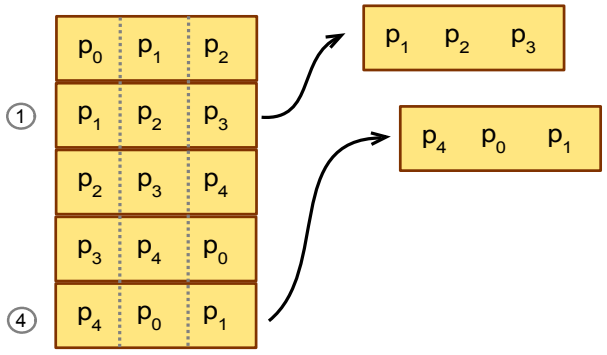


FIGURE 3.5: Dans cet exemple, nous travaillons dans GF(2^5) avec des mots de $w = 3$ bits. En lisant les mots aux adresses 1 et 4, nous obtenons les mots p_1, p_2, p_3 et p_4, p_0, p_1 qui, concaténés donnent $p_1, p_2, p_3, p_4, p_0, p_1$. C'est un décalage circulaire de p_0, p_1, p_2, p_3, p_4 si nous ignorons le dernier bit p_1.

Algorithme 13 : Multiplication de Massey-Omura avec un tampon de w bits.

Données : $A, B \in \mathrm{GF}(2^m)$
Résultat : $P = A \times B$

1 $G \leftarrow 0$
2 **pour** i **de** 0 **à** $m + w - 2$ **faire**
3 $\quad G \leftarrow \mathtt{SHL}(G, 1)$; $G[w-1] \leftarrow A \times M_0 \times B^T$
4 $\quad$ **si** $i \geq (w-1)$ **alors**
5 $\quad\quad P[i - w + 1] \leftarrow G$ *(Écrire en BRAM)*
6 $\quad A \leftarrow \mathtt{ROL}(A, 1)$; $B \leftarrow \mathtt{ROL}(B, 1)$;
7 **retourner** P

3.3.2 Exploitation des formes $A^{2^k} \times A$

Dans la séquence d'exponentiation d'Itoh-Tsujii, des termes multiplicatifs spécifiques, que nous nommerons **SMP** (*Specific Multiplication Pattern*), apparaissent régulièrement : $A^{2^k} \times A$. Ce motif est lié à l'apparition de $U_i = U_j + U_j$ dans la chaîne d'addition considérée. Nous modifions l'algorithme de Massey-Omura (et l'opérateur induit) afin qu'il supporte à la fois des multiplications standards ($A \times B$) ainsi que des SMPs. Lorsque nous avons à calculer des SMPs, nous remarquons que $\texttt{ROL}(A, -i) \times M_0$ et $M_0 \times \texttt{ROL}(A, -i)^T$ apparaissent dans la suite des opérations. Puisque ces deux valeurs sont égales (en fait, l'une est la transposée de l'autre), nous pourrions factoriser un produit matrice-vecteur à chaque itération de l'algorithme. À chaque cycle d'horloge, nous évaluons $V = M_0 \times \texttt{ROL}(A, -i)^T$ et retournons ($p_i = \texttt{ROL}(A, k - i) \times V, p_{i+k} = \texttt{ROL}(A, -i - k) \times V$). Par exemple, si $k = 1$, l'algorithme retournera à chaque cycle d'horloge les couples de bits (p_i, p_{i+1}). Au cycle d'horloge 0, la méthode renvoie (p_0, p_1), au cycle d'horloge 1, (p_1, p_2) ainsi de suite. Dans cette exemple, quasi pathologique, il faudra $m - 1$ cycles d'horloge afin d'obtenir entièrement le produit. Nous aurions pu nous attendre à réduire drastiquement le temps de calcul en produisant deux bits simultanément : nous aurions pu envisager un temps de calcul en $\approx m/2$ cycles d'horloge. Cela dit, la redondance des sorties pourra être minimisée grâce à l'astuce que nous présenterons dans la session suivante. Notez que la sortie s'effectue toujours en série (1 ou 2 bits à la fois) et qu'une telle particularité peut être exploitée en lançant parallèlement des calculs annexes basés sur les bits produits à chaque cycle d'horloge (comme présenté dans [55]).

3.3.3 Multiplication et inversion en base normale permutée

Comme nous l'avons vu précédemment, lors du calcul d'un SMP, une redondance dans les bits de sorties apparaît. Dans l'exemple préalablement donné, ce niveau de redondance approche 2 : chacun des bits du produit est généré deux fois lors des $m - 1$ cycles d'horloge nécessaires (sauf p_{m-1} qui lui, n'est produit qu'une fois). Cette redondance est notamment liée à la constante de décalage dans l'algorithme de Massey-Omura. Jusqu'ici, à chaque cycle d'horloge, les registres A, B et P (voir Algo. 10) sont décalés de 1 bit vers la gauche. Que se passerait-il si nous changions ce décalage ? Si nous reprenons l'exemple avec $k = 1$ et choisissons un décalage de 2, les sorties seraient (p_0, p_1) au cycle d'horloge 0, (p_2, p_3) au cycle d'horloge 1, (p_4, p_5) au cycle d'horloge 2, etc. Ainsi, en considérant une autre valeur de décalage, nous parvenons à réduire la redondance puisqu'ici, en $m/2$ (m pair) ou $(m + 1)/2$ (m impair) cycles d'horloge, tous les bits du produit ont été générés. Nous proposons ainsi de changer la constante de décalage θ de

façon à réduire la redondance (et ainsi, le temps de calcul). Pour des raisons matérielles, ce θ devra être constant pour chacune des multiplications de la séquence d'Itoh-Tsujii. Il n'est possible de choisir un θ particulier pour une multiplication particulière : cela engendrerait une complexité circuit que nous tentons d'éviter jusqu'alors. Aussi, nous avons écrit un programme Python, qui à partir d'une chaîne d'addition, teste l'ensemble des θ possibles et retourne celui qui minimise le temps de calcul (en terme de cycles d'horloge). Pour des raisons pratiques, nous avons choisi des chaînes d'addition binaires (c'est à dire que chaque terme est calculé comme $U_i = U_{i-1} + U_{i-1}$ ou $U_i = U_{i-1} + U_0$ avec $U_0 = 1$) : implanter l'algorithme d'Itoh-Tsujii avec de telles chaînes ne requiert que deux registres temporaires (qui sera ici notre BRAM pour U_{i-1} et un registre spécifique de taille m pour U_0). Les chaînes d'addition générées par la méthode binaire ne sont pas toujours les plus courtes, mais en pratique et pour les corps du NIST [51], toutes nos chaines sont au pire un terme plus longues que les plus courtes trouvées sur [60]. Ce décalage nouveau nous amène à proposer l'introduction d'une **base normale permutée** (PNB : *Permuted Normal Basis*) :

Definition 3.1. PNB : *Permuted Normal Basis*
Un élément $A = [a_0, a_1, a_2, \ldots, a_{m-1}]$ (base normale) est représenté comme $A' = [a_0, a_\theta, a_{2\theta \bmod m}, \ldots, a_{(m-1)\theta \bmod m}]$ (base normale permutée).

Il est important de signaler qu'une mise au carré en PNB est, en outre, un décalage circulaire (la valeur de décalage n'est pas de 1 bit mais de θ). Pareillement, la nature de l'addition entre deux éléments du corps n'est pas chamboulée par la représentation en base normale permutée. Notre adaptation de l'algorithme de Massey-Omura en PNB (et pour les SMPs) est détaillée dans l'algorithme 14 où $N(k)$ dénote le nombre de cycles d'horloges nécessaire pour calculer une SMP $A^{2^k} \times A$ (au bout de combien de cycles obtiendrons-nous l'ensemble des bits du produit $A^{2^k} \times A$? Cela varie d'un k à l'autre).

Algorithme 14 : Multiplication de Massey-Omura en PNB.
Données : $A, B \in \mathrm{GF}(2^m)$ en PNB, et k
Résultat : $P = A \times B$
1 $C \leftarrow \texttt{ROL}(A, -k)$, $G \leftarrow 0$, $H \leftarrow 0$, $j \leftarrow k \cdot \theta^{-1} \bmod m$
2 **pour** i **de** 0 **à** $(w + N(k) - 2)$ **faire**
3 $\quad G \leftarrow \texttt{SHL}(G, 1)$; $H \leftarrow \texttt{SHL}(H, 1)$
4 $\quad V \leftarrow A \times M'_0$
5 $\quad G[w-1] \leftarrow V \times B^T$; $H[w-1] \leftarrow V \times C^T$
6 $\quad$ **si** $i \geq (w-1)$ **alors**
7 $\quad\quad P[i - w + 1] \leftarrow G$ (*écrire en DP-BRAM*)
8 $\quad\quad P[j + i - w + 1] \leftarrow H$ (*écrire en DP-BRAM*)
9 $\quad A \leftarrow \texttt{ROL}(A, 1)$; $B \leftarrow \texttt{ROL}(B, 1)$; $C \leftarrow \texttt{ROL}(C, 1)$;
10 **retourner** P

Dans notre algorithme 14, la matrice M'_0 est une matrice M_0 « transformée » et adaptée à notre nouvelle base PNB. La matrice M'_0 est toujours symétrique[‡] (si ce n'était pas le cas, notre factorisation du produit matrice-vecteur ne pourrait être appliquée). Elle est une permutation des lignes et colonnes de la matrice M_0. La ligne i dans M_0 se retrouve à la ligne $i \times \theta^{-1} \mod m$ dans M'_0 tandis que la colonne j dans M_0 se retrouve à la colonne $j \times \theta^{-1} \mod m$ dans M'_0. Notez que θ^{-1} est l'inverse modulaire de θ modulo m. Un exemple est donné dans la figure 3.6 pour GF(2^7).

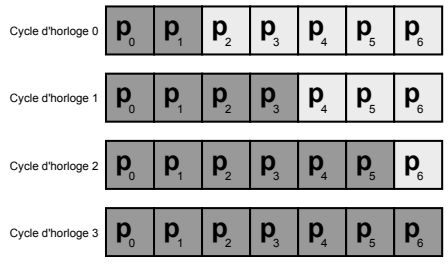
$$M_0 = \begin{pmatrix} 0&1&0&0&0&0&0 \\ 1&0&1&0&0&1&1 \\ 0&1&0&1&1&1&0 \\ 0&0&1&0&0&1&0 \\ 0&0&1&0&0&0&1 \\ 0&1&1&1&0&0&1 \\ 0&1&0&0&1&1&1 \end{pmatrix} \quad \Rightarrow \quad M'_0 = \begin{pmatrix} 0&0&0&0&0&1&0 \\ 0&0&0&1&1&0&0 \\ 0&0&1&0&1&1&1 \\ 0&1&0&0&1&1&1 \\ 0&1&1&1&0&1&0 \\ 1&0&1&1&1&0&0 \\ 0&0&1&1&0&0&0 \end{pmatrix}$$

FIGURE 3.6: M_0 et la matrice associée M'_0 (PNB) pour GF(2^7) et $\theta = 3$.

Notre algorithme 14 produit les bits $(p_{i\theta}, p_{i\theta+k})$ à chaque cycle d'horloge. Puisque m est premier, les indexes $j\theta \bmod m$ génèrent GF(m) vu comme un groupe additif. En conséquence, il existe pour chaque couple d'entiers $i < m$ et $k < m$ un autre entier $j < m$ tel que $(i\theta + k) \bmod m = j\theta \bmod m$. Nous écrivons alors dans une mémoire double-ports (DP-BRAM : *Dual Port BRAM*) les mots $[p_{i\theta}, p_{(i+1)\theta}, \ldots, p_{(i+w-1)\theta}]$ et $[p_{j\theta}, p_{(j+1)\theta}, \ldots, p_{(j+w-1)\theta}]$ aux adresses (respectivement) @ $= i$, @ $= j$.

Nous allons présenter un exemple dans un corps de taille réduite GF(2^7) ($m = 7$) dans le but de clarifier les esprits. Ce corps nous conduit à la chaîne d'addition binaire $(U) = (1, 2, 3, 6)$ (voir tableau 3.2). Dans cette chaîne, il y a deux SMPs : $k = 1$ (pour $2 = 1+1$) et $k = 3$ (pour $6 = 3+3$). Notre programme Python retourne $\theta = 2$. Pour la SMP $k = 1$, les couples de bits rendus à chaque cycle d'horloge sont les suivants :

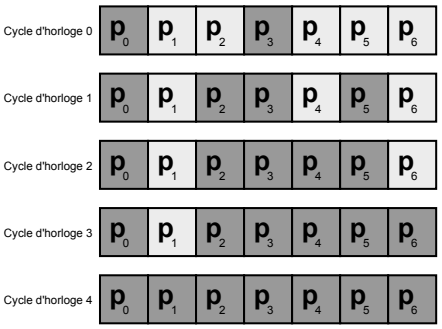
- Cycle d'horloge 0 : $(\underline{p_0}, p_1)$
- Cycle d'horloge 1 : (p_2, p_3)
- Cycle d'horloge 2 : (p_4, p_5)
- Cycle d'horloge 3 : $(p_6, \underline{p_0})$



Il suffit ici de 4 cycles d'horloge pour obtenir la totalité des bits du produit (nous noterons $N(1) = 4$). Les redondances sont soulignées. Enfin, pour le deuxième SMP ($k = 3$), les bits du produit sont rendus dans cet ordre :

‡. car m est ici choisi premier.

– Cycle d'horloge 0 : $(\underline{p_0}, p_3)$
– Cycle d'horloge 1 : $(\underline{p_2}, p_5)$
– Cycle d'horloge 2 : $(\underline{p_4}, \underline{p_0})$
– Cycle d'horloge 3 : $(p_6, \underline{p_2})$
– Cycle d'horloge 4 : $(p_1, \underline{p_4})$



Ici, il faut 5 cycles d'horloge pour obtenir l'ensemble des bits du produit ($N(3) = 5$). Ainsi, en profitant des SMPs dans l'exponentiation d'Itoh-Tsujii, nous avons besoin de 16 cycles d'horloge (sans compter tous les cycles qui pourraient être liés au contrôle) pour calculer un inverse. En effet, 4 cycles sont nécessaires pour le premier SMP ($k = 1$), 7 cycles requis pour une multiplication classique (pour le calcul de $P_2 = P_1^2 \times P_0 = A^{(110)_2} \times A^{(1)_2} = A^{(111)_2}$) et enfin, 5 cycles pour le dernier SMP ($k = 3$). Sans l'exploitation des SMPs, il faudrait $7 + 7 + 7 = 21$ cycles d'horloge (3 multiplications successives) pour obtenir le même résultat. Le gain, dans cet exemple « jouet » est d'environ 24%.
Un autre exemple, avec le corps de taille cryptographique GF(2^{163}) : nous trouvons aisément la chaîne d'addition binaire suivante : $(U) = (1, 2, 4, 5, 10, 20, 40, 80,$
$81, 162)$. Notre programme retourne $\theta = 72$. Pour ce corps, les SMPs et leurs coûts en termes de cycles d'horloge sont les suivants : $N(k = 1) = 120$, $N(k = 2) = 86$,$N(k = 5) = 111$, $N(k = 10) = 104$, $N(k = 20) = 118$, $N(k = 40) = 90$ et $N(k = 81) = 103$. Le gain grâce à la solution PNB est ici estimé à 21% (voir section suivante Sec. 3.3.4).

3.3.4 Estimation du coût

Nous noterons N_{SMP} le nombre de SMPs dans la séquence d'Itoh-Tsujii. Nous estimons le nombre de cycles d'horloge nécessaire pour calculer une inversion par la formule $C_{\text{inv}} = C_{\text{IO}} + C_{\text{SMPs}} + C_{\text{nonSMPs}}$ où :
– $C_{\text{IO}} = 2\lceil \frac{m}{w} \rceil$ est le temps passé à la lecture/écriture des entrées/sorties ;
– $C_{\text{SMPs}} = D + 2N_{\text{SMP}}\lceil \frac{m}{w} \rceil$, D est le nombre de cycles d'horloge que l'architecture consacre au calcul des SMPs. Des valeurs de D sont données dans le tableau 3.5 : elles ont été calculées par notre programme Python (en quelques minutes sur un PC standard) et nous n'avons pas aujourd'hui de formule explicite pour exprimer cette quantité. Le deuxième terme énonce le temps passé lors des transferts de données internes (typiquement, récupération de l'élément stocké en BRAM) avant le lancement d'un SMP.
– $C_{\text{nonSMPs}} = (|(U)| - N_{\text{SMP}}) \times (m + \lceil \frac{m}{w} \rceil)$, cette quantité est similaire à celle précédemment détaillée mais elle ne concerne, cette fois, que les multiplications classiques non-SMP.

TABLE 3.5: Détails et comparaisons de l'inversion pour les corps du NIST [51]. L'unité *ch* correspond au nombre de cycles d'horloge requis pour l'inversion.

m	(θ, D)	original (ch)	modifié (ch)	gain
163	$(72, 732)$	1702	1335	$\approx 21\%$
233	$(36, 1046)$	2667	2082	$\approx 21\%$
283	$(28, 1431)$	3522	2761	$\approx 21\%$
409	$(35, 2263)$	5090	4185	$\approx 17\%$
571	$(171, 3221)$	8282	5973	$\approx 27\%$

Dans le tableau 3.5, nous exposons pour tous les corps cryptographique du NIST [51] le paramètre θ optimal (pour le temps d'inversion) ainsi que le temps d'inversion (en cycles d'horloge) requis en utilisant, respectivement, notre multiplieur PNB et le multiplieur de Massey-Omura original. Ici, w, la taille des mots, a été choisie égale à 32 (pour des raisons matérielles). Notre méthode permet, en moyenne, un gain de 20% face à une approche non-SMP.

Notre MIU prend en charge les multiplications non-SMP de façon toute aussi efficace qu'un multiplieur de Massey-Omura : il suffit pour cela d'ignorer la deuxième sortie et de récupérer, séquentiellement, les bits $p_{i\theta}$. Enfin, notons que l'obtention du paramètre θ par l'intermédiaire de notre programme Python se fait très efficacement puisqu'il ne suffit que d'une poignée de minutes pour les corps les plus grands (Core i7 @ 3 GHz avec 8 GO de mémoire vive). Nous nous sommes également demandés si notre stratégie de base normale permutée était généralisable à des architectures contenant plus d'un bloc matriciel $\times M_0$. La réponse est positive, il ne demande d'ailleurs que peu d'effort d'ajouter un bloc et d'exploiter les motifs SMPs. Cependant, l'efficacité décroit en fonction du nombre de blocs matriciels ajoutés. Nous obtenons les courbes suivantes en figure 3.7, avec en abscisse la taille du corps (seuls les cas où m est premier ont été traités) et en ordonnée le nombre de cycles nécessaires pour réaliser une inversion. Les blocs supplémentaires calculent respectivement les bits du produit $c_{(i+1)\theta}$, $c_{(i+2)\theta}$, ... et $c_{(i+d-1)\theta}$ dans la représentation en base normale permutée (tandis que le bloc initial lui se charge de $c_{i\theta}$). Nous avons observé que ce choix n'est pas forcément optimal, le vrai problème *d'optimisation* serait de rechercher les constantes entières s_j à attribuer aux blocs $M_0\times$ calculant, alors, les bits $c_{(i+s_j)\theta}$ (tout en tachant de trouver le décalage θ optimum). Quoi qu'il en soit, les variations semblent anecdotiques et ne paraissent pas changer le comportement asymptotique de l'ajout de blocs sur notre algorithme SMP. La forme en « dents de scie » des courbes de la figure 3.7 est liée à la proportionnalité qu'il existe entre le poids de Hamming de $m-1$ (m est la « taille » du corps GF(2^m)) et le temps d'inversion. Nous pouvons avoir $m_0 > m_1$ mais $\mathbf{HW}(m_0) \leq \mathbf{HW}(m_1)$ où **HW** est une fonction qui détermine le poids de Hamming de la décomposition binaire

de m_0 et de m_1. La progression du coût n'est donc pas strictement linéaire en fonction de m.

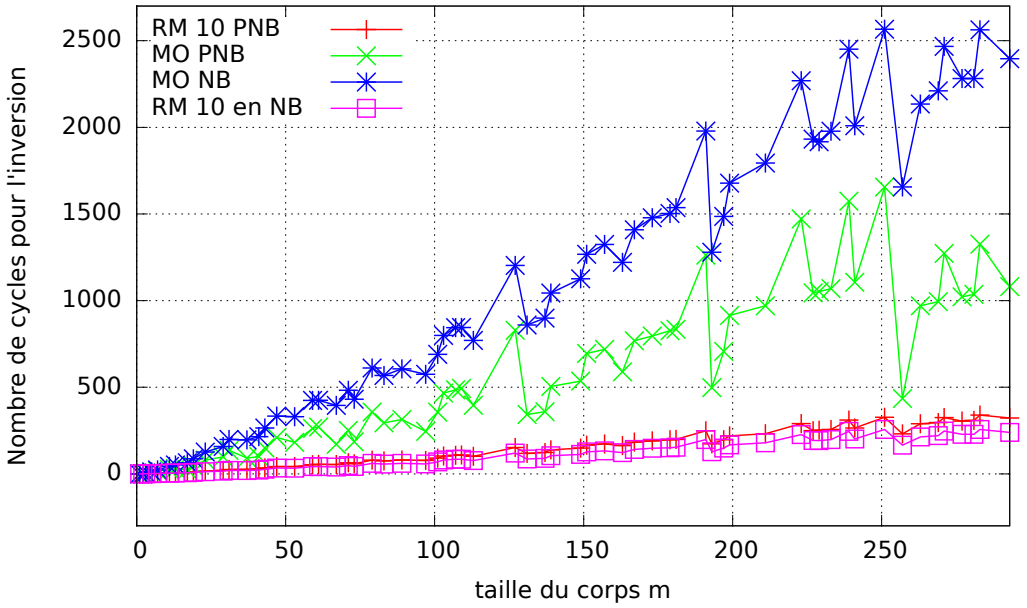


FIGURE 3.7: **Influence de l'ajout de blocs $M_0\times$ sur le temps d'inversion :** *RM 10 en NB* est une inversion réalisée à l'aide d'un multiplieur de Reyhani-Masoleh [52] avec 10 blocs $\times M_0$, *MO PNB* est une inversion réalisée avec notre multiplieur PNB modifié, *MO NB* est une inversion réalisée avec un multiplieur de Massey-Omura [49] et enfin, *RM 10 en PNB* est une inversion réalisée à l'aide d'un multiplieur de Reyhani-Masoleh [52] utilisant notre astuce concernant les **SMPs** avec 10 blocs $\times M_0$.

3.4 Détails de l'implémentation sur FPGA et résultats

L'architecture de notre unité multiplieur-inverseur (MIU) est décrite à travers le schéma de la figure 3.8 où $\ell = \lceil \log_2 m \rceil$ et w désigne la taille des bus. Notre architecture est capable de calculer des produits (en base normale permutée) mais aussi des inversions (également en base normale permutée). Lors de l'inversion, notre architecture utilise les SMPs afin d'accélérer le processus. Tout au long de cette inversion, des adresses sont calculées (notamment pour l'écriture dans la BRAM). Afin de réduire la latence et en notant que le calcul d'adresse suit toujours le même schéma (addition et réduction modulo m), nous avons choisi de pipeliner ces deux opérations. Le bloc principal de notre architecture est un multiplieur de Massey-Omura, modifié dans le but de prendre en considération les SMPs. Ce multiplieur est présenté dans la figure 3.9. Il comporte deux blocs dédiés au produit scalaire (au lieu d'un vis à vis d'un Massey-Omura classique, voir figure 3.1), de trois registres à décalage circulaire (au lieu de deux). Nous noterons également la présence de deux petits registres de w bits (en bas du schéma) qui nous

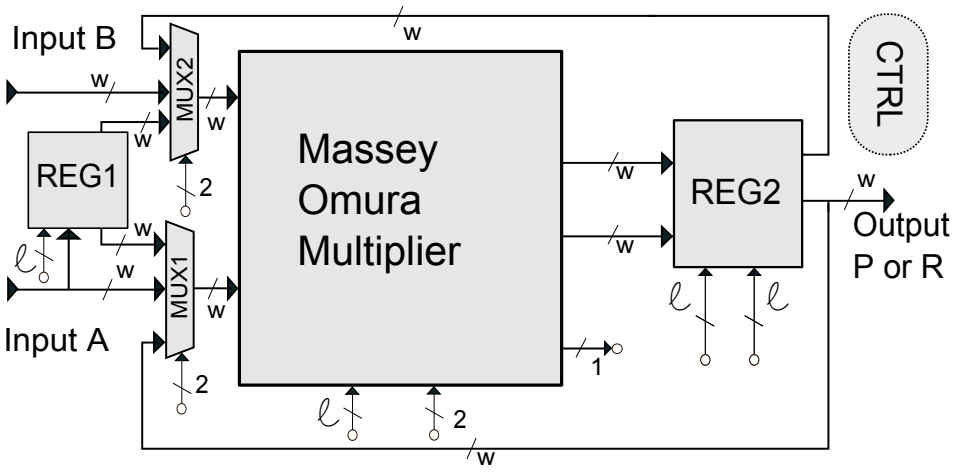


FIGURE 3.8: Architecture de notre unité multiplieur-inverseur *(MIU)*.

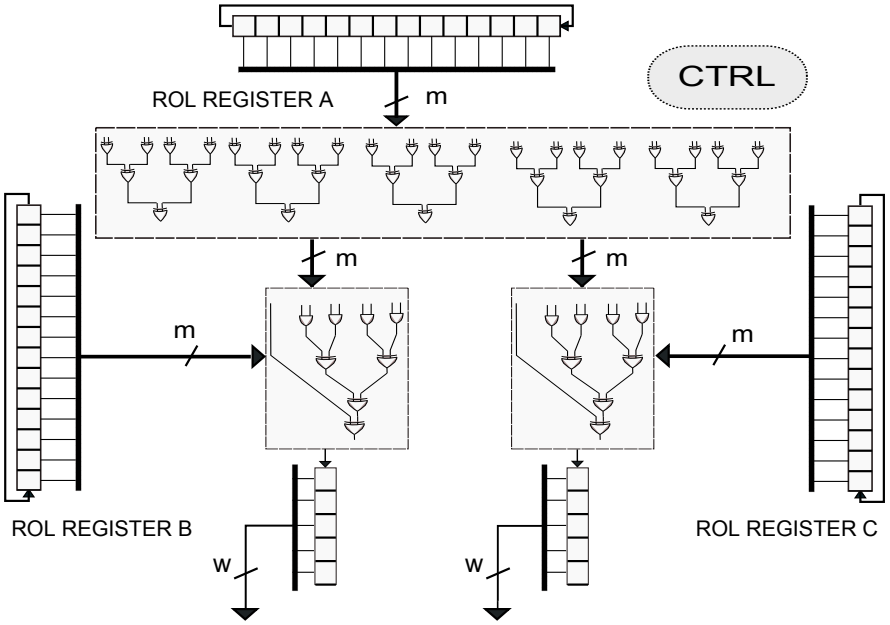


FIGURE 3.9: Vue haut niveau de notre multiplieur modifié de Massey-Omura pour la représentation PNB.

servent de mémoire tampon. Ces mémoires tampons sont destinées à stocker les mots $[p_i, p_{i+1}, \ldots, p_{i+w-1}]$ et $[p_j, p_{j+1}, \ldots, p_{j+w-1}]$ qui seront ensuite envoyés au bloc-mémoire (portant ici le nom de REG2 dans le dessin Fig. 3.8). Le REG1 est quant à lui un registre de m bits qui permet de stocker A, l'élément à inverser. Ce registre REG1 est indispensable quand apparaît dans la chaîne d'addition des termes faisant intervenir U_0 (comme dans, par exemple, pour $u_1 = u_0 + u_0$ nous avons $P_1 = P_0^2 \times P_0 = A^{(10)_2} \times A^{(1)_2} = A^{(11)_2}$, voir exemple . 3.2). Dans ce travail, nous avons choisi $w = 32$ bits mais il est à noter qu'il s'agit d'un paramètre de notre architecture et que cette valeur peut être modifiée. Évidemment, il faut éviter des tailles extravagantes pour que la taille du bus corresponde à la taille d'un mot d'un BRAM $(16, 18, 32, 36$ bits) où à l'un de ses multiples.

L'ordonnancement des calculs est contrôlé par une machine d'état (FSM : *Finite-State Machine*), présentée en Fig. 3.10. Elle comporte différents macro-états (chaque état se compose en fait d'un ensemble de sous-états, mais qui ne seront pas explicités ici) décrits ci-dessous :

- L'état initial `LOAD_OP` permet de charger les opérandes $A \in \mathrm{GF}(2^m)$ (en PNB) et $B \in \mathrm{GF}(2^m)$ (en PNB) dans l'unité. L'envoi des données s'effectue de façon séquentielle sous la forme d'une suite de sous-mots de w bits composant respectivement A et B. Il n'est pas obligatoire de charger B dans le cadre d'une inversion.
- L'état `MULT` lance une multiplication standard $A \times B$.
- L'état `DONE` est atteint quand le calcul de l'inversion ou de la multiplication est terminé. Dans cet état, l'unité attend un signal de l'utilisateur (un flag) avant de retourner, de façon séquentielle également (suite de sous-mots de w bits), le résultat.
- L'état `REG×REG` débute le calcul de la première multiplication de la séquence d'Itoh-Tsujii $A \times A$ en utilisant l'unité de Massey-Omura modifiée. En base normale (permutée), un carré, comme nous l'avons vu auparavant est trivial : c'est un décalage circulaire. Cependant, pour le bon fonctionnement de notre approche (issue de la section Sec. 3.3.1), la valeur de A^2 doit être stockée en BRAM sous la forme $[p_0, p_1, \ldots, p_{w-1}]$ $[p_1, p_2, \ldots, p_w], \ldots [p_{m-1}, p_0, \ldots, p_{m+w-2}]$. C'est pourquoi cette première étape est une étape clef et c'est de cette « conversion » que nait la réelle complexité du décalage : elle coute m cycles d'horloge, soit le temps d'une multiplication non SMP. Une fois cette conversion achevée, tous les décalages seront faits à partir de lectures en mémoire. Il est bon de remarquer que si A^2 était disponible dans un registre de m bits avant tout calcul (impliquant un peu de contrôle dans l'unité de Massey-Omura), la première étape d'Itoh-Tsujii serait la multiplication $A^2 \times A$ dont le résultat aurait été stocké en BRAM. Dans ces conditions, aucun temps de conversion n'aurait été exigé. Il s'agit là d'une optimisation que nous n'avons pas encore incorporé dans notre travail.
- L'état `BRAM×BRAM` lance le calcul d'une multiplication SMP $A^{2^k} \times A$. La valeur $N(k)$ est envoyée au multiplieur.
- L'état `REG×BRAM` débute le calcul d'une multiplication $A \times B$ standard.
- L'état `WAIT` est atteint après le lancement d'un calcul (`REG×REG`, `BRAM×BRAM`, `REG×BRAM`) et attend que celui-ci se termine avant de passer à l'étape suivante.
- `NEXT` détermine quelle sera l'étape prochaine de la séquence d'Itoh-Tsujii.

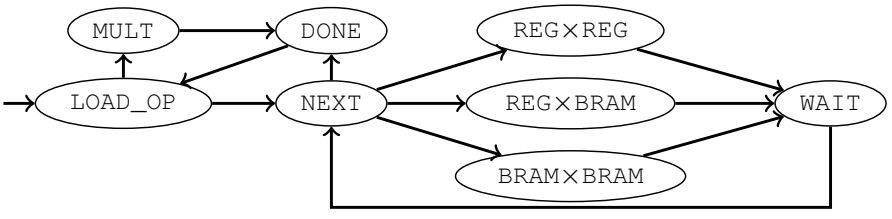


FIGURE 3.10: La machine d'états de notre unité multiplieur-inverseur.

TABLE 3.6: Implémentations FPGA de notre MIU pour différents blocs de multiplications (Spartan 6 LX75T).

m	Algo.	Surface Slices (LUT, FF)	Fréq. MHz	Temps Inv. μs	#block RAM
163	MO1 [49]*	337 (1004, 763)	217	7	1
	RM2 [52]*	423 (1348, 862)	140	6	1
	PNB	469 (1411, 1034)	196	5	1
233	MO1 [49]*	420 (1231, 966)	229	11	1
	RM2 [52]*	569 (1765, 1078)	138	10	1
	PNB	526 (1703, 1296)	181	10	1
283	MO1 [49]*	484 (1659, 1107)	160	22	1
	RM2 [52]*	668 (2164, 1209)	131	13	1
	PNB	719 (2230, 1498)	159	15	1
409	MO1 [49]*	647 (2178, 1501)	163	31	1
	RM2 [52]*	917 (2768, 1610)	139	18	1
	PNB	980 (3167, 1993)	159	22	1
571	MO1 [49]*	941 (3336, 1968)	109	75	2
	RM2 [52]*	1656 (4911, 2727)	80	53	2
	PNB	1190 (4422, 2634)	94	63	2

Nous avons implanté la même architecture Fig. 3.8 avec trois unités de multiplication différentes : une unité de Massey-Omura classique (que nous nommerons MO1 dans la suite du document), une unité de Massey-Omura modifiée (PNB) et enfin, un Massey-Omura avec deux blocs $\times M_0$ (RM2) d'une complexité matérielle réduite grâce aux suggestions de Reyhani-Masoleh (voir [52]). Pour RM2, l'unité de multiplication produit deux bits simultanément qui formeront deux mots de w bits que nous enverrons en BRAM aux adresses $@ = i, @ = i + \lceil m/2 \rceil$. Dans les tableaux 3.6, 3.7 et 3.8, les lignes accompagnées d'une astérisque (*) correspondent à des architectures que nous avons implémentées et optimisées sur le même FPGA que notre solution PNB. Les lignes qui n'en portent pas (le bas du tableau 3.7) rapportent des chiffres issus des documents respectivement indiqués ([55] et [59]). Nous reportons dans ces tableaux la surface en Slices, LUTs et bascules (FF : *Flip-Flops*), le temps nécessaire pour le calcul d'une inversion, le nombre de BRAMS ainsi que la fréquence maximale de fonctionnement des architectures considérées. Pour les architectures issues de [55] et de [59], nous rapportons les données trouvées telles quelles dans les deux papiers respectifs. La cible de leur implémentation est aussi un Virtex-4, ce qui rend notre comparaison légitime.

Nous remarquons que la solution RM2 nous conduit à une baisse importante de la fréquence de fonctionnement du circuit vis à vis des autres implémentations. Notre solution PNB à un impact relativement réduit sur les fréquences de fonctionnement comparées à son design original : celui de Massey-Omura (MO1). Ces fréquences de fonctionnement plus élevées peuvent être désirables lors de la conception d'accélérateur. Un

TABLE 3.7: Implémentations FPGA de notre MIU pour différents blocs de multiplications (Virtex-4 LX100).

m	Algo.	Surface Slices (LUT, FF)	Fréq. MHz	Temps Inv. μs	#block RAM
163	MO1 [49]*	906 (1636, 743)	250	6	1
	RM2 [52]*	1220 (2068, 808)	210	3	1
	PNB	1227 (227, 1058)	247	4	1
233	MO1 [49]*	1256 (2233, 1009)	202	13	1
	RM2 [52]*	1430 (2435, 1016)	204	6	1
	PNB	1654 (2792, 1329)	212	8	1
283	MO1 [49]*	1577 (2839, 1149)	191	18	1
	RM2 [52]*	1924 (2435, 1147)	165	9	1
	PNB	2073 (3741, 1525)	186	12	1
409	MO1 [49]*	2283 (2839, 1149)	169	31	1
	RM2 [52]*	2729 (4833, 1532)	150	15	1
	PNB	2482 (4627, 2024)	155	21	1
571	MO1 [49]*	3378 (5615, 2016)	125	64	2
	RM2 [52]*	4976 (9445, 2090)	107	39	2
	PNB	4308 (5928, 2650)	125	48	2
571	Hybride ($d = 13$) [55]	#LUTs = 85268	74	5	–
	Parallèle ($d = 13$) [59]	#LUTs = 56657	82	5	–

même accélérateur peut contenir plusieurs unités fonctionnelles qu'il ne faut, à l'évidence, pas ralentir. Notre solution PNB rentre dans ce cadre.

Dans [45], l'auteur donne des estimations des couts en terme de temps d'une multiplication scalaire basée sur un algorithme de type *halve-and-add*. Dans le cas de la base normale et d'un recodage r-NAF de la clef, ce coût peut être très approximativement approché par $(m/(r+1) \times I + (m + 3 \times m/(r+1)) \times M)\mu s$ où I est le temps d'une inversion et M le temps d'une multiplication, les deux quantités exprimées en microsecondes. Nous avons aussi évalué la surface des opérateurs de trace Tr, d'additions en nombre de LUTs (une centaine de LUTs). À partir de ces chiffres, nous donnons des estimations de la taille d'un opérateur de multiplication scalaire de type *halve-and-add*. Une telle approche néglige la surface du contrôle mais permet tout de même de se faire une idée des gains que pourrait apporter notre algorithme *PNB*. Pour nous comparer aux solutions de la littérature, nous avons calculé un produit temps surface (PTS), qui est le produit entre le nombre de LUTs du circuit et le temps nécessaire afin de réaliser une multiplication scalaire. Nos estimations sont données dans le tableau 3.8. Nous n'avons pas choisi le corps GF(2^{571}) innocemment, c'est le corps sur-lequel nous avons remarqué une vraie plus-value de notre architecture PNB. En soi, le résultat n'est pas étonnant : l'algorithme d'inversion que nous proposons se base sur une « factorisation » de l'usage du bloc $M_0\times$. Plus cette dernière est « grosse », plus notre méthode sera adaptée. Le

TABLE 3.8: Estimations du Coût/performance pour de multiples unités de multiplication scalaire ($m = 571$,Virtex-4).

	Algorithmes	halving ms	surface #LUTs	PTS $\times 10^{-3}$
NAF	MO1 [49]*	17	5742	95
	RM2 [52]*	13	9572	122
	PNB	**14**	6055	**82**
	Parallel IT (d=13) [59]	1.6	56784	90
	Hybrid IT (d=13) [55]	1.6	85395	136
3-NAF	MO1 [49]*	14		79
	RM2 [52]*	8		76
	PNB	**11**	similaire	**65**
	Parallel IT (d=13) [59]	1.3		74
	Hybrid IT (d=13) [55]	1.4		119

tableau 3.8 montre que notre solution *PNB* se retrouve être plus efficace si nous tenons compte à la fois du temps de calcul et de la surface. Nous pensons dès lors que nos approche pourrait être utilisée pour des applications nécessitant un haut niveau de sécurité mais dont la surface silicium est limitée. À noter qu'il n'y a pas de différences fondamentales entre le NAF et le 3-NAF concernant la surface occupée sur le circuit, c'est pourquoi nous indiquons « similaire » dans la deuxième partie du tableau.

3.5 Conclusion

Nous avons proposé dans ce chapitre une unité de calculs nommée MIU pour « *Multiplication-Inversion* Unit ». Nous avons suggéré l'emploi d'une base normale permutée (PNB), une modification du multiplieur de Massey-Omura ainsi que l'exploitation de motifs spécifiques dans l'algorithme d'inversion d'Itoh-Tsujii. Notre solution permet des inversions, théoriquement, 20% plus rapides face à des procédés plus classiques. Le surplus de matériel nécessaire à notre approche n'a qu'un impact modéré sur les fréquences de fonctionnement du circuit. Notre solution a été implémentée sur FPGA et peut s'avérer être une alternative efficace pour des corps de grandes tailles et plus spécifiquement pour des corps dont le type $t \geq 10$. Nous pensons que notre approche peut être adoptée sur des circuits de taille restreinte. Notons aussi que nous pouvons coupler notre stratégie au parallélisme de [59].

Chapitre 4

Crypto-processeur intégrant une unité de *Halving*

Nous avons proposé dans le chapitre précédent une unité combinant les opérations d'inversion et de multiplication. Ici, nous tentons d'aller plus loin, en proposant une architecture complète d'un crypto-processeur opérant dans GF(2^m) dont les éléments seront représentés en base normale. Nous nous intéresserons aussi au calcul parallèle qu'implique le *halving*. En effet, comme nous l'avons noté auparavant, le halving permet de casser, en partie, l'aspect séquentiel du *double-and-add* habituel (ou l'échelle de Montgomery). Comme indiqué dans [46], nous avons la possibilité de lever cette dépendance en lançant parallèlement sur une partie de la clef un algorithme de type *halve-and-add* et sur l'autre morceau un algorithme de type *double-and-add*. Ce parallélisme devrait nous apporter une protection contre certaines attaques par canaux cachés (au moins celles de type **SPA**) puisque seront lancés deux calculs indépendants simultanément : c'est une façon de semer le trouble chez l'attaquant, les informations fuyantes devraient dès lors se mêler. Nous estimerons aussi la sécurité physique de notre crypto-processeur en mettant en œuvre une attaque dite par *templates*. Par commodité, cette attaque a été menée sur des simulations plutôt que sur un banc d'attaques réel.

4.1 Opération de *Halving*

Nous aurons dans ce chapitre besoin de quelques notions issues de l'arithmétique des corps finis, notamment concernant la trace Tr que nous avons abordée dans 3.2.3. Si nous reprenons les opérations de doublement $Q = P + P$ en coordonnées affines (x, y)

dans $\mathbb{E}(\mathrm{GF}(2^m))$, courbe définie par :

$$y^2 + x \times y = x^3 + a \times x + b \text{ avec } a, b \in \mathrm{GF}(2^m), \tag{4.1}$$

nous avons les calculs suivants :

$$\begin{aligned} \lambda &= x + y/x \\ u &= \lambda^2 + \lambda + a \\ v &= x^2 + u \times (\lambda + 1), \end{aligned}$$

avec $Q = (u, v)$. Nous voulons maintenant trouver le point P tel que $Q = 2 \times P$. Nous nous assurons, tout d'abord, de son unicité et de son existence dans $\mathbb{E}(\mathrm{GF}(2^m))$ en remarquant que, si un tel P existe, alors $Q/2 = Q \times (2^{-1} \bmod N_P)$ où N_P est l'ordre de P dans le groupe $\mathbb{E}(\mathrm{GF}(2^m))$. L'existence de $(2^{-1} \bmod N_P)$ est relié au fait que 2 doit être premier avec N_P, c'est à dire que N_P se doit d'être impair. Dans ce cas, le doublement du point P et la division de P est un automorphisme du groupe généré par ce même point [20]. La première chose à faire pour trouver P est de résoudre, pour λ, l'équation :

$$u = \lambda^2 + \lambda + a \tag{4.2}$$

autrement écrite $\lambda^2 + \lambda = c$ avec $c = u + a$. En base normale et, si m est impair, cette équation se résout facilement. En base normale, cela revient à trouver $\lambda = (\lambda_0, \lambda_1, \ldots, \lambda_{m-1})$ où $\lambda_i \in \{0, 1\}$ tel que :

$$\begin{array}{rcl} \lambda_0 + \lambda_{m-1} & = & c_0 \\ \lambda_1 + \lambda_0 & = & c_1 \\ \lambda_2 + \lambda_1 & = & c_2 \\ \vdots & \vdots & \vdots \\ \lambda_{m-1} + \lambda_{m-2} & = & c_{m-1}. \end{array} \tag{4.3}$$

Nous avons, en appliquant la fonction trace Tr à l'équation 4.2 que $\mathrm{Tr}(\lambda^2 + \lambda) = \mathrm{Tr}(c)$, soit $\mathrm{Tr}^2(\lambda) + \mathrm{Tr}(\lambda) = \mathrm{Tr}(c)$, or $\mathrm{Tr}(A) \in \{0, 1\}$ pour tout $A \in \mathrm{GF}(2^m)$ et de fait $\mathrm{Tr}^2(\lambda) + \mathrm{Tr}(\lambda) = \mathrm{Tr}(\lambda) + \mathrm{Tr}(\lambda) = 0 = \mathrm{Tr}(c)$. En d'autres termes, l'équation $\lambda^2 + \lambda = c$ a une solution si et seulement si $\mathrm{Tr}(c) = 0$. Il est à noter que si λ est une solution de $\lambda^2 + \lambda = c$ alors $\lambda + 1$ l'est également. L'élément $\lambda + 1$ est l'inverse logique de $\lambda = (\lambda_0, \lambda_1, \ldots, \lambda_{m-1})$, que nous noterons $\lambda + 1 = (\overline{\lambda_0}, \overline{\lambda_1}, \ldots, \overline{\lambda_{m-1}})$ où $\overline{0} = 1$ et $\overline{1} = 0$. Si nous supposons que $\lambda_0 = 0$ (ce que nous pouvons faire puisque l'une des solutions de l'équation $\lambda^2 + \lambda = c$

aura son premier bit égal à 0), nous pouvons réécrire l'équation 4.4 comme :

$$\begin{array}{lllllll} \lambda_{m-1} & = & & & c_0 & & \\ \lambda_1 & = & & & c_1 & & \\ \lambda_2 + \lambda_1 & = & \lambda_2 + c_1 & = & c_2 & \Rightarrow & \lambda_2 = c_1 + c_2 \\ \lambda_3 + \lambda_2 & = & \lambda_3 + c_1 + c_2 & = & c_3 & \Rightarrow & \lambda_3 = c_1 + c_2 + c_3 \\ \vdots & \vdots & & & \vdots & & \\ \lambda_{m-1} + \lambda_{m-2} & = & \lambda_{m-1} + \sum_{i=1}^{m-2} c_i & = & c_{m-1} & \Rightarrow & \lambda_{m-1} = \sum_{i=1}^{m-1} c_i \end{array} \tag{4.4}$$

Pour que le système ait une solution, il faut que $\lambda_{m-1} = \sum_{i=1}^{m-1} c_i = c_0$, ce qui est le cas dans l'hypothèse où $\mathrm{Tr}(c) = 0$. En effet, nous avons $\sum_{i=1}^{m-1} c_i = \mathrm{Tr}(c) + c_0 = 0 + c_0 = c_0$. Autrement dit, si $\mathrm{Tr}(c) = 0$ alors une solution à l'équation 4.2, que nous noterons $\lambda_P = (\lambda_0, \lambda_1, \ldots, \lambda_{m-1})$ est

$$\begin{array}{lll} \lambda_0 & = & 0 \\ \lambda_1 & = & c_1 \\ \lambda_2 & = & c_1 + c_2 \\ \lambda_3 & = & c_1 + c_2 + c_3 \\ \vdots & \vdots & \vdots \\ \lambda_{m-1} & = & \sum_{i=1}^{m-1} c_i. \end{array} \tag{4.5}$$

Exemple 4.1.1. Dans GF(2^5) , si $c = (c_0, c_1, c_2, c_3, c_4) = (0, 0, 1, 1, 0)$ alors $\lambda_0 = 0$, $\lambda_1 = c_1 = 0$, $\lambda_2 = c_1 + c_2 = 0 + 1 = 1$, $\lambda_3 = c_1 + c_2 + c_3 = 0 + 1 + 1 = 0$, $\lambda_4 = c_1 + c_2 + c_3 + c_4 = 0 + 1 + 1 + 0 = 0$. Une solution à $\lambda^2 + \lambda = c$ est donc $(\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4) = (0, 0, 1, 0, 0)$. Ce que nous pouvons vérifier puisque $(0, 0, 1, 0, 0)^2 + (0, 0, 1, 0, 0) = (0, 0, 0, 1, 0) + (0, 0, 1, 0, 0) = (0, 0, 1, 1, 0)$.

Une fois λ_P et $(\lambda_P + 1)$ trouvés (les deux solutions de l'équation 4.2), nous pouvons extraire la coordonnée x à partir de l'équation $v = x^2 + u \times (\lambda + 1)$. Nous avons donc deux solutions, qui nous conduiront à un élément x différent ; il faut choisir la bonne parmi λ_P et $\lambda_P + 1$. La fonction trace Tr nous permet de déterminer celle qui est correcte et ainsi, de trouver la coordonnée x convenable. En effet, la trace Tr de la coordonnée x de P doit être égale à celle de a [20]. Ce critère est suffisant pour sélectionner la bonne solution puisque qu'en extrayant x de cette façon :

$$x_0 = \sqrt{v + u \times (\lambda_P + 1)}$$

ou

$$x_1 = \sqrt{v + u \times (\lambda_P)},$$

nous pouvons prouver que les traces Tr de x_0 et x_1 seront différentes et donc des traces Tr de x_0 ou x_1, une seule sera égale à Tr(a). Enfin, à partir de $\lambda = x + y/x$ nous extrayons la coordonnée y en écrivant $\lambda \times x + x^2 = y$. L'un des intérêts du *halving* est que l'opération $Q/2$ est très peu couteuse vis à vis d'un doublement de point (que cela soit en coordonnées affines ou projectives) : une telle démarche ne coûte généralement qu'une résolution de l'équation quadratique $\lambda^2 + \lambda = c$ (peu onéreux que cela soit en surface ou en temps, nous y reviendrons), deux multiplications, une racine carrée (un simple décalage circulaire en base normale) et une poignée d'additions dans le corps GF(2^m). À titre de comparaison, un doublement de base en coordonnées affines requiert une inversion (l'équivalent de 10 à 100 multiplications sur le corps fini GF(2^m)), deux multiplications et six additions dans le corps GF(2^m). En coordonnées projectives, un doublement nécessite (au moins) sept multiplications, trois carrés et trois additions [21]. Autant dire que l'approche *halving* semble être une alternative très intéressante aux méthodes de *doubling*.

Pour rendre le *halving* encore plus avantageux, il est possible de travailler avec un nouveau type de coordonnée (u, λ_Q) avec $\lambda_Q = u + \frac{v}{u}$. Cela permet d'économiser une multiplication au niveau du corps GF(2^m). L'algorithme est décrit en Algo. 15.

Algorithme 15 : Algorithme de *halving* [45].

Données : (u, λ_Q)
Résultat : (x, λ_P)
1 Trouver une solution de $\lambda^2 + \lambda = u + a$
2 $t \leftarrow u \times (u + \lambda_Q + \lambda)$
3 **si** $\mathrm{Tr}(t) = 0$ **alors**
4 $\quad \lambda_P \leftarrow \lambda\,;\, x \leftarrow \sqrt{t + u}$
5 **sinon**
6 $\quad \lambda_P \leftarrow \lambda + 1\,;\, x \leftarrow \sqrt{t}$
7 **retourner** (x, λ_P)

Il faudra, cependant, à chaque addition de points $P + Q$, reconvertir (u, λ_Q) en coordonnées affines standards (x, y).

4.2 Opérateur de résolution de $\lambda^2 + \lambda = c$

Nous avons vu dans la section précédente 4.1 une façon de calculer une solution à l'équation $\lambda^2 + \lambda = c$. C'est sur cette méthode, qui consiste à sommer séquentiellement les bits de c_i que nous nous sommes basés. Il existe une autre approche [20] se fondant

sur la *half-trace* HT définie comme :

$$\mathrm{HT}(c) = \sum_{i=0}^{(m-1)/2} c^{2^{(2i)}}. \tag{4.6}$$

La valeur HT(c) est en fait une solution de l'équation $\lambda^2 + \lambda = c$ dans le cas où m est impair. Nous avons effectivement :

$$\begin{aligned}
\mathrm{HT}(c)^2 + \mathrm{HT}(c) &= \left(\textstyle\sum_{i=0}^{(m-1)/2} c^{2^{(2i)}}\right)^2 + \sum_{i=0}^{(m-1)/2} c^{2^{(2i)}} \\
&= \textstyle\sum_{i=0}^{(m-1)/2} c^{2^{(2i+1)}} + \sum_{i=0}^{(m-1)/2} c^{2^{(2i)}} && \textit{par linéarité du carré} \\
&= \textstyle\sum_{i=0}^{(m-1)} c^{2^i} + c^{2^m} \\
&= \mathrm{Tr}(c) + c \\
&= c && \textit{du fait que } \mathrm{Tr}(c) = 0
\end{aligned}$$

La fonction HT est une fonction linéaire (par la linéarité du carré, ou autrement dit, par l'action du morphisme de Frobenius sur les éléments de GF(2^m) [61]), c'est à dire que si $c = \sum_{i=0}^{m-1} c_i \times \beta^{2^i}$ alors $\mathrm{HT}(c) = \sum_{i=0}^{m-1}(c_i) \times \mathrm{HT}(\beta^{2^i}) = \sum_{i=0}^{m-1} c_i{}^{*} \times \mathrm{HT}(\beta^{2^i})$. Il est possible de pré-calculer les HT(β^{2^i}) ; le calcul de la *half-trace* est résumé à sommer les HT(β^{2^i}) dont les c_i correspondants valent 1. C'est l'une des solutions suggérées dans [20], dans le cas d'une base polynomiale.

Dans le but d'implémenter un opérateur de résolution de l'équation quadratique $\lambda^2+\lambda = c$, pour notre *crypto-processeur*, nous avons donc opté pour la méthode évoquée dans la partie précédente et l'équation 4.5. Notre crypto-processeur travaillera avec des mots de w bits. L'objectif est maintenant d'écrire un algorithme manipulant des mots de w bits, réalisant un tel calcul. Dans l'algorithme 16, nous sommons les w premiers bits de c en utilisant la relation de récurrence

$$\lambda_{i+1} = \lambda_i + c_{i+1}.$$

Puis, nous sommons les w bits suivants, en ayant au préalablement mémorisé λ_{w-1} dans une variable temporaire t, en se basant, cette fois-ci, sur la relation

$$\lambda_{i+1} = \sum_{j=w}^{i} c_j + c_{i+1} + t,$$

pour i compris entre $w+1$ et $2 \times w - 1$. Nous employons cette stratégie pour les w bits suivants pour récupérer l'ensemble des λ_i (i compris entre 0 et $m-1$). La figure 4.1 est une projection matérielle de l'algorithme 16. La bascule fait office de variable t,

∗. HT(c_i) = c_i est m est supposé impair.

le registre à décalage se décale circulairement de w bits à chaque cycle d'horloge et permet de stocker les w bits $\lambda_i, \lambda_{i+1}, \ldots, \lambda_{i+w-1}$ de λ ainsi calculés. Nous ignorons, éventuellement, les derniers bits du registre si la taille de ce dernier est supérieure à la taille du corps m (car la taille du registre sera toujours un multiple de w). Dans les faits, l'architecture ici présentée n'est pas implémentée de cette façon au sein du circuit FPGA, l'algorithme de routage se charge de réduire le chemin critique et le nombre de portes utilisées en transformant la suite de portes `XOR` en arbre binaire. Ce problème qui consiste à faire des sommes partielles, comme nous le faisons ici, est connu sous le nom de somme à préfixe parallèle (*Prefix sum*) [62].

Algorithme 16 : Calcul d'une solution à l'équation quadratique $\lambda^2 + \lambda = c$ [20].

Données : $c = (c_0, c_1, \ldots, c_{m-1})$ dans GF(2^m) en base normale
Résultat : λ, solution de l'équation quadrique $\lambda^2 + \lambda = c$
Nous posons $T_w = \lceil m/w \rceil$.

$t \leftarrow 0$
pour i **de** 0 **à** $T_w - 1$ **faire**
  $\lambda_{i\times w} \leftarrow t + c_{i\times w}$
  pour j **de** 1 **à** w **faire**
    $\lambda_{i\times w+j} \leftarrow \lambda_{i\times w+j-1} + c_{i\times w+j}$
  $t \leftarrow \lambda_{i\times w+w-1}$
retourner λ

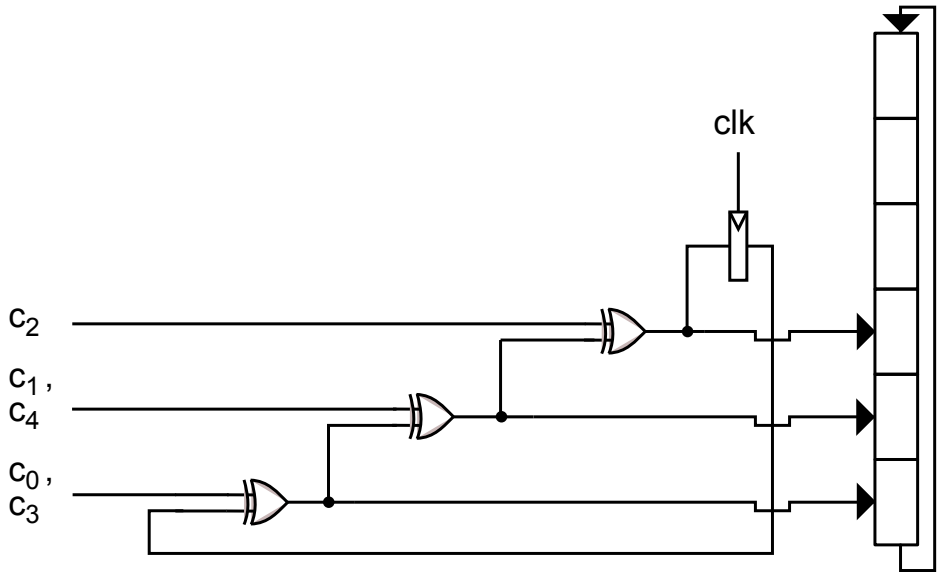


FIGURE 4.1: Architecture de l'unité de résolution de l'équation $\lambda^2+\lambda = c$ pour GF(2^5).

À noter que le circuit permet de récupérer la trace Tr d'un élément c puisque la trace $\mathrm{Tr}(c) = c_0+c_1+\ldots+c_{m-1}$ soit exactement la quantité $\lambda_{m-1}+c_0$. Le circuit est présenté dans la figure 4.1. Les entrées sont au premier cycle d'horloge (c_0, c_1, c_2) (le premier mot de 3 bits) puis (c_3, c_4, X) (pour le second). La valeur X signifie que cette dernière n'a pas d'importance pour conserver un bon fonctionnement du circuit.

4.3 Additionneur en base normale dans GF(2^m)

En base normale, l'addition de deux éléments du corps GF(2^m) se fait bit à bit. Par exemple, si $A = (a_0, a_1, \ldots, a_{m-1})$ et $B = (b_0, b_1, \ldots, b_{m-1})$ alors $C = A + B = (a_0 + b_0, a_1 + b_1, \ldots, a_{m-1} + b_{m-1}) = (c_0, c_1, \ldots, c_{m-1})$. Cela se fait très bien en implémentant en parallèle autant de portes `XOR` que nécessaire produisant les bits de $c_i = a_i + b_i$. Nous avons choisi de travailler avec des mots de w bits (par commodité et homogénéité mais aussi parce que cela réduira la complexité matérielle de nos unités). Nous envoyons à notre additionneur $T_w = \lceil m/w \rceil$ mots de w bits formant les éléments A et B. Au premier envoi de ces mots, nous calculons les bits $c_0, c_1, \ldots, c_{w-1}$ que nous stockons dans un registre à décalage (se décalant de w bits à chaque cycle d'horloge), puis calculons au second mot les bits $c_w, c_1, \ldots, c_{2w-1}$ et ainsi de suite. Nous ignorons éventuellement les derniers bits du registre si la taille du corps est inférieure à celle du registre. Nous noterons que la taille du registre sera toujours un multiple de w pour garantir le bon fonctionnement de l'architecture. L'architecture est décrite dans la figure 4.2. Nous envoyons d'abord (a_0, a_1, a_2) et (b_0, b_1, b_2) puis (a_3, a_4, X) et (b_3, b_3, X).

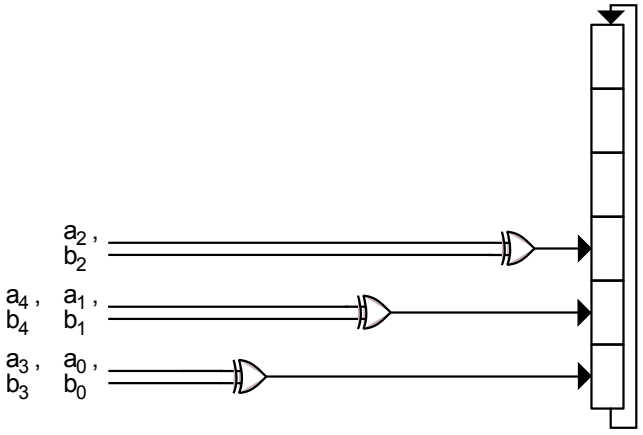


FIGURE 4.2: Architecture de l'unité d'addition pour GF(2^5).

4.4 Racine carrée

En base normale, comme nous l'avons vu dans la section 3.2.3, la racine est un décalage circulaire. Si $A = (a_0, a_1, \ldots, a_{m-1})$ alors $\sqrt{A} = (a_1, a_2, \ldots a_0)$. Dans chacune de nos unités de calculs, nous essayons d'être le plus uniforme possible : nous souhaitons suivre un même schéma de calcul. Nous chargeons les opérateurs, mot de w bits par mot de w bits, nous lançons le calcul et venons recueillir le résultat, là aussi, mot de w bits par mot de w bits. Nous avons donc créé l'opérateur décrit dans la figure 4.3. Nous venons remplir le registre à décalage, qui une fois plein, de par le routage des sorties, permet

de rendre le résultat escompté. Le circuit peut paraître complexe pour faire un simple décalage circulaire sur un opérande A, mais le cheminement nous a semblé plus pratique, de par homogénéité de notre architecture.

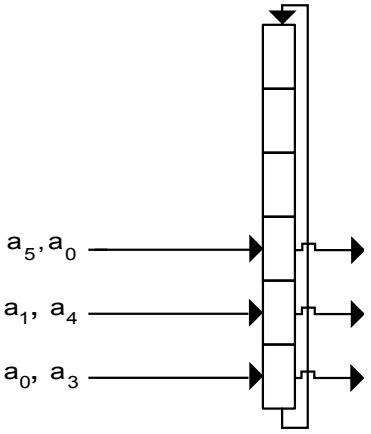


FIGURE 4.3: Architecture de l'unité de racine carrée pour GF(2^5).

4.5 Parallélisme *halve-and-add* et *double-and-add*

Il est possible de casser le côté séquentiel de l'algorithme d'Hörner pour le calcul de $[k]P$ en « séparant » la clef k en deux parties (pas forcément égales). L'une des façons les plus naïves est d'écrire $k = (2^n)\mathcal{K}_1 + \mathcal{K}_0$ où si k est représenté sur l bits, $\mathcal{K}_0$ l'est sur n bits et $\mathcal{K}_1$ sur $l - n$. Auquel cas, il est possible de lancer en parallèle les multiplication scalaires de $[\mathcal{K}_0]P$ et de $[\mathcal{K}_1]P$. Il restera, cependant, à calculer $[\mathcal{K}_1]P \times 2^n$ avant de l'additionner à $[\mathcal{K}_0]P$. Si nous estimons le coût temporel d'une addition $P+Q$ par α_0 cycles d'horloge et celui d'un doublement $P+P$ par α_1 cycles (avec $\alpha_0 > \alpha_1$), un tel cheminement coutera, en moyenne $\max((\alpha_0) \times \frac{n}{2}, (\alpha_0) \times \frac{(l-n)}{2}) + \max((\alpha_1) \times n, (\alpha_1) \times (l-n)) + \alpha_1 \times n$.

	$[\mathcal{K}_0]P$	$[\mathcal{K}_1]P$
Additions $P + Q$ (en parallèle)	$(\alpha_0) \times \frac{n}{2}$	$(\alpha_0) \times \frac{(l-n)}{2}$
Doublements $P + P$ (en parallèle)	$(\alpha_1) \times n$	$(\alpha_1) \times (l - n)$
Partie séquentielle $[2^n]([\mathcal{K}_1]P)$	$\alpha_1 \times n$	–

Il y a aussi l'addition finale, que nous négligerons ici. Si nous supposons avoir coupé k en deux parties égales, nous avons donc le coût temporel $\alpha_0 \times \frac{l}{4} + \alpha_1 \times \frac{l}{2} + \alpha_1 \times \frac{l}{2} = \alpha_0 \times \frac{l}{4} + \alpha_1 \times l = l \times (\frac{\alpha_0}{4} + \alpha_1)$. Nous voyons que nous ne parvenons pas à nous défaire des l doublements $P + P$. Le coût temporel d'un $[k]P$ non-parallèle est de $l \times (\frac{\alpha_0}{2} + \alpha_1)$. Techniquement, sur FPGA, il faudrait deux portions du circuit dédiées à la multiplication scalaire $[k]P$ pour lancer les calculs de $[\mathcal{K}_0]P$ et $[\mathcal{K}_1]P$ simultanément. Cela veut dire

doubler la surface (face à une version entièrement séquentielle) pour ne gagner *que* $\frac{\alpha_0}{4}$ cycles d'horloge, peu face à l'ensemble des opérations. Une telle stratégie ne nous semble pas viable, du moins sur un circuit reconfigurable.
L'approche avait été, cela dit, proposée dans [63] dans lequel les auteurs remarquent que le calcul parallèle ($[\mathcal{K}_0]P$ et $[\mathcal{K}_1]P$) peut être une façon de protéger le circuit contre les attaques *SPA*. Les auteurs n'étayent pas davantage ce propos. La partie séquentielle (qui consiste à calculer $2^n[\mathcal{K}_1]P + [\mathcal{K}_0]P$) n'a aucun lien avec le contenu de la clef, et si de l'information fuit pendant cette séquence, elle ne sera, normalement, pas révélatrice de secret.
Dans la même philosophie, nous proposons d'exploiter un schéma similaire en réécrivant k de la façon suivante :

$$\begin{aligned} k &= (2^n 2^{-n})((2^n)\mathcal{K}_1 + \mathcal{K}_0) \\ &= (2^n)(\mathcal{K}_1 + 2^{-n}\mathcal{K}_0). \end{aligned}$$

Si nous avons connaissance du point P avant la multiplication scalaire, il est alors possible de pré-calculer $\tilde{P} = 2^n P$. Auquel cas, nous pouvons calculer $[k]P$ comme

$$\begin{aligned} kP &= (2^n)(\mathcal{K}_1 + 2^{-n}\mathcal{K}_0)P \\ &= (\mathcal{K}_1 + 2^{-n}\mathcal{K}_0)\tilde{P} \\ &= (\mathcal{K}_1\tilde{P}) + (2^{-n})(\mathcal{K}_0\tilde{P}). \end{aligned}$$

Il est envisageable de calculer $(\mathcal{K}_1\tilde{P})$ et $(\mathcal{K}_0\tilde{P})$ en parallèle et de venir, ensuite calculer $2^{-n}(\mathcal{K}_0\tilde{P})$. Nous pouvons aussi recoder k comme somme de puissances négatives 2^{-i} auquel cas, il est possible d'utiliser une stratégie de calculs parallèles sans avoir, au préalable, à calculer $\tilde{P}$, nous y reviendrons. L'intérêt réside ici dans le $\ll 2^{-n} \gg$ qui nous permettra d'utiliser un algorithme de type *halving* pour calculer $2^{-n}(\mathcal{K}_0\tilde{P})$. L'opération de *halving* est bien plus rapide qu'une opération de *doubling*. Il est tout à fait raisonnable de calculer, aussi, $(\mathcal{K}_0\tilde{P})$ et $(\mathcal{K}_1\tilde{P})$ de par un algorithme *halve-and-add*. Nous ferons référence à cette approche par l'appellation ***parallel halve-and-add*** .

Une solution suggérée par J.Taverne et al. [64] (et reprise et améliorée par Christophe Negre et Jean-Marc Robert dans [46]) est de, toujours, couper k en deux parties mais, cette fois-ci, de lancer sur chacune d'entre elles un algorithme de multiplication scalaire différent, pour ainsi lever les dépendances intrinsèques au schéma d'Hörner. Cela est rendu possible en calculant d'une part avec un algorithme de type *double-and-add* et de l'autre avec un algorithme de type *halve-and-add*.

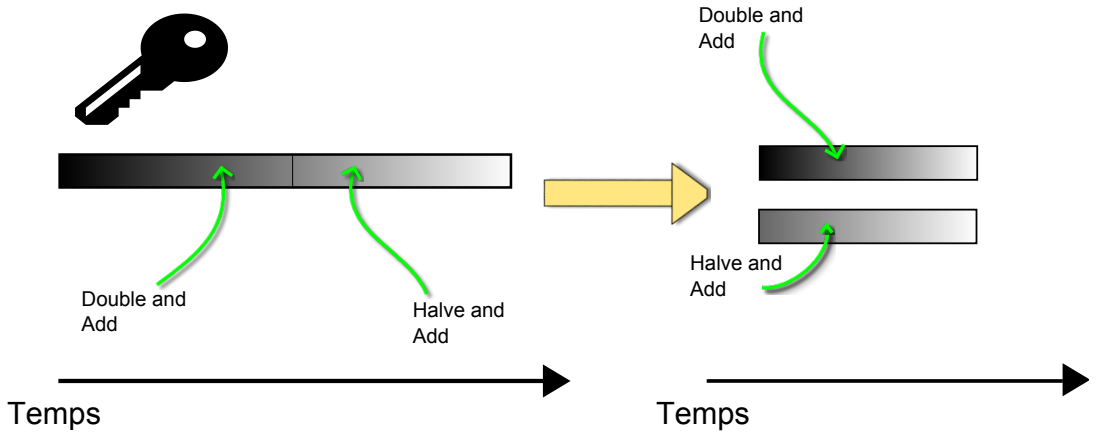


FIGURE 4.4: Parallélisme de la multiplication scalaire $[k]P$.

Pour cela, nous aurons besoin de « recoder » le scalaire k de la façon suivante :

$$k' = k \times 2^l \bmod N = \sum_{i=0}^{l-1} \underline{k_i} \times 2^i$$

où N est l'ordre de P dans $\mathbb{E}(\mathrm{GF}(2^m))$ et l le nombre de bits nécessaires pour représenter N. Nous pouvons ainsi écrire k comme

$$k = 2^{-l} \times k' \bmod N = 2^{-l} \times \sum_{i=0}^{l-1} \underline{k_i} \times 2^i = \sum_{i=0}^{l-1} \underline{k_i} \times 2^{i-l}.$$

La représentation $(\underline{k_0}, \underline{k_1}, \ldots, \underline{k_{N-1}})$ est en fait une réécriture de k en base $2^{-1} \bmod N$. Nous avons, dès lors, la possibilité de lancer un algorithme de type *halve-and-add* sur cette nouvelle représentation de la clef k, comme mentionné dans l'algorithme 17.

Algorithme 17 : Algorithme *halve-and-add* pour la multiplication scalaire ECC [45].

Données : P un point de la courbe $\mathbb{E}$, $\underline{k} = (\underline{k_0}, \underline{k_1}, \ldots, \underline{k_{m-1}})$ un entier, avec $\underline{k} = k \times 2^{-l} \bmod N$ où N est l'ordre du point P et l le nombre de bits minimal nécessaire pour stocker N

Résultat : $[\underline{k}]P$

```
Q ← O
pour i de 0 à m − 1 faire
    Q ← Q/2
    si k_{m−1−i} = 1 alors
        Q ← Q + P
retourner Q
```

Il est envisageable, maintenant, d'utiliser un autre facteur multiplicatif que 2^{-l} comme nous l'avons juste évoqué ; il est, en soit, tout à fait raisonnable de choisir un facteur 2^{-n} avec $n < l$. Nous obtiendrions alors une écriture de k dont une partie s'exprimera

avec des puissances de 2 négatives (base 2^{-1}) et l'autre avec des puissances positives de 2 (base 2) :

$$k' = k \times 2^n \bmod N = \sum_{i=0}^{l-1} \underline{k_i} \times 2^i,$$

pour finalement obtenir,

$$k = 2^{-n} \times k' = 2^{-n} \times \sum_{i=0}^{l-1} \underline{k_i} \times 2^i = \sum_{i=0}^{l-1} \underline{k_i} \times 2^{i-n}.$$

Nous reformulons l'égalité précédente en disjoignant les puissances positives de celles négatives :

$$k = \underbrace{\sum_{i=0}^{l-1} \underline{k_i} \times 2^{i-n}}_{\text{puissances négatives } \mathcal{K}_0} + \underbrace{\sum_{i=n}^{l-1} \underline{k_i} \times 2^{i-n}}_{\text{puissances positives } \mathcal{K}_1}.$$

Il est dorénavant possible de calculer $[\mathcal{K}_0]P$ via un algorithme de type *halve-and-add* et de calculer $[\mathcal{K}_1]P$ via un algorithme de type *double-and-add*, le tout, en parallèle, comme illustré dans la figure 4.4. Il restera une addition finale que nous pouvons considérer comme négligeable face à la totalité des calculs jusqu'alors effectués.

4.6 Architecture proposée

Nous présenterons dans cette partie l'architecture de notre crypto-processeur. Notre crypto-processeur est doté de deux cœurs d'exécution identiques pouvant partager des données par le biais d'une mémoire partagée (voir figure 4.6). Deux fils les relient l'un l'autre directement afin qu'ils puissent se synchroniser ; une façon, pour un cœur, de dire à l'autre « j'ai terminé mon calcul et ai envoyé le résultat sur le banc de mémoire partagée, tu peux aller y récupérer des données ». Chaque cœur contient en son sein quatre unités de calculs : une unité de multiplication-inversion (MIU, voir chapitre 3), une unité d'addition, une unité de résolution quadratique $\lambda^2 + \lambda = c$ (que nous venons de décrire en 4.2, notée *Halving* sur le schéma 4.5) et enfin, une unité de racine carrée (notée *sqrt*) qui est, en fait, en base normale, un décaleur-circulaire de un bit. Chaque cœur possède également un registre qui lui est propre (ce sont des blocs *ram* en double-port), lui permettant de stocker les données dont il a besoin pour assurer le bon déroulement des algorithmes cryptographiques.

Notre crypto-processeur est programmable. Chaque cœur lancera le programme contenu dans la mémoire du micro-code (dans les faits, un bloc *ram* de 32×256 bits). Le processeur possède son propre assembleur, inspiré des assembleurs les plus classiques. Il est

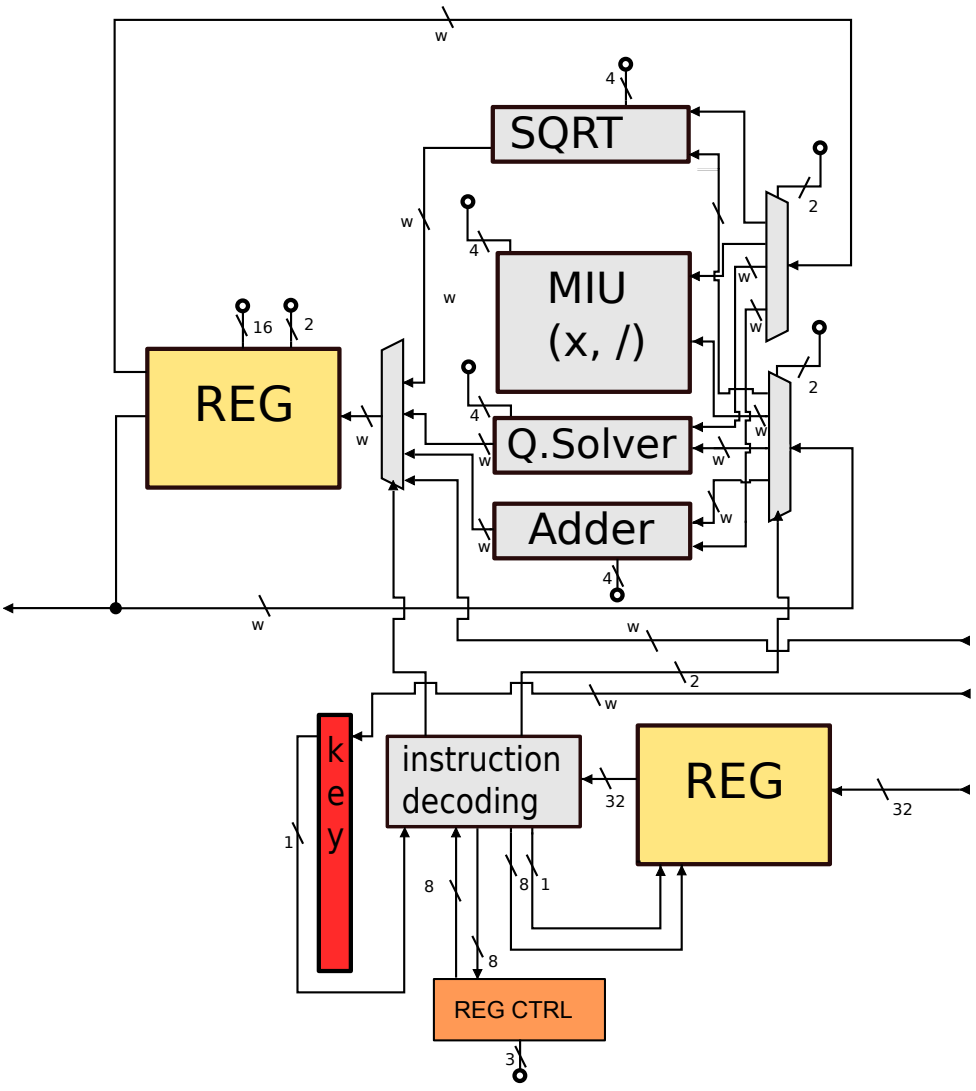


FIGURE 4.5: Architecture d'un cœur de notre processeur.

.

possible de définir des étiquettes (*label*), d'y sauter ou non selon un test conditionnel (pouvant dépendre de la clef courante, de la valeur de la trace du dernier élément dont on a calculé λ tel que $\lambda^2 + \lambda = c$ et de la valeur de 8 registres 8 bits de contrôle). Les instructions du crypto-processeur utilisent un format 32 bits.

31	30	29	28	27	26	25	24	...	17	16	...	9	8	...	1	0
Code Opération				*unit*			*A*			*B*			*C*			*Flag*	

Nous listons, ci-dessous, le jeu d'instructions de notre assembleur. Il est suffisamment étoffé pour gérer la majorité des algorithmes de multiplications scalaires. Nous sommes cependant limités par un recodage binaire de la clef.

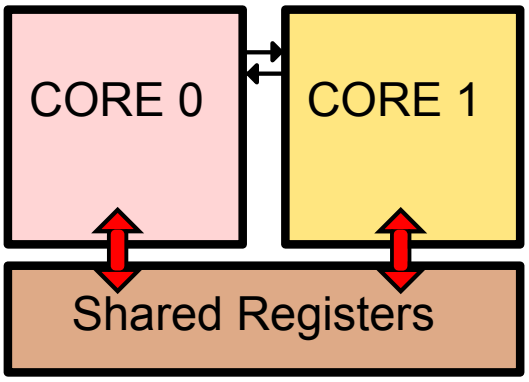


FIGURE 4.6: Les deux cœurs communiquent par le biais d'une mémoire partagée.

Les données sont stockées sous la forme de mots de w bits. Ainsi, chaque élément du corps fini GF(2^m) nécessite $T_w = \lceil m/w \rceil$ mots. L'adressage est gérée de façon automatique par notre « compilateur », l'utilisateur ne manipulera pas des adresses physiques (il serait très enclin à faire des erreurs et à venir superposer des données) mais des noms de variables.

```
1  def Variable_Name
2  SET_REG REG 8_BITS_INT X X X
3  LOAD_OP UNIT 8_BITS_ADDR 8_BITS_ADDR X X
4  RUN UNIT X X X F
5  SEND_R UNIT 8_BITS_ADDR X X X
6  GOTO X 8_BITS_ADDR X X X
7  IF_KEY_JUMP X 8_BITS_ADDR 8_BITS_ADDR X X
8  REG_JUMP REG 8_BITS_INT 8_BITS_ADDR X X
9  REG_ADD REG 8_BITS_INT X X X
10 GET_NEXT_KEY_BIT X X X X X
11 OVER X X X X X
12 TRACE_JUMP X 8_BITS_ADDR 8_BITS_ADDR X X
13 SYNC X X X X X
```

– **SET_REG REG A X X X** : Cette commande permet d'initialiser l'un des huit registres 8 bits de contrôle (REG0, REG1, REG2, REG3, REG4, REG5, REG6, REG7) avec la valeur précisée dans A.

– **LOAD_OP UNIT A B X X** : Cette commande permet de charger le contenu des opérandes stockées aux adresses A et B dans l'unité UNIT. Seront envoyés à l'unité UNIT les mots de w bits stockés aux adresses $A, A+1, \ldots, A+T_w-1$ ainsi que les mots situés en $B, B+1, \ldots, B+T_w-1$ représentant deux éléments du corps fini GF(2^m). L'assembleur permet de définir des pointeurs et de manipuler des noms plutôt que des

nombres. L'assembleur permet de cette façon d'abstraire la notion d'élément : chacun d'eux est référencé par un nom, ce qui permet en outre d'éviter que deux éléments se chevauchent créant collatéralement des conflits en mémoire.

Exemple : Si $m = 7$ et $w = 3$ alors $T_w = 3$. En définissant à l'aide de la commande **def** deux éléments ElmtA et ElmtB, l'assembleur leur aura attribué, par exemple, respectivement les adresses 18 et 21. Les deux codes suivants sont équivalents :

```
1    def ElmtA
2    def ElmtB
3    LOAD_OP ADD ElmtA ElmtB X X
```

```
1    LOAD_OP ADD 18 21 X X
```

- **RUN UNIT X X X F** : Cette commande permet de lancer l'unité UNIT. L'unité UNIT utilisera les opérandes auparavant chargés via la commande LOAD_OP. L'instruction est bloquante, le cœur sur lequel cette commande est exécutée ne lira la prochaine instruction que lorsque l'unité UNIT aura terminé son calcul. Le flag F (0 ou 1 sur 1 bit) précise, dans le cas de l'unité MIU (**MULT**), le mode qui sera employé. Si $F = 0$ alors il s'agira d'une multiplication classique tandis qu'un flag levé $F = 1$ lancera une inversion de l'opérande A.
- **SEND_R UNIT A X X X** : Cette commande envoie à l'adresse A du registre de données le résultat précédemment calculé par l'instruction RUN.
- **GOTO X A X X X** : Cette commande permet de sauter directement à la ligne A du programme contenu dans la mémoire du micro-code.
- **IF_KEY_JUMP X A B X X** : Cette commande permet de sauter de façon conditionnelle aux lignes A ou B. Si le bit courant de la clef est égal à 1 alors la suite de l'exécution reprendra de la ligne A sinon (bit courant égal à 0) à la ligne B.
- **REG_JUMP REG A B C X** : Cette commande compare la valeur contenue dans le registre REG avec la valeur A. Si les deux sont égales alors le programme continuera de la ligne B, sinon, si ces deux valeurs diffèrent, le programme reprendra de la ligne C.
- **REG_ADD REG A X X X** : Cette commande ajoute à la valeur contenue dans le registre REG la valeur A. Il s'agit d'un calcul non-signé. Les calculs sont effectués modulo $256 = 2^8$, les dépassements de capacité (*overflows*) ne sont ici pas gérés.
- **GET_NEXT_KEY_BIT X X X X X** : Cette commande permet de récupérer le prochain bit de la clef, qui deviendra alors le bit de clef courante. La clef est stockée du poids faible au poids fort ; ainsi si $k = k_0 + k_1 \times 2 + k_2 \times 2^2 + \ldots + k_{m-1} \times 2^{m-1}$. Le bit initialement « courant » est le bit k_0. En appliquant une fois la fonction GET_NEXT_KEY_BIT, le bit « courant » deviendra le bit k_1 et ainsi de suite.

– **OVER X X X X X** : Cette commande permet de terminer le programme. Le cœur restera dès lors en mode IDLE. Cette commande envoie également un signal de synchronisation au deuxième cœur de calcul.
– **TRACE_JUMP X A B X X** : Cette commande permet de sauter de façon conditionnelle aux lignes A ou B. Si la trace Tr du dernier élément c dont l'utilisateur a calculé la solution $\lambda^2 + \lambda = c$ est égale à 1 alors la suite de l'exécution reprendra de la ligne A sinon à la ligne B.
– **SYNC X X X X X** : Cette instruction est bloquante et attend que l'autre cœur ait terminé l'exécution de son programme avant de poursuivre.

4.7 Performances

Le but principal de notre démarche est d'évaluer l'apport relatif du parallélisme aussi bien pour le gain en vitesse qu'il peut amener que pour la sécurité apparente que semble fournir la superposition temporelle de deux calculs indépendants.

Nous pouvons citer une implémentation matérielle basée sur du *halving* qu'est [65]. Dans ce papier, les auteurs proposent d'utiliser le parallélisme naturellement présent au sein des formules d'addition de points et de « division par deux » de points. Leur architecture est pipelinée de manière à rendre ces opérations très rapides : l'opération la plus couteuse en temps n'occupera que 8 cycles d'horloge. Ces solutions occupent une grande surface silicium et ont été conçues pour la performance pure. Nous ne nous comparerons pas à cette implémentation car ce que nous souhaitons vraiment mesurer est l'apport relatif du parallélisme et non la performance brute. D'une manière générale, le calcul d'un *halving* (trouver P tel que $2 \times P = Q$) est bien plus rapide que celui d'un *doubling* (pour P, établir la valeur de $2 \times P$). Nous utilisons, en effet, ici, un système de coordonnées affines : un *doubling* requiert une inversion au niveau du corps fini et trois multiplications. Il faut spécifiquement dix multiplications pour calculer A^{-1} dans GF(2^{233}). Le *halving*, quant à lui, ne nécessite que deux multiplications (en coordonnées affines, toujours), le calcul de la solution quadratique $\lambda^2 + \lambda = c$ (qui est réalisé en $\lceil m/w \rceil$ cycles, très peu face aux m cycles de la multiplication ou aux $10 \times m$ -environ- cycles de l'inversion), le calcul d'une racine carrée (un décalage circulaire) et une poignée d'additions (très rapides dans GF(2^m)). Étant donnée la disparité des opérations de *doubling* et de *halving*, couper la clef en deux parties égales ne sera évidemment pas l'optimal en terme de rapidité d'exécution. Nous devons maintenant trouver « où » couper la clef (dans l'hypothèse ou la clef n'a pas été recodée et qu'il y a donc, en moyenne, autant de 0 que de 1 dans l'écriture binaire de k). Si nous négligeons toutes les opérations au niveau du corps

à l'exceptions faites des multiplications et des inversions, nous avons donc les coûts suivants :

halving	$2 \times \mathtt{M}$
doubling	$\mathtt{I} + 3 \times \mathtt{M}$
add	$\mathtt{I} + 3 \times \mathtt{M}$

où `M` est le coût d'une multiplication et `I` celui d'une inversion. Pour un recodage binaire de la clef (ce qui sera notre cas ici), il y a, en moyenne, autant de 0 que de 1. Si l représente le nombre de bits de la clef dont nous nous servons pour calculer un *halve-and-add* (et donc $m-l$ bits pour la portion *double-and-add*), nous obtiendrons les coûts temporels (en nombre de cycles d'horloges) suivants :

$$H(l) = \frac{l}{2} \times add + l \times halving,$$

pour la partie *halve-and-add* et

$$D(l) = \frac{m-l}{2} \times add + (m-l) \times doubling,$$

pour la partie *double-and-add* . Si les deux calculs sont lancés en parallèle, le coût global de l'opération sera

$$\mathcal{C}(l) = \max(H(l), D(l)) + \mathcal{E}$$

où $\mathcal{E}$ est un coût, là encore négligeable, de l'opération qui consiste à additionner les deux points respectivement obtenus par les deux algorithmes de multiplications scalaires. La présence de max provient du fait que le temps d'exécution est plafonné par l'algorithme le plus lent. Nous obtenons les courbes de $\mathcal{C}(l)$ représentées dans la figure 4.7.

Nous observons, dans la figure 4.7, un optimum autour de $0.7 \times m$, pour une clef dont le poids de Hamming est de $m/2$. D'après les mesures réelles que nous avons obtenues sur *ModelSim*, nous approchons un optimal voisin de cette valeur théorique (voir figure 4.8).

Nous reportons dans le tableau 4.1 les valeurs (*slices*, LUTs, *flip-flops* (bascules)) tirées du logiciel *ISE* après synthèse, placement et routage. Nous avons aussi incorporé dans ces chiffres une version mono-core de notre processeur (implémentation matérielle d'un seul des deux cœurs) afin de montrer l'apport de la version parallèle de notre multiplication

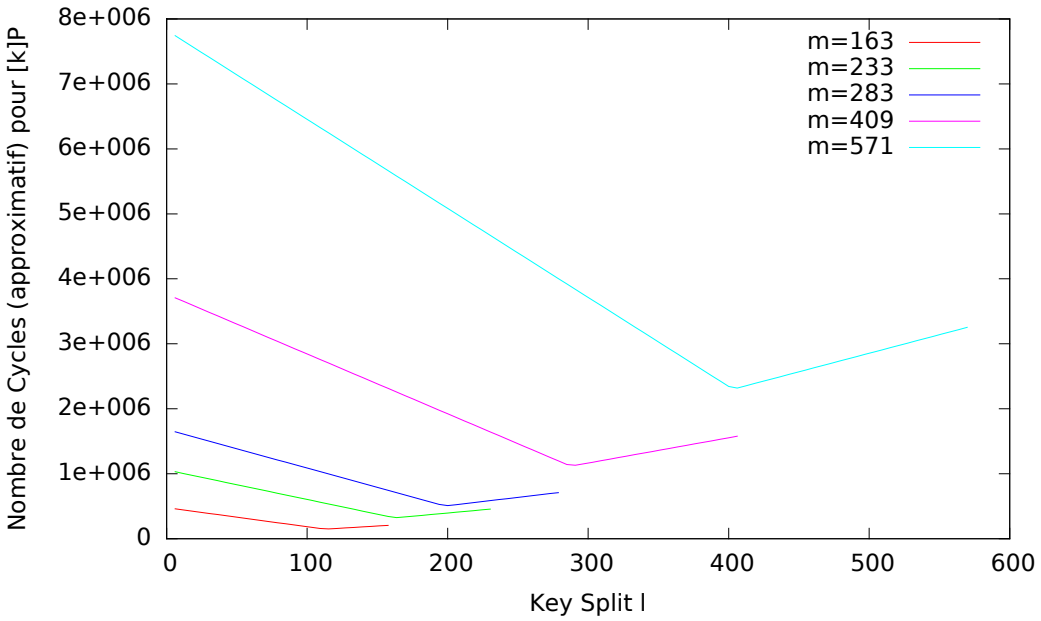


FIGURE 4.7: Nombres de cycles (approximatifs et théoriques) nécessaires pour le calcul d'un $[k]P$ selon l'endroit où la clef est coupée.

	m	Slices	LUTs	FFs	Frq (MHz)	PTS ($\times 10^{-9}$)
Dual-Core	163	1763	5365	4892	270	2345
	233	2216	7035	5972	264	4731
	283	2662	8339	6593	259	7061
	409	3464	10794	8637	263	11309
	571	5052	15669	11210	218	27420
Single-Core	163	791	2659	2467	278	1311
	233	1154	3253	3004	264	3178
	283	1366	3897	3335	250	3754
	409	1453	5069	4335	271	5140
	571	2571	8340	5627	229	12789

TABLE 4.1: Surface et fréquence de notre crypto-processeur.

scalaire $[k]P$. Dans la version *Dual-Core*, nous prenons le paramètre l optimal ($\approx 0.7m$) pour réaliser la multiplication scalaire. Dans la version *Mono-Core*, nous lançons une multiplication scalaire à l'aide de l'algorithme *halve-and-add* (plus efficace que le *double-and-add* standard) sur l'entièreté de la clef k. Le terme **PTS** est le sigle pour « produit temps surface », c'est à dire le produit entre le temps requis pour le calcul d'un $[k]P$ et la surface (ici en *slices*) occupée par notre crypto-processeur. Plus le **PTS** est petit, plus la solution est avantageuse. Nous voyons que la version *Dual-Core* a un produit temps-surface moins intéressant que la version *Mono-core* (nous obtenons même une version parallèle plus lente qu'une version strictement séquentielle pour GF(2^{571})). Cela est largement explicable par l'hyper-compétitivité du *halve-and-add* face au *double-and-add* .

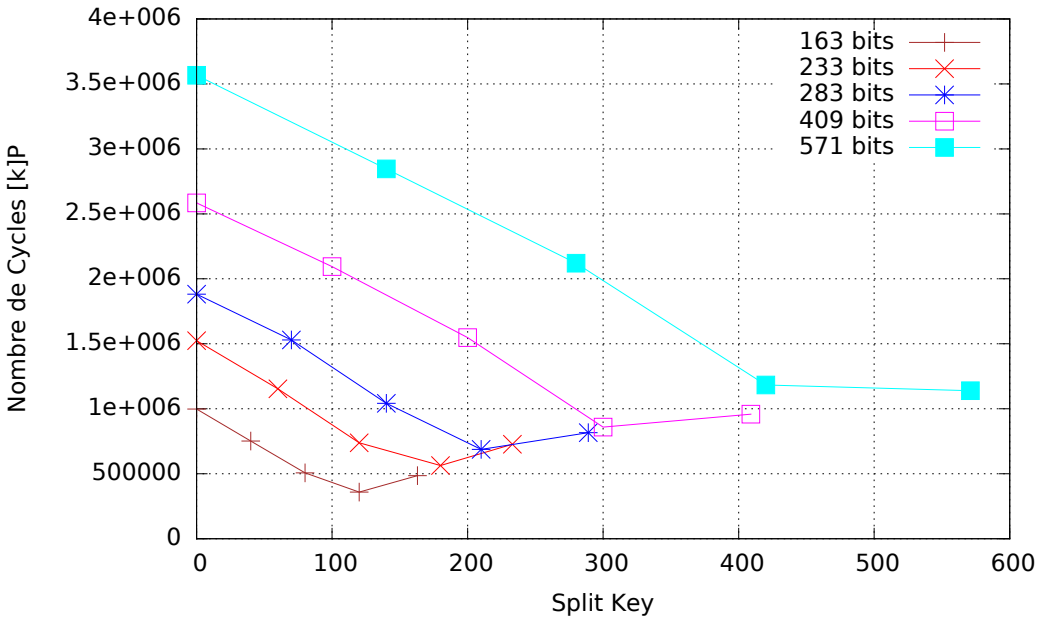


FIGURE 4.8: Nombres de cycles mesurés pour le calcul d'un $[k]P$ selon l'endroit où la clef est coupée.

m	Nombre de Cycles pour un [k]P	PTS
163	318987	2082
233	474733	3984
283	593433	6099
409	887709	11692
571	1349080	31264

TABLE 4.2: Performance de l'algorithme *parallel halve-and-add*.

Malgré un apport pas forcément intéressant du calcul parallèle (le PTS est bien supérieur au PTS d'une version mono-cœur), celui-ci apporte en plus de sa vitesse une forme de protection contre les attaques SPAs, comme cela a été supposé dans [63]. Évidemment, l'architecture que nous avons proposée permet de lancer deux multiplications scalaires en même temps. Dépendant du contexte et des besoins, notre processeur peut se programmer différemment.

Nous avons aussi codé, à l'aide de notre assembleur, l'algorithme du ***parallel halve-and-add*** dont nous faisions référence dans la section 4.5. Les résultats sont rassemblés dans le tableau 4.2. Nous y apercevons que cette approche est plus attrayante que l'approche parallèle précédente pour les corps GF(2^{163}), GF(2^{233}), GF(2^{283}) de par un PTS plus bas que celui d'un *double-and-add* et *halve-and-add* parallèle.

4.8 Évaluation de la sécurité physique de notre crypto-processeur.

Les attaques dites par *templates* [10] permettent de retrouver des bits secrets (typiquement, les bits de clefs privées) en observant le comportement de l'appareil sur lequel le calcul est effectué. Il est nécessaire avant toutes choses de créer *un profil du dispositif*, à partir de traces de consommation, de traces de rayonnement électromagnétique etc. Nous allons, pour ainsi dire, venir apprendre le comportement de l'appareil par un modèle probabiliste dont nous expliciterons le fonctionnement dans cette section. Les attaques par *templates* sont génériques, dans le sens où il n'est pas indispensable, au préalable, de comprendre ou de connaitre le fonctionnement de la cible. Nous travaillons en ≪ boite noire ≫. Le but de ce chapitre est, aussi, d'évaluer la sécurité de la solution parallèle que nous venons de proposer dans la partie précédente 4.6. Nous pensions, peut-être naïvement, que le parallélisme des calculs, à lui-seul, pouvait être une façon de nous prémunir des attaques les plus évoluées. Peut-être avons-nous été trop optimistes ? Dans tous les cas, nous montrerons ici que les attaques *templates* permettent de venir récupérer de manière quasi-systématique des bits de clefs. Nous suggérons également une méthode pour tenter de réduire *l'efficacité* de ces attaques en essayant ≪ d'homogénéiser ≫ les fuites de consommation, en ne rendant pas un calcul plus caractéristique qu'un autre. Enfin, il nous a été fait la remarque qu'il était peut-être possible de venir attaquer directement le circuit en *SPA*. En fait, nous pensons effectivement qu'il est possible d'extraire les deux courbes de consommation (celle en *double-and-add* et celle en *halve-and-add*) à partir d'un algorithme de traitement du signal dans la version parallèle non protégée ; nous pensons, en revanche, qu'il sera plus difficile d'y parvenir sur celle protégée, sans pouvoir, toutefois, quantifier ni vraiment analyser ces hypothèses.

4.8.1 La génération des *templates*

Afin de ≪ modéliser ≫ le comportement de l'appareil sur lequel nous travaillons, il est, bien sûr, nécessaire d'avoir accès au dispositif pour en extraire des données physiques (comme la consommation d'énergie, ou le rayonnement électromagnétique) mais aussi, pour pouvoir lancer, à volonté, l'ensemble des calculs requis pour son profilage. Ce sont là des *hypothèses fortes*.

Une trace T_j est un vecteur de nombres réels décrivant une quantité physique (disons la consommation d'énergie P) en fonction du temps. Il s'agit d'un sous-échantillonnage de la fonction $P(t)$, la puissance dissipée par l'appareil en fonction du temps. Nous pourrions écrire T_j comme un vecteur $(t_0^{(j)}, t_1^{(j)}, t_2^{(j)}, \ldots, t_{v-1}^{(j)})$ où chaque $t_h^{(j)}$ est un nombre réel

représentant la puissance dissipée à l'instant $h \times \Delta$ où Δ est le pas d'échantillonnage exprimé en *ps*. Nous nous servirons de ces traces pour créer un *template*.

Notre but est de créer un *template* (profil), de caractériser le comportement du circuit pour chaque hypothèse de clef. Il parait aberrant, à première vue, de créer autant de profils que de clefs possibles ; l'idée reste de travailler sur un nombre restreint de bits (sous-clefs). Nous pouvons, par exemple, créer 8 profils correspondant aux trois premiers bits ($2^3 = 8$) de la clef $\mathcal{K}_1$. Si notre attaque permet véritablement de retrouver ces trois bits, il est possible de réitérer l'attaque sur les trois bits suivants de $\mathcal{K}_1$. Ainsi, il n'est pas nécessaire de créer 2^l *templates* différents (où l est la taille de la clef $\mathcal{K}_1$ en bits), mais $8 \times \lceil l/3 \rceil$ afin d'identifier tous les bits secrets de $\mathcal{K}_1$. Nous considérerons, quoi qu'il en soit, une attaque comme réussie si nous parvenons à extraire correctement (et avec une quasi-certitude) un bit de la clef $\mathcal{K}_1$.

Un *template* peut être vu comme une signature physique et caractéristique de l'appareil cryptographique. Nous extrayons pour chaque hypothèse sur des trois premiers bits de la clef $\mathcal{K}_1$ $(000, 001, 010, 011, 100, 101, 110, 111)$ un nombre u de traces. Nous lançons u calculs $[k]P$ sur notre crypto-processeur (P variant d'une itération à l'autre) avec les trois premiers bits de $\mathcal{K}_1$ valant 000 et récupérons les traces correspondantes puis, nous lançons u calculs de $[k]P$ sur notre crypto-processeur avec les trois premiers bits de $\mathcal{K}_1$ valant 001 et récupérons les traces correspondantes, etc. Nous avons donc pour chaque hypothèse de clef un ensemble de traces que nous noterons $E^{000}, E^{001}, \ldots, E^{111}$ où $E^{xyz} = \{T_0^{xyz}, T_1^{xyz}, \ldots, T_{u-1}^{xyz}\}$ $(x, y, z \in \{0, 1\})$. Cela signifie par exemple, que nous avons pour l'hypothèse de clef 000 un ensemble E^{000} qui regroupe u traces distinctes $(T_0^{000}, T_1^{000}, \ldots, T_{u-1}^{000})$. En ayant ces données, il est possible de déterminer la trace moyenne M_{xyz} (dans chacune des hypothèses de clef de 000 à 111) du dispositif en écrivant

$$M_{xyz} = \frac{1}{u} \times \sum_{i=0}^{u-1} T_i^{xyz}$$

où l'addition de trace T_i^{xyz} et T_j^{xyz} est une addition vectorielle, c'est à dire une addition qui consiste à sommer les composantes indépendamment les unes des autres. Une fois cette quantité calculée, nous définissons la matrice de corrélations C_{xyz} définie comme :

$$C_{xyz} = \frac{1}{u-1} \sum_{i=0}^{u-1} (T_i^{xyz} - M_{xyz})^{(t)} (T_i^{xyz} - M_{xyz}).$$

Un *template* est la donnée des valeurs C_{xyz} et M_{xyz}. Plus il y aura de traces, plus le *template* sera convenable et cohérent. De plus, avoir un nombre important de traces

permet d'avoir une idée plus précise du bruit qui entoure chacune des mesures. Dans nos expérimentations, nous avons accès à un modèle parfait. La « consommation » mesurée est faite via la *switching activities*, soit l'activité électrique créée au sein du circuit quand un signal passe de 0 à 1 ou inversement. Il y a effectivement une corrélation entre la puissance dynamique de l'appareil et cette quantité [66]. En technologie *CMOS*, la puissance dynamique P_{dyn} est égale à

$$P_{\text{dyn}} = \tau \times C_l \times V_{\text{dd}}^2 \times f,$$

où V_{dd} est la tension d'alimentation, f est la fréquence de fonctionnement du circuit, C_l est la charge et τ l'activité du circuit. Il y a un lien direct entre les communications des signaux internes du circuit et la puissance dynamique P_{dyn}. L'activité interne τ est en particulier attachée aux variations des poids de Hamming des différents registres entre deux cycles d'horloge.

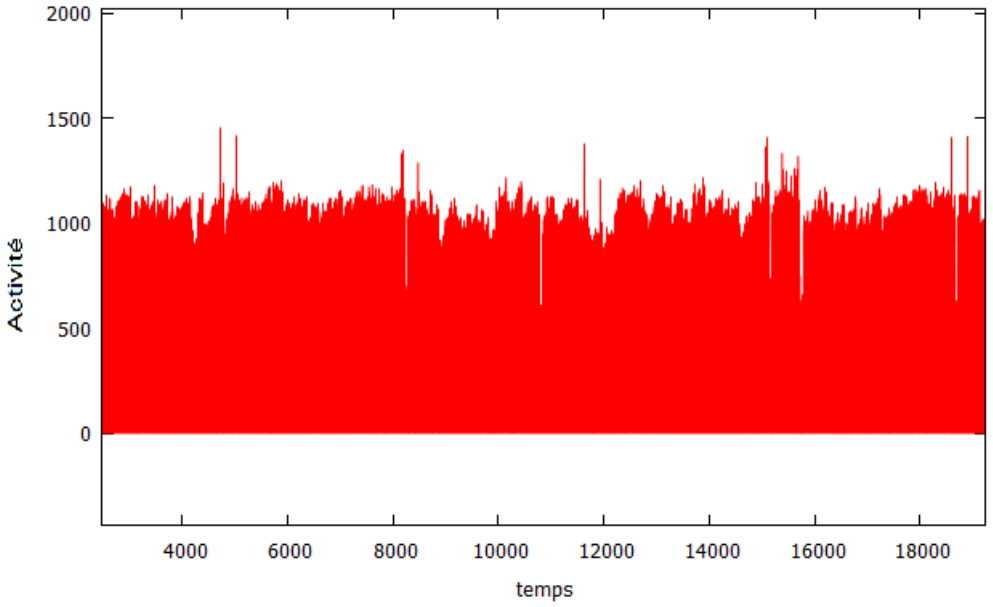


FIGURE 4.9: Un exemple de courbes obtenues à partir d'un fichier **.vcd**. Nous pouvons remarquer, dans cet exemple, que les creux d'activités coïncident avec la fin d'une opération et le début d'une autre.

Dans notre approche, cette mesure (du poids de Hamming) est totalement théorique et se fait par l'intermédiaire de *ModelSim*, un simulateur de circuit dans lequel il est possible de créer un fichier **.vcd** (*Value change dump*) enregistrant à chaque cycle d'horloge l'ensemble des signaux et, de fait, de pouvoir déterminer ceux qui ont changé. Nous avons en plus une simulation *CABA* (*Cycle Accurate Bit Accurate*) qui nous permet d'avoir la main sur tous les aspects de notre crypto-processeur. Un exemple de trace est donné en figure 4.9. Il s'agit là d'un modèle parfait puisqu'aucun bruit n'ira entacher nos diverses mesures. Nous ne pouvons nous assurer que le circuit effectif se comportera

comme nous le supposons ici. Tout dépendra du placement/routage de l'architecture implémentée sur FPGA. Ce modèle théorique ne tient pas, non plus, compte du *fan in/out*. Pourtant, ce modèle est suffisamment cohérent pour pouvoir y formuler des hypothèses.

4.8.2 Exploitation des *templates*

Une fois l'ensemble des *templates* créé, nous pouvons « attaquer » le système. Nous observons une trace T qui a été générée avec une clef inconnue k et nous dérisons, maintenant, la déterminer. En fait, ici nous tenterons de déterminer $\mathcal{K}_1$ qui est la partie avec les puissances positives dans l'écriture de

$$k = \underbrace{\sum_{i=0}^{l-1} \underline{k_i} \times 2^{i-n}}_{\text{puissances négatives } \mathcal{K}_0} + \underbrace{\sum_{i=n}^{l-1} \underline{k_i} \times 2^{i-n}}_{\text{puissances positives } \mathcal{K}_1} .$$

L'idée est de comparer T à la totalité des *templates* (C_{xyz}, M_{xyz}) et de choisir celui avec lequel elle semble le mieux correspondre. Cela se fait à partir de la formule suivante :

$$pr(T|\mathcal{K}_1^{(xyz)}) = pr(T|(C_{xyz}, M_{xyz})) = \frac{1}{\sqrt{2\pi \det(C_{xyz})}} e^{-\frac{1}{2}(T-M_{xyz})C_{xyz}^{-1}(T-M_{xyz})^{(t)}} \quad (4.7)$$

qui détermine la probabilité que la trace observée T soit issue du *template* (C_{xyz}, M_{xyz}). Il existe des raffinements de ce modèle, comme l'emploi du théorème de Bayes. De plus, en pratique, une seule trace T ne peut, à priori, être suffisante pour déterminer la clef $\mathcal{K}_1$ (et par extension k), il est souvent obligatoire de concevoir et de construire des schémas d'attaques plus complexes [67]. Cela dit, nous concernant, nous sommes dans des hypothèses parfaites, sans bruit (gaussien) aucun. Nous montrerons plus tard qu'une seule trace observée et une poignée de traces d'apprentissages (une dizaine) suffit à attaquer notre crypto-processeur (non protégé). Par contre, nous notons dans la formule 4.7 la présence de la quantité C_{xyz}^{-1}. Inverser une matrice peut être un processus coûteux quand la taille de celle-ci est grande. Une trace peut contenir des dizaines de milliers de points. Il sera indispensable de traiter les différentes traces pour en extraire les points saillants, les points d'intérêts. Pour ce faire, il y a bien sûr de nombreuses variantes ; c'est en fait un choix quasi-heuristique qui fixera, en partie, l'efficacité de l'attaque par *templates*. Pour notre part, nous avons choisi de calculer, pour chaque paire M_{xyz} et M_{abc} la différence de vecteurs. Nous sommons l'intégralité des différences obtenues (voir l'équation 4.8), et nous devrions obtenir une nouvelle trace T_p dans laquelle les points les

plus hauts sont les points auxquels nous observons une réelle variation entre les différents *templates*, en d'autres termes, des points qui signent ou caractérisent les dits *templates* (voir figure 4.10).

$$T_p = \sum_{xyz \neq abc} |M_{xyz} - M_{abc}| \tag{4.8}$$

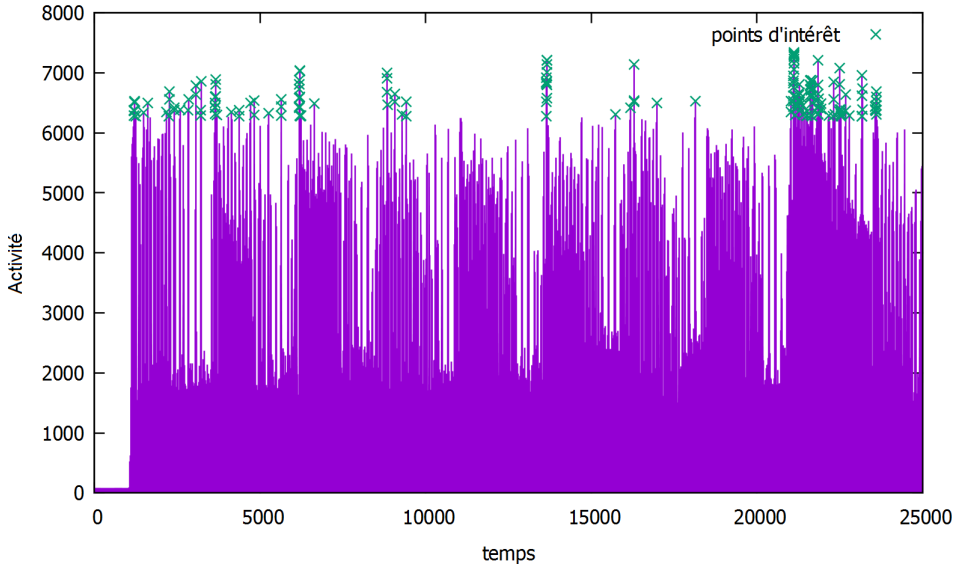


FIGURE 4.10: Le vecteur T_p dont nous extrayons 300 points d'intérêt.

À partir du vecteur T_p, si nous désirons, par exemple 100 points d'intérêts, nous retenons de T_p les 100 valeurs les plus grandes et les points correspondants. Par exemple, si $T_p = (t_0, t_1, \ldots, t_h)$ avec h grand, nous obtiendrons le nouveau vecteur $T_p^{(100)} = (t_{\delta_0}, t_{\delta_1}, \ldots, t_{\delta_{99}})$ où $0 < \delta_0 < \delta_1 < \ldots < \delta_{99} < h$ et où les t_{δ_i} sont parmi les 100 plus grandes valeurs de l'ensemble des t_i. Les δ_i forment une indication temporelle qui révèlent quand, dans le temps, l'appareil « fuit ». Ils seront nos points d'intérêt.

4.8.3 Des idées pour réduire la vulnérabilité du crypto-processeur

Comme nous allons le voir dans la section 4.8.5, le parallélisme à lui-seul ne peut être une vraie protection contre les attaques *templates*. L'un des principaux problèmes auxquels notre architecture est confrontée est que, même si la puissance instantanée des deux cœurs est sensée s'additionner, l'activité d'un cœur ne peut parvenir, seul, a réellement « cacher » l'activité de l'autre. Nous pouvons déceler sur une trace des grands sauts de consommation, signe que les deux cœurs œuvrent au même moment. Ces sursauts suffisent à eux seuls à caractériser une trace et à en déduire la clef (par une approche statistique telle que l'attaque par *templates*). Nous avions différentes idées pour contrer

cela, comme l'instauration de *Dummy operations*, c'est à dire des opérations totalement fictives (inutiles pour le bon déroulement des calculs cryptographiques) pour faire en sorte qu'un même $[k]P$ signe différemment à chaque exécution. Cela rendrait ainsi les sursauts caractéristiques moins flagrants. Nous montrerons que cette approche consolide vraiment la sécurité de notre système, quitte à légèrement ralentir le calcul d'une multiplication scalaire $[k]P$. Nous sommes partis du principe que nous pouvions nous permettre de ralentir les opérations en exploitant le gain (en vitesse) obtenu par l'approche parallèle de la multiplication scalaire $[k]P$. Il est bon de remarquer que les *Dummy operations* se doivent d'être suffisamment conséquentes pour qu'elles soient considérées comme une contre-mesure. Nous avions choisi, en premier lieu, de retarder l'exécution de certaines instructions (de manière totalement aléatoire) de façon à rendre les pics de consommation plus ou moins aléatoire dans le temps. Nous avons, malheureusement remarqué, que retarder le lancement d'une instruction de quelques cycles ne suffit pas, là encore. Le pic sera, somme toute, lui aussi retardé. Nous avons donc la stratégie suivante : avant chaque décodage d'instruction de notre crypto-processeur, nous choisirons de lancer ou non, totalement aléatoirement, avec une probabilité p_{dummy}, une multiplication « pour rien ». Cette multiplication génère de l'activité, elle n'est pas dissociable d'une multiplication classique et permet d'étaler le spectre de la consommation dans le temps. La combinatoire est raisonnablement grande pour que diverses multiplications scalaires $[k]P$ aient un profil de consommation différent.

Nous avions également une autre idée, mais qui n'a pas été implémentée, faute de temps. Nous voulions travailler avec un seul cœur de notre architecture, que nous aurions modifié pour qu'il ait accès à deux registres mémoires (stockant les données du programme) et à deux mémoires contenant les micro-codes. Le cœur aurait été pioché aléatoirement les instructions, soit dans la mémoire contenant le micro-code du *double-and-add* soit dans la mémoire contenant le micro-code du *halve-and-add* (et irait lire/écrire les valeurs dans le bloc mémoire alloué à chacun des micro-codes). Les dépendances entre les instructions doivent bien sûr être conservées (un **LOAD_OP** sera toujours suivi d'un **RUN** et d'un **SEND_R**). Le calcul d'un $[k]P$ aurait été différent à chaque exécution et la localisation temporelle des calculs aurait été aléatoire. Nous pensons qu'il s'agit d'un stratagème viable pour se protéger contre les attaques SPA mais aussi contre les attaques *templates*, dans le sens où il faudra très probablement beaucoup de traces pour assimiler le circuit. Notons qu'il aurait été tout aussi naturel d'inclure des algorithmes du type « Échelle de Montgomery » ; il faut toutefois garder à l'esprit que ce type d'algorithme augmente considérablement le temps de calcul.

4.8.4 Génération de l'aléa

Nous avons choisi d'utiliser une combinaison logique de LFSRs [68] afin de générer notre aléa. Cet aléa permettra au crypto-processeur de choisir entre une instructions utiles et « opérations pour rien ». Un **LFSR** (*Linear-Feedback-Shift-Register*) est un registre à décalage à rétro-action linéaire. Derrière ce terme se cache un mécanisme simple pour générer des suites de bits pseudo-aléatoire. Ils sont constitués à partir d'un petit nombre de portes `XOR` et d'un registre à décalage. Ils sont faciles à implémenter en circuit, mais aussi en logiciel.

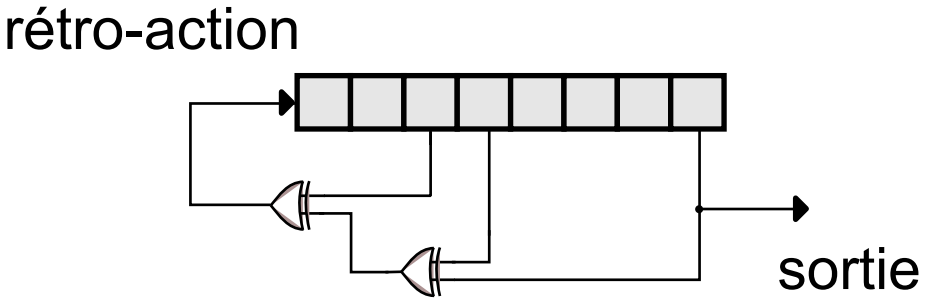


FIGURE 4.11: LFSR de Fibonacci [68] sur 8 bits.

Un exemple est donné dans la figure 4.11. À chaque cycle d'horloge, le registre est décalé de 1 bit vers la droite. La nouvelle entrée (à gauche) est alimentée par une combinaison de portes `XOR`s dont la sortie dépend de l'état registre lui-même. Les portes `XOR`s ne sont pas reliées aux bascules du registre aléatoirement, la connexion dépend d'un polynôme de « rétro-action ». Si nous reprenons l'exemple de la figure 4.11, le polynôme de GF(2)[x] correspondant est $x^8 + x^4 + x^3 + 1$. En effet, la huitième bascule (tout à droite) est reliée à la quatrième par le biais d'une porte `XOR` dont la sortie est, enfin, reliée à la troisième bascule. Le 1 ($= x^0$) de ce polynôme n'intervient pas dans la conception du LFSR. Au bout d'un certain nombre de cycle, le LFSR produira la même séquence de 0 et 1. La périodicité de cette séquence est au maximum de $2^d - 1$ cycles (où d est la taille du registre). Cette même séquence est entièrement déterminée par sa valeur initiale (parfois appelée IV comme *Initialization Vector*), c'est à dire l'état du registre à l'instant 0. Pour assurer le bon fonctionnement du circuit, les polynômes de rétro-action sont des polynômes primitifs [69]. Pour rappel, un polynôme primitif est le polynôme minimal d'un élément x primitif, c'est à dire, un élément qui génère le groupe multiplicatif de GF(2^m)*. Ils permettent de maximiser la durée du cycle tout en évitant des convergences pathologiques. Typiquement, si le registre atteint la valeur 0, tous les bits qui sortiront du LFSR seront eux aussi, égaux à 0. Il y a autant de 0 que de 1 en sortie du LFSR (sauf dans le cas où $IV = 0$), $pr(0) = 1/2$ et $pr(1) = 1/2$. Il est possible

de combiner les LFSR entre eux pour obtenir deux sorties et ainsi pouvoir générer d'autres probabilités $pr(00) = pr(01) = pr(10) = pr(11) = 1/4$ (voir figure 4.12). Pour maximiser la période de cette combinaison de LSFR, nous choisirons habituellement des LSFRs dont les périodes p_0 et p_1 sont premières entre elles. Nous aurons donc une période égale au *ppcm* de p_0 et p_1, c'est à dire $p_0 \times p_1$.

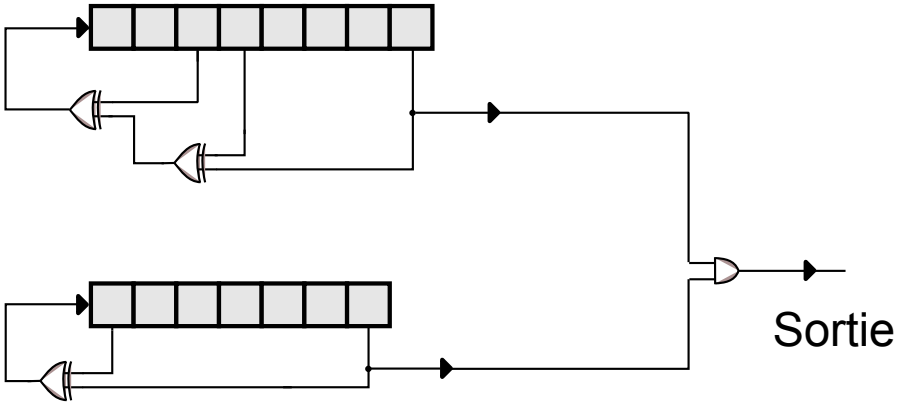


FIGURE 4.12: Deux LFSRs de Fibonacci [68] combinés.

Le circuit présenté dans la figure 4.12 permet de générer un circuit produisant 25% de 1 et 75% de 0. Ces circuits, simples, vont nous permettre de générer l'aléa dont nous avons besoin pour insérer des *dummy operations* au cours du calcul d'un $[k]P$.
Nous avons choisi de relier ces *LFSRs* pour générer, au niveau du décodeur d'instruction, les « opérations pour rien » (*Dummy Operations*) (comme le montre la figure 4.13). Ces LFSRs sont initialisables, même si les connexions ne sont pas ici représentées. Les LFSRs sont, en soi, de « mauvais » générateurs d'aléas : des algorithmes (comme celui de Berlekamp-Massey [70]) permettent, par exemple, de remonter jusqu'au polynôme primitif à partir des séquences « d'aléas » observées. Ils resteront, pour nous, une « preuve de concept ».

4.8.5 Résultats des attaques par Templates

Nous avons attaqué notre processeur à l'aide d'une attaque par *templates*. Nous nous sommes fixés le corps GF(2^{233}) et l'algorithme de *halving-parallèle*. Nous avons fixé le nombre de traces d'apprentissage à 1000 (par hypothèse de clefs) et le nombre de points d'intérêt à 300. Nous n'observons **qu'une** trace d'exécution (mais cela n'est pas aberrant puisque, rappelons-le, nous jouissons d'un modèle idéal, sans bruit). De plus, le comportement que nous n'observons ici que sur une trace est logiquement assimilable à n'importe quelle autre trace. Le but était de montrer que les « opérations pour rien » ont

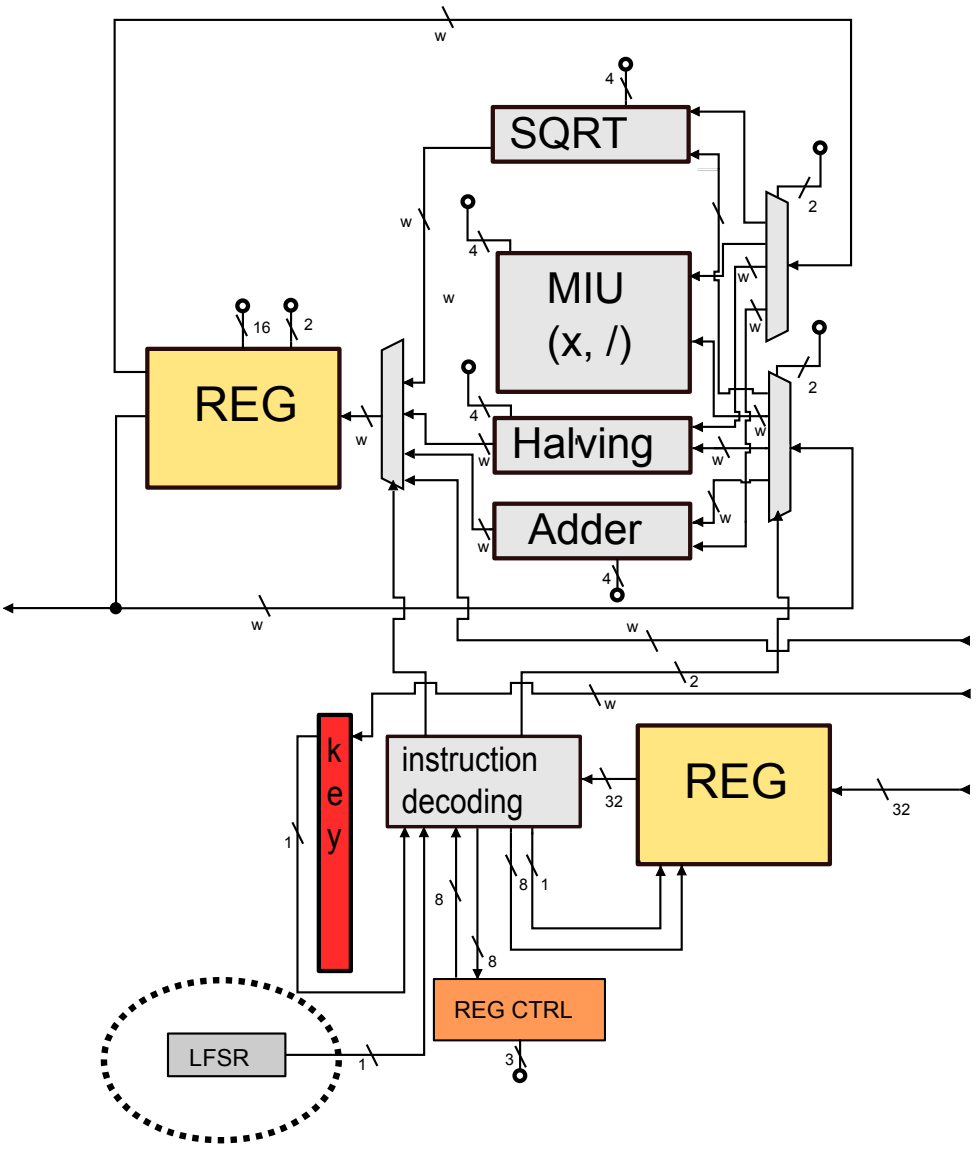


FIGURE 4.13: Une unité d'exécution dont le décodeur d'instruction est relié à des LSFR.

suffisamment d'impact pour protéger le circuit. Évidemment, plus la probabilité de lancement d'une « opération pour rien » est grande, plus le calcul d'une multiplication scalaire sera, lui aussi, grand (voir tableau 4.3). La probabilité p_{dummy} est la probabilité (construite à partir de LFSRs) que le circuit lance, pendant le décodage d'instruction, une « opération pour rien ».

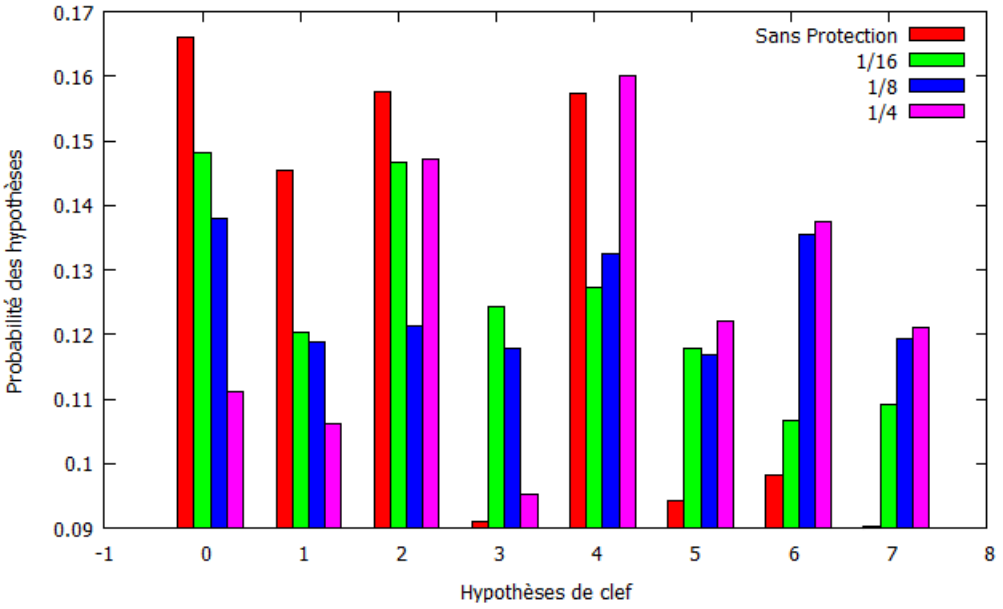


FIGURE 4.14: Probabilité des hypothèses de clefs construites par l'attaque par *templates* (1000 traces d'apprentissage).

p_{dummy}	Nombre de Cycles pour un [k]P	Surcout temporel
0	474733	-
1/16	519596	9%
1/8	570985	20%
1/4	732284	36%

TABLE 4.3: Surcouts temporels [a] des « opérations pour rien » en fonction de la probabilité p_{dummy}.

a. moyennés à partir de 10 exécutions

Dans la figure 4.14, nous montrons les probabilités respectives établies par l'attaque par *templates* des clefs $0 = (000)_2, 1 = (001)_2, \ldots 7 = (111)_2$. La « bonne hypothèse de clef » est $0 = (000)_2$, à laquelle les templates associent la plus grande probabilité dans une version parallèle non-protégée par le biais des *Dummy Operations*.

Nous voyons à l'inverse qu'une version protégée avec 25% de chance de lancer une multiplication inutile (à chaque instruction) ne permet pas à l'attaque par *templates* de récupérer l'hypothèse correcte de la clef. Pire, la probabilité de l'hypothèse 000 n'est ni seconde, ni troisième dans l'ordre des probabilités, mais sixième. Nous pouvons dès lors imaginer que cette contre-mesure permet également de contrer les attaques (possiblement) par énumération de clefs [71]. Nous observons par contre qu'une probabilité trop faible sur l'exécution d'opérations inutiles rétablit une certaine forme de vulnérabilité : avec 1/8ème de chance de lancer une opération pour rien, la première hypothèse 000

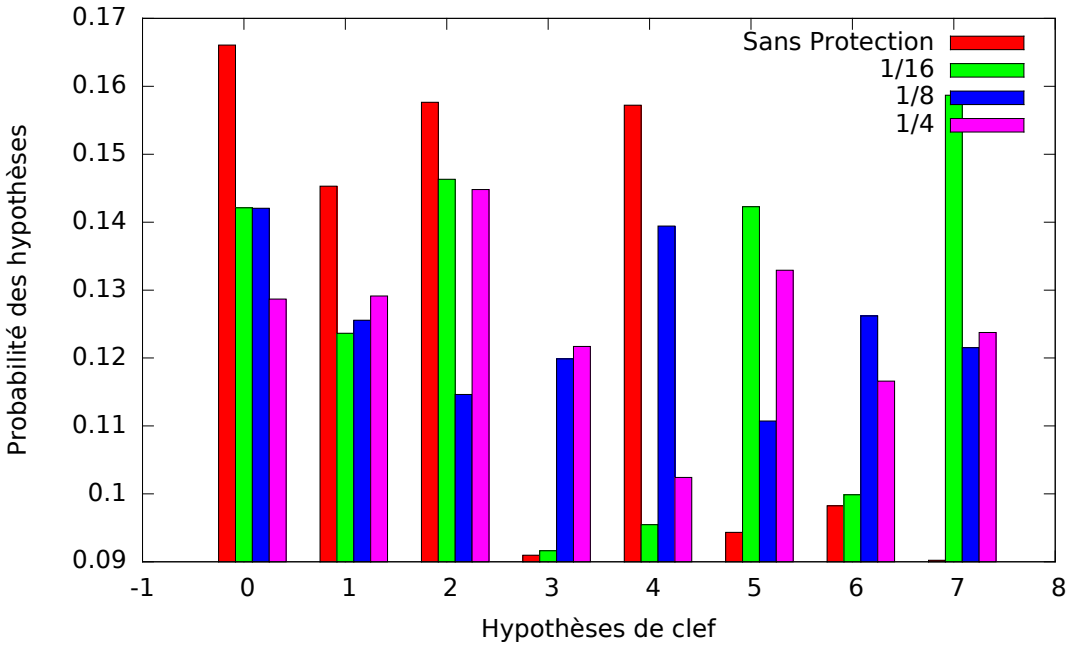


FIGURE 4.15: Probabilité des hypothèses de clefs construites par l'attaque par *templates* (500 traces d'apprentissage).

redevient celle à qui il est attribué la plus grande probabilité. Le phénomène est encore plus remarquable et identifiable avec une probabilité de 1/16 sur l'exécution de *Dummy operations*. Il est alors évident qu'il existe un **compromis** entre sécurité (face aux canaux cachés) et performance brute. Avec une probabilité $p_{\text{dummy}} = 1/4$, le circuit semble, en partie, protégé mais le temps d'exécution de la multiplication scalaire se rapproche d'un algorithme purement séquentiel de *halve-and-add*. Dans la figure 4.15, il s'agit des mêmes données que fournies précédemment, sauf que le nombre de traces d'apprentissage a été réduit à 500 par hypothèse de clef. Nous pouvons remarquer l'impact significatif des traces sur la qualité des attaques. Dans une version non protégée (sans « opérations pour rien »), la probabilité affectée à la bonne hypothèse 000 est moins marquée qu'auparavant.

Avec un nombre de traces trop faible, même un p_{dummy} petit ($\approx 1/16$) permet de « protéger » le circuit. Avec un $p_{\text{dummy}} = 1/16$, la probabilité attribuée à l'hypothèse de clef 000 n'est que la troisième plus grande.

Nous avons également tracé dans 4.16, l'écart-type des probabilités de clefs estimées par les *templates* pour chaque probabilité $p_{\text{dummy}} \in \{0, 1/16, 1/8, 1/4\}$.

Plus cet écart-type est faible, plus la distribution tend à se concentrer autour de la « moyenne » (1/8). Ce qui est remarquable, c'est que cet écart-type tend à se réduire lorsque nous augmentons la probabilité p_{dummy}. Autrement dit, il semble qu'ajouter des

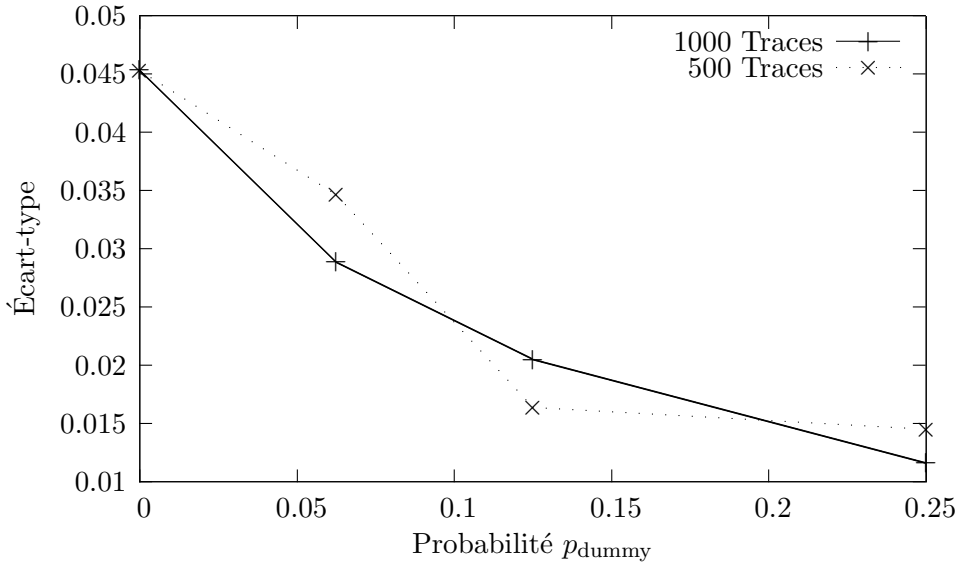


FIGURE 4.16: Écart-type des hypothèses de clef en fonction de la probabilité p_{dummy}.

« opérations pour rien » tend à rendre les probabilités attribuées à chacune des hypothèses de clef plus uniformes et permettrait ainsi de réduire les biais.
C'est d'ailleurs ce que nous observons au travers les chiffres du tableau 4.4. Nous avons, pour 100 attaques, estimé le nombre d'attaques réussies en fonction de la probabilité p_{dummy}. Chacune des instances des attaques a été réalisée dans les mêmes conditions que précédemment, c'est à dire que nous disposons de 1000 traces d'apprentissages par templates (soit un total de $2^3 \times 1000 = 8000$ traces) et travaillons avec 300 points d'intérêts. Nous considérons une attaque réussie quand la probabilité $pr(T|\mathcal{K}_1^{(xyz)})$ de la bonne hypothèse de clef est la plus grande. Ce qui semble émerger de ce tableau est la confirmation qu'augmenter la probabilité p_{dummy} rend les probabilités $pr(T|\mathcal{K}_1^{(xyz)})$ uniformes (ce qui va de pair avec le taux de réussite). Il apparait que chacune des probabilités $pr(T|\mathcal{K}_1^{(xyz)})$ tend vers $1/2^3$.
Évidemment, nous sommes conscients que nous avons choisi des paramètres arbitraires

p_{dummy}	0	1/16	1/8	1/4	1/2
Taux de Réussite	100	76	62	11	13

TABLE 4.4: Taux de réussite en fonction de p_{dummy} sur 100 essais.

concernant les attaques. Il serait en l'occurrence très intéressant d'étudier la sensibilité de l'efficacité des « opérations pour rien » en fonction du nombre de traces d'apprentissages, du nombre de points d'intérêts, du nombre de traces observées pour la phase d'attaque, etc. Bref, cette étude est loin d'être exhaustive ; elle montre à l'inverse une tendance. Comme attendu, augmenter le nombre « d'opérations pour rien » réduit l'efficacité des attaques. Cette étude montre surtout que le parallélisme des calculs dans

la multiplication scalaire ne permet pas, seul, de se prémunir face à des attaques par canaux cachés.

4.9 Conclusion

Nous avons présenté dans ce chapitre un crypto-processeur réalisant des calculs sur des extensions du corps fini binaire GF(2) dans une représentation en base normale. Nous avons montré que le calcul parallèle permis par le *halving* et proposé par Christophe Negre et Jean-Marc Robert dans [46] est intéressant sans pour autant être très efficace sur l'architecture ici proposée. L'algorithme de *parallel halve-and-add* 4.5 s'est montré plus rapide. Nous avons aussi montré que le parallélisme des calculs, seul, ne parvient pas à protéger le circuit face à des attaques par canaux cachés, notamment par des attaques par *templates*. Pour apporter une résistance supplémentaire, nous avons suggéré l'emploi « d'opérations pour rien » cachant le calcul utile dans un flot de calculs, lui, inutile. Évidemment, rajouter des opérations pour rien, c'est allonger, implicitement, le temps d'exécution d'une multiplication scalaire $[k]P$. Nous nous sommes aperçus qu'il y avait un compromis sécurité/performance, dépendant de l'abondance de ces « opérations inutiles ». Il serait intéressant, en perspectives, de s'intéresser à ces différents compromis. Pourrions-nous, par exemple, n'appliquer les « opérations pour rien » que sur la partie la plus rapide de la multiplication scalaire (c'est à dire sur le *halve-and-add*) tout en conservant une sécurité équivalente ? Pourrions nous marier des « opérations pour rien » avec d'autres types de protections (échelle de Montgomery, atomicité, etc) ? Qu'en est-il des autres systèmes de coordonnées ?

Chapitre 5

RNS dans GF(2^m)

5.1 Réduction modulaire en RNS

Dans ce chapitre, nous étudierons la représentation **RNS** (*Residue Number System*) et la pertinence de son utilisation pour les extensions du corps fini binaire GF(2). Nous présenterons notamment le « Φ-RNS », une représentation mono-base des éléments de GF(2^m) (jusqu'ici, représenter un élément en RNS supposait l'existence de deux bases —au moins—, nous y reviendrons). Nous irons jusqu'à l'implémentation FPGA de l'algorithme de multiplication modulaire en « Φ-RNS ». Nous détaillerons un algorithme d'extraction de racines carrées, qui n'est d'ailleurs pas une opération si triviale dans cette représentation, ainsi qu'un algorithme d'inversion. Ces deux derniers points n'ayant, par contre, pas fait l'objet d'une implémentation matérielle.

5.1.1 Algorithme de Montgomery

L'algorithme de Montgomery [12] est probablement l'un des algorithmes de réduction modulaire les plus populaires. Il permet notamment d'éviter la (couteuse) division généralement requise dans un algorithme de réduction plus naïf. L'algorithme de Montgomery ne travaille pas avec une représentation standard des éléments, ces derniers sont d'abord convertis dans le « domaine de Montgomery ». Nous noterons cette transformation M(A) :

$$\begin{array}{lcl} \mathrm{GF}(2^m) & \rightarrow & \mathrm{GF}(2^m) \\ \mathrm{M}(A) & \mapsto & A \times R \end{array}$$

La quantité R est généralement fixée à x^m dans le cadre de GF(2^m) [72]. La multiplication de Montgomery prend en entrée deux éléments A et B de GF(2^m) et calcule la quantité $\text{MM}(A, B) = A \times B \times R^{-1} \bmod f$. Si les deux opérandes ont été auparavant convertis dans « le domaine de Montgomery », le résultat sera, lui aussi, représenté dans ce même domaine :

$$\text{MM}(\text{M}(A), \text{M}(B)) = (A \times R) \times (B \times R) \times R^{-1} \bmod f = A \times B \times R.$$

L'algorithme est décrit en Algo. 18. La clef de l'algorithme est de rajouter autant de fois f que requis au produit $A \times B$ pour que $A \times B + q \times f$ soit divisible par $R = x^m$. Diviser par R est très simple, c'est un décalage des coefficients composant le nombre $A \times B + q \times f$. Par exemple $(110000)_2 = x^5 + x^4$ est divisible par x^3, la division de $x^5 + x^4$ par x^3 est égale à $x^2 + x = (000110)_2$. Il s'agit de la même représentation, sauf que les bits ont été décalés de trois rangs vers la droite. La division intrinsèque à la réduction est toujours présente ... mais elle a été astucieusement remplacée par un décalage.

Algorithme 18 : Algorithme de multiplication de *Montgomery* [45].

Données : $\tilde{A} = A \times R$ et $\tilde{B} = B \times R$ dans le domaine de Montgomery dans GF(2^m).
Résultat : $\tilde{A} \times \tilde{B} \times R$ dans le domaine de Montgomery.

1 $t_0 \leftarrow (\tilde{A} \times \tilde{B} \times f^{-1}) \bmod R$
2 $t_1 \leftarrow \tilde{A} \times \tilde{B} + f \times t_0$
3 $t_2 \leftarrow t_1 / R$
4 **retourner** t_2

Dans l'algorithme 18, f^{-1} est l'inverse modulaire de $f \bmod R$. La quantité f^{-1} est calculable à l'avance. À noter que l'opération modulo $R = x^m$ ligne 1 est triviale. Réaliser une opération modulo x^m consiste à ne garder que les m bits de poids faibles (plus petits que x^m) de l'élément donné. Par exemple $(x^4+x+1) \bmod x^3 = x+1$. La conversion d'un élément A dans le domaine de Montgomery à partir de l'algorithme de multiplication de Montgomery se fait par le calcul de $\text{MM}(A, R^2)$ et l'opération inverse par le calcul de $\text{MM}(A \times R, 1)$.

Exemple 5.1.1. Nous travaillons dans cet exemple avec le corps GF(2^3) et le polynôme de réduction $f = x^3 + x + 1$. Nous posons $R = x^3$ et trouvons que $f^{-1} = x^2 + x + 1$. Si nous souhaitons calculer le produit de $A = x^2 + 1$ et $B = x^2$ par l'algorithme de Montgomery, il nous faut d'abord convertir A et B dans le domaine de Montgomery. Nous supposons que cela soit déjà fait (nous laisserons le lecteur vérifier les calculs) et nous trouvons $\tilde{A} = A \times R = x^2$ et $\tilde{B} = B \times R = 1 + x + x^2$. Appliquons maintenant l'algorithme de multiplication de Montgomery :

– $t_0 \leftarrow \tilde{A}\times\tilde{B}\times f^{-1} \bmod x^3 = (x^4+x^3+x^2)\times(x^2+x+1) \bmod x^3 = x^6+x^4+x^2 \bmod x^3 = x^2$.
– $t_1 \leftarrow \tilde{A}\times\tilde{B}+f\times t_0 = x^4+x^3+x^2+(x^2)\times(x^3+x+1) = (x^4+x^3+x^2)+(x^5+x^3+x^2) = x^5+x^4$
– $t_2 \leftarrow t_1/R = (x^5+x^4)/x^3 = x^2+x$

Nous avons donc $A \times B \times R = x^2 + x$.

L'algorithme de Montgomery permet donc d'avoir une réduction modulaire en n'effectuant que des multiplications et des décalages sur les éléments. Les calculs sont comparativement simples vis-à-vis d'une division (typiquement utilisée pour une réduction de Barrett [73]) ce qui rend la méthode attrayante.

5.1.2 Multiplication de Mastrovito

L'algorithme de Mastrovito a été proposé par Mastrovito lui-même durant sa thèse [74]. Il est basé sur du calcul matriciel. L'idée est de construire une matrice de multiplication qui concentre multiplication et réduction modulaire. Si A et B sont les deux opérandes (appartenant à GF(2^m)), la méthode construit d'abord une matrice M_B qui sera (vectoriellement) multipliée par A. Autrement dit $A\times B \bmod f = M_B \times A$ (à droite de l'égalité, les quantités manipulées sont des vecteurs, que nous ne distinguerons pas, par abus de notation, des éléments en représentation standard). Nous noterons, tout d'abord, que C [*] $= A \times B =$

$$
\begin{array}{llllllllllllll}
 & (& b_0 & + & b_1x & + & \dots & + & b_{m-1}x^{m-1} & & & & &) \times a_0 \\
+ & (& & & b_0x & + & \dots & + & b_{m-2}x^{m-1} & + & b_{m-1}x^m & & &) \times a_1 \\
 & & & & & \vdots & & & \vdots & & & & & \\
+ & (& & & & & & + & b_0x^{m-1} & + & b_1x^m & + \dots + & b_{m-1}x^{2m-2} &) \times a_{m-1}
\end{array}
$$

qui peut s'écrire sous forme matricielle comme :

*. avec $C = (c_0, c_1, \dots, c_{2m-2})$.

$$\begin{matrix} M_U \\ \\ \\ \\ \\ M_D \\ \\ \\ \\ \\ \end{matrix} \left\{ \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \right. \left(\begin{array}{cccc} b_0 & 0 & \dots & 0 \\ b_1 & b_0 & \ddots & 0 \\ \vdots & \vdots & \ddots & 0 \\ b_{m-1} & b_{m-2} & \ddots & 0 \\ \hline 0 & b_{m-1} & \ddots & b_1 \\ \vdots & 0 & \ddots & b_2 \\ 0 & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & b_{m-1} \end{array} \right) \times \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ a_{m-1} \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{2m-2} \end{pmatrix}.$$

Cette matrice se décompose en deux sous-matrices : la partie supérieure M_U et la partie inférieure M_D. La matrice M_U génère la fraction du produit qui n'a pas besoin d'être réduite modulo f (les degrés des monômes correspondants sont situés entre 0 et $m-1$). À l'inverse, la matrice M_D forme les monômes de degrés compris entre m et $2m-2$. Il est nécessaire d'appliquer un traitement particulier à M_D avant de pouvoir construire la matrice M_B auparavant évoquée. La première ligne de M_D nous dit que $a_1 \times b_{m-1} + a_2 \times b_{m-2} + \ldots + a_{m-1} \times b_{m-2}$ est le poids attribué au monôme x^m. Il suffit d'exprimer x^m mod f en fonction de la base polynomiale $1, x, x^2, \ldots, x^{m-1}$ pour réduire ce bit du produit $A \times B$. Nous créons alors un vecteur colonne $X^{(0)} = x^m \bmod f$. Nous réitérons le procédé sur chacune des lignes de M_D jusqu'à obtenir $m-1$ vecteurs $X^{(0)}, X^{(1)} = x^{m+1} \bmod f, X^{(2)} = x^{m+2} \bmod f, \ldots, X^{(m-2)} = x^{2m-2} \bmod f$ auxquels nous attribuerons le poids binaire $\sum_{j+k=i} a_j \times b_k$ correspondant.

Nous savons donc que le produit $A \times B \bmod f$ vaut :

$$A \times B \bmod f = M_U \times A + \sum_{i=m}^{2m-2} \Big(\sum_{j+k=i} a_j \times b_k \Big) X^{(i-m)}$$

qui peut également, une nouvelle fois, s'écrire sous forme matricielle comme

$$A \times B \bmod f = \begin{pmatrix} b_0 & 0 & \dots & 0 \\ b_1 & b_0 & \ddots & 0 \\ \vdots & \vdots & \ddots & 0 \\ b_{m-1} & b_{m-2} & \ddots & 0 \end{pmatrix} \times A$$

$$+ \underbrace{\left(X^{(0)} \,\middle|\, X^{(1)} \,\middle|\, X^{(2)} \,\middle|\, \dots \,\middle|\, X^{(m-1)} \right) \begin{pmatrix} 0 & b_{m-1} & \ddots & b_0 \\ \vdots & 0 & \ddots & b_1 \\ 0 & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & b_{m-1} \end{pmatrix}}_{=M_R} \times A.$$

Nous pouvons au final, écrire que $A \times B = (M_U + M_R) \times A = M_B \times A$. La matrice M_B peut dans certain cas (selon le polynôme de réduction f) avoir une forme particulière (typiquement, une matrice de Toeplitz). Une matrice de Toeplitz est une matrice du type :

$$M = \begin{pmatrix} M_0 & M_{-1} & M_{-2} & \dots & \dots & M_{-n+1} \\ M_1 & M_0 & M_{-1} & \ddots & & \vdots \\ M_2 & M_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & M_{-1} & M_{-2} \\ \vdots & & \ddots & M_1 & M_0 & M_{-1} \\ M_{l-1} & \dots & \dots & M_2 & M_1 & M_0 \end{pmatrix}$$

En exploitant ces propriétés de symétries, il est possible d'accélérer la construction de M_B (il y a un nombre limité de sous-blocs M_i, il suffit de les créer pour composer l'intégralité de M_B). C'est ce qui a été notamment fait dans la thèse de Danuta Pamula [28] et dans l'article [75]. Pour résumer, l'algorithme de multiplication de Mastrovito consiste en deux principales étapes (qui peuvent elles-mêmes être décomposées selon diverses approches) : une étape de construction de la matrice M_B et une étape de multiplication matrice-vecteur. À noter qu'un polynôme de réduction creux (trinomial, pentanomial) apporte une simplicité certaine pendant la phase de construction de M_B.

Exemple 5.1.2. Dans cet exemple, nous choisissons le polynôme de réduction $f = x^3 + x + 1$. Nous choisissons également les opérandes $A = x^2 + 1$ et $B = x^2$. La première

chose à faire est de construire la matrice M_U (qui dépend de B) :

$$M_U = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Nous avons maintenant à générer les vecteurs $X^{(i)}$:

- $X^{(0)} = x^3 \bmod f = x + 1 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$
- $X^{(1)} = x^4 \bmod f = x^2 + x = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$

d'où

$$M_R = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

Nous en déduisons la matrice M_B suivante :

$$M_B = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}.$$

Le produit $A \times B \bmod f = M_B \times A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = x.$

L'algorithme de Mastrovito permet un parallélisme important, principalement modulé par la stratégie de la construction de la matrice M_B. C'est d'ailleurs cette modularité qui apporte à l'algorithme un intérêt certain.

5.1.3 Théorème chinois des restes

Le théorème chinois des restes est un théorème fondamental en arithmétique modulaire. Son origine proviendrait du livre « Sunzi Suanjing » [76] datant du III$^{\text{ème}}$ siècle. Il a été décrit sous forme de problème dans les entiers.

> *Je possède une armée de h hommes. Si j'organise mon armée en lignes de* 5*, il restera une ligne qui ne contiendra que* 2 *hommes. Si j'organise mon armée en lignes de* 7*, il restera une ligne qui ne contiendra que* 3 *hommes. Combien ai-je d'hommes disponibles dans mon armée ? (au minimum).*

En termes mathématiques, le problème précédent pourrait s'écrire comme le système de congruences suivant :

- $h \bmod 5 \equiv 2$
- $h \bmod 7 \equiv 3$

Une des solutions à ce problème est $h = 17$. En effet, $17 \bmod 5 = 2$ et $\bmod 7 = 3$. Il ne s'agit pas de la seule solution puisque 52 le sera également, 87 aussi ... et d'une manière générale, $17 + q \times (5 \times 7)$. Le théorème chinois des restes stipule que tout système de congruence :

$$\begin{cases} h \bmod n_0 = s_0 \\ h \bmod n_1 = s_1 \\ h \bmod n_2 = s_2 \\ \qquad \vdots \\ h \bmod n_{v-1} = s_{v-1} \end{cases}$$

admet une solution unique modulo le produit des moduli $N = \prod_{i=0}^{v-1} n_i$ si les n_i sont premiers entre eux. Il existe une formule qui fournit une solution :

$$s = \sum_{i=0}^{v-1} |\tilde{s}_i \times N_i^{-1}|_{n_i} \times N_i$$

avec $N_i = \prod_{i \neq j} n_j$ et N_i^{-1} l'inverse modulaire de $N_i \bmod n_i$. Cet inverse existe puisque N_i est premier avec n_i (par hypothèse sur les n_i). La notation $|.|_{n_i}$ est une autre façon de noter la réduction modulaire : $|a|_{n_i} = a \bmod n_i$. Nous pouvons vérifier que la formule nous procure bien une solution en calculant :

$$s \bmod n_i = \sum_{i=0}^{v-1} (|\tilde{s}_i \times N_i^{-1}|_{N_i} \times n_i) \bmod n_i = (|\tilde{s}_i \times N_i^{-1}|_{n_i} \times N_i) \bmod n_i = s_i.$$

Le théorème chinois des restes s'applique aux *anneaux principaux* ; en somme, il s'applique à la représentation polynomiale des éléments d'un corps fini dans GF(2^m). De par cette propriété d'unicité de la solution modulo N, nous pouvons construire une arithmétique simplifiant bon nombre de calculs. Pour un nombre $x < N$ (ou $\deg(x) < \deg(N)$), nous notons sa représentation modulaire (couramment appelée représentation **RNS** (*Residue-Number System*)) par

$$A = (A \bmod n_0, A \bmod n_1, \ldots, A \bmod n_{v-1}) = (\tilde{A}_0, \tilde{A}_1, \ldots, \tilde{A}_{v-1}).$$

Cette représentation est très pratique puisque $A + B \bmod N = (\tilde{A}_0, \tilde{A}_1, \ldots, \tilde{A}_{v-1}) + (\tilde{B}_0, \tilde{B}_1, \ldots, \tilde{B}_{v-1}) = (\tilde{A}_0 + \tilde{B}_0 \bmod n_0, \tilde{A}_1 + \tilde{B}_1 \bmod n_1, \ldots, \tilde{A}_{v-1} + \tilde{A}_{v-1} \bmod n_{v-1})$. Nous avons le même genre de relation pour le produit : $A \times B \bmod N = (\tilde{A}_0, \tilde{A}_1, \ldots, \tilde{A}_{v-1}) \times (\tilde{B}_0, \tilde{B}_1, \ldots, \tilde{B}_{v-1}) = (\tilde{A}_0 \times \tilde{B}_0 \bmod n_0, \tilde{A}_1 \times \tilde{B}_1 \bmod n_1, \ldots, \tilde{A}_{v-1} \times \tilde{B}_{v-1} \bmod n_{v-1})$. Cette propriété est séduisante puisque cela veut dire que la multiplication et l'addition se font sans propagation de retenues entre les différents restes modulo n_i. Le coût de la multiplication peut être découpée en $\mathcal{O}((m/w)^2 \times w)$ face à $\mathcal{O}(m^2)$ sans découpage aucun (où m est le nombre de bits des opérandes considérés et w la taille des canaux). Il est tout à fait faisable de paralléliser les calculs en travaillant sur chacun des résidus simultanément : le parallélisme est d'ailleurs l'un des buts premiers de la représentation RNS.

La représentation RNS est idéale lorsque nous devons travailler modulo un nombre qui se décompose en de nombreux moduli n_i (il suffit d'appliquer mécaniquement le théorème des restes chinois pour gérer cette arithmétique). Notre problématique s'avère différente puisque l'arithmétique des corps finis (que cela soit GF(p) ou GF(2^m)) requiert de travailler modulo un élément premier/irréductible. C'est alors que les auteurs de [11] suggèrent d'entremêler représentation RNS et réduction de Montgomery. Le concept est simple puisqu'il « suffit » d'employer l'algorithme de Montgomery (que nous avons décrit en 5.1.1) avec l'arithmétique RNS. Malgré l'aspect simpliste de la proposition, il y a de nombreuses subtilités. Ce qui est généralement fait si nous voulons multiplier deux éléments x et y en Montgomery-RNS est de trouver q tel que $x \times y + q \times f$ soit divisible par N. Cela revient à trouver q tel que $(A \times B + q \times f) \bmod N = (0, 0, \ldots, 0)$. La prochaine étape est de diviser la quantité $A \times B + q \times f$ par N. Or $N = (0, 0, \ldots, 0)$ (exprimé comme $\bmod n_0$, $\bmod n_1, \ldots, \bmod n_{v-1}$), nous avons donc une opération du type $0/0 \ldots$ Évidemment, c'est absurde. L'astuce est de se donner une nouvelle base de moduli $n'_0, n'_1, \ldots, n'_{w-1}$ formant le modulo $N' = \prod_{i=0}^{w-1} n'_i$. Les n'_i sont supposés premiers entre eux et premiers avec les n_i. Trouver la quantité q est facilement réalisable dans la première base $(n_0, n_1, \ldots, n_{v-1})$. Une fois ce q trouver, la valeur q est traduite dans la seconde base $(n'_0, n'_1, \ldots, n'_{w-1})$ (à partir d'un processus nommé **extension de base** [77]). Le calcul de $A \times B + q \times f$ est effectué dans cette seconde base dans laquelle la division par N est possible (puisque $N \neq 0 \bmod N'$). Le nouveau résultat est enfin retraduit dans la première base au travers d'une nouvelle extension de base. Pour faciliter la lecture nous dirons que l'élément x représenté dans la base $n_0, n_1, \ldots, n_{v-1}$ est *écrit dans la base* $\mathcal{B}$ et l'élément A représenté dans la base $n'_0, n'_1, \ldots, n'_{w-1}$ est *écrit dans la base* $\mathcal{B}'$. L'algorithme RNS-Montgomery est décrit en 19 pour GF(2^m). La fonction

BE$(D, \mathcal{B}, \mathcal{B}')$ est une fonction qui convertit l'élément D exprimé dans la base $\mathcal{B}$ dans la base $\mathcal{B}'$.

Algorithme 19 : Multiplication de Montgomery en RNS dans GF(2^m).

Données : A et B en base $\mathcal{B}$ et $\mathcal{B}'$
Résultat : $A \times B \times M^{-1} \mod f$ dans les bases $\mathcal{B}$ et $\mathcal{B}'$

1 $C_0 \leftarrow A \times B \mod N$ *en base* $\mathcal{B}$
2 $C_1 \leftarrow A \times B \mod N'$ *en base* $\mathcal{B}'$
3 $D \leftarrow C_0 \times f^{-1} \mod N$ *en base* $\mathcal{B}$
4 $D \leftarrow \mathbf{BE}(D, \mathcal{B}, \mathcal{B}')$
5 $D \leftarrow (C_1 + D \times f) \mod N'$ *en base* $\mathcal{B}'$
6 $D \leftarrow (D \times N^{-1}) \mod N'$ *en base* $\mathcal{B}'$
7 $E \leftarrow \mathbf{BE}(D, \mathcal{B}', \mathcal{B})$
8 **retourner** (D, E)

Il faut surement quelques restrictions sur le degré des polynômes N et N' pour certifier du bon déroulement de l'algorithme de Montgomery en RNS. Typiquement, il faut que $\deg(N) > \deg(A)$ et $\deg(N') > \deg(A)$ pour tout élément A dans GF(2^m) afin de garantir que les quantités manipulées soient dans la portée des éléments qui peuvent être représentés par les bases $\mathcal{B}$ et $\mathcal{B}'$. L'extension de base **BE** peut être effectuée de différentes façons. Nous en citerons deux. L'une est basée sur le théorème chinois des restes (c'est celle que nous emploierons) et l'une basée sur une représentation intermédiaire dite **MRS** (*Mixed-Radix System*) [11]. En MRS, A est représenté comme

$$A = \sum_{i=0}^{v-1} \tilde{A}_i \times \mathcal{N}_{i-1} \tag{5.1}$$

où $\mathcal{N}_i = \prod_{j=0}^{i} n_j$ avec $\mathcal{N}_{-1} = 0$. Connaissant x dans la base $\mathcal{B}$, il est possible d'extraire chacun des coefficients dans la nouvelle base MRS en utilisant un schéma de type Hörner [78].

$$\begin{cases} \tilde{A}_0 \leftarrow \tilde{A}_0 \\ \tilde{A}_1 \leftarrow (\tilde{A}_1 - \tilde{A}_0) \times (n_0)^{-1} \bmod n_1 \\ \tilde{A}_2 \leftarrow ((\tilde{A}_2 - \tilde{A}_0 - \tilde{A}_1 \times n_0) \times (n_0 \times n_1)^{-1} \bmod n_2 \\ \qquad \vdots \\ \tilde{A}_{v-1} \leftarrow (\tilde{A}_{v-1} - \tilde{A}_0 - n_0 \times (\tilde{A}_1 - n_1 \times (\tilde{A}_2 \ldots - n_{v-3} \times \tilde{A}_{v-2}))) \times (n_0 \times \ldots \times n_{v-2})^{-1} \bmod n_{v-1} \end{cases}$$

Une fois l'ensemble des $\tilde{A}_i$ calculé, il suffit de reconstruire A à partir de l'équation 5.1 (qui peut être réécrite en suivant en schéma d'Hörner) en y appliquant les moduli de la nouvelle base $n'_0, n'_1, \ldots, n'_{v-1}$. Cette approche permet de ne pas avoir (dans GF(p)),

comme c'est le cas dans la reconstruction de A par le théorème des restes chinois, l'apparition de la quantité α. En effet, si nous reconstruisons A par le théorème des restes chinois [77], nous avons :

$$A = \sum_{i=0}^{v-1} \left| \tilde{A}_i \times N_i^{-1} \right|_{n_i} \times N_i - \alpha \times p.$$

Ce $\alpha \times p$ permet au système de rester dans la dynamique de la représentation RNS et de ne pas « déborder ». Ce α disparait dans GF(2^m) puisque la « non propagation » de retenues nous permet de nous assurer de la cohérence du système RNS (voir équation 5.2).

$$A = \sum_{i=0}^{v-1} \left| \tilde{A}_i \times N_i^{-1} \right|_{n_i} \times N_i - \cancel{\alpha \times f} \tag{5.2}$$

Utiliser le théorème chinois des restes pour l'extension de base apparait être une excellente solution dans GF(2^m). Il suffit de reconstruire A à partir de l'équation 5.2 et d'y appliquer les modulis $n'_0, n'_1, \ldots, n'_{w-1}$ afin de transposer A dans sa nouvelle base $\mathcal{B}'$.

5.2 Φ-RNS

5.2.1 Quelques notions mathématiques

Pour introduire le concept du Φ-RNS, nous aurons besoin de quelques définitions et propriétés mathématiques. Nous définirons tout d'abord l'endomorphisme qui à un polynôme en x à coefficients dans GF(2) associe le polynôme obtenu par le changement de variable $x \leftarrow x + 1$.

Definition 1. Définissons l'endomorphisme Φ suivant :

$$\begin{array}{rcccc} \Phi & : & \mathrm{GF}(2)[x] & \rightarrow & \mathrm{GF}(2)[x] \\ & & P(x) & \mapsto & P(x+1) \end{array}$$

L'application de Φ peut se faire par l'application de la matrice A_n sur la représentation vectorielle de l'élément P. La matrice A_n est définie par récurrence par la formule :

$$A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix}$$

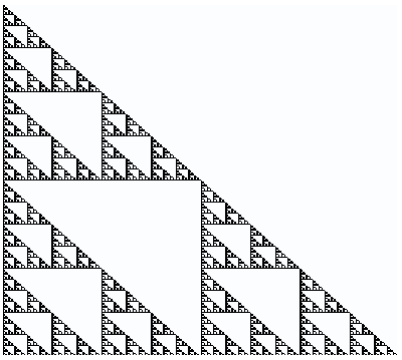


FIGURE 5.1: Matrice A_8, un pixel noir correspond à la valeur 1 et un pixel blanc à la valeur 0.

Avec $A_0 = 1$. Par exemple, nous avons

$$A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix},$$

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

La taille de la matrice A_i est toujours une matrice carrée de côté 2^i.

Quand n tend vers l'infini A_∞ est un espace fractal de type « triangle de Sierpinsky » [79]. La taille des éléments n'est pas toujours une puissance de 2, il suffit alors de tronquer les colonnes et les lignes inutiles. Pour illustrer cela, si nous travaillons avec GF(2^3), nous pouvons reprendre la matrice A_2 et supprimons la dernière colonne et la dernière ligne.

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & \not{0} \\ 1 & 1 & 0 & \not{0} \\ 1 & 0 & 1 & \not{0} \\ \not{1} & \not{1} & \not{1} & \not{1} \end{pmatrix}$$

Si nous voulons calculer $\Phi(x+1)$, nous posons

$$\Phi(x+1) = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = x.$$

Nous pouvons aussi nous servir de cette relation de récurrence pour trouver un compromis quant à l'application de Φ. En l'occurrence, nous pouvons n'avoir qu'un circuit ou qu'un fragment de code spécialisé dans l'application de A_{n-1}. Les vecteurs u_0 et u_1 correspondent respectivement aux $m/2$ bits de poids forts et aux $m/2$ bits de poids faibles d'un élément P (avec m une puissance de 2).

$$\Phi(P) = \begin{pmatrix} u_0 & u_1 \end{pmatrix} \times \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix} = \begin{pmatrix} v_0 = A_{n-1} \times (u_1 + u_0) \\ v_1 = A_{n-1} \times u_1 \end{pmatrix}$$

Proposition 5.1. *Pour tout $P \in GF(2)[x]$, $\Phi(\Phi(P)) = P$.*

Démonstration. Nous avons $\Phi(P) \Leftrightarrow \Phi(P(x)) = P(x+1)$. Enfin, $\Phi(P(x+1)) = P(x)$. □

Proposition 5.2. *Si $P \in GF(2)[x]$ est irréductible alors $\Phi(P)$ l'est aussi.*

Démonstration. Raisonnons par l'absurde. Si $\Phi(P)$ était réductible, alors $\Phi(P) = S \times T$ où S, T appartiennent à GF(2)[x]. Alors, nous avons :

$$\begin{aligned} \Phi(\Phi(P)) &= \Phi(S \times T) \\ &= \Phi(S) \times \Phi(T). \end{aligned}$$

Or, d'après la proposition 5.1, $\Phi(\Phi(P)) = P$, nous avons donc $P = \Phi(T) \times \Phi(S)$. Le polynôme P n'est donc pas irréductible, ce qui est absurde puisque cela contredit l'hypothèse. □

Proposition 5.3. *Une généralisation de la proposition précédente est de dire que si P et Q sont premiers entre eux, $\Phi(P)$ et $\Phi(Q)$ le seront aussi.*

Démonstration. Raisonnons par l'absurde. Si $\Phi(P)$ et $\Phi(Q)$ n'étaient pas premiers entre eux, cela signifierait qu'il existe un diviseur commun $D \neq 1$ tel que $\Phi(P) = z_0 \times D$ et $\Phi(Q) = z_1 \times D$. En appliquant Φ, nous trouvons $P = \Phi(z_0) \times \Phi(D)$ et $Q = \Phi(z_1) \times \Phi(D)$ et donc que P et Q ont $\Phi(D)(\neq 1)$ comme diviseur commun. Cela est absurde par hypothèse sur P et Q. □

Definition 2. Soit L_P l'ensemble des polynômes de GF(2)[x] que Φ laisse invariant.

$$L_P = \{P \in \mathrm{GF}(2)[x], \Phi(P) = P\}$$

Exemple 5.2.1. $x^6+x^5+x^4+x^3 \in L_P$. En effet, $\Phi(x^6+x^5+x^4+x^3) = (x+1)^6+(x+1)^5+(x+1)^4+(x+1)^3 = (x^6+x^4+x^2+1)+(x^5+x^4+x+1)+(x^4+1)+(x^3+x^2+x+1) = x^6+x^5+x^4+x^3$.

Proposition 5.4. *L_P est un sous-anneau généré par $1, s, s^2, s^3, \ldots$ où $s = x^2+x$. Autrement dit, les éléments de L_P sont les polynômes de la forme $\sum_{i=0}^{d} a_i \times s^i$ avec $a_i \in GF(2)$.*

Proposition 5.5. *Représentation Linéaire*

Tout polynôme dans $GF(2)[x]$ peut s'écrire comme

$$P = \mathcal{L}_0 + x \times \mathcal{L}_1$$

où $\mathcal{L}_0, \mathcal{L}_1 \in L_P$.

Démonstration. Soit $\mathcal{L}_1 = \Phi(P) + P$. Il est évident que $\mathcal{L}_1$ appartient à L_P. Soit $\mathcal{L}_0 = P + x \times \mathcal{L}_1$. Alors :

$$\begin{aligned}
\Phi(\mathcal{L}_0) &= \Phi(P + x \times \mathcal{L}_1) \\
&= \Phi(P) + (x+1) \times \mathcal{L}_1 \\
&= \Phi(P) + x \times \mathcal{L}_1 + \mathcal{L}_1 \\
&= \Phi(P) + x \times \mathcal{L}_1 + \Phi(P) + P \\
&= P + x \times \mathcal{L}_1 \\
&= \mathcal{L}_0
\end{aligned}$$

Cela signifie que $\mathcal{L}_0$ appartient aussi à L_P. Ainsi, nous obtenons $P = \mathcal{L}_0 + x \times \mathcal{L}_1$ ce qui conclut la preuve. □

Exemple 5.2.2. Soit $P = x^7 + x$. Calculons $\mathcal{L}_1 = \Phi(x^7+x) + (x^7+x) = x^6+x^5+x^4+x^3+x^2+x$. Maintenant, $\mathcal{L}_0 = P + x \times \mathcal{L}_1 = x^6+x^5+x^4+x^3+x^2+x$. Ainsi, $P = (x^6+x^5+x^4+x^3+x^2+x) + x \times (x^6+x^5+x^4+x^3+x^2+x)$. Chacun vérifiera que $\mathcal{L}_0$ et $\mathcal{L}_1$ appartiennent à L_P.

Proposition 5.6. *Soit $\mathcal{B} = \{n_0, n_1, \ldots, n_{v-1}\}$ une base* RNS*, c'est à dire que les n_i sont premiers entre eux. Si aucun diviseur D de n_i n'implique que $\Phi(D)$ soit diviseur d'un autre n_j, alors $\Phi(\mathcal{B}) = \{\Phi(n_0), \Phi(n_1), \ldots, \Phi(n_{v-1})\}$ est également une base* RNS *première à $\mathcal{B}$. Aussi, nous noterons $N = \prod_{i=0}^{v-1} n_i$ et $N' = \prod_{i=0}^{v-1} \Phi(n_i)$.*

Démonstration. Considérons le polynôme n_0 ; n_0 est par définition premier aux autres n_i. Nous avons donc $\Phi(n_0)$ premiers avec les autres $\Phi(n_i)$ de par la proposition 5.3. Il faut

maintenant vérifier que $\Phi(n_0)$ est premier avec les n_i ($i \neq 0$). S'il n'était pas premier, cela voudrait dire qu'il existe un j tel que $\Phi(n_0)$ et n_j aient un diviseur commun D. Autrement dit que $\Phi(n_0) = z_0 \times D$ et $n_j = z_1 \times D$. Cela nous dit que aussi que $n_0 = \Phi(z_0) \times \Phi(D)$. Or, par hypothèse, $\Phi(D)$ ne doit diviser aucun des n_i (avec $j \neq i$). Cela est absurde et donc $\Phi(n_0)$ est premier avec les n_i. Nous pouvons reprendre le raisonnement pour chacun des n_i pour conclure la preuve. □

En pratique, trouver des n_i respectant ces contraintes est facile. Nous avons codé un algorithme en *Maple* qui nous a toujours donné des dizaines de solutions pour les moduli de degrés 28 ou 32 (en une poignée de secondes sur un ordinateur de bureau standard).

Nous rappellerons ici la définition d'extension de base auparavant évoquée dans 5.1.3.

Definition 3. Extension de Base
Une extension de base est une fonction qui à $P \in \mathrm{GF}(2^m)[x]$ exprimé en base $\mathcal{B}$ ($P_{\mathcal{B}}$) associe le même polynôme P exprimé dans la base $\mathcal{B}'$ ($P_{\mathcal{B}'}$).

$$\mathbf{BE}(P_{\mathcal{B}}, \mathcal{B}, \mathcal{B}') \rightarrow P_{\mathcal{B}'}$$

Proposition 5.7. *Théorème chinois des restes pour les polynômes*
Soit $P \in GF(2)[x]$ *tel que* $\mathrm{Deg}(P) < \sum_{i=0}^{v-1} \mathrm{Deg}(n_i)$.
Considérons $P = (\tilde{p}_0, \tilde{p}_1, \ldots, \tilde{p}_{v-1})$ *où* $\tilde{p}_i = P \mod n_i$. *Alors*

$$P = \sum_{i=0}^{v-1} |\tilde{p}_i \times N_i^{-1}|_{n_i} \times N_i$$

où $N_i = \prod_{j \neq i} n_i$.

Proposition 5.8. *Nous avons que pour tout* $A, B, N \in GF(2)[x]$ *alors*

$$\Phi(\Phi(A) \times \Phi(B) \mod N) = A \times B \mod \Phi(N)$$

Démonstration. Nous avons $\Phi(A) \times \Phi(B) \mod N = \Phi(A) \times \Phi(B) + q \times N$.
D'où $\Phi(\Phi(A) \times \Phi(B) \mod N) = \Phi(\Phi(A) \times \Phi(B) + q \times N) = A \times B + \Phi(q) \times \Phi(N) = A \times B \mod \Phi(N)$ □

Exemple 5.2.3. Posons $A = x^2 + 1$ et $B = x + 1$ ainsi que $N = (x^3 + x + 1)$.
Nous avons d'une part $A \times B \mod N = x^2$.
D'autre part, nous avons $\Phi(A) \times \Phi(B) \mod \Phi(N) = (x^2) \times (x) \mod x^3 + x^2 + 1 = x^3 = x^2 + 1 = \Phi(x^2)$.

L'idée, derrière la proposition 5.8, est qu'un multiplieur optimisé et spécialisé dans la multiplication modulo N (dans la base $\mathcal{B}$) peut également être utilisé pour effectuer des calculs modulo $\Phi(N)$ (dans la base $\mathcal{B}'$), en appliquant aux opérandes et au résultat le morphisme Φ. C'est notamment ce qui nous intéresse dans le cadre d'une extension de base dans laquelle deux modes de calculs sont nécessaires (modulo N et N'). Nous proposons donc de réécrire l'algorithme de réduction de Montgomery-RNS en intégrant ce fait, en tentant de supprimer l'emploi d'une seconde base. Il n'y aura donc qu'une seule base $\mathcal{B}$ mais nous ferons subir, selon les besoins, la transformation Φ aux opérandes. L'algorithme est décrit dans 20. La ligne $\mathbf{BE}(D, \Phi(D))$ est une extension de base un peu différente. Il s'agit connaissant D dans la base $\mathcal{B}$ d'exprimer $\Phi(D)$ dans cette même base $\mathcal{B}$. Cette étape est réalisée à travers le théorème des restes chinois.

Algorithme 20 : Multiplication de Montgomery en ≪ Φ-RNS ≫.

Données : $A, \Phi(A)$ et $B, \Phi(B)$ en base $\mathcal{B}$
Résultat : $A \times B \times N^{-1} \mod f$ dans les bases $\mathcal{B}$ et $\mathcal{B}'$

1 $C_0 \leftarrow A \times B \mod N$
2 $C_1 \leftarrow \Phi(A) \times \Phi(B) \bmod N$ $\Leftrightarrow A \times B \bmod \Phi(N)$
3 $D \leftarrow C_0 \times f^{-1} \mod N$
4 $D \leftarrow \mathbf{BE}(D, \Phi(D))$
5 $D \leftarrow D \times \Phi(f) \mod N$ $\Leftrightarrow (A \times B \times f^{-1} \bmod N) \times f \bmod \Phi(N)$
6 $D \leftarrow D + C_1$ $\Leftrightarrow (A \times B \times f^{-1} \bmod N) \times f + A \times B \bmod \Phi(N)$
7 $D \leftarrow D \times \Phi(N)^{-1} \mod N$ $\Leftrightarrow ((A \times B \times f^{-1} \bmod N) \times f + A \times B) \times N^{-1} \bmod \Phi(N)$
8 $E \leftarrow \mathbf{BE}(D, \Phi(D))$
9 **retourner** (E, D)

Exemple 5.2.4. Appliquons l'algorithme de ≪ Φ-RNS ≫ à travers un exemple dans GF(2^5). Considérons le polynôme de réduction $f = x^5 + x^3 + 1$ (qui est bien irréductible). Considérons la base RNS $\mathcal{B} = \{x^3, x^3 + x + 1\}$. Nous avons $f^{-1} \bmod N = x^3 + 1$ et $\Phi(N)^{-1} \bmod N = x^4 + x^3 + x^2 + x + 1$. Nous souhaitons multiplier $A = x^3 + 1$ et $B = x^2$ en ≪ Φ-RNS ≫. La première chose à faire est de convertir A et $\Phi(A)$ dans la base $\mathcal{B}$. De même pour B et $\Phi(B)$. Nous avons donc $A = (1, x), \Phi(A) = (x^2 + x, x^2 + 1)$, $B = (x^2, x^2)$ et $\Phi(B) = (x^2 + 1, x^2 + 1)$. Nous déroulons l'algorithme :

1. $C_0 \leftarrow A \times B \bmod N = (1, x) \times (x^2, x^2) = (x^2, x + 1)$
2. $C_1 \leftarrow \Phi(A) \times \Phi(B) \bmod N = (x^2 + x, x^2 + 1) \times (x^2 + 1, x^2 + 1) = (x^2 + x, x^2 + x + 1)$
3. $D \leftarrow (C_0 \times f^{-1}) \bmod N = (x^2, x + 1) \times (1, x) = (x^2, x^2 + x)$
4. $D \leftarrow \mathbf{BE}(D, \Phi(D))$ Nous reconstruisons $\Phi(D)$ à partir du théorème chinois des restes : $\Phi(D) = \Phi(|(x^2) \times (1 + x + x^2)|_{n_0} \times n_1 + |(x^2 + x) \times (x + x^2)|_{n_1} \times n_0)$. Nous avons donc $\Phi(D) = (0, x^2)$.
5. $D \leftarrow D \times \Phi(f) \mod N = (0, x^2) \times (x^2 + 1, x^2 + x + 1) = (0, x^2 + x + 1) = (0, 1)$
6. $D \leftarrow D + C_1 = (x^2 + x, x^2 + x + 1) + (0, 1) = (x^2 + x, x^2 + x)$

7. $D \leftarrow (D \times \Phi(N)^{-1} \bmod N) = (x^2+x, x^2+x) \times (x^2+x+1, x) = (x, x^2+x+1)$
8. $E \leftarrow \mathbf{BE}(D, \Phi(D)) = (1+x^2, 0)$
9. Retourner (E, D)

La valeur de E de l'exemple 5.2.5 correspond au polynôme $x^4+x^3+x^2+1$ qui est bien égal à $A \times B \times N^{-1} \bmod f$. Nous toucherons un petit mot sur la nouvelle extension de base présente à la ligne 4 et 8. Le but de cette opération est de représenter pour $A \in \mathrm{GF}(2)[x]$ la quantité $\Phi(A)$ exprimée dans la base $\mathcal{B}$. Il s'agit donc d'évaluer (pour chaque j compris entre 0 et $v-1$) la quantité $\Phi(A) \bmod n_j = \Phi(\sum_{i=0}^{v-1} |\tilde{a}_i \times N_i^{-1}|_{n_i} \times N_i) \bmod n_j = \Phi(\sum_{i=0}^{v-1} |\tilde{a}_i \times N_i^{-1}|_{n_i}) \times \Phi(N_i) \bmod n_i$. Le lecteur remarquera que tous les calculs de l'algorithme de ≪ Φ-RNS ≫ 20 sont réalisés modulo N (et donc dans la première base $\mathcal{B}$). Implémenter cet algorithme ne requiert qu'un multiplieur spécialisé pour la base $\mathcal{B}$ et une portion de circuit s'occupant de l'application de Φ sur les opérandes. Comme nous le verrons, le circuit s'attelant à cette tâche est petit, comparativement à la taille d'un multiplieur. L'algorithme 20 ne fonctionne que dans les conditions explicitées à la proposition 5.6, cela assure notamment que l'inverse de $\Phi(N)$ existe modulo N.

Nous avons donc un algorithme de multiplication modulaire en RNS mono-base. Cela a au moins deux avantages évidents :
– Cela permet notamment de réduire le nombre de constantes à stocker (nous divisons ce nombre par deux face à une représentation RNS standard).
– Cela permet d'avoir des multiplieurs spécialisés et très rapides dans une **seule base**. Avoir des multiplieurs spécialisés dans deux bases signifierait de devoir doubler, probablement, la surface silicium leur étant allouée.

5.2.2 Architecture proposée

Nous avons proposé une architecture implémentant l'algorithme 20 ≪ Φ-RNS ≫. Elle est composée de différents canaux (généralement appelés *rower* en anglais [80]). Chacun de ces canaux est chargé du support de la l'addition et de la multiplication modulo un n_i. Les tailles de nos canaux se doivent d'être identiques pour garantir le bon fonctionnement de l'architecture. Dans les faits, nous avons choisi des canaux de 28 ou 32 bits afin d'avoir une taille de mots correspondant à ceux gérés par les blocs mémoires de type BRAM — présents sur le FPGA— mais aussi afin de limiter la complexité des blocs multiplicatifs de Mastrovito.

Le canal (*rower*) de la figure 5.2 que nous proposons possède deux étages de calculs. Un étage dédié à la multiplication modulo n_i et un autre étage, quant à lui, dédié à l'application du morphisme Φ. Toutes les unités de mémoire ne sont pas indiquées sur le schéma pour une question de lisibilité. Notons que Φ est appliqué après chacune des

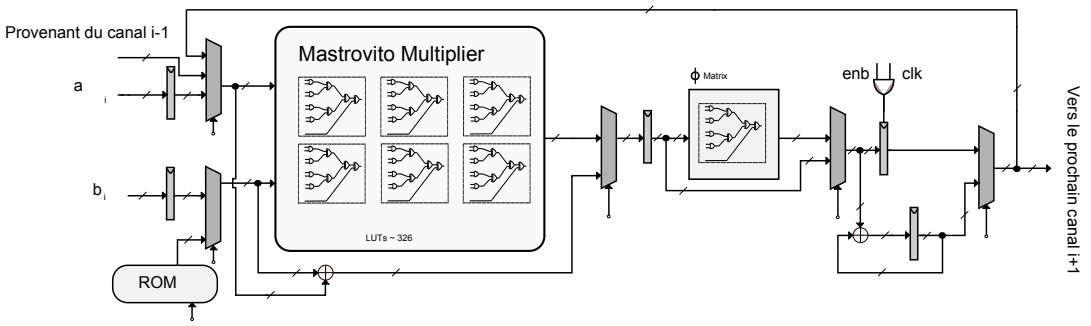


FIGURE 5.2: Architecture de notre canal.

deux extensions de base et est, finalement, assez peu usité tout au long des différentes étapes composant la multiplication RNS de Montgomery. Cette portion du circuit est d'ailleurs relativement petite puisqu'elle n'occupera qu'une trentaine de LUTs comparée aux $\approx$ 350 LUTs pour un multiplieur de Mastrovito (sur un canal de 32 bits). L'étage se chargeant du calcul modulo n_i est basé sur une matrice de Mastrovito, qui permet de multiplier deux éléments et de réduire le produit en un nombre réduit d'étapes. En règle générale, le procédé proposé par Mastrovito peut être décomposé en deux étapes. À un élément $B \in \mathrm{GF}(2^w)$, nous associons la matrice M_B à laquelle nous viendrons multiplier, par le biais d'un produit matrice-vecteur, l'élément A interprété comme vecteur. Le résultat de cette opération sera le produit $A \times B \mod n_i$. La construction de M_B n'est pas forcément faite en un seul cycle d'horloge, elle peut prendre plus ou moins de temps selon le matériel que le concepteur lui aura dédié. Danuta Pamula a suggéré, entre autres, elle aussi lors de sa thèse [28], de découper la matrice de Mastrovito en un ensemble de sous-matrices. Un nombre important de ces matrices sont identiques ce qui permet d'économiser grandement de la surface de silicium au détriment d'une exécution légèrement plus lente. En ce qui nous concerne, la taille du corps dont le canal a à se charger est relativement faible (une trentaine de bits) ce qui permet, à moindre coût, que cela soit en termes de surface et/ou en fréquence, de combiner la construction de M_B et la multiplication vecteur-matrice en un seul cycle d'horloge. Nous avons donc un canal qui calcule une multiplication modulaire en deux cycles : les opérandes passeront en premier lieu par le multiplieur de Mastrovito dont la sortie, sera, ou non, traitée par Φ. Nous noterons la présence d'un accumulateur dans la figure 5.2 ; celui-ci va notamment être utilisé lors de l'extension de base **BE**. La mémoire ROM permet de stocker les constantes comme $f^{-1} \mod n_i$, $\Phi(N)^{-1} \mod n_i$ ainsi que l'ensemble des constantes $N_i^{-1} \mod n_i$ et $\Phi(N_i)$ présentes lors d'une extension de base **BE**. Nous avons aussi modifié l'algorithme de multiplication « Φ-RNS » pour tenir compte de la présence de deux étages dans le canal. En organisant les opérations comme nous le suggérons dans l'algorithme 21, il est possible d'avoir un taux d'utilisation du pipeline plus important. Il y a deux idées primordiales à saisir. À la ligne 3, nous multiplions C_1 par $(f^{-1} \times S)$. Cette constante S

correspond à la constante $(N_0^{-1}, N_1^{-1}, \ldots, N_{v-1}^{-1})$ en représentation RNS. Cela permet, en avance de phase, d'effectuer les multiplications par les N_i^{-1}, nécessaires dans une extensions de base **BE**. La valeur de $f^{-1} \times S$ est donc pré-calculée. Ces pré-calculs rendent l'extension de base **BE** plus rapide puisque les $\Phi(\left|\tilde{D}_i \times N_i^{-1}\right|_{n_i})$ seront directement accessibles (chaque canal ayant calculé et mémorisé cette donnée : cette valeur est d'ailleurs stockée dans une bascule. Il suffit ensuite de faire circuler ces quantités (illustré dans la figure 5.4) pour que chaque canal j ait accès à chacun de ces $\Phi(\hat{D}_i) = \Phi(\left|\tilde{D}_i \times N_i^{-1}\right|_{n_i})$ et ainsi user du théorème chinois des restes pour évaluer $(\sum_{i=0}^{v-1} \Phi(\hat{D}_i) \times \Phi(N_i))$ mod n_j. Nous avons choisi de distribuer circulairement les données au sein de circuit comme le montre le schéma de l'architecture 5.3.

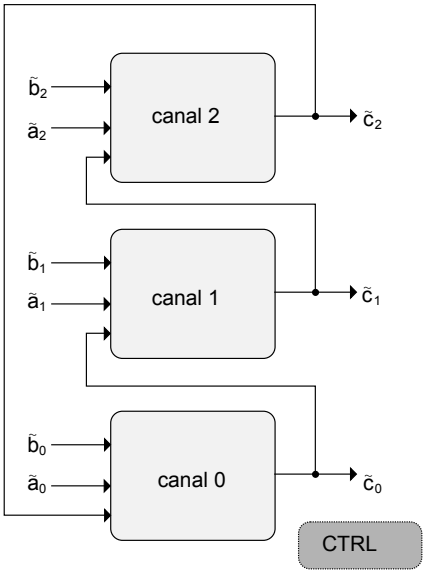


FIGURE 5.3: Vue haut niveau de notre architecture (multiplieur) Φ-RNS.

Ainsi, si nous ne nous intéressons qu'au premier canal, celui-ci calculera au préalable $\Phi(\hat{D}_{v-1}) \times \Phi(N_{v-1})$ mod n_0 puis $\Phi(\hat{D}_{v-2}) \times \Phi(N_{v-2}$ mod $n_0)$ ... Le second canal lui calculera premièrement $\Phi(\hat{D}_0) \times \Phi(N_0$ mod $n_1)$ puis $\Phi(\hat{D}_{v-1}) \times \Phi(N_{v-1})$ mod n_1, ... Un exemple sur trois canaux est donné dans le tableau 5.1. Cette architecture en anneau a notamment été proposée dans [11]. Cela permet d'éviter la présence de gros multiplixeurs normalement requis pour faire circuler les données entre les canaux dans une approche plus standard.

Nous avons donc un pipeline à deux étages que nous remplissons de la façon suivante (chaque chiffre i se rapporte à ligne i de l'algorithme 21). Dans l'algorithme 21, l'extension $\mathbf{BE}'$ est une fonction dont le but est aussi de calculer $\Phi(D)$ dans la base $\mathcal{B}$. Cela dit, nous avons multiplié, avant cette extension de base les quantités $\hat{D}_i = |\tilde{D}_i \times N_i^{-1}|_{n_i}$, cette extension n'a, de ce fait, plus qu'à calculer les produits $\Phi(\hat{D}_i) \times \Phi(N_i)$ mod n_j et à venir accumuler ces valeurs.

Algorithme 21 : Algorithme de multiplication Φ-RNS ré-ordonnancé pour profiter des deux étages de pipeline du canal.

Données : $A, \Phi(A)$ et $B, \Phi(B)$ en base $\mathcal{B}$
Résultat : $A \times B \times N^{-1} \mod f$ et $\Phi(A \times B \times N^{-1} \mod f)$ dans la base $\mathcal{B}$

1 $C_0 \leftarrow A \times B \mod N$
2 $C_1 \leftarrow \Phi(A) \times \Phi(B) \mod N$
3 $D \leftarrow C_0 \times (f^{-1} \times S) \bmod N$
4 $D \leftarrow \mathbf{BE}'(D, \Phi(D))$
5 $D \leftarrow D \times (\Phi(f)) \mod N$
6 $D \leftarrow (D + C_1) \mod N$
7 $D \leftarrow (D \times \Phi(N)^{-1}) \mod N$
8 $E \leftarrow D \times S \mod N$
9 $E \leftarrow \mathbf{BE}'(E, \Phi(E))$
10 **retourner** (E, D)

	Cycle de multiplication 0	Cycle de multiplication 1	Cycle de multiplication 2
canal 0	$\hat{D}_2 \times N_2 \bmod n_0$	$\hat{D}_1 \times N_1 \bmod n_0$	$\hat{D}_0 \times N_0 \bmod n_0$
canal 1	$\hat{D}_0 \times N_0 \bmod n_1$	$\hat{D}_2 \times N_2 \bmod n_1$	$\hat{D}_1 \times N_1 \bmod n_1$
canal 2	$\hat{D}_1 \times N_1 \bmod n_2$	$\hat{D}_0 \times N_0 \bmod n_1$	$\hat{D}_2 \times N_2 \bmod n_1$

TABLE 5.1: Un exemple d'ordonnancement de la base d'extension **BE** sur trois canaux.

Ét. 1	1	2	3	X	$\mathbf{BE}_0$	...	$\mathbf{BE}_{v-1}$	X	5	X
Ét. 2	X	1	2	3	X	$\mathbf{BE}_0$	...	$\mathbf{BE}_{v-1}$	X	5
Ét.1	6	X	7	X	8	X	$\mathbf{BE}_0$	...	$\mathbf{BE}_{v-1}$	X
Ét.2	X	6	X	7	X	8	X	$\mathbf{BE}_0$	...	$\mathbf{BE}_{v-1}$

TABLE 5.2: Remplissage du pipeline d'un canal.

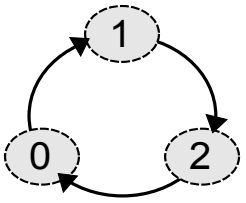


FIGURE 5.4: Architecture en anneau pour l'extension de base.

Dans le tableau 5.2, Ét.1 correspond aux calculs en cours dans le premier étage du canal et Ét.2 correspond aux calculs du second. La notation X permet de signifier la présence d'une bulle dans le pipeline. Les annotations $\mathbf{BE}_i$ coïncident à l'ensemble des calculs nécessaires pour l'extension de base **BE**. En effet, il faut faire circuler l'ensemble des $\hat{D}_i$ antérieurement calculés dans chacun des canaux. Cela prend $v + 1$ cycles où v est le nombre de canaux. Nous avons donc un algorithme qui nécessite $2 \times (v + 1) + 12$ cycles avant de renvoyer le résultat (E, D).

Exemple 5.2.5. Appliquons l'algorithme de « Φ-RNS » 21 à travers le même exemple que précédemment. Considérons le polynôme de réduction $f = x^5 + x^3 + 1$ (qui est bien

irréductible). Considérons la base RNS $\mathcal{B} = \{x^3, x^3 + x + 1\}$. Nous avons $f^{-1} \bmod N = x^3 + 1$ et $\Phi(N)^{-1} \bmod N = x^4 + x^3 + x^2 + x + 1$. Nous souhaitons multiplier $A = x^3 + 1$ et $B = x^2$ en « Φ-RNS ». La première chose à faire est de convertir A et $\Phi(A)$ dans la base $\mathcal{B}$. De même pour B et $\Phi(B)$. Nous avons donc $A = (1, x), \Phi(A) = (x^2 + x, x^2 + 1)$, $B = (x^2, x^2)$ et $\Phi(B) = (x^2 + 1, x^2 + 1)$. Nous déroulons l'algorithme :

1. $C_0 \leftarrow A \times B \bmod N = (1, x) \times (x^2, x^2) = (x^2, x + 1)$
2. $C_1 \leftarrow \Phi(A) \times \Phi(B) \bmod N = (x^2 + x, x^2 + 1) \times (x^2 + 1, x^2 + 1) = (x^2 + x, x^2 + x + 1)$
3. $D \leftarrow (C_0 \times (f^{-1} \times S)) \bmod N = (x^2, x + 1) \times (x^2 + x + 1, x^2 + x + 1) = (x^2, x)$
4. $D \leftarrow \mathbf{BE}'(D, \Phi(D))$ Nous reconstruisons $\Phi(D)$ à partir du théorème chinois des restes : $\Phi(D) = \Phi(x^2) \times \Phi(n_1) + \Phi(x + 1) \times \Phi(n_0) = (0, x^2)$. Nous avons donc $\Phi(D) = (0, x^2)$.
5. $D \leftarrow D \times \Phi(f) \mod N = (0, x^2) \times (x^2 + 1, x^2 + x + 1) = (0, x^2 + x + 1) = (0, 1)$
6. $D \leftarrow D + C_1 = (x^2 + x, x^2 + x + 1) + (0, 1) = (x^2 + x, x^2 + x)$
7. $D \leftarrow (D \times \Phi(N)^{-1} \bmod N) = (x^2 + x, x^2 + x) \times (x^2 + x + 1, x) = (x, x^2 + x + 1)$
8. $E \leftarrow D \times S = (x, x^2 + x + 1) \times (x^2 + x + 1, x^2 + x) = (x^2 + x, x^2)$
9. $E \leftarrow \mathbf{BE}'(D, \Phi(D))$. Nous avons donc $\Phi(x^2+x) \times \Phi(n_1) + \Phi(x^2) \times \Phi(n_0) = (1+x^2, 0)$
10. Retourner (E, D)

Nous sommes conscients que placer l'unité Φ dans le pipeline n'est sans doute pas la meilleure des approches. Nous pensons malgré tout que cela facilite le contrôle du canal et plus sommairement, le contrôle de l'architecture du multiplieur tout entier.

5.2.3 Résultats d'implémentation et comparaisons

Nous comparerons dans cette section notre architecture aux solutions de multiplication modulaire déjà existantes dans GF(2^m).

Nous avons trouvé deux architectures RNS implémentées sur FPGA [81] et [36] opérant dans GF(2^m). Dans ces papiers, les auteurs ont besoin de reconstruire P en base polynomiale pour effectuer la réduction modulaire (modulo f) avant de reconvertir le tout en représentation RNS. Il s'agit là d'une approche hybride : dans l'approche que nous proposons, toutes les quantités manipulées sont de tailles w, contrairement aux deux précédentes solutions, où des quantités de tailles m sont reconstruites. L'idée générale des deux précédentes implémentations [81] et [36] est, qu'il est « facile » dans GF(2^m) de n'obtenir que les bits de poids forts à partir du théorème chinois des restes. Les auteurs utilisent une base qui permet de gérer une dynamique de $2m$ bits. Il est donc multiplié deux éléments de GF(2^m), vus comme des polynômes de degrés $m - 1$. La multiplication produit donc un résultat de degré au plus $2m - 2$, ce qui ne provoque donc pas

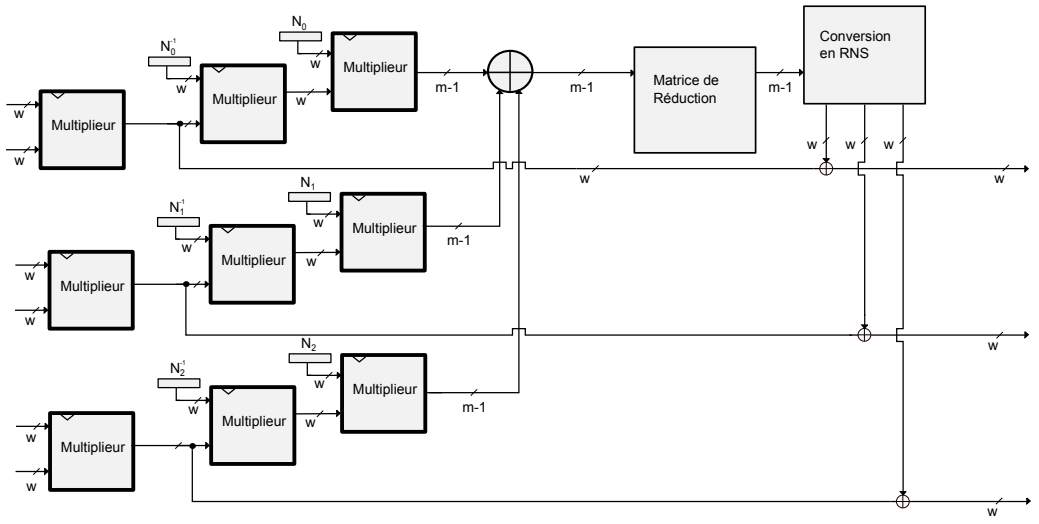


FIGURE 5.5: Architecture RNS hybride de [36].

de « dépassement » des capacités du système RNS. Une fois la multiplication effectuée, l'architecture reconstruit le polynôme correspondant (à partir du théorème chinois des restes) sur une longueur de $2 \times m - 1$ bits. Les poids forts de ce produit sont rapidement déductibles du théorème chinois des restes : il suffit de regarder dans la formule les termes qui produisent des monômes de degré supérieur ou égal à m). Il est donc extirpé les monômes de degrés m à $2 \times m - 2$ (nécessitant d'être ainsi réduits modulo f). Cette quantité est ensuite réduite par une matrice de réduction (la matrice M_R dont nous faisions référence en 5.1.2), entièrement réalisée par un arbre de `XOR`. Une fois réduite, cette valeur est de nouveau convertie en RNS et additionnée avec le produit $A \times B$ non réduit, déjà représenté en RNS. Nous présentons leur idée à travers la figure 5.5. Dans cet exemple, nous avons représenté 3 canaux. Les deux premières couches de multiplieurs (à gauche) calculent modulo n_i. La dernière, quant à elle, effectue la multiplication par N_i sur $2 \times m - 1$ bits, sauf que nous ne considérons que les degrés supérieurs ou égaux à m (les monômes qui auront besoin d'être réduits). La sortie s'effectue donc sur $m - 1$ bits. Il est difficile de nous comparer à leur architecture étant donnée l'utilisation de technologies vieillissantes (Spartan-3). Cela dit, le nombre de cycles d'horloge estimés pour une multiplication modulaire pour un corps de 163 bits (GF(2^{163})) est de 168, ce qui est bien au delà de ce que nos proposons ici (26 cycles) pour une surface qui est à priori totalement équivalente.

Dans le tableau 5.6, les chiffres représentent la surface (Slices, LUTs et *Flip-flops*) occupée par un multiplieur seul. Nous pouvons voir que nous obtenons un très bon produit temps-surface (PTS). Pour nous permettre une comparaison la plus exhaustive possible, nous choisissons la LUTs comme « unité de surface ». Plus le PTS est petit, plus l'architecture proposée est intéressante. Les chiffres ici présents ne comportent, cependant,

pas le matériel chargé de convertir un polynôme en représentation « Φ-RNS » (et vice-versa). Nous avons besoin pour $m = 163$ de 6 canaux de 28 bits, de 9 pour $m = 233$ et de 11 pour $m = 283$. Pour les tailles supérieures ($m = 409, 571$) nous avons choisi des canaux de 32 bits.

Nous avons implémenté notre architecture sur FPGA Xilinx Virtex-4 et Virtex-6 et donnons des résultats d'implémentation dans le tableau 5.6. Nous nous comparons à l'implémentation de l'algorithme de Mastrovito de [28] et d'un algorithme plus naïf de « Multiplication Matrice-Vecteur » (toujours issu de [28]). L'architecture proposée dans la thèse [82] est une architecture basée sur un algorithme de « décalages et additions » (*shift-and-add*) sur la représentation polynomiale des éléments de GF(2^m). Plutôt que de travailler bit par bit, le multiplicande est scanné mot par mot (dans leur cas, 32 bits par 32 bits). L'algorithme est explicité en 22. La valeur T_{32} est fixée à $\lceil m/32 \rceil$. Les opérandes A et B sont composés de mots de 32 bits qui concaténés forment les éléments respectifs en représentation polynomiale ; $A = (A_0, A_1, \ldots, A_{T_{32}-1})$ et $B = (B_0, B_1, \ldots, B_{T_{32}-1})$ avec A_i et B_i des mots de 32 bits.

Algorithme 22 : Multiplication « décalages et additions » de [82]

```
1 c ← 0
2 pour i de 0 à T_32 − 1 faire
3 |   c = c + B_i × a ;
4 |   a ← a × x^32 mod f
5 retourner c mod f
```

Leur algorithme de multiplication s'effectue donc en T_{32} itérations. Les auteurs ont implémenté une architecture sur GF(2^{283}). Ils choisissent un pentanomial $f = x^{283} + x^{12} + x^7 + x^5 + 1$ composé de monômes de faibles degrés. Cela permet une réduction rapide qu'ils estiment à 4 cycles avec leur méthode. Leur solution propose un PTS bien meilleur que notre solution RNS. Cela est en particulier lié au choix du polynome de réduction. L'un des vecteurs de recherche serait de trouver un polynôme de réduction qui permet de réduire/faciliter les calculs en Φ-RNS ; un « équivalent » des polynômes creux pour RNS dans GF(2^m).
Nous nous sommes aussi comparés à l'architecture proposée dans l'article [83]. Ici, les auteurs proposent une architecture complète pour réaliser une multiplication scalaire, le détail en slices, LUTs du multiplieur seul n'est pas donné. Cela dit, le nombre de cycles reste une information intéressante.

m	Type de multiplieur ($m = 163$)	Slices (LUTS, FFs)	Frq (Mhz) (Nb. Cycles)	PTS	FPGA Cible
163	MV Multiplication [28]	? (1050, ?)	520 (326 cycles)	658	V-6
	Phi RNS	1677 (3832, 2457)	300 (26 cycles)	332	V-6
	RNS-Hybride [36]	2752(?, ?)	? (168 cycles)	–	S-3
233	Mastrovito [28]	? (3760, ?)	295 (75 cycles)	930	V-6
	Phi RNS	2051 (5838, 3635)	294 (30 cycles)	591	V-6
283	Fast Reduction [82]	1781(3367, 2156)	246 (13 cycles)	177	V-4
	Phi RNS	5440 (10218, 4493)	164 (36 cycles)	2242	V-4
	Phi RNS	2475 (7067, 4421)	268 (36 cycles)	949	V-6
409	Phi RNS	2885(8231, 5208)	238 (40 cycles)	1383	V-6
	FFMULT [83]	?, (?, ?)	142 (181 cycles)	?	V-4
571	Phi RNS	6202(13394, 8345)	242 (50 cycles)	2767	V-6
	FFMULT [83]	?, (?, ?)	142 (332 cycles)	?	V-4

FIGURE 5.6: Comparaison de divers multiplieurs dans GF(2^m). Ici V-6 = Virtex-6, V-4 = Virtex-4 et S-3 = Spartan-3.

Nous souhaitions également estimer l'impact qu'aurait l'implémentation de notre multiplieur dans une multiplication scalaire $[k]P$ complète. Nous avons donc incorporé ce multiplieur dans une architecture similaire à celle décrite dans le chapitre 4 (voir figure 4.5). Évidemment, nous avons retiré les unités de résolution d'équation de $\lambda^2 + \lambda = c$ et de racines carrées. Tous les éléments sont représentés en « Φ-RNS ». Nous avons utilisé les coordonnées projectives pour calculer un produit scalaire $[k]P$. Les formules que nous avons utilisées sont présentées dans le tableau 5.3. Il est bon de remarquer que le schéma de calcul $A \times B + C$ apparait relativement souvent. Pour tenir compte de cet état de fait et éviter le transfert inutile de données, nous avons choisi d'adapter notre multiplieur RNS lui permettant ainsi de réaliser des calculs de ce type pour le même cout qu'une « simple » opération $A \times B$. Parce qu'il n'y a aucune propagation de retenues dans GF(2^m), ajouter deux éléments en RNS est facile : il ne sera pas nécessaire de réduire le résultat obtenu (en relançant par exemple, un processus, couteux, d'extension de base). Nous nous sommes comparés aux architectures issues de [84] et de [83]. Nous avons choisi la même cible FPGA afin de rendre cette comparaison valable (Virtex-4). Dans [84], Morales et al. présentent un co-processeur entièrement programmable (très similairement à ce que nous avons fait dans le chapitre 4). Ils utilisent, dans ce papier, des formules de courbes elliptiques unifiées provenant de [85]. Dans [83], les auteurs adoptent des coordonnées de Lopez-Dahab modifiées. Leur architecture (spécialisée pour ces coordonnées)

m	Cycles ADD	Cycles DBL	Frq (Mhz)	ADD (μs)	DBL (μs)	Mult. Scalaire (μs)
163	531	474	243	2,18	1,95	436,6
233	603	540	215	2,80	2,51	803,0
283	648	584	207	3,13	2,82	1093,7
409	747	672	206	3,62	3,26	1828,5
571	901	891	154	5,85	5,78	4417,2

TABLE 5.3: Détails des coûts d'une multiplication scalaire $[k]P$ sur notre architecture en 2-NAF.

est la même pour toutes les tailles de corps considérées ; le même circuit servira, en itérant bien sûr plusieurs fois, à calculer des multiplications dans GF(2^{163}) mais aussi dans GF(2^{571}). Les résultats sont reportés dans le tableau 5.4. Nous avons obtenu en moyenne des meilleurs chiffres que [84]. Malheureusement, malgré la rapidité de notre multiplication, nous ne parvenons pas à nous hisser aux performances de [83]. Dans [83], tout est « câblé » pour amorcer la même série de calculs. Nous avons détaillé dans 5.3 le temps requis pour un doublement de point, une addition de points.

Ce manque de performance est principalement lié à la structure de notre processeur (nous avons malgré tout une multiplication plus rapide que la leur basée sur un Karatsuba [86]). Nous avons à décoder l'instruction, charger les bonnes valeurs dans les unités fonctionnelles, etc. C'est d'ailleurs le vrai goulot d'étranglement de notre architecture : le transfert de données (chargement des opérandes et récupération du résultats) est quasiment aussi chronophage que le calcul lui-même. Pour avoir une architecture tirant profit de la rapidité de notre multiplieur RNS, il faudrait implémenter des bus de données plus conséquents (de l'ordre de 128 bits). Cela est, nous semble t-il, une piste à creuser. En conclusion, le « Φ-RNS » nous parait être une solution prometteuse pour le calcul dans GF(2^m). Nous n'avons pas su tirer profit de la rapidité de notre multiplieur dans un crypto-processeur complet. Augmenter la taille des bus permettrait surement d'atteindre un bien meilleur compromis temps/surface.

Addition	Doublement
$A = Y1 + Z1 \times Y2$	$A = X1^2$
$B = X1 + Z1 \times X2$	$B = A + Y1 \times Z1$
$AB = A + B$	$C = X1 \times Z1$
$C = B^2$	$BC = B + C$
$E = B \times C$	$D = C^2$
$F = (A \times AB + a \times C) \times Z1 + E$	$E = B \times BC + a \times D$
$X3 = B \times F$	$X3 = C \times E$
$Y3 = C \times (A \times X1 + B \times Y1) + AB \times F$	$Y3 = BC \times E + A^2 \times C$
$Z3 = E \times Z1$	$Z3 = C \times D$

(5.3)

Arch.	m	Slices (LUTs,FF)	Frq (Mhz)	temps (ms)	PTS
[84] (programmable)	163	3034 (?, ?)	87	1.07	3246
	233	4236 (?, ?)	78	2.11	8937
	283	5743 (?, ?)	84	3.18	18262
[83]	163	2648 (?, ?)	142	0.483	1278
	233	2648 (?, ?)	142	1.093	2894
	283	2648 (?, ?)	142	1.404	3717
	409	2648 (?, ?)	142	3.861	10233
	571	2648 (?, ?)	142	9.208	24382
Notre architecture : 2-NAF	163 (28 bits)	3514 (5102, 2743)	243	0.436	1534
	233 (28 bits)	5547 (8118, 4468)	215	0.803	4454
	283 (28 bits)	6815 (10009, 5568)	207	1.09	7453
	409 (32 bits)	9479 (13468, 7688)	206	1.82	17333
	571 (32 bits)	10238 (19475, 10911)	154	4.41	45233
Notre architecture : 3-NAF	163 (28 bits)	3514 (5102, 2743)	243	0.406	1430
	233 (28 bits)	5547 (8118, 4468)	215	0.748	3344
	283 (28 bits)	6815 (10009, 5568)	207	1.01	5678
	409 (32 bits)	9479 (13468, 7688)	206	1.70	13107
	571 (32 bits)	10238 (19475, 10911)	154	4.13	45158

TABLE 5.4: Comparaisons de différentes architectures permettant le calcul d'un $[k]P$ sur Virtex-4.

5.2.4 Racine carrée en RNS

Dans cette partie nous présenterons un algorithme d'extraction de racines carrées en Φ-RNS. Nous n'avons pas implémenté l'opérateur sur FPGA.

Dans GF(2^m), la racine carrée est un opérateur linéaire. Ce phénomène est directement relié au fait que la mise au carré est, elle aussi, linéaire. Explicitement, nous avons pour tout $A \in \mathrm{GF}(2^m)$ et pour tout $B \in \mathrm{GF}(2^m)$ l'égalité suivante :

$$\sqrt{A+B} = \sqrt{A} + \sqrt{B}$$

Si nous écrivons A en base polynomiale, c'est à dire, sous la forme, $A = \sum_{i=0}^{m-1} a_i \times x^i$, nous pouvons appliquer la relation précédente pour obtenir :

$$\begin{aligned}
\sqrt{A} &= \sqrt{\textstyle\sum_{i<m \text{ pair}} a_i \times x^i + \sum_{j<m \text{ impair}} a_j \times x^j} \\
&= \sqrt{\textstyle\sum_{i<m \text{ pair}} a_i \times x^i} + \sqrt{\textstyle\sum_{j<m \text{ impair}} a_j \times x^j} \\
&= \textstyle\sum_{i\leq(m-1)/2} a_{2\times i} \times x^i + (\sum_{j<(m-1)/2} a_{2\times j+1} \times x^j) \times \sqrt{x}
\end{aligned}$$

Cela signifie que pour réaliser une racine carrée, il suffit de pouvoir extraire de A les coefficients d'indices pairs représentant l'élément $\sqrt{\mathcal{P}} = \sum_{i \leq (m-1)/2} a_{2\times i} \times x^i \in \mathrm{GF}(2^m)$, ainsi que les coefficients d'indices impairs représentant, quant à eux, l'élément $\sqrt{\mathcal{I}} = \sum_{j<(m-1)/2} a_{2\times j+1} \times x^j \in \mathrm{GF}(2^m)$. Une fois chaque quantité connue, nous appliquons la formule :

$$\sqrt{A} = \sqrt{\mathcal{P}} + \sqrt{\mathcal{I}} \times \sqrt{x}\,^{\dagger}$$

Exemple 5.2.6. Nous cherchons à extraire la racine carrée de $A = x^5 + x + 1 \bmod x^7 + x + 1$. Nous réécrivons A comme $x \times \underbrace{(x^4+1)}_{\mathcal{I}} + \underbrace{1}_{\mathcal{P}}$.Nous pouvons dès lors poser $\sqrt{A} = \sqrt{x \times (x^4+1) + 1} = \sqrt{x} \times \sqrt{x^4+1} + \sqrt{1} = \sqrt{x} \times (x^2+1) + 1$. Nous avons pré-calculé $\sqrt{x} \bmod x^7 + x + 1 = x^4 + x$. Nous avons donc $\sqrt{A} = (x^4 + x) \times (x^2 + 1) + 1 = x^6 + x^4 + x + x^3 + 1$. Nous pouvons finalement vérifier que $(x^6 + x^4 + x + x^3 + 1)^2 \bmod x^7 + x + 1 = x^5 + x + 1$.

Autant il est facile d'extraire des coefficients dans le cadre d'une base polynomiale, autant en RNS la problématique est légèrement différente. Heureusement, des outils rudimentaires vont nous donner une solution simple (que je trouve élégante) : la notion de dérivées.

Definition 5.9. Dérivées de polynômes
Soit $A = \sum_{i=0}^{m-1} a_i \times x^i$, nous noterons sa dérivée comme $d(A) = \sum_{i \text{ pair}} a_{i+1} \times x^i$.

Cette définition est issue des notions de dérivées traditionnelles, elle conserve des propriétés équivalentes, comme $d(A \times B) = A \times d(B) + d(A) \times B$, entre autres. Ce qui nous intéresse ici, c'est que compte-tenu de la caractéristique du corps, dériver A revient à éliminer les coefficients d'indices pairs. La question qui se pose maintenant va de soi : comment dériver un élément A représenté en RNS ? Quelles sont les conditions qui pourraient rendre cette opération élémentaire ?

Proposition 5.10. *Soit $A \in GF(2)[x]$ représenté en RNS sous la forme $A = (\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_{v-1})$. Nous exigerons ici que tous les moduli de la base RNS soient des carrés, autrement dit, $n_i = (g_i)^2$ pour $i \in [0, \ldots, n-1]$. Alors, $d(A \mod N) = d(A) \mod N$.*

Cela revient à dire qu'il suffit de dériver **indépendamment** dans chacun des canaux la coordonnée correspondante.

†. $\sqrt{x}$ est pré-calculé.

Démonstration. Si A est représenté en RNS, il peut alors s'écrire comme : $A = \sum_{i=0}^{v-1} |\tilde{a}_i \times N_i^{-1}|_{n_i} \times N_i$. Appliquons la formule de la dérivée.

$$\begin{aligned} d(A) &= d(\textstyle\sum_{i=0}^{v-1} |\tilde{a}_i \times N_i^{-1}|_{n_i} \times N_i) \\ &= \textstyle\sum_{i=0}^{v-1} d(|\tilde{a}_i \times N_i^{-1}|_{n_i} \times N_i) \\ &= \textstyle\sum_{i=0}^{v-1} d(|\tilde{a}_i \times N_i^{-1}|_{n_i}) \times N_i + \sum_{i=0}^{v-1} (|\tilde{a}_i \times N_i^{-1}|_{m_i}) \times d(N_i) \end{aligned}$$

Or ici, $d(N_i) = d(\prod_{j\neq i} n_j) = d(\prod_{j\neq i} (g_j)^2) = \sum_k d((g_k)^2) \times \frac{N_i}{n_k}$. Notons que $d((g_k)^2) = 0$. Cela implique que $\sum_k d((g_k)^2) \times \frac{N_i}{n_k} = \sum_k 0 \times \frac{N_i}{n_k} = 0$. Ainsi,

$$\begin{aligned} d(A) &= \textstyle\sum_{i=0}^{v-1} d(|\tilde{a}_i \times N_i^{-1}|_{n_i}) \times N_i + \sum_{i=0}^{v-1} (|\tilde{a}_i \times N_i^{-1}|_{n_i}) \times \underbrace{d(N_i)}_{=0} \\ &= \textstyle\sum_{i=0}^{n-1} d(|\underline{a}_i \times N_i^{-1}|_{n_i}) \times N_i \\ &= (d(\tilde{a}_0), d(\tilde{a}_1), \ldots, d(\tilde{a}_{v-1})) \end{aligned}$$

□

Nous disposons dès lors, pour le calcul de dérivées, d'un outil puissant, qui sous la condition sus-citée ($n_i = (g_i)^2$), n'impose aucun calcul complexe. Si la condition n'est pas respectée, il est nécessaire d'appliquer une suite d'opérations impliquant une communication entre les différents canaux (comme peut l'exiger, par exemple, un changement de base), ce qui alourdit considérablement la dérivation.

Exemple 5.2.7. Posons $n_0 = x^4$ et $n_1 = x^4 + x^2 + 1$. Les moduli n_0 et n_1 sont premiers entre eux et ce sont bien des carrés $n_0 = (x^2)^2$ et $n_1 = (x^2 + x + 1)^2$. Posons aussi $A = x^5 + x + 1$. Nous avons $A = (x + 1, x^3 + 1)$. De par la proposition 5.10, $d(A) = (d(x + 1), d(x^3 + 1)) = (1, x^2)$ dans la base $\mathcal{B} = \{n_0, n_1\}$. Nous pouvons en effet vérifier que $d(A) = d(x^5 + x + 1) = (x^4 + 1) = (1, x^2)$.

Une fois l'élément B dérivé ($d(B)$), (tous les monômes sont d'indices pairs), il faut appliquer la racine carrée afin de récupérer $\sqrt{\mathcal{I}}$ [‡] :

$$\begin{aligned} \sqrt{d(B)} &= \sqrt{d(\textstyle\sum_{i=0}^{m-1} b_i \times x^i)} \qquad = \sqrt{\textstyle\sum_{i<m \text{ impair}} b_i \times x^{i-1}} \\ &= \textstyle\sum_{i=0}^{(m+1)/2} b_{(2\times i+1)} \times x^i \end{aligned}$$

Appliquer cette racine carrée en RNS va impliquer de devoir communiquer entre les différents canaux. Cela est principalement dû au fait qu'extraire une racine carrée modulo $N = G^2$ [§] un carré amène un ensemble de solutions (du type $\sqrt{d(B)} + q \times$

‡. m est supposé impair.
§. $G = \prod_{i=0}^{v-1} g_i$.

G) que nous ne savons pas facilement traiter. Nous savons, par contre, que $d(B) = (d(\tilde{b}_0), d(\tilde{b}_1), \ldots, d(\tilde{b}_{v-1})) = \sum_{i=0}^{v-1} |d(\tilde{b}_i) \times (N_i)^{-1}|_{n_i} \times N_i$ nous amenant à

$$\sqrt{d(B)} = \sum_{i=0}^{v-1} \sqrt{|d(\tilde{b}_i) \times (N_i)^{-1}|_{n_i}} \times G_i, \tag{5.4}$$

avec $G_i = \prod_{j \neq i} g_j$.

Démonstration. Nous prétendons que la racine de $Z = |d(\tilde{b}_i) \times (N_i)^{-1}|_{n_i}$ existe dans GF(2)[x] (c'est à dire que Z n'est constitué que de monômes d'indices pairs). Par définition de la fonction dérivée, $\sqrt{d(\tilde{b}_i)}$ existe dans GF(2)[x]. Si $T = |(G_i)^{-1}|_{g_i}$ est l'inverse de G_i modulo g_i alors $T \times G_i = 1 \bmod g_i \equiv 1 + q \times g_i$ et donc que $T^2 \times (G_i)^2 \equiv 1 + q^2 \times g_i^2 \equiv T^2 \times N_i \bmod n_i$. Ici T^2 est égal $|(N_i)^{-1}|_{n_i}$. Autrement dit $|(N_i)^{-1}|_{n_i}$ est un carré, sa racine existe dès lors dans GF(2)[x]. Nous avons donc le produit de deux carrés $d(\tilde{b}_i)$ et $|(N_i)^{-1}|_{n_i}$ modulo un carré $N = G^2$: il s'agit là aussi d'un carré. □

Nous avons alors la possibilité d'employer la relation 5.4 modulo n_i pour convertir $\sqrt{d(B)}$ dans la base $\mathcal{B}$. Nous calculons aussi $\Phi(d(B))$ dans la base $\mathcal{B}$ afin de pouvoir utiliser l'algorithme de multiplication en « Φ-RNS ».

En RNS, nous proposons de calculer la racine carrée d'un élément $A \in$ GF(2^m) de la façon suivante :

Algorithme 23 : Algorithme d'extraction de racines carrées en représentation RNS

Données : $A \in$ GF(2^m) représenté en Φ-RNS
Résultat : $\sqrt{A}$ représente en Φ-RNS

1 $\mathcal{I} \leftarrow d(A)$ *Extraction des coefficients d'indice impair en Φ-RNS*
2 $\mathcal{P} \leftarrow A + \mathcal{I} \times x$ *Extraction des coefficients d'indice pair*
3 **retourner** $\sqrt{\mathcal{P}} + \sqrt{\mathcal{I}} \times \underbrace{(\sqrt{x} \times \Phi(N))}_{\text{pré-calculé}} \bmod f$ *Multiplication en « Φ-RNS »*

Toutes les lignes de l'algorithme 3 impliquent de manipuler les quantités en représentation Φ-RNS. C'est à dire qu'à chaque étape, nous avons accès, pour une variable D, à D et $\Phi(D)$ dans la base $\mathcal{B}$. Notons que la ligne 2 de l'algorithme ne requiert pas une multiplication RNS. Nous savons que A est au plus de degré $m-1$, et donc que $d(A)$ de degré $m-2$. Ainsi, multiplier $d(A)$ par x produit un polynôme de degré au plus $m-1$ et reste dans la représentation dynamique de la base $\mathcal{B}$. L'extraction des racines carrées se fait par la relation 5.4 modulo n_i pour i compris entre 0 et $v-1$ (inclus).

Exemple 5.2.8. Reprenons l'exemple précédent avec $n_0 = x^4, n_1 = x^4 + x^2 + 1, A = x^5 + x + 1$. Nous avons $N_0^{-1} = n_1^{-1} \bmod n_0 = x^2 + 1$ et $N_1 = n_0^{-1} \bmod n_1 = x^2$. Nous remarquons que N_0^{-1} et N_1^{-1} sont bien des carrés.

Nous avons $A = (x+1, x^3+1)$ et $\Phi(A) = (1, x^3+x^2+x)$ dans la base $\mathcal{B}$. Nous avons pour $\mathcal{I}$ les valeurs $d(A) = (1, x^2)$ et $d(\Phi(A)) = (0, x^2+1)$. Pour $\mathcal{P}$ nous trouvons après calculs $A + \mathcal{I} \times x = (x+1, x^3+1) + x \times (1, x^2) = (x+1, x^3+1) + (x, x^3) = (1, 1)$. Nous effectuons une démarche similaire pour la représentation $\Phi(d(A) \times x + A)$ pour laquelle nous trouvons $(1, x^3+x^2+x) + x \times (0, x^2+1) = (1, x^3+x^2+x) + (0, x^3+x) = (1, x^2)$. Nous savons donc que le polynôme $\mathcal{P}$ représentant les monômes d'indices pairs est représenté par le couple Φ-RNS $(1, 1)$ et $(1, x^2)$ et le polynôme $\mathcal{I}$ représentant les monômes d'indices impairs par le couple Φ-RNS $(1, x^2)$ et $(0, x^2+1)$.
Il faut maintenant calculer $\sqrt{\mathcal{I}}$ et $\sqrt{\mathcal{P}}$. Nous ne détaillerons le calcul que pour $\sqrt{\mathcal{I}}$.

En reprenant la relation 5.4, nous avons $\sqrt{\mathcal{I}} = \sqrt{|d(\tilde{a}_0) \times (x^2+1)|_{n_0}} \times (x^2+x+1) + \sqrt{|d(\tilde{a}_1) \times (x^2)|_{n_0}} \times (x^2) = \sqrt{|1 \times (x^2+1)|_{n_0}} \times (x^2+x+1) + \sqrt{|x^2 \times x^2|_{n_1}} \times x^2 = (x+1) \times (x^2+x+1) + (x+1) \times x^2 = (x^3+1) + (x^3+x^2) = x^2+1$. Nous retrouvons bien $\mathcal{I} = x^4+1$ d'où $\sqrt{I} = x^2+1$. Ce calcul est normalement fait $\mod n_0$ et $\mod n_1$.

Nous complétons l'algorithme par le calcul RNS de $\sqrt{A} = \sqrt{\mathcal{P}} + \sqrt{\mathcal{I}} \times (\sqrt{x} \times \Phi(N)) \mod f = (x^3+x+1, x^3+x^2+x+1)$ et $\Phi(\sqrt{A}) = (x^3+1, x^3)$.

Nous avons donc proposé dans cette section un algorithme permettant d'extraire, sous certaines conditions sur la base $\mathcal{B}$, la racine carrée d'un élément représenté en « Φ-RNS ». Cette méthode fait appel à la notion de dérivée et permet d'accomplir le calcul en une seule multiplication « Φ-RNS » et un équivalent de « quatre extensions de base » pour l'extraction de $\sqrt{\mathcal{P}}$ et $\sqrt{\mathcal{I}}$ via la relation 5.4. Ce sont des opérations relativement lourdes et conséquentes face à extraction dans une représentation polynomiale standard.

5.2.5 Inversion en RNS

Comme nous l'avons vu précédemment lors de cette thèse au chapitre 4, les algorithmes d'inversions modulaires reposent sur deux principales méthodes (et leurs dérivés). La première est basée sur le petit théorème de Fermat (*FLT*) tandis que la seconde se fonde sur la séquence d'Euclide. Réaliser une exponentiation rapide n'est pas spécialement efficace en RNS étant donné le fait qu'une opération de type $A \times A$ est malheureusement tout aussi couteuse qu'une opération de type $A \times B$ avec $A \neq B$. Cet « handicap » nuit particulièrement à cette première approche s'appuyant sur le *FLT*. L'algorithme d'Euclide (binaire) nous a semblé singulièrement adapté : aucune extension de base ne sera en effet nécessaire. Dans l'algorithme traditionnelle d'Euclide, il est requis, au cours de l'exécution, de comparer les degrés des polynômes U et V. Or, en représentation RNS, il n'est pas évident d'extraire ce degré sans avoir recours à une reconstruction entière du polynôme ; c'est à dire, plus laconiquement, de convertir un élément en représentation RNS en base polynomiale. L'opération est bien évidemment couteuse ! L'idée introduite

par Karim Bigou [87] dans sa thèse et l'article [88] est que, dans les corps premiers GF(p), si A et B sont impairs alors $A+B$ ou $A-B$ est divisible par 4. L'auteur parvient à se libérer des comparaisons en ne faisant que des tests de divisibilité par des petites constantes (par 2, 4). Ces tests de divisibilités reviennent à ne considérer que les bits de poids faibles de l'entier représenté en RNS, plus aisés à manipuler. Ces tests sont tout droit inspirés du ruban de *Pascal*. Le ruban de Pascal est une méthode simple pour connaitre très rapidement la divisibilité d'un entier par un autre. En fait, nous apprenons au collège cette technique indirectement quand on nous enseigne que « si la somme des chiffres que composent le nombre est divisible par 3 alors ce nombre est lui même divisible par 3 ». Typiquement, si nous appliquons la méthode à 163 707 nous avons $1+6+3+7+0+7=24$. Le nombre 24 est divisible par 3 (nous aurions pu calculer par récursion $2+4=6$ et vérifier que 6 est bien divisible par 3). Pourquoi cela fonctionne t-il ? L'idée est de décomposer 163 707 comme somme de puissance de 10 auxquelles nous avons appliqué un modulo 3 (si 163 707 mod 3 = 0 alors 163 707 est divisible par 3). Nous avons donc $163\,707 \bmod 3 = (1\times10^5+6\times10^4+3\times10^3+7\times10^2+0\times10^1+7\times10^0) \bmod 3 = (1\times(10^5 \bmod 3)+6\times(10^4 \bmod 3)+3\times(10^3 \bmod 3)+7\times(10^2 \bmod 3)+0\times(10^1 \bmod 3)+7\times(10^0 \bmod 3))$. Nous avons que $10 \bmod 3 = 1$ et de fait $10^i \bmod 3 = 1$. Finalement, nous écrivons $163\,707 \bmod 3 = (1+6+3+7+0+7 \bmod 3)$. Différemment exprimé, la somme des chiffres de 163 707 est divisible par 3 si et seulement si 163 707 est lui même divisible par 3. Le ruban de Pascal est une généralisation de ce procédé, si un nombre D est écrit en base $\mathbf{B}$ sous la forme $\sum_{i=0}^{n} d_i \times \mathbf{B}^i$, tester sa divisibilité par une petite constante c revient à calculer $\sum_{i=0}^{n} d_i \times (\mathbf{B}^i \bmod c) \bmod c$. L'idée est donc de pré-calculer les quantités $Z_i = (\mathbf{B}^i \bmod c)$ et de calculer $D \bmod c = \sum_{i=0}^{n} d_i \times Z_i \bmod c$. Dans l'algorithme d'inversion 24 que nous proposons en Φ-RNS, nous n'aurons besoin que de calculer la divisibilité du polynôme « $Current$ » par x. Nous employons finalement un Ruban de Pascal sur la formule des restes chinois sur la représentation du polynôme $Current$. Savoir si $Current$ est divisible par x revient à calculer $\sum_{i=0}^{v-1} |\widetilde{Current}_i \times N_i^{-1}|_{n_i} \times N_i \bmod x = \sum_{i=0}^{v-1} |\widetilde{Current}_i \times N_i^{-1}|_{n_i} \times Z_i$ où $Z_i = N_i \bmod x \in \{0,1\}$.

Exemple 5.2.9. Si nous avons un canal (disons le premier) dont le modulo s'écrit $n_0 = x \times g_0$ alors tous les N_i (sauf N_0) seront divisibles par x, tous les Z_i sauf Z_0 seront égaux à 0. Pour connaitre la divisibilité de $Current$ par x, il ne suffira que de calculer $|\widetilde{Current}_0 \times N_i^{-1}|_{n_0} \bmod x$.

Le pire des cas que nous puissions avoir est le cas où $n_0 = x$. Posons la base $\mathcal{B} = \{n_0, n_1, n_2\}$ avec $n_0 = x, n_1 = x^3+x+1, n_2 = x^2+1$. Nous avons, d'après le théorème chinois des restes, pour un élément $A = (\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_{v-1})$ que $A = |\tilde{a}_0 \times 1|_{n_0} \times (x^5 + x^2 + x + 1) + |\tilde{a}_1 \times 1|_{n_1} \times (x^3 + x) + |\tilde{a}_2 \times x|_{n_2} \times (x^4 + x^2 + x)$ et plus particulièrement $A \bmod x = |\tilde{a}_0 \times 1|_{n_0} \times (x^5 + x^2 + x + 1) \bmod x = |\tilde{a}_0|_{n_0}$.

Le principal problème de l'exemple 5.2.9 est qu'il nous faudra diviser par x dans l'algorithme 24. En RNS, cette division par x se fera par la multiplication par l'inverse de x mod N. Or si $pgcd(x, N) = x$, cet inverse n'existe pas. En choisissant des moduli qui ne sont pas divisibles par x, nous pouvons prouver que les Z_i sont tous égaux à 1.

Nous proposons l'algorithme Alg.24, dans lequel ne figure aucune comparaison, mais des tests de visibilités par x, tests qui ne seront pas abrupts en circuit. Cet algorithme avait aussi été proposé en base polynomiale dans [89] dans GF(2^m) : la comparaison de degrés peut être couteuse en matériel (notamment sur des corps de grandes tailles), il convient donc de s'en passer.

Algorithme 24 : Algorithme d'Euclide Binaire RNS

Données : $A \in \mathrm{GF}(2)[x]$, l'élément à inverser, $f \in \mathrm{GF}(2)[x]$ le polynôme irréductible définissant le corps GF(2^m)
Résultat : $A^{-1} \mod f$

```
1  Current ← A ; LastOdd ← f ; ULastOdd ← 0 ;                    Initialisation
2  UCurrent ← 1 ;                                                Boucle
3  tant que Current ≠ 1 faire
4      si Current mod x=0 alors
5          Current ← Current / x
6          si UCurrent mod x = 1 alors
7              UCurrent ← (UCurrent+f) / x
8          sinon
9              UCurrent ← (UCurrent) / x
10     sinon
11         tmp ← Current
12         Current ← Current + LastOdd
13         LastOdd ← tmp
14         tmp ← UCurrent
15         UCurrent ← UCurrent + ULastOdd
16         ULastOdd ← tmp
17 retourner UCurrent
```

Démonstration. Observons tout d'abord que la condition $Current \neq 1$ est facilement implémentable en RNS puisqu'il suffit de vérifier que la valeur stockée dans chaque canal est égale à 1. L'algorithme est inspiré de l'algorithme étendu d'Euclide. Nous nous sommes cependant passés des comparaisons des degrés, difficiles en RNS. Le degré maximal du système $deg(Current)$ peut donc osciller durant l'algorithme. Nous prétendons cependant qu'il va, au final, attendre 0. Tout au long de l'algorithme, nous avons une relation du type $Current = UCurrent \times A + VCurrent \times f$. Ici $VCurrent$ n'est pas sauvegardé. À l'issu du déroulement de cet algorithme, $Current = 1$ et donc nous obtenons une égalité $1 = UCurrent \times A + VCurrent \times f$. La variable $UCurrent$ est, de fait, l'inverse de A modulo f. Nous devons cependant montrer que $Current$ atteint 1

après un certain nombre d'itérations. La variable *Current* est tout d'abord initialisée à A, l'élément à inverser, d'où $deg(Current) = deg(A)$. Nous avons déroulé 5 étapes de l'algorithme pour en déduire une règle sur les degrés de *Current* et de *LastOdd*.

- À la première itération, *Current* s'écrit comme $x^{z_0} \times D_0$ ou D_0 est premier avec x. Nous noterons g_0 le degré de *Current* et g_1 le degré de *LastOdd*. À noter que $deg(LastOdd) > deg(Current)$ à chacune des itérations de l'algorithme.
- Pendant z_0 itérations, l'algorithme passera par la première partie de la condition « **si** *Current* mod $x = 0$ **alors** ». Ensuite *Current* ne sera plus divisible par x. Le registre *Current* prendra la valeur de $LastOdd + Current$, et sera donc, à cet instant, de degré g_1. Le registre *LastOdd* prendra l'ancienne valeur contenue dans *Current* (c'est à dire D_0), qui sera de degré $g_2 = g_0 - z_0$. À cet instant, *Current* s'écrit sous la forme $x^{z_1} \times D_1$, D_1 premier avec x.
- Pendant z_1 itérations, l'algorithme passera par la première partie de la condition « **si** *Current* mod $x = 0$ **alors** ». Ensuite *Current* ne sera plus divisible par x. Le registre *Current* prendra la valeur de $LastOdd+Current$, et sera donc, à cet instant, de degré $g_2 = g_0 - z_0$. Le registre *LastOdd* prendra l'ancienne valeur contenue dans *Current* (c'est à dire D_1), qui sera de degré $g_3 = g_1 - z_1$. À cet instant, *Current* s'écrit sous la forme $x^{z_2} \times D_2$, D_2 premier avec x.
- Pendant z_2 itérations, l'algorithme passera par la première partie de la condition « **si** *Current* mod $x = 0$ **alors** ». Ensuite *Current* ne sera plus divisible par x. Le registre *Current* prendra la valeur de $LastOdd+Current$, et sera donc, à cet instant, de degré $g_3 = g_1 - z_1$. Le registre *LastOdd* prendra l'ancienne valeur contenue dans *Current* (c'est à dire D_2), qui sera de degré $g_4 = g_2 - z_2 = g_0 - z_0 - z_2$. À cet instant, *Current* s'écrit sous la forme $x^{z_3} \times D_3$, D_3 premier avec x.
- Pendant z_3 itérations, l'algorithme passera par la première partie de la condition « **si** *Current* mod $x = 0$ **alors** ». Ensuite *Current* ne sera plus divisible par x. Le registre *Current* prendra la valeur de $LastOdd + Current$, et sera donc, à cet instant, de degré $g_4 = g_0 - z_0 - z_2$. Le registre *LastOdd* prendra l'ancienne valeur contenue dans *Current* (c'est à dire D_3), qui sera de degré $g_5 = g_3 - z_3 = g_1 - z_1 - z_3$. À cet instant, *Current* s'écrit sous la forme $x^{z_4} \times D_4$, D_3 premier avec x.
- etc

En raisonnant par induction, nous pouvons prouver que $g_{2\times d+1} = g_1 - z_1 - z_3 - z_5 - \ldots - z_{2\times(d-1)+1}$ et que $g_{2\times d} = g_0 - z_0 - z_2 - \ldots - z_{2\times(d-1)}$. Le registre *Current* prendra des valeurs dont le degré sera successivement soit de $g_{2\times d}$ soit de $g_{2\times d+1}$ avec d s'incrémentant à chaque fois que nous passons par la deuxième partie de la condition (c'est à dire *Current* mod $x \neq 0$). Nous obtenons dès lors deux suites décroissantes $((g_{2\times d+1})$ et $(g_{2\times d}))$ avec $g_{2\times d+1} \geq g_{2\times(d+1)+1}$ et $g_{2\times d} \geq g_{2\times(d+1)}$. Ces deux suites, dans les entiers, sont décroissantes et minorées par 0. Elles admettent donc une limite. Il y a alors un rang j à partir duquel $z_j = 0$. Or $z_j = 0$ signifie que *Current* n'est pas divisible par x,

ce qui est absurde étant donnée la construction $Current \leftarrow Current + LastOdd$ à la ligne 12 de l'algorithme 24 (qui nous assure un $Current$ divisible par x). À ce niveau $Current$ est donc égal à 0. L'algorithme se termine donc avant ce rang j et se conclut logiquement en un nombre fini d'itérations. En pratique, d'après nos expérimentations (basées sur un programme écrit en Python), $4 \times m$ itérations de la boucle principale semblent être le maximum. Cela doit être formellement prouvable. □

Exemple 5.2.10. Nous fournissons un exemple de déroulement de l'algorithme. La variable $LastOdd$ est initialisée à $x^3 + x + 1$.

En représentation polynomiale

$x^3 + x + 1$	0	
$x^2 + 1$	1	
$x^3 + x^2 + x$	1	$Current \leftarrow Current + LastOdd$
$x^2 + x + 1$	$((x^3 + x + 1) + 1)/x = x^2 + 1$	$Current \leftarrow Current/x$
$x^2 + x + 1 + LastOdd = x$	$(x^2 + 1) + 1 = x^2$	$Current \leftarrow Current + LastOdd$
1	x	$Current \leftarrow Current/x$

En représentation RNS avec la base $\mathcal{B} = \{n_0 = x^2 + 1, n_1 = x^2 + x + 1\}$

$(1, x)$	$(0, 0)$	
$(0, x)$	$(1, 1)$	
$(1, 0)$	$(1, 1)$	$Current \leftarrow Current + LastOdd$
$(x, 0)$	$((1, x) + (1, 1))/x = (0, x)$	$Current \leftarrow Current \times x^{-1} \bmod N$
$(x, 0) + LastOdd = (x, x)$	$(0, x) + (1, 1) = (1, x + 1)$	$Current \leftarrow Current + LastOdd$
$(1, 1)$	(x, x)	$Current \leftarrow Current \times x^{-1} \bmod N$

Nous trouvons donc que x est l'inverse de x^2+1 (nous avons bien, en effet, $x \times (x^2+1) = x^3+x = 1 \bmod x^3 + x + 1$). Les parties grises sont les polynômes qui seront stockés dans $LastOdd$.

Nous avons, dans cette section, présenté un algorithme d'inversion dans GF(2^m) en représentation RNS (et par extension, en « Φ-RNS »)

5.3 Conclusion

Nous avons proposé dans ce chapitre un nouvel algorithme de multiplication modulaire en RNS que nous avons nommé « Φ-RNS ». Cette multiplication est basée sur une nouvelle représentation des éléments en RNS. Nous ne stockons plus les polynômes comme $(A \bmod N, A \bmod N')$ mais comme $(A \bmod N, \Phi(A) \bmod N)$. Nous montrons que notre solution est très compétitive face à des algorithmes plus classiques de multiplications

dans GF(2^m). Nous avons aussi proposé un algorithme d'extraction de racines carrées et d'inversion en « Φ-RNS ».

Chapitre 6

Conclusion

À travers ce manuscrit, nous avons tenté d'apporter de nouvelles idées algorithmiques quant aux calculs sur GF(2^m). Nous avons introduit le concept de base normale permutée (*PNB*) (section 3) ainsi qu'une nouvelle représentation RNS (section 5) que nous avons nommée « Φ-RNS ». La base normale permutée permet d'exploiter des schémas multiplicatifs utilisés durant l'inversion et nous accorde l'opportunité d'accélérer sensiblement le calcul. Le « Φ-RNS » permet de diviser par deux le nombre de constantes normalement utilisées dans une représentation RNS standard. Il nous donne également la possibilité de n'employer qu'une seule base ce qui nous permet d'implémenter des multiplieurs spécialisés à moindre coût (en surface silicium).
Nous avons aussi conçu un crypto-processeur complet et avons intégré un algorithme de multiplication scalaire parallèle sur courbes elliptiques. La clef est coupée en deux : le *halve-and-add* permet de casser le côté séquentiel d'un schéma de Hörner. Nous montrons que le gain en vitesse du parallélisme est intéressant sans pour autant atteindre un ratio de deux (nous avons doublé la surface sans doubler la vitesse). Nous voulions aussi estimer la sécurité que pouvait apporter le parallélisme dans les calculs. Nous avons mené une attaque de type *templates* et montrons que le parallélisme seul des calculs ne suffit pas au système à se prémunir face à des attaques par canaux cachés. Nous avons alors suggéré l'emploi « d'opérations pour rien », inutiles pour le bon déroulement du calcul pour apporter une sécurité (qui s'avère ici efficace), quitte à ralentir la vitesse d'exécution des algorithmes de multiplications scalaires.

Les perspectives sont nombreuses : il serait intéressant d'étudier la représentation Φ-RNS dans d'autres corps finis (en petites caractéristiques). Les transformations (Φ) seraient plus nombreuses encore, pourraient-elles être exploitées (typiquement $x \to x + 2$ dans GF(3^m)) ?

Nous avons remarqué qu'il existait un compromis entre vitesse et sécurité. Il serait intéressant de se pencher davantage sur ce compromis.

Annexe A

Multiplications en Base Normale sur CPU

Dans cette annexe, nous nous sommes penchés sur les implémentations CPU de la multiplication et de l'inversion en base normale dans GF(2^m). À travers le chapitre Chp.3, nous avons remarqué qu'il était relativement aisé, pour un certain nombre d'algorithmes de multiplications de type PISO [*], de paralléliser le calcul. Typiquement, pour l'algorithme de multiplication de Massey-Omura [49], la génération de chacun des bits du produit $C = A \times B = (c_0, c_1, \ldots, c_{m-1})$ est indépendant. Il n'est, par exemple, pas nécessaire de connaitre le bit c_i pour déduire le bit c_{i+1}. Il est certain que l'algorithme de Massey-Omura (et ses dérivés) permette(nt) d'implémenter rapidement le parallélisme, mais à quel prix ? Gérer le parallélisme dans une application n'est pas gratuit, il prend du temps machine, temps lié à la gestion même du partage des tâches et à la synchronisation des processus. En somme, qu'apportent le multithreading, le multi-cores aux calculs sur GF(2^m) dont les éléments sont représentés en base normale ? Pour répondre à cette question, nous avons implémenté et validé différents algorithmes écrits en Python. Pour la gestion même du parallélisme, nous avons utilisé la bibliothèque **multiprocessing** de Python. La configuration qui sera employée durant la phase de tests est basée sur un processeur Intel i-7 3720 cadencé à 2.6 Ghz (4 cœurs physiques, 4 cœurs virtuels) épaulé de 8 Go de mémoire vive. La version de Python est la version 3.4 64 bits tournant sur Windows 7 Professionnel, lui aussi, 64 bits.

∗. Parallel-Input/Serial-Output

A.1 État de l'art

Comme en matériel, la multiplication en base normale repose sur du calcul matriciel. La méthode de Massey-Omura est facilement adaptable sur CPU. Ning Yin ont proposé dans [90] une façon efficace de le faire tout en tenant compte de la particularité essentielle d'un processeur : la taille des bus est fixe. Nous la noterons w. Nous ne pouvons travailler simultanément sur les m bits des opérandes si m excède la taille des bus. La première chose à réaliser est qu'il existe une formule simple qui lie chaque bit c_k du produit $C = A \times B = [c_0, c_1, \ldots, c_{m-1}]$ aux opérandes A et B. Nous avons, en effet, la relation suivante :

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} \times b_{j+k} \times M_0[i,j] \tag{A.1}$$

où $M_0[i,j]$ est le coefficient à l'index i,j de la matrice M_0. Tous les calculs ici présents s'effectuent dans GF(2). L'approche de Ning Yin se base sur cette formule, sauf qu'au lieu d'évaluer cette somme pour chacun des c_k les uns à la suite des autres, nous le faisons sur w bits simultanément (il s'agit, en quelque sorte, d'une approche bit-slicée [91]). Une succession d'opérations (XOR, AND, etc) sur des mots de w bits sera lancée, les bits manipulés seront, par contre, totalement indépendants les uns des autres. Au final, à l'issu de l'algorithme 25, nous obtiendrons un ensemble de sous-mots de w bits ayant pour valeurs $[c_0, c_1, \ldots, c_{w-1}], [c_w, c_{w+1}, \ldots, c_{2w-1}], \ldots, [c_{w(T_w-1)}, c_{w(T_w-1)+1}, \ldots, c_{w(T_w-1)+w-1}]$ (les indices sont à prendre modulo m). En premier lieu, il est, comme dit auparavant in-envisageable de travailler sur des mots de taille m sur ordinateur. Les opérandes A et B seront découpées en $T_w = \lceil m/w \rceil$ sous-mots de w bits (le dernier sous-mot sera éventuellement comblé par des 0 si m n'est pas un multiple de w).

$$\begin{array}{lcl} A(0) & = & [a_0, a_1, \ldots, a_{w-1}] \\ A(1) & = & [a_w, a_{w+1}, \ldots, a_{2w-1}] \\ \vdots & \vdots & \vdots \\ A(\lfloor m/w \rfloor) & = & [a_{\lfloor m/w \rfloor w}, a_{\lfloor m/w \rfloor w+1}, \ldots, a_{m-1}, 0, \ldots, 0] \end{array} \tag{A.2}$$

Nous avons une représentation similaire pour B. Dans l'algorithme de Massey-Omura, les registres A et B sont décalés de 1 bit vers la gauche à chaque cycle d'horloge. Nous avons une problématique similaire en logiciel, il nous faudra décaler les opérandes A et B. Décaler un registre qui stocke les m bits d'un bit vers la gauche n'est pas difficile (il existe des circuits spécialisés pour cela) mais quand les éléments sont représentés en sous-mots de w bits, le labeur n'est pas le même. Ning et Yin proposent de pré-calculer,

à la manière que nous avons suggérée dans 3.3.1, l'ensemble des décalages possibles à travers la représentation « redondante » :

$$\begin{array}{lcl} RotA(0) & = & [a_0, a_1, \ldots, a_{w-1}] \\ RotA(1) & = & [a_1, a_2, \ldots, a_w] \\ \vdots & \vdots & \vdots \\ RotA(m-1) & = & [a_{m-1}, a_0, \ldots, a_{w-2}] \end{array} \tag{A.3}$$

Ce précalcul peut sembler superflu, il n'en est rien. Nous ferons appel de nombreuses fois à chacun des RotA/RotB dans l'algorithme 25, autant les conserver une bonne fois pour toute. Notons aussi que l'opérateur & présent à la ligne 10 de l'algorithme 25 effectue une fonction « ET Logique » bit à bit sur l'ensemble du sous-mot de w bits :

$$(0011)\ \&\ (0101) = (0001)$$

Nous voyons que la rapidité est de l'algorithme 25 est notamment reliée au nombre de coefficients égaux à 1 dans la matrice M_0 puisque des instructions supplémentaires seront lancées, ou non, selon la valeur de $M_0[i, j]$ (lignes 8 et 9). L'algorithme 25 ne permet pas d'exploiter les motifs **SMP**s de type $A^{\text{shf}} \times A$ dont nous avons parlés dans le chapitre Chp. 3. Nous pourrions en tenir rigueur en utilisant un algorithme de multiplication différent, notamment celui décrit dans Algo. 26. L'idée est de décomposer la matrice de Massey-Omura de façon à non plus la parcourir bit par bit, mais mot par mot. En effet, en notant que

Algorithme 25 : Algorithme de multiplication de Ning et Yin (1) [90].

Données : $A, B \in \mathrm{GF}(2^m)$
Résultat : $P = A \times B$

```
1  Précalculer RotA et RotB.
2  T_w ← ⌈m/w⌉
3  pour t de 0 à T_w − 1 faire
4  |  P[t] ← 0
5  |  pour i de 0 à m − 1 faire
6  |  |  temp ← 0
7  |  |  pour j de 0 à m − 1 faire
8  |  |  |  si M_0[i, j] = 1 alors
9  |  |  |  |  temp ← temp + RotB[(j ∗ w + t)%m]
10 |  |  P[t] ← P[t] + (temp & RotA[(i ∗ w + t)%m])
11 retourner P
```

$$c_k = \sum_{i=0}^{m-1} a_i \times (\sum_{j=0}^{m-1} b_j \times M_0[i,j]) \tag{A.4}$$

Le terme $\sum_{j=0}^{m-1} b_j \times M_0[i,j]$ pour chaque i est évalué en T_w passes de boucle par la formule :

$$\sum_{j=0}^{Tw-1} \left(\sum_{k=0}^{w-1} b_{k+j\times w} \times M_0[i, k+j\times w] \right) \tag{A.5}$$

La partie entre parenthèses de l'expression est évaluée en une seule instruction, à ligne 8 de l'algorithme 26 via l'application de &. Ainsi, pour chaque c_i, le calcul implique $T_w \times m$ passes de boucles. Par extension, la génération de l'ensemble des c_i est donc $\mathcal{O}(T_w \times m^2)$.

$$M_0 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

FIGURE A.1: Décomposition de la matrice M_0 ($\mathrm{GF}(2^7)$) en sous-mots de 3 bits.

Algorithme 26 : Notre méthode de multiplication $A \times B$ en base normale.

Données : $A, B \in \mathrm{GF}(2^m)$
Résultat : $P = A \times B$

1 Précalculer $RotB$.
2 $T_w \leftarrow \lceil m/w \rceil$
3 **pour** t **de** 0 **à** $m-1$ **faire**
4 **pour** i **de** 0 **à** $m-1$ **faire**
5 $temp \leftarrow 0$
6 **pour** j **de** 0 **à** $T_w - 1$ **faire**
7 **si** $a_i = 1$ **alors**
8 $temp \leftarrow$
  $temp + M_0[i][j\times w, j\times w+1, \ldots, j\times w+w-1]$ & $RotB[(j*w+t)\%m]$
9 $acc \leftarrow 0$
10 **pour** k **de** 0 **à** $w-1$ **faire**
11 $acc \leftarrow (((temp >> k)\ \&\ 1) + acc)$
12 $c_t \leftarrow acc$
13 **retourner** $c_0, c_1, \ldots, c_{m-1}$

L'algorithme 26 nous permet aussi d'exploiter la symétrie à laquelle nous avons fait référence dans la section Sec .3.3.2. Dérouler l'algorithme 26 de multiplication de cette façon va nous donner la possibilité de calculer efficacement les motifs $A^{2^{\text{shf}}} \times A$. En effet, à chaque tour de la boucle principale (ligne 3), il est calculé un c_i. Nous avons dès lors le contrôle sur l'ordre de sortie de ces bits. Nous pourrions en effet calculer à chaque tour de boucle (ligne 3) les bits :

- Cycle d'horloge 0 : (c_0, c_{shf})
- Cycle d'horloge 1 : $(c_{2\times\text{shf}}, c_{2\times\text{shf}+\text{shf}})$
- Cycle d'horloge 2 : $(c_{4\times\text{shf}}, c_{4\times\text{shf}+\text{shf}})$
- Cycle d'horloge 3 : $(c_{6\times\text{shf}}, c_{6\times\text{shf}+\text{shf}})$
- etc

Cet ordonnancement des bits de sortie permet d'exploiter les **SMP**s du chapitre Chp. 3. De plus, contrairement à notre approche matérielle du Chp. 3, il nous sera possible, en logiciel, de décaler par la valeur qui convient $(2 \times \text{shf})$ pour générer l'ensemble des bits c_i et $c_{i+\text{shf}}$, sans redondance aucune, en seulement $\lceil m/2 \rceil$ tours de boucle. Une preuve de cette affirmation est donnée dans le cadre suivant.

Démonstration. Notons que si m est impair, l'inverse modulaire de $2 \mod m$ est égal à $2^{-1} \equiv (m+1)/2 \mod m$. L'ensemble (qui génère $\mathbb{Z}/m\mathbb{Z}$ comme groupe additif) $0, 2\times\text{shf}, 4\times\text{shf}, 6\times\text{shf}, \ldots, ((m+1)/2)\times 2\times\text{shf}$, $3\times\text{shf}, 5\times\text{shf}, \ldots, (-2\times\text{shf})$ peut être découpé en deux ensembles :
- $\Psi_0 = \{0, 2\times\text{shf}, 4\times\text{shf}, 6\times\text{shf}, \ldots, ((m+1)/2)\times 2\times\text{shf}\}$
- $\Psi_1 = \{3\times\text{shf}, 5\times\text{shf}, \ldots, (-2\times\text{shf})\}$

L'ensemble Ψ_0 correspond à l'ensemble $i \times 2 \times \text{shf}$ pour i compris entre 0 et $(m+1)/2$, bornes incluses. La partie Ψ_1, quant à elle, est l'ensemble $i \times 2 \times \text{shf} + \text{shf}$ pour i compris entre 1 et $(m+1)/2$, bornes incluses également. Ainsi, en sortant deux bits du produit $C = A \times B$ par tour de la boucle principale, il est possible de « paver » $\mathbb{Z}/m\mathbb{Z}$ en retournant $c_{i\times 2\times\text{shf}}$ et $c_{i\times 2\times\text{shf}+\text{shf}}$ séquentiellement pour l'ensemble des $i \in [0, 1, \ldots, ((m+1)/2)]$. □

Malheureusement, cet algorithme est bien plus lent que l'algorithme Algo. 25. Des résultats sont donnés les figures Fig. A.2 et Fig. A.3.

La méthode Algo. 25 que nous avons présentée fonctionne dans le cas général, peu importe la forme, la structure de la matrice M_0 : elle ne tient en aucun cas compte du caractère « creux » de M_0 quand nous choisissons de travailler en base normale gaussienne. Dans ce que nous avons présenté jusqu'à maintenant, l'entièreté de la matrice est parcourue tout en sachant pertinemment qu'une majorité de ses coefficients sera égale à 0. Les matrices M_0 dérivées d'une base normale gaussienne contiennent, sur chacune de leurs lignes, un maximum de t coefficients 1 (le reste est nul) : c'est en notant cela

que Ning et Yin [90] ont suggéré un raffinement de l'algorithme Algo. 25 qui simplifie grandement la complexité. L'algorithme Algo. 27 est destiné aux corps qui disposent d'une base normale gaussienne de type 2 (nous pouvons parler de base optimale de type 2 : Optimal Normal Basis II, alias ONB II). Dans ces circonstances, si ce n'est la première ligne de la matrice M_0 qui n'est composée que d'un seul 1, les autres lignes sont, elles, constituées de deux coefficients 1. Les positions de ces '1' sur chacune des lignes sont stockées dans les tableaux t_0 et t_1 (ligne 6 de Algo. 27). L'approche peut être avec certitude généralisée à un type t donné en considérant les tableaux t_0, t_1, ..., t_{t-1}. Cet apport algorithmique autorise une nette augmentation des performances.

Algorithme 27 : Algorithme de multiplication de Ning et Yin (2) en ONB II [90].

Données : $A, B \in \mathrm{GF}(2^m)$
Résultat : $P = A \times B$

```
1 Précalculer RotA et RotB.
2 T_w ← ⌈m/w⌉
3 pour t de 0 à T_w − 1 faire
4 |   temp ← RotA[0] & RotB[t_0[0]]
5 |   pour i de 1 à m − 1 faire
6 |   |   temp ← RotA[i] & (RotB[t_1[i]] + RotB[t_0[i]])
7 |   P[t] ← temp
8 retourner P
```

A.2 Parallélisme de la Multiplication en Base Normale

Intéressons nous maintenant au parallélisme, objet même de cet appendice. Pour l'algorithme 25 et l'algorithme 27, il est retourné à chaque tour de la boucle principale (ligne 3) un mot de w bits représentant une portion du produit $C = A \times B$.

Cycle horloge 0	Cycle horloge 1	...	Cycle horloge $T_w - 1$
$c_0, c_1, \ldots, c_{w-1}$	$c_w, c_{w+1}, \ldots, c_{2w-1}$	...	$c_{w(T_w-1)}, c_{w(T_w-1)+1}, \ldots, c_{w(T_w-1)+w-1}$

L'idée présentée ici est simple, il s'agit de répartir équitablement le calcul de ces portions dans p_n processus différents (ce qui est faisable car, rappelons-le, la génération de chacun de ces mots est indépendante des autres bits du produit C). Typiquement, si la taille w des mots est égale à 3, nous pourrions avoir le découpage suivant dans $\mathrm{GF}(2^7)$ avec $p_n = 2$ (une couleur symbolisant un processus différent).

Cycle horloge 0	Cycle horloge 0	Cycle horloge 1
c_0, c_1, c_2	c_3, c_4, c_5	c_6, c_0, c_1

Concernant notre suggestion d'algorithme 26, le découpage en p_n portions est très légèrement différent. Étant donné que nous sortons les bits c_i un à un, nous attribuons à chacun des processus P_i $(0 \leq i < p_n)$ le calcul des bits $c_0, c_1, \ldots, c_{z-1}$ pour P_0, $c_z, c_{z+1}, \ldots, c_{2z-1}$ pour P_1 (avec $z = \lceil m/p_n \rceil$) et ainsi de suite. Nous avons mesuré des temps d'exécution pour les corps du NIST ($m \in \{163, 233, 283, 409, 571\}$).

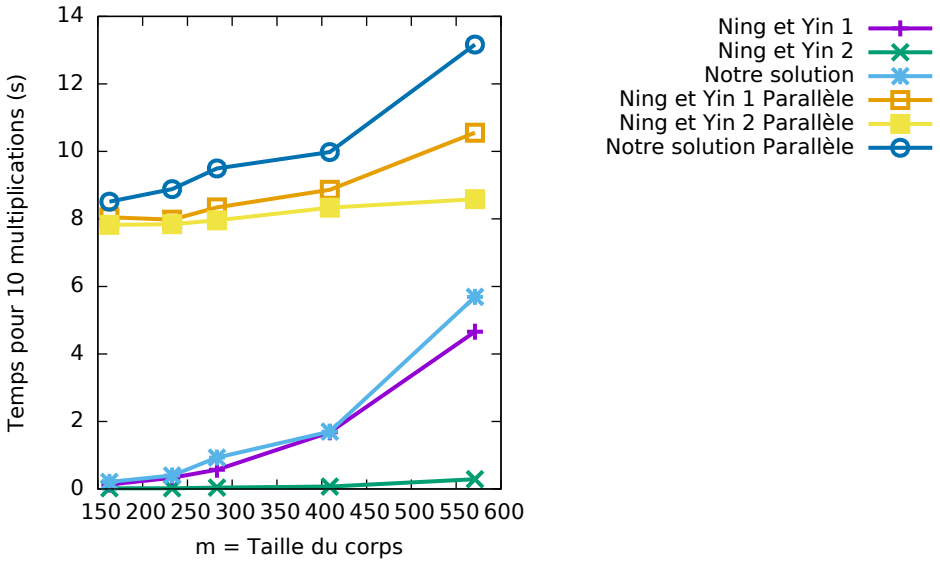


FIGURE A.2: Temps d'exécution pour 10 multiplications en base normale employant divers algorithmes (2 processus)

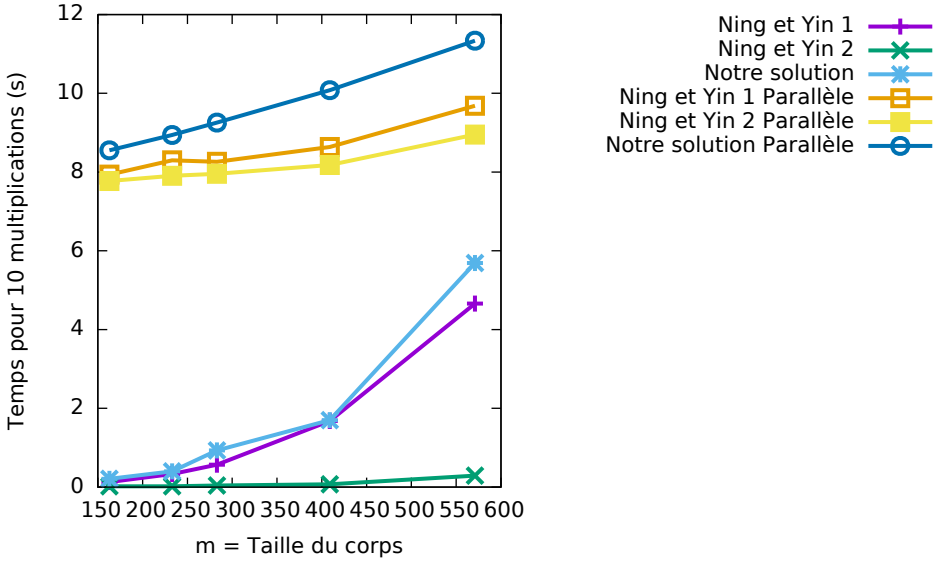


FIGURE A.3: Temps d'exécution pour 10 multiplications en base normale employant divers algorithmes (4 processus)

Ce qui frappe avec les figures Fig. A.2 et Fig. A.3 est l'impact significatif du parallélisme dans les performances de la multiplication en base normale et pas forcément dans le sens

que nous imaginions. L'explication nous semble relativement simple : synchroniser les multiples processus n'est pas une tâche facile et cela occupe une part très importante du temps CPU disponible. Les algorithmes déployés ici sont trop légers pour que la gestion du parallélisme ne soit un poids. Nous avons tenté de travailler avec des corps de taille très importante (> 10000 bits) mais aucun résultat obtenu ne semble pourvoir l'approche parallèle d'un quelconque intérêt. Il existe probablement une frontière concernant la taille des corps m qui une fois dépassée, met en exergue la plus-value du parallélisme, mais elle apparait comme suffisamment lointaine pour que l'approche multi-cpus ne soit pas considérée comme une solution viable. Nous noterons également la piètre performance de « notre solution ». Ce n'est pas étonnant puisque le nombre d'opérations élémentaires (AND, XOR) est plus élevé que dans les algorithmes de Ning et Yin (bien que les complexités soient les mêmes entre Algo. 25 et Algo. 26, c'est à dire en $\mathcal{O}(m^3/w)$). Autant dire que l'astuce présentée dans le chapitre Chp. 3 basée sur les **SMP**s n'aura, ici, aucun attrait. Il est bon aussi de noter que l'algorithme Algo. 27 surplombe les autres implémentations notamment de par sa faible complexité ($\mathcal{O}(tm^2/w)$ où t est le type de la base normale).

Dans la séquence d'exponentiation d'Itoh-Tsujii (sur laquelle se base l'opération d'inversion), il apparait un certain nombre de termes qui ne dépendent pas directement l'un de l'autre. Dans ce cas précis, il est alors tout à fait possible de réaliser ces deux calculs simultanément sur un processeur multi-cœurs. Nous allons maintenant tenter de voir si le calcul parallèle a dans ce cas précis, une quelconque utilité.

A.3 Parallélisme dans l'Inversion en Base Normale

Il existe en effet, dans la séquence d'Itoh-Tsujii un certain nombre de multiplications pouvant être lancées simultanément. Reprenons un exemple dans $\mathrm{GF}(2^{233})$ et choisissons la chaine d'addition $U = (1, 2, 4, 16, 32, 40, 64, 104, 128, 232)$. Le tableau Tab. A.1 résume l'ensemble des opérations réalisées nous conduisant au résultat $A^{-1} = A^{2^m-2}$. Les lignes en couleurs grises signifient que ces calculs peuvent être effectués en parallèle, indépendamment sur un processus P_0 et un processus P_1. Il est possible, en effet, de calculer $40 = 32 + 8$ et $64 = 32 + 32$ ensemble, nous avons toutes les données nécessaires pour le faire à cet endroit précis de la séquence.

CM 0	$\beta_0 = A$	U_0		1
CM 1	$\beta_1 = \beta_0^2 \times \beta_0$	$U_1 = U_0 + U_0$	$V_1 = (0,0)$	2
CM 2	$\beta_2 = \beta_1^{2^2} \times \beta_1$	$U_2 = U_1 + U_1$	$V_2 = (1,1)$	4
CM 3	$\beta_3 = \beta_2^{2^4} \times \beta_2$	$U_3 = U_2 + U_2$	$V_3 = (2,2)$	8
CM 4	$\beta_4 = \beta_3^{2^8} \times \beta_3$	$U_4 = U_3 + U_3$	$V_4 = (3,3)$	16
CM 5	$\beta_5 = \beta_4^{2^{16}} \times \beta_4$	$U_5 = U_4 + U_4$	$V_4 = (4,4)$	32
CM 6	$\gamma_0 = \beta_5^{2^8} \times \beta_3$	$U_6 = U_5 + U_3$	$V_4 = (5,3)$	40
CM 6	$\beta_6 = \beta_5^{2^{32}} \times \beta_5$	$U_7 = U_5 + U_5$	$V_4 = (5,5)$	64
CM 7	$\gamma_1 = \beta_6^{2^{40}} \times \gamma_0$	$U_8 = U_7 + U_6$	$V_4 = (7,6)$	104
CM 7	$\beta_7 = \beta_6^{2^{64}} \times \beta_6$	$U_9 = U_7 + U_7$	$V_5 = (7,7)$	128
CM 8	$\gamma_2 = \beta_6^{2^{128}} \times \gamma_1$	$U_{10} = U_9 + U_8$	$V_4 = (9,8)$	232
-	$\gamma_3 = \gamma_2^2$	-	-	-

TABLE A.1: Parallélisme dans la séquence d'Itoh-Tsujii. La signification de CM est « cycle de multiplication ».

Ainsi, dans cet exemple, nous avons la possibilité de calculer en même temps (au même cycle de multiplication) la quantité β_6 et λ_0, l'une dans le processus P_0 et l'autre dans le processus P_1. Ce parallélisme apparait naturellement dans les chaines d'additions binaires dans lesquelles chaque puissance de 2 apparait. Ces puissances y sont ensuite combinées pour former l'exposant souhaité. Autrement dit, la combinaison de ces puissances $(40 = 32 + 8)$ et leurs calculs $(2, 4, 8, 16, \ldots)$ sont indépendants faisant apparaitre un parallélisme natif. L'algorithme est décrit dans Algo. 28.

Algorithme 28 : Algorithme d'Itoh-Tsujii parallèle

Données : A nonzero element $A \in \mathrm{GF}(2^m)$

Résultat : $\alpha^{-1} \in \mathrm{GF}(2^m)$

1 **Initialisation** : $Reg1 \leftarrow A$; $Reg2 \leftarrow 1$; $Pow \leftarrow 0$;
2 **pour** i ***de*** 1 ***à*** $\lfloor log_2(m-1) \rfloor$ **faire**
3 **si** $m_{i-1} = 1$ **alors**
4 $Reg3 \leftarrow Reg1$;
5 $Reg2 \leftarrow (Reg3)^{2^{Pow}} * Reg1$;
6 $Pow \leftarrow Pow + 2^{i-1}$
7 $Reg1 \leftarrow Reg1^{2^{2^{i-1}}} * Reg1$ (en parallèle avec la ligne 3)
8 **retourner** $(Reg1^{2^{Pow}} * Reg2)^2$

Nous avons donc implémenté un algorithme d'Itoh-Tsujii sans parallélisme, dénoté **IT** et une version parallèle, quant à elle, dénotée **P-IT**. Les résultats sont reportés dans la figure Fig. A.4. L'algorithme d'Itoh-Tsujii est une exponentiation, c'est à dire une série

de multiplications. Nous avons utilisé l'algorithme de multiplication *Ning et Yin 2* le plus rapide afin de réaliser cette exponentiation (que cela soit dans la version P-IT ou dans la version standard).

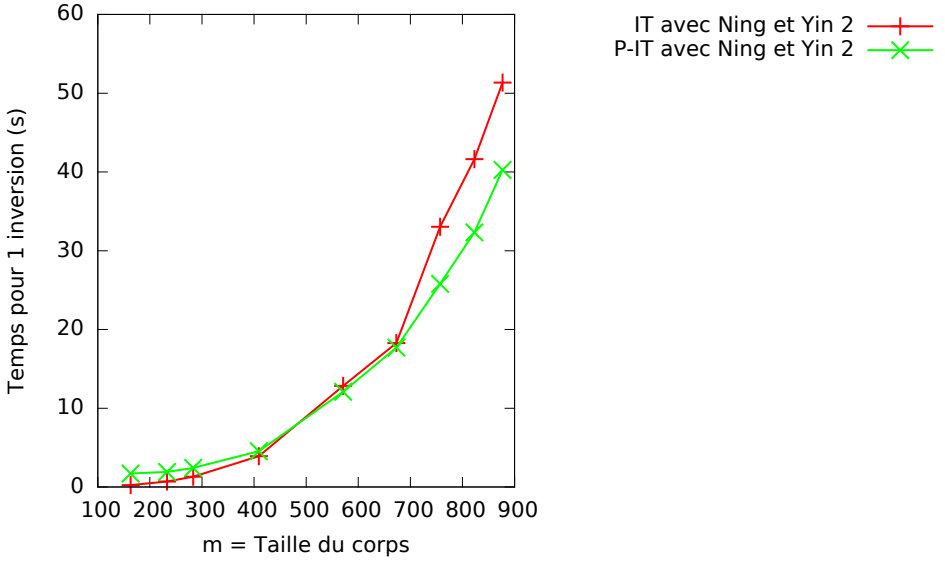


FIGURE A.4: Temps d'exécution pour 1 inversion en base normale en *IT* et *P-IT*.

Nous observons ici que le parallélisme offre un léger gain de vitesse quand le corps GF(2^m) devient suffisamment grand pour combler le surcout du parallélisme. Nous avons d'ailleurs rajouté des corps de tailles supérieures à celle du plus gros corps conseillé par le *NIST*. Le gain est notamment relié à l'écriture binaire de $m-1$, plus le poids de Hamming de cette représentation sera grande, plus le parallélisme aura un impact sur le calcul global. Pourquoi une telle différence entre l'inversion et la multiplication ? Il faut savoir que, concernant l'inversion, l'objet *Pool* créé pour la gestion du parallélisme en Python est partagé tout au long de l'exponentiation d'Itoh-Tsujii. C'est notamment la création de cet objet, qui est gourmand en temps. Dans nos mesures de performances sur la multiplication, cet objet était généré avant chacune d'elle. Nous aurions pu faire en sorte que la « *Pool* » soit là aussi partagée, mais le but était avant tout de montrer l'apport du parallélisme sur une multiplication, seule. En effet, il serait, à priori, tout aussi efficace de lancer n multiplications en parallèle que d'exploiter le parallélisme natif de l'algorithme de multiplication en base normale.

A.4 Conclusion

Nous avons montré dans ce chapitre que le parallélisme induit par les algorithmes de multiplication en base normale n'est pas toujours exploitable étant donné le surcout

que représente la synchronisation des processus sur un ordinateur. Nous avons montré, à l'inverse, que le parallélisme induit par l'exponentiation d'Itoh-Tsujii est exploitable sur nos ordinateurs modernes, à partir d'une taille, certes, relativement conséquente des corps $\mathrm{GF}(2^m)$ ($m \geq 571$).

Bibliographie

[1] Cybersecurity market report. Website. URL http://cybersecurityventures.com/cybersecurity-market-report/.

[2] S. Singh and C. Coqueret. *Histoire des codes secrets : de l'Égypte des Pharaons à l'ordinateur quantique*. Le Livre de poche. Librairie générale française, 2001. ISBN 9782253150978. URL https://books.google.fr/books?id=awSxHAAACAAJ.

[3] S. Perifel. *Complexité Algorithmique*. 2014. URL http://www.liafa.jussieu.fr/~sperifel/livre_complexite.html.

[4] E. Maiwald. *Fundamentals of Network Security*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2004. ISBN 0072230932, 9780072230932.

[5] A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX :5–38, 1883.

[6] Loi n 90-1170 du 29 décembre 1990 sur la réglementation des télćommunications. Loi française. URL http://legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000000533747.

[7] P.C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *CRYPTO'96 Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113.

[8] Koh ichi Nagao. Equations system coming from weil descent and subexponential attack for algebraic curve cryptosystem. Cryptology ePrint Archive, Report 2013/549, 2013. http://eprint.iacr.org/.

[9] Speeding the pollard and elliptic curve methods of factorization. *Math. Comp.*, 48, 1987. doi : 10.2307/2007888.

[10] S. Chari, J.R. Rao, and P. Rahatgi. Template attacks. *Cryptographic Hardware and Embedded Systems - CHES*, pages 13–28, 2002. doi : 10.1007/3-540-36400-5_3.

[11] J.C Bajard, L.S Didier, and P. Kornerup. An RNS montgomery modular multiplication algorithm. *IEEE TRANSACTIONS ON COMPUTERS*, 47 :766–776, 1998.

[12] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44, 1985. doi : 10.1090/S0025-5718-1985-0777282-X.

[13] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22. doi : 10.1109/TIT.1976.1055638.

[14] J.M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics 15*, pages 331–334, 1975. doi : 10.1007/bf01933667.

[15] H. Cohen. *A course in computational algebraic number theory.* 1996.

[16] R. Rivest, A. Shamir, and L. Adleman. Method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21 :120–126, 1978.

[17] C. Pomerance. Analysis and comparison of some integer factoring algorithms. *Computational Methods in Number Theory*, pages 89–139, 1982.

[18] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Pierrick Gaudry, Peter L. Montgomery, Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, Paul Zimmermann, and et al. Factorization of a 768-bit rsa modulus, 2010.

[19] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31*, pages 469–472, 1985. doi : 10.1109/TIT.1985.1057074.

[20] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[21] D. J. Bernstein and T. Lange. Explicit-formulas database. Website. URL https://www.hyperelliptic.org/EFD/.

[22] M. Joye. Fast point multiplication on elliptic curves without precomputation. *2nd International Workshop, WAIFI*, pages 33–46, 2008. doi : 10.1007/978-3-540-69499-1_4.

[23] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiaion using mixed coordinates. *International Conference on the Theory and Application of Cryptology and Information*, pages 51–65, 1998. doi : 10.1007/3-540-49649-1_6.

[24] N. P. Smart. The discrete logarithm problem on elliptic curves of trace one. *Journal of Cryptology*, pages 193–196, 1999. doi : 10.1007/s001459900052.

[25] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94, 2006. doi : 10.1109/JPROC.2005.862424.

[26] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. Springer, 2007.

[27] J.S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. *Cryptographic Hardware and Embedded Systems - CHES*, pages 292–302, 1999. doi : 10.1007/3-540-48059-5_25.

[28] D. Pamula. *Arithmetic Operators on GF(2^m) for Cryptographic Applications : Performance - Power Consumption - Security Tradeoffs*. Phd thesis, Univ. Rennes and Silesian Univ. of Technology, 2012.

[29] T. Chabrier. *Arithmetic recodings for ECC cryptoprocessors with protections against side-channel attacks*. Phd thesis, Univ. Rennes, 2013.

[30] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.) : Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2.

[31] F Herbaut, P.Y. Liardet, N. Méloni, Y. Teglia, and P. Véron. Random Euclidean Addition Chain Generation and Its Application to Point Multiplication. In *INDOCRYPT 2010*, volume 6498, pages 238–261, Hyderabad, India, 2010. Springer.

[32] Virtex-II platform FPGA user guide. URL http://www.xilinx.com/support/documentation/user_guides/ug002.pdf.

[33] O. Sentieys and A. Tisserand. Architectures reconfigurables FPGA. *Technologies logicielles Architectures des systèmes, Techniques de l'Ingénieur*, pages 1–22, 2012.

[34] A. P. Fournaris and O. G. Koufopavlou. GF(2^k) multipliers based on Montgomery multiplication algorithm. In *ISCAS (2)*, pages 849–852, 2004. URL http://dblp.uni-trier.de/db/conf/iscas/iscas2004-2.html#FournarisK04.

[35] C. Grabbe, M. Bednara, J. Teich, J. von zur Gathen, and J. Shokrollahi. Fpga designs of parallel high performance GF(2^{233}) multipliers. In *ISCAS (2)*, pages 268–271, 2003. URL http://dblp.uni-trier.de/db/conf/iscas/iscas2003-2.html#GrabbeBTGS03.

[36] J. Chu and M. Benaissa. GF(2^m) multiplier using polynomial residue number system. *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, 2008.

[37] G. Sutter, J.P. Deschamps, and J.L. Imaña. Efficient elliptic curve point multiplication using digit-serial binary field operations. *IEEE Transactions on Industrial Electronics*, 60(1) :217–225, 2013. URL http://dblp.uni-trier.de/db/journals/tie/tie60.html#SutterDI13.

[38] D. Pamula and E.Hrynkiewicz. Area-speed efficient modular architecture for GF(2^m) multipliers dedicated for cryptographic applications. *Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 13, 2013.

[39] M.A. Garcia-Martinez, R. R. Posada-Gomez, G. Morales-Luna, and F.Rodriguez-Henriquez. Fpga implementation of an efficient multiplier over finite fields GF(2^m). *Reconfigurable Computing and FPGAs (Reconfig)*, 2005.

[40] A. El-Sisi, S. M. Shohdy, and N. A. Ismail. Reconfigurable implementation of Karatsuba multiplier for Galois field in elliptic curves. In *TeNe*, pages 87–92. Springer, 2008. URL http://dblp.uni-trier.de/db/conf/cisse/tene2008.html#El-SisiSI08.

[41] E. Ferrer, D. Bollman, and O. Moreno. A fast finite field multiplier. In *ARC*, volume 4419 of *Lecture Notes in Computer Science*, pages 238–246. Springer, 2007. URL http://dblp.uni-trier.de/db/conf/arc/arc2007.html#FerrerBM07.

[42] J. L. Imaña and J. M. Sánchez. Efficient reconfigurable implementation of canonical and normal basis multipliers over galois fields GF(2^m) generated by aops. *J. VLSI Signal Process. Syst.*, 42 :285–296, 2006. doi : 10.1007/s11266-006-4189-x.

[43] R. Azarderakhsh and A. Reyhani-Masoleh. Low-complexity multiplier architectures for single and hybrid-double multiplications in Gaussian normal bases. *IEEE Trans. Comp.*, 62 :744–757, 2013. doi : 10.1109/TC.2012.22.

[44] J. Xie, P. K. Meher, and Z.H. Mao. High-throughput finite field multipliers using redundant basis for FPGA and ASIC implementations. *IEEE Trans. on Circuits and Systems*, 62-I :110–119, 2015.

[45] E. W. Knudsen. Elliptic scalar multiplication using point halving. In *Proc. Int. Conf. Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 135—-149, 1999. doi : 10.1007/978-3-540-48000-6_12.

[46] C. Negre and J.-M. Robert. New parallel approaches for scalar multiplication in elliptic curve over fields of small characteristic. Technical report, LIRMM, University of Perpignan UPVD, 2013.

[47] M. A. Hasan and C. Negre. Low space complexity multiplication over binary fields with dickson polynomial representation. *IEEE Trans. Comp.*, 60 :602–607, 2011. doi : 10.1109/TC.2010.132.

[48] H. W. Chang, W.-Y. Liang, and C. W. Chiou. Low cost dual-basis multiplier over GF(2^m) using multiplexer approach. In *Knowledge Discovery and Data Mining*, pages 185–192. Springer, 2012. doi : 10.1007/978-3-642-27708-5_25.

[49] J. K. Omura and J. L. Massey. Computational method and apparatus for finite field arithmetic. US Patent US4587627 A, 1986.

[50] Q. Liao. The Gaussian normal basis and its trace basis over finite fields. *J. Number Theory*, 132 :1507—1518, 2012. doi : 10.1016/j.jnt.2012.01.013.

[51] NIST. FIPS 186-2, digital signature standard (DSS), 2000.

[52] A. Reyhani-Masoleh. Efficient algorithms and architectures for field multiplication using Gaussian normal bases. *IEEE Trans. Comp.*, 55(1) :34–47, 2006.

[53] G.-L. Feng. A VLSI architecture for fast inversion in GF(2^m). *IEEE Trans. Comp.*, 38(10) :1383–1386, 1989. doi : 10.1109/12.35833.

[54] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3 :63–79, 1991. doi : 10.1007/BF00196789.

[55] R. Azarderakhsh, K. Jarvinen, and V. Dimitrov. Fast inversion in GF(2^m) with normal basis using hybrid-double multipliers. *IEEE Trans. Comp.*, 63 :1041–1047, 2014. doi : 10.1109/TC.2012.265.

[56] W. Drescher, K. Bachmann, and G. Fettweis. VLSI architecture for non-sequential inversion over GF(2^m) using the euclidean algorithm. In *Proc. Conf. Signal Processing Applications and Technology*, 1997.

[57] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and Computation*, 78 :171—-177, 1988. doi : 10.1016/0890-5401(88)90024-7.

[58] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1997.

[59] J. Hu, W. Guo J. Wei, and R.C.C. Cheung. Fast and generic inversion architectures over GF(2^m) using modified Itoh-Tsujii algorithms. *IEEE Transactions on Circuits and Systems II : Express Briefs*, 2015. doi : 10.1109/TCSII.2014.2387612. Accepted paper.

[60] Math. Dept. Bielefeld University. Shortest addition chains. Website, 2011. URL http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html.

[61] A.Weil. *Basic Number Theory*. Springer, 1995. doi : 10.1007/978-3-642-61945-8.

[62] A. Brodnik, J. Karlsson, J. I. Munro, and A. Nilsson. An O(1) solution to the prefix sum problem on a specialized memory architecture. *CoRR*, abs/cs/0601081, 2006. URL http://dblp.uni-trier.de/db/journals/corr/corr0601.html#abs-cs-0601081.

[63] S. Aravamuthan and V.R. Thumparthy. A parallelization of ECDSA resistant to simple power analysis attacks. *Communication Systems Software and Middleware COMSWARE*, 94 :1–7, 2007. doi : 10.1109/COMSWA.2007.382592.

[64] J. Taverne, A. Faz-Hernández, and F. Rodríguez-Henríquez. Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. doi : 10.1007/s13389-011-0017-8.

[65] S. M. H. Rodríguez and Francisco R.H. An FPGA arithmetic logic unit for computing scalar multiplication using the half-and-add method. In *ReConFig*. IEEE Computer Society, 2005. URL http://dblp.uni-trier.de/db/conf/reconfig/reconfig2005.html#RodriguezR05.

[66] L. Bossuet. Approche didactique pour l'enseignement de l'attaque DPA ciblant l'algorithme de chiffrement AES. *Journal sur l'enseignement des sciences et technologies de l'information et des systèmes*, 11, 2012. URL https://hal.archives-ouvertes.fr/hal-00753215.

[67] O. Choudary and M. G. Kuhn. Template attacks on different devices. In *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, pages 179–198, 2014. doi : 10.1007/978-3-319-10175-0_13.

[68] A. Klein. *Linear Feedback Shift Registers*. 2013. doi : 10.1007/978-1-4471-5079-4_2.

[69] Stephen D. Cohen. Primitive polynomials with a prescribed coefficient. *Finite Fields and Their Applications*, 12(3) :425 – 491, 2006. ISSN 1071-5797. doi : http://dx.doi.org/10.1016/j.ffa.2005.08.001. URL http://www.sciencedirect.com/science/article/pii/S1071579705000663.

[70] J.L Massey. Shift-register synthesis and bch decoding. *IEEE Trans. Information Theory*, pages 122–127, 1969.

[71] N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.X. Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, pages 390–406, 2012. doi : 10.1007/978-3-642-35999-6_25. URL http://dx.doi.org/10.1007/978-3-642-35999-6_25.

[72] C. K. Koc and T. Acar. Montgomery multiplication in GF(2^k). *Des. Codes Cryptography*, 14 :57–69, 1998. ISSN 0925-1022. doi : 10.1023/A:1008208521515. URL http://dx.doi.org/10.1023/A:1008208521515.

[73] A. J. Menezes, S.A. Vanstone, and P.C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

[74] E. Mastrovito. *VLSI Architectures for Computation in Galois Fields*. Phd thesis, Dep. Electrical Eng., Linkoping Univ., Sweden, 1991.

[75] D. Pamula, E.A. Hrynkiewicz, and A. Tisserand. Analysis of GF(2^{233}) multipliers regarding elliptic curve cryptosystem applications. In *PDeS - 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems*, pages 252–257, Brno, Czech Republic, 2012. URL https://hal.inria.fr/hal-00702622.

[76] J. Eynard. *RNS arithmetic approach of asymmetric cryptography*. Theses, Université Pierre et Marie Curie - Paris VI, 2015. URL https://tel.archives-ouvertes.fr/tel-01187925.

[77] A. P. Shenoy and Ramdas Kumaresan. Fast base extension using a redundant modulus in rns. *IEEE Trans. Computers*, 38 :292–297, 1989. URL http://dblp.uni-trier.de/db/journals/tc/tc38.html#ShenoyK89.

[78] J.C Bajard, M. Kaihara, and T. Plantard. Selected RNS bases for modular multiplication. *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pages 25–32. doi : 10.1109/ARITH.2009.20.

[79] H.O. Peitgen and D. Saupe, editors. *The Science of Fractal Images*. Springer-Verlag New York, Inc., New York, NY, USA, 1988. ISBN 0-387-96608-0.

[80] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 523–538, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3-540-67517-5. URL http://dl.acm.org/citation.cfm?id=1756169.1756220.

[81] Junfeng Chu and Mohammed Benaissa. Polynomial residue number system GF(2^m) multiplier using trinomials.

[82] J. Huang and University of North Texas. *FPGA Implementations of Elliptic Curve Cryptography and Tate Pairing Over Binary Field*. University of North Texas, 2007. ISBN 9780549318996. URL https://books.google.fr/books?id=VXAdJFe6688C.

[83] K.C.C Loi and Seok-Bum Ko. High performance scalable elliptic curve cryptosystem processor in GF(2^m). *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, 2013.

[84] Morales-Sandoval Feregrino-Uribe C. Cumplido R. and ; Algredo-Badillo I. A reconfigurable GF(2^m) elliptic curve cryptographic coprocessor. *Programmable Logic (SPL), 2011 VII Southern Conference*, 2011.

[85] M. Morales-Sandoval, C. Feregrino-Uribe, R. Cumplido, and I. Algredo-Badillo. A single formula and its implementation in FPGA for elliptic curve point addition using affine representation. *Journal of Circuits, Systems, and Computers*, 19(2) : 425–433, 2010. URL http://dblp.uni-trier.de/db/journals/jcsc/jcsc19.html#Morales-SandovalFCA10.

[86] P. G. Comba. Exponentiation cryptosystems on the ibm pc. *IBM Syst. J.*, 29(4) : 526–538, 1990. ISSN 0018-8670. doi : 10.1147/sj.294.0526. URL http://dx.doi.org/10.1147/sj.294.0526.

[87] K. Bigou. *Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques*. PhD thesis, 2014. URL http://www.theses.fr/2015REN1S087. Thèse de doctorat dirigée par Tisserand, Arnaud et Guillermin, Nicolas Informatique Rennes 1 2014.

[88] K. Bigou and A. Tisserand. Improving modular inversion in rns using the plus-minus method. *Cryptographic Hardware and Embedded Systems - CHES 2013*, 2013.

[89] Che Wun Chiou, Fu Hua Chou, and Yun Chi Yeh. Speeding up euclid's gcd algorithm with no magnitude comparisons. *Int. J. Inf. Comput. Secur.*, 4 :1–8, 2010. ISSN 1744-1765. doi : 10.1504/IJICS.2010.031855. URL http://dx.doi.org/10.1504/IJICS.2010.031855.

[90] Peng Ning and Yiqun Lisa Yin. Efficient software implementation for finite field multiplication in normal basis. In *In Information and Communications Security (ICICS), Springer-Verlag LNCS 2229*, pages 177–188. Springer-Verlag, 2001.

[91] C. Rebeiro, D. Selvakumar, and A. S. L. Devi. *Bitslice Implementation of AES*. 2006. doi : 10.1007/11935070_14.

Résumé/Abstract

La cryptographie et la problématique de la sécurité informatique deviennent des sujets de plus en plus prépondérants dans un monde hyper connecté et souvent embarqué. La cryptographie est un domaine dont l'objectif principal est de ≪ protéger ≫ l'information, de la rendre inintelligible à ceux ou à celles à qui elle n'est pas destinée. La cryptographie repose sur des algorithmes solides qui s'appuient eux-mêmes sur des problèmes mathématiques réputés difficiles (logarithme discret, factorisation des grands nombres etc). Bien qu'il soit complexe, sur papier, d'attaquer ces systèmes de protection, l'implantation matérielle ou logicielle, si elle est négligée (non protégée contre les attaques physiques), peut apporter à des entités malveillantes des renseignements complémentaires (temps d'exécution, consommation d'énergie etc) : on parle de canaux cachés ou de canaux auxiliaires. Nous avons, dans cette thèse, étudié deux aspects. Le premier est l'apport de nouvelles idées algorithmiques pour le calcul dans les corps finis binaires GF(2^m) utilisés dans le cadre de la cryptographie sur courbes elliptiques. Nous avons proposé deux nouvelles représentations des éléments du corps : la base normale permutée et le Phi-RNS. Ces deux nouveautés algorithmiques ont fait l'objet d'implémentations matérielles en FPGA à partir desquelles nous montrons que ces premières, sous certaines conditions, apportent un meilleur compromis temps-surface. Le deuxième aspect est la protection d'un crypto-processeur face à une attaque par canaux cachés (dite attaque par *templates*). Nous avons implémenté, en VHDL, un crypto-processeur complet et nous y avons exécuté, en parallèle, des algorithmes de *double-and-add* et *halve-and-add* afin d'accélérer le calcul de la multiplication scalaire et de rendre, de par ce même parallélisme, notre crypto-processeur moins vulnérable face à certaines attaques par canaux auxiliaires. Nous montrons que le parallélisme seul des calculs ne suffira pas et qu'il faudra marier le parallélisme à des méthodes plus conventionnelles pour assurer, à l'implémentation, une sécurité raisonnable.

Cryptography and security market is growing up at an annual rate of 17% according to some recent studies. Cryptography is known to be the science of secret. It is based on mathematical hard problems as integer factorization, the well-known discrete logarithm problem. Although those problems are trusted, software or hardware implementations of cryptographic algorithms can suffer from inherent weaknesses. Execution time, power consumption (...) can differ depending on secret informations such as the secret key. Because of that, some malicious attacks could be used to exploit these weak points and therefore can be used to break the whole crypto-system. In this thesis, we are interested in protecting our physical device from the so called side channel attacks as well as interested in proposing new GF(2^m) multiplication algorithms used over elliptic curves cryptography. As a protection, we first thought that parallel scalar multiplication (using halve-and-add and double-and-add algorithms both executed at the same time) would be a great countermeasure against template attacks. We showed that it was not the case and that parallelism could not be used as protection by itself : it had to be combined with more conventional countermeasures. We also proposed two new GF(2^m) representations we respectively named permuted normal basis (PNB) and Phi-RNS. Those two representations, under some requirements, can offer a great time-area trade-off on FPGAs.