Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe

Vincent Migliore

▶ **To cite this version:**

Vincent Migliore. Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe. Cryptographie et sécurité [cs.CR]. Université de Bretagne Sud, 2017. Français. NNT : 2017LORIS456 . tel-01791940

HAL Id: tel-01791940

https://theses.hal.science/tel-01791940v1

Submitted on 15 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE / UNIVERSITÉ DE BRETAGNE SUD

sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de :
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : STIC
École Doctorale SICMA

présentée par

Vincent MIGLIORE

Cybersécurité matérielle et conception de composants dédiés au calcul homomorphe

Thèse soutenue le 26-09-2017,
devant le jury composé de :

Dr. Guy GOGNIAT
Professeur, Université de Bretagne-Sud / Directeur de thèse
Dr. Arnaud TISSERAND
Directeur de Recherche CNRS / Co-directeur de thèse
Dr. Caroline FONTAINE
Chargée de Recherche CNRS / Co-directrice de thèse
Dr. Régis LEVEUGLE
Professeur, Grenoble INP - Phelma / Rapporteur
Dr. Nicolas SENDRIER
Directeur de recherche, INRIA / Rapporteur
Dr. Renaud SIRDEY
Directeur de recherche, CEA / Examinateur
Dr. Russell TESSIER
Professeur, University of Massachusetts (USA) / Examinateur
Dr. Vianney LAPOTRE
Maître de Conférences, l'Université de Bretagne-Sud / Examinateur
Dr. Sébastien CANARD

Résumé

L'émergence d'internet et l'amélioration des infrastructures de communication ont considérablement encouragé l'explosion des flux d'informations au niveau mondial. Cette évolution a été accompagnée par l'apparition de nouveaux besoins et de nouvelles attentes de la part des consommateurs. Communiquer avec ses proches ou ses collaborateurs, stocker des documents de travail, des fichiers multimédia, utiliser des services innovants traitant nos documents personnels, tout cela se traduit immanquablement par le partage, avec des tiers, d'informations potentiellement sensibles. Ces tiers, s'ils ne sont pas de confiance, peuvent réutiliser à notre insu les données sensibles que l'on leur a confiées.
Dans ce contexte, le chiffrement homomorphe apporte une bonne solution. Il permet de cacher aux yeux des tiers les données qu'ils sont en train de manipuler. Cependant, à l'heure actuelle, le chiffrement homomorphe reste complexe. Pour faire des opérations sur des données de quelques bits (données en clair), il est nécessaire de manipuler des opérandes sur quelques millions de bits (données chiffrées). Ainsi, une opération normalement simple devient longue en termes de temps de calcul.
Dans cette étude, nous avons cherché à rendre le chiffrement homomorphe plus pratique en concevant un accélérateur spécifique. Nous nous sommes basés sur une approche de type co-conception logicielle/matérielle utilisant l'algorithme de Karatsuba. En particulier, notre approche est compatible avec le *batching*, qui permet de stocker plusieurs bits d'informations dans un même chiffré.
Notre étude démontre que le *batching* peut être implémenté sans surcoût important comparé à l'approche sans *batching*, et permet à la fois de réduire les temps de calcul (calculs effectués en parallèle) et de réduire le rapport entre la taille des données chiffrées et des données en clair.

Abstract

The emergence of internet and the improvement of communication infrastructures have considerably increased the information flow around the world. This development has come with the emergence of new needs and new expectations from consumers. Communicate with family or colleagues, store documents or multimedia files, using innovative services which processes our personal data, all of this implies sharing with third parties some potentially sensitive data. If third parties are untrusted, they can manipulate without our agreement data we share with them.
In this context, homomorphic encryption can be a good solution. Homomorphic encryption can hide to the third parties the data they are processing. However, at this point, homomorphic encryption is still complex. To process a few bits of clear data (cleartext), one needs to manage a few million bits of encrypted data (ciphertext). Thus, a computation which is usually simple becomes very costly in terms of computation time.
In this work, we have improved the practicability of homomorphic encryption by implementing a specific accelerator. We have followed a software/hardware co-design approach with the help of Karatsuba algorithm. In particular, our approach is compatible with batching, a technique that "packs" several messages into one ciphertext.
Our work demonstrates that the batching can be implemented at no important additional cost compared to non-batching approaches, and allows both reducing computation time (operations are processed in parallel) and the ciphertext/cleartext ratio.

n d'ordre : 456

Université de Bretagne Sud
Centre d'Enseignement et de Recherche C. Hyugens - rue de Saint Maudé - 56321 LORIENT
Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70

Table des matières

Liste des Figures 6

Liste des Tables 8

Notations 9

1 Le chiffrement homomorphe : Introduction et état de l'art 11
- 1.1 Enjeux autour de la cryptographie actuelle et future . . . 11
  - 1.1.1 Cryptographie actuelle . . . 11
  - 1.1.2 La cryptographie post-quantique . . . 14
- 1.2 Le chiffrement homomorphe . . . 14
  - 1.2.1 Prémices . . . 14
  - 1.2.2 Les différentes familles de chiffrement homomorphe . . . 16
  - 1.2.3 Utilisation du chiffrement homomorphe en pratique . . . 18
- 1.3 Cas d'étude : le chiffrement FV . . . 19
  - 1.3.1 R-LWE . . . 20
  - 1.3.2 FV . . . 21
  - 1.3.3 Choix des paramètres . . . 27
  - 1.3.4 La technique du *batching* . . . 29
- 1.4 Multiplication de grands opérandes : état de l'art . . . 30
  - 1.4.1 L'algorithme naïf . . . 31
  - 1.4.2 L'algorithme de Karatsuba-Ofman [KO62] . . . 32
  - 1.4.3 L'algorithme de Toom-Cook [Too63] . . . 33
  - 1.4.4 L'algorithme de la NTT [Pol71] . . . 35
  - 1.4.5 Optimiser la NTT pour l'homomorphe : l'utilisation de la NWC 37
  - 1.4.6 Quelques mots à propos du système RNS . . . 38
- 1.5 Implémentations logicielles et/ou matérielles . . . 39
  - 1.5.1 Les implémentations logicielles . . . 39
  - 1.5.2 Les implémentations matérielles . . . 41
- 1.6 Conclusion . . . 43
- 1.7 Approche choisie . . . 45

2 Implémentation logicielle des primitives de FV 47

2.1 Bibliothèques logicielles existantes 47
2.2 Approche retenue 48
2.3 Représentation des nombres 49
2.4 Implémentation des opérations arithmétiques sur les entiers 51
2.4.1 L'addition de nombres dans la représentation en complément à deux 51
2.4.2 La soustraction de nombres dans la représentation en complément à deux 52
2.4.3 Détermination du nombre d'opérations successives possibles avant propagation de retenue 54
2.5 Représentation des polynômes 55
2.6 Présentation de la programmation vectorielle 56
2.7 Opérations arithmétiques 58
2.7.1 Addition/soustraction polynomiale 58
2.7.2 Réduction modulaire polynomiale 58
2.7.3 *Batching* 60
2.7.4 Le choix du module 62
2.8 Conclusion 62

3 Co-conception logicielle/matérielle pour l'accélération de l'opération de chiffrement et de multiplication homomorphe de FV 63

3.1 Conception d'un accélérateur de la multiplication polynomiale par l'utilisation de l'algorithme de Karatsuba 63
3.1.1 Présentation de l'architecture 64
3.1.2 Présentation de la plateforme retenue pour l'accélérateur matériel 65
3.1.3 Présentation du bus PCI-E 67
3.1.4 Présentation de RIFFA 69
3.1.5 Implémentation logicielle de Karatsuba 71
3.1.6 Implémentation matérielle de Karatsuba 74
3.2 Adaptation de l'accélérateur de Karatsuba pour la multiplication homomorphe 85
3.2.1 Nouvelle architecture de haut-niveau 85
3.2.2 Adaptation de FV.Mult 87

3.2.3 Adaptation de FV.Relin 91
3.3 Adaptation de l'accélérateur de Karatsuba pour le chiffrement 97
3.3.1 Nouvelle architecture de haut-niveau 97
3.3.2 Adaptation de FV.Encrypt 97
3.3.3 Stratégie d'implémentation 98
3.3.4 Modification de l'application logicielle 99
3.3.5 Modification de l'accélérateur matériel 99
3.4 Conclusion 100

4 Résultats d'implémentation 103
4.1 Réduction modulaire polynomiale et *batching* 103
4.2 Accélération matérielle 105
4.2.1 Efficacité du lien PCI-E 107
4.2.2 FV.Mult et FV.Relin 107
4.2.3 FV.Encrypt 117
4.3 Conclusion 122

5 Conclusion de l'étude et perspectives 125
5.1 Conclusion 125
5.2 Perspectives 126

Liste des publications 129

Liste des Figures

1.1 Exemple de la signature numérique d'un document. 12
1.2 Scénario d'utilisation du chiffrement homomorphe. 15
1.3 Carte mentale représentant la diversité des chiffrements homomorphes. 18
1.4 Additionneur complet 1 bit. 19
1.5 Additionneur 4 bits. 20
1.6 Organigramme de l'utilisation du chiffrement FV avec les noms des fonctions associées. 22
1.7 Architecture de l'accélérateur matériel proposé dans [PG14]. 44
1.8 Architecture de l'accélérateur matériel proposé dans [SRJV$^+$]. 45

2.1 Représentation d'un nombre entier a en base β avec une notation redondante. 50
2.2 Addition de nombres positifs. 52
2.3 Propagation de la retenue. 53
2.4 Problème d'inversion dans la représentation en complément à deux. . 55
2.5 Représentation des polynômes. 56
2.6 Exemple d'une opération d'addition 32 bits de deux vecteurs de 128 bits avec la programmation vectorielle. 58
2.7 Représentation d'un polynôme de 3072 coefficients sur 140 bits en mémoire, optimisé pour la programmation vectorielle. 58

3.1 Stratégie d'implémentation de l'accélération de la multiplication polynomiale avec Karatsuba dans une approche de co-conception. 66
3.2 Présentation de la plateforme DE5-net, possédant notamment un puissant FPGA Stratix V GX de chez Altera. 67
3.3 Architecture du bus PCI-E comme implémenté dans un ordinateur généraliste. 68
3.4 Performances de RIFFA d'après la documentation [JRHK15]. 70
3.5 Présentation de l'arborescence des opérations dans l'algorithme de Karatsuba pour trois pré-récursions. 73
3.6 Présentation schématique des post-traitements de Karatsuba. 75
3.7 Architecture retenue pour l'accélérateur matériel de Karatsuba pour une multiplication polynomiale de polynômes de 2560 coefficients, codés sur 108 bits. 76

3.8 Présentation schématique des pré-traitements de Karatsuba dans le composant matériel. 77

3.9 Représentation de l'ordonnancement des sous-polynômes en sortie du pré-traitement et du pré-crossbar. 79

3.10 Ordonnancement du multiplieur de sous-polynômes en utilisant l'algorithme de multiplication classique. 82

3.11 Présentation schématique de l'implémentation des post-traitements de Karatsuba dans le composant matériel. 83

3.12 Ordonnancement des sous-polynômes après multiplication des sous-produits de Karatsuba. 84

3.13 Ordonnancement des sous-polynômes après les post-récursions 3, 4 et 5. 84

3.14 Évaluation de la quantité de mémoire requise en fonction du nombre de post-récursions implémentées sur le composant matériel. 86

3.15 Stratégie d'implémentation de l'accélération de FV.mult et FV.relin avec Karatsuba dans une approche de co-conception. 89

3.16 Stratégie d'implémentation de l'accélération FV.encrypt avec Karatsuba dans une approche de co-conception. 101

3.17 Ordonnancement de 4×{Multiplieur Entier 10×135 bits}. 102

3.18 Représentation en mémoire du polynôme U. 102

4.1 Comparaison du temps de transfert annoncé dans la documentation de RIFFA, et le temps de transfert mesuré sur notre machine. 108

4.2 Temps de calcul de la multiplication homomorphe de FV avec notre accélérateur basé sur Karatsuba. 112

4.3 Temps de calcul de un à quatre produits/accumulations successifs dans FV.Encrypt. 120

Liste des Tables

1.1 Valeur maximale pour $\log_2 q$ pour une dimension donnée n d'une instance R-LWE, avec λ le niveau de sécurité et $\sigma_{err} = 2\sqrt{n}$. 21

1.2 Valeur minimale de $\log_2 q$ après L multiplications homomorphes permettant une opération de déchiffrement cohérente dans le schéma de chiffrement FV, pour les paramètres suivants : $n = 4096$, $\omega = 2^{32}$, $t = 2$. 28

1.3 Paramètres de sécurité pour le schéma FV. 29

1.4 Temps de calcul des opérations de déchiffrement et de multiplication homomorphe pour différentes bibliothèques actuelles. 40

1.5 Temps de calcul des opérations de multiplication homomorphe pour FV-FULL-RNS présenté dans [BEHZ16]. 41

1.6 Présentation des résultats d'implémentations matérielles de la littérature. 43

1.7 Exemples de multiplications polynomiales possibles avec l'algorithme de Karatsuba. 46

2.1 Influence des paramètres α et β de la représentation proposée équation 2.3 sur la taille maximale des nombres avec un nombre fixé de chiffres. 52

2.2 Exemples d'implémentations existantes d'opérateurs vectoriels sur processeurs. 56

3.1 Comparaison des différentes générations de PCI-E. 69

3.2 Configurations possibles des FPGA de chez Altera (famille Stratix V) pour une interface avec le PCI-E. 70

3.3 Présentation de la réduction maximale du nombre de lignes de sous-polynômes avec l'aide d'un pré-crossbar. 80

3.4 Impact des approximations (1) et (2) introduites dans FV sur le niveau de bruit pour l'algorithme de Karatsuba. 90

4.1 Exemples de polynômes cyclotomiques et leur temps de calcul en utilisant la démarché présentée section 1.3.4. 106

4.2 Configuration de l'accélérateur matériel pour les profondeurs multiplicatives 3, 4, 7 et 8. (tailles et volumes représentés en bits) 110

4.3 Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (2560, 108)$. Les temps de calcul sont donnés pour 1000 essais. 113

4.4 Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (3072, 135)$. Les temps de calcul sont donnés pour 1000 essais. 113

4.5 Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (5120, 216)$. Les temps de calcul sont donnés pour 1000 essais. 114

4.6 Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (6144, 243)$. Les temps de calcul sont donnés pour 1000 essais. 114

4.7 Résumé des résultats d'implémentation pour la multiplication homomorphe de FV. 118

4.8 Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (2560, 108)$. Les temps de calcul sont donnés pour 1000 essais. 119

4.9 Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (3072, 135)$. Les temps de calcul sont donnés pour 1000 essais. 121

4.10 Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (5120, 216)$. Les temps de calcul sont donnés pour 1000 essais. 121

4.11 Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (6144, 243)$. Les temps de calcul sont donnés pour 1000 essais. 122

4.12 Résumé des résultats de l'implémentation du produit/accumulation nécessaire lors du chiffrement de FV (quatre produits/accumulations). 123

Notations

Nous présentons ici l'ensemble des notations qui seront utilisées par la suite. Un polynôme est représenté en majuscule et un nombre entier en minuscule. Soit A un polynôme, on représente par $a_{(i)}$ son $i^{\text{ème}}$ coefficient. Les vecteurs sont représentés en **gras**. Ainsi, $\mathbf{A}$ représente un vecteur de polynômes, et $\mathbf{a}$ un vecteur d'entiers. Soit $\mathbf{A}$ un vecteur, $\mathbf{A}[i]$ représente son $i^{\text{ème}}$ élément.

Les autres notations qui seront utilisées par la suite sont résumées ci-dessous :

Opérations sur les entiers/polynômes/vecteurs		
$a+b$ (resp. $A+B$)	$\rightarrow$	Addition entière entre a et b (resp. addition polynomiale entre A et B)
$a \cdot b$ (resp. $A \cdot B$)	$\rightarrow$	Multiplication entière entre a et b (resp. multiplication polynomiale entre A et B)
$a_{(j_0 \rightarrow j_1)}$	$\rightarrow$	Entier dont la séquence binaire correspond aux bits de a compris entre les indices j_0 et j_1
$A_{(j_0 \rightarrow j_1)}$	$\rightarrow$	Polynôme dont le coefficient d'indice i correspond à $a_{i,(j_0 \rightarrow j_1)}$
$\langle \cdot, \cdot \rangle$	$\rightarrow$	Produit scalaire : $\langle \mathbf{A}, \mathbf{B} \rangle = \sum_i \mathbf{A}[i] \cdot \mathbf{B}[i]$
$a = (1623)_b$	$\rightarrow$	Représentation de a dans la base b
$\bar{a}$	$\rightarrow$	Inverse binaire de a ($(0000)_2 \rightarrow (1111)_2$)

Opérateurs		
$[\cdot]_q$	$\rightarrow$	Modulo l'entier q
$\lfloor \cdot \rfloor$	$\rightarrow$	Arrondi à l'entier inférieur
$\lceil \cdot \rceil$	$\rightarrow$	Arrondi à l'entier supérieur
$\lfloor \cdot \rceil$	$\rightarrow$	Arrondi à l'entier au plus proche avec arrondi à l'entier supérieur en cas d'égalité
$\|\| \cdot \|\|_\infty$	$\rightarrow$	Norme infinie : $\|\|A\|\|_\infty = \max_{(i)} \|a_i\|$

Fonctions		
ln	$\rightarrow$	Logarithme en base e
log	$\rightarrow$	Logarithme en base 10
$\log_2$	$\rightarrow$	Logarithme en base 2

Distributions		
R	$\rightarrow$	$\mathbb{Z}[x]/f(x)$, pour $f(x)$ polynôme cyclotomique de degré n
R_q	$\rightarrow$	$\mathbb{Z}_q[x]/f(x)$, pour un entier positif q
$A \leftarrow U_R$	$\rightarrow$	Polynôme uniforme et aléatoire sur R
$A \leftarrow B_R$	$\rightarrow$	Polynôme binaire et aléatoire sur R ($U_R \cap \{1\}$)
$A \leftarrow B_{\{-1,0,1\},R}$	$\rightarrow$	Polynôme dont les coefficients appartiennent à $\{-1, 0, 1\}$ et aléatoire sur R ($U_R \cap \{-1, 0, 1\}$)
$A \leftarrow D_{R,\sigma}$	$\rightarrow$	Polynôme sur R dont les coefficients sont générés aléatoirement à partir d'une distribution gaussienne discrète de paramètre σ
$A \leftarrow \chi_R$	$\rightarrow$	Polynôme sur R suivant la loi de probabilité χ

Par la suite, et sauf mention contraire explicite, $R_q = \mathbb{Z}_q[x]/f(x)$, avec $f(x)$ un polynôme cyclotomique de degré n, et q un entier positif également nommé module.

1

Le chiffrement homomorphe : Introduction et état de l'art

L'objectif de ce chapitre est d'introduire les schémas de chiffrement homomorphe, leurs contextes d'utilisation ainsi que les enjeux et défis scientifiques à relever. Pour ce faire, ce chapitre est organisé comme suit :

— Une présentation succincte de l'utilisation actuelle de la cryptographie usuelle, ainsi que ses limites amenant à s'intéresser au chiffrement homomorphe ;
— Une étude plus approfondie sur le chiffrement homomorphe afin de déterminer les enjeux et défis à relever autour de ces nouveaux schémas (notamment sur la multiplication homomorphe) ;
— Un focus sur le schéma FV qui semble être l'un des schémas les plus prometteurs à l'heure actuelle ;
— Une présentation des différents algorithmes de multiplication de polynômes, ainsi qu'une analyse de leurs avantages et inconvénients ;
— Un état de l'art sur les implémentations de chiffrements homomorphes, quelles soient logicielles ou matérielles ;
— Une présentation de notre démarche pour l'accélération du chiffrement homomorphe.

1.1 Enjeux autour de la cryptographie actuelle et future

1.1.1 Cryptographie actuelle

L'émergence d'internet et l'amélioration des infrastructures de communication ont considérablement encouragé l'explosion des flux d'informations au niveau mondial. Cette évolution a été accompagnée par l'apparition de nouveaux besoins et de nouvelles attentes de la part des consommateurs. Communiquer avec ses proches ou ses collaborateurs, stocker des documents de travail, des fichiers multimédia, utiliser des services innovants traitant nos documents personnels, tout cela se traduit immanquablement par le partage, avec des tiers, d'informations potentiellement

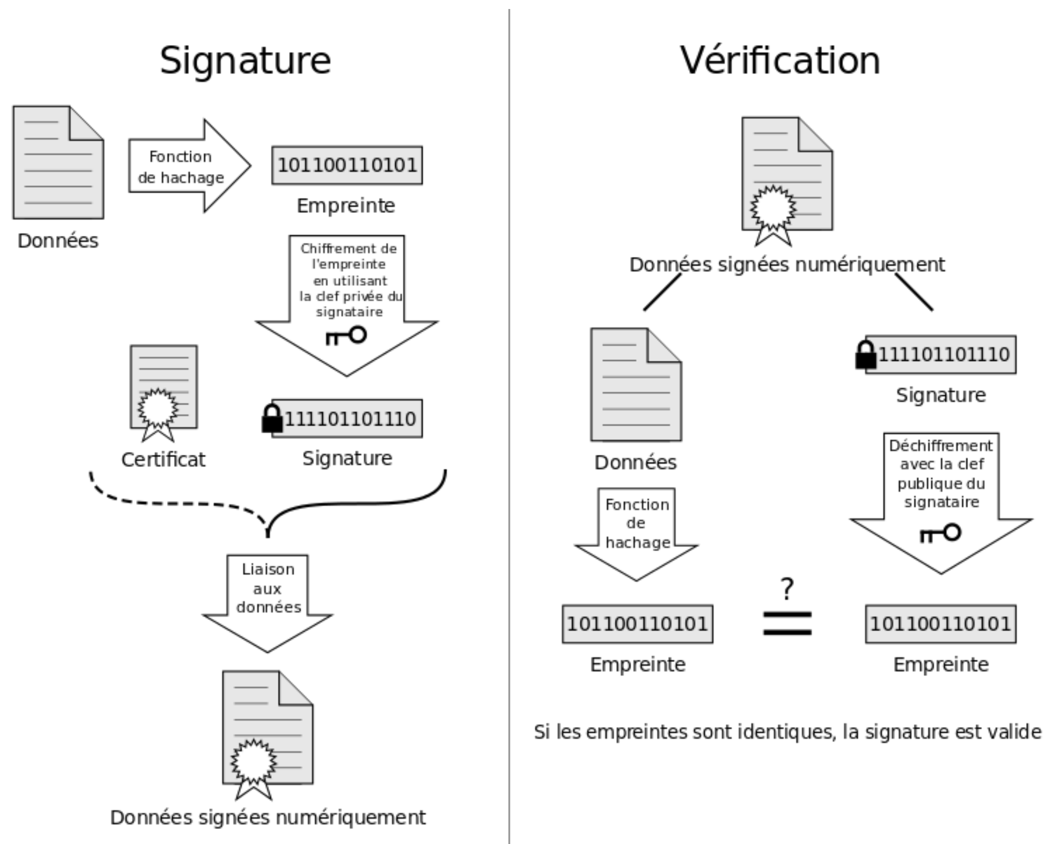


Figure 1.1 – Exemple de la signature numérique d'un document.
Source : `https://fr.wikipedia.org/wiki/Signature_numérique`.

sensibles, impliquant une nécessité de protéger ces données.

La cryptographie actuelle permet d'apporter une première réponse à ces nouveaux besoins, et permet de garantir certains points essentiels, notamment :

1) L'authentification : vérifier que la communication s'effectue avec le bon interlocuteur (pas d'usurpation d'identité) ;
2) L'intégrité : le document reçu n'a pas été altéré ou modifié ;
3) La confidentialité : la communication est confidentielle (seul le destinataire a accès aux données que j'envoie).

Pour ce faire, la cryptographie se base sur trois grandes familles d'algorithmes :

1) Les chiffrements asymétriques ;
2) Les chiffrements symétriques ;
3) Les fonctions de hachage.

Les algorithmes de chiffrements asymétriques sont basés sur deux fonctions, une première fonction qui permet de chiffrer une information (permettant de cacher l'information) et une seconde fonction de déchiffrement qui permet à partir d'une donnée chiffrée, de récupérer le message en clair. Pour chacune de ces opérations, une clé est associée permettant de mener à bien les calculs. Selon les protocoles utilisés, la clé de chiffrement ou de déchiffrement peut être soit publique (utilisable par

tout le monde), soit privée. Ces algorithmes permettent, par exemple, une communication sécurisée entre deux interlocuteurs (mais de manière unidirectionnelle). Les algorithmes les plus connus sont RSA [RSA78], El-Gamal [Gam85] et ECC [Mil85].

Les algorithmes de chiffrements symétriques permettent, contrairement au chiffrement asymétrique, un échange bidirectionnel d'information via l'utilisation d'une unique clé. L'algorithme le plus connu est l'AES [DR01].

Les fonctions de hachage sont des fonctions qui permettent de récupérer des propriétés caractéristiques d'un message sans avoir accès au message lui-même. Lorsque l'on applique une telle fonction sur un message, l'algorithme va nous renvoyer une valeur que l'on nomme empreinte (ou condensat) et qui est unique par message. Cette valeur (ou haché) peut alors être comparée à une valeur connue pour déterminer si le message haché possède les propriétés requises. L'algorithme le plus connu est SHA2 [SHA].

En utilisant de manière conjointe ces différents algorithmes via des protocoles de communication, il est alors possible de proposer des services sécurisés. Afin d'illustrer ce propos, prenons l'exemple figure 1.1 qui présente le cas de la signature numérique d'un document. Cette signature a pour but de vérifier que le document reçu n'a été ni altéré, ni modifié par un tier. La personne va recevoir, adjoint au document, une signature du document. Pour ce faire, les deux interlocuteurs se sont entendus sur une fonction de hachage, et une méthode de chiffrement asymétrique. La personne qui possède le document possède également une clé permettant de chiffrer des données, et la personne qui doit réceptionner le document une clé permettant de déchiffrer des données. Pour obtenir cette signature, deux algorithmes vont être appliqués au document : Une fonction de hachage pour obtenir une empreinte du document, puis cette empreinte va être chiffrée. Le client, pour vérifier l'authenticité de la donnée, va alors en parallèle hacher le document qu'il a reçu, et déchiffrer la signature. Si la donnée n'a pas été altérée, alors les empreintes seront identiques.

Il existe bien entendu un certain nombre d'autres protocoles de sécurité qui sont adaptés à des contextes donnés. On peut citer par exemple SSL qui permet de sécuriser la navigation sur des pages web, ou encore TLS qui permet le transfert sécurisé de documents.

Malgré la diversité des mécanismes de protection, il reste un problème majeur que ne résout pas la cryptographie actuelle. Supposons que l'on souhaite utiliser un service internet qui permet de monitorer notre activité physique. Ce service est effectué par un serveur distant qui reçoit nos données quotidiennes et fait le bilan de notre activité. Ces données sont transmises de manière sécurisée au serveur via le protocole SSL. Cependant, pour traiter nos données, le serveur doit déchiffrer nos données. À partir de ce moment, le serveur possède donc l'intégralité de nos données personnelles, et si le serveur n'est pas de confiance, ces données peuvent très bien être utilisées à d'autres fins (ciblage publicitaire, revente, ...). C'est dans ce contexte que le chiffrement homomorphe apporte une solution à ce problème. Il permet de faire des calculs sur des données, en ne manipulant qu'une version chiffrée

des données. Ainsi, sans la connaissance de la clé de déchiffrement, le serveur est dans l'incapacité d'avoir accès aux données personnelles. On peut alors considérer le chiffrement homomorphe comme une famille d'algorithmes cryptographiques et pouvoir bâtir à partir d'eux de nouveaux services soucieux de la protection des données personnelles.

1.1.2 La cryptographie post-quantique

Depuis les recommandations du NIST [NIS], il ne semble plus raisonnable à ce jour de proposer des algorithmes de cryptographie qui ne sont pas résistants à l'ordinateur quantique. Les ordinateurs quantiques exploitent les propriétés de la physique quantique afin de résoudre des problèmes mathématiques réputés difficiles sur des ordinateurs standards. L'ordinateur quantique étant capable de casser de nombreux chiffrements asymétriques, son déploiement à grande échelle menace la sécurité et l'intégrité des données sur le web. Pour donner un exemple concret, un ordinateur quantique est capable de résoudre en un temps polynomial la décomposition en facteurs premiers d'un entier donné N (algorithme de Shor [Sho97]), ce qui casse le chiffrement El-Gamal [Gam85].

À l'heure actuelle, les schémas de chiffrement homomorphe que nous présenterons sont robustes aux attaques quantiques, et donc leur application à moyen/long terme n'est pas encore remis en cause par la cryptographie post-quantique.

1.2 Le chiffrement homomorphe

1.2.1 Prémices

Le chiffrement homomorphe est apparu en 2009 avec les travaux de Carlos Aguilar et al. [AMGH10] ainsi que de Craig Gentry [Gen09]. L'objectif était de proposer un schéma de chiffrement capable de cacher aux yeux du serveur de calcul les données qu'il était en train de manipuler (dit opérations homomorphes). Le caractère homomorphe de certains schémas de chiffrement n'est cependant pas nouveau, ni même récent. Par exemple, le schéma RSA datant de 1977 est homomorphe pour la multiplication, et le schéma de Pailler [Pai99] datant de 1999 est homomorphe pour l'addition. En revanche, la vraie révolution des travaux datant de 2009 est de pouvoir permettre avec un même schéma à la fois les opérations d'addition et de multiplication. La figure 1.2 présente le fonctionnement de base du chiffrement homomorphe. Le client va tout d'abord générer deux types de clés, une clé publique pour chiffrer les données, et une clé secrète pour les déchiffrer. Le client va ensuite chiffrer ses données avec sa clé publique, et envoyer le résultat au serveur de calcul. Le serveur va alors procéder au traitement des données reçues en fonction du service proposé au client (traitement d'image, monitoring, ...). Le résultat du calcul, qui est donc chiffré, va être renvoyé au client qui récupèrera le résultat en clair après

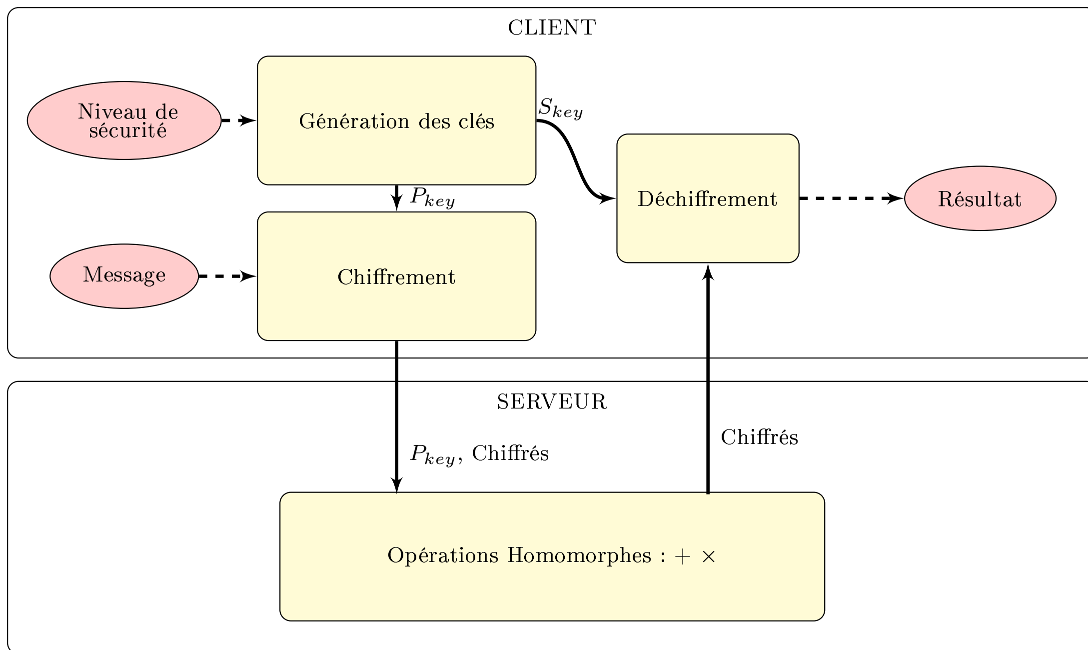


FIGURE 1.2 – Scénario d'utilisation du chiffrement homomorphe.

déchiffrement. La clé publique peut également être divulguée au serveur si celui-ci à besoin de chiffrer ses propres données pour les inclure dans les traitements.

Quelques spécificités viennent cependant limiter le caractère pratique du chiffrement homomorphe :

(1) Les opérandes à manipuler pour les calculs homomorphes peuvent atteindre plusieurs millions de bits ;
(2) Les données doivent être représentées comme une suite de nombres entiers, avec un chiffré par entier ;
(3) Seules des opérations élémentaires sur les données sont possibles (addition, soustraction et multiplication) ;
(4) Le nombre d'opérations successives possibles est limité ;

La limitation (1) est la limitation la plus impactante du chiffrement homomorphe. Contrairement à la cryptographie actuelle où les opérations à effectuer se font sur des données de plusieurs centaines de bits, le chiffrement homomorphe demande la manipulation de données jusqu'à plusieurs millions de bits. Cela va impacter tous les étages de la cryptographie homomorphe : du client qui va devoir chiffrer les données, au serveur qui va devoir traiter les données, en passant par le réseau qui va devoir acheminer les chiffrés.

Les limitations (2) et (3) viennent également compliquer son implémentation. Le chiffrement homomorphe actuel n'est capable de traiter que des données représentées comme des entiers, et seulement avec des opérations élémentaires (addition, soustraction et multiplication entière). Pour calculer un algorithme avec le chiffrement homomorphe (qui possède par exemple des conditions, des boucles, des comparaisons), il va falloir transformer complètement les calculs pour ne comporter

que ces opérations élémentaires. Ainsi, un algorithme simple peut très rapidement devenir extrêmement complexe. Dans [FSF+13], des exemples d'implémentations d'algorithmes simples en homomorphe sont présentés. Pour une simple opération de tri par ordre croissant de 10 valeurs sur 4 bits, 1620 additions et 1350 multiplications sur des opérandes de plusieurs millions de bits sont requises. Ces opérations vont donc demander à la fois de nombreuses ressources de calculs, des temps de traitements qui peuvent atteindre plusieurs secondes par algorithme, et par la même un coût énergétique non négligeable.

La limitation (4) réduit également le nombre d'opérations que l'on peut effectuer dans le domaine de l'homomorphe. De manière simplifiée, chaque chiffré possède un bruit initial. Ce bruit, au fur et à mesure des opérations, va s'amplifier jusqu'à atteindre un seuil limite avant de rendre le résultat de l'opération de déchiffrement incohérent. L'amplification du bruit étant plus marqué lors des multiplications homomorphes que lors des additions, on donne souvent le nombre d'opérations possibles pour des paramètres donnés en fonction du nombre de multiplications successives uniquement. On appelle cette valeur la profondeur multiplicative. Ces schémas, limités en nombre d'opérations, sont de type *Somewhat-Homomorphic Encryption* (SWHE). Cependant, avec la méthode du *bootstrapping* proposée dans [Gen09], il est possible de dépasser cette limite et permettre un nombre illimité d'opérations. Ces schémas sont dits de type *Fully-Homomorphic Encryption* (FHE). Cette méthode réalise une opération de déchiffrement puis de chiffrement de manière homomorphe afin de générer de nouveaux chiffrés neufs. Cette technique a longtemps été considérée comme bien trop complexe et coûteuse pour être applicable. Au jour d'aujourd'hui, le *bootstrapping* reste particulièrement complexe, mais son caractère peu pratique est à nuancer grâce aux travaux présentés dans [CGGI16]. Cette implémentation réduit l'opération de boostraping à 0.1 s alors que précédemment, cette opération pouvait durer plusieurs secondes.

À la vue des différentes limitations présentées, il est indéniable que le chiffrement homomorphe n'est pas une solution adaptée à tous les services en ligne. Cependant, pour certaines applications spécifiques (par exemple la gestion de données médicales ou de santé), le chiffrement homomorphe peut proposer une solution pour le traitement de données sensibles.

1.2.2 Les différentes familles de chiffrement homomorphe

Le chiffrement homomorphe reste encore à ce jour, bien que cela semble se stabiliser, un domaine de recherche en pleine évolution. Ainsi, il existe une multitude de schémas homomorphes basés sur différents problèmes mathématiques réputés difficiles à résoudre (souvent réduits à un problème difficile sur les réseaux euclidiens [Ajt96]). Hormis les premiers schémas basés sur une application directe d'un problème de réseau euclidien, on peut distinguer trois grandes familles de problèmes : le problème de l'*Approximate-GCD* (A-GCD) [HG01], le problème *Learning With Errors* (LWE) [Reg09] ainsi que son homologue basé sur les polynômes *Ring-*

LWE (R-LWE) [LPR10], et finalement le problème NTRU [HPS98].

L'application du problème A-GCD pour le chiffrement homomorphe est apparu en 2010 avec le schéma DHGV [DGHV10], dont le but était de simplifier les schémas homomorphes datant de 2009 en les appliquant à des opérandes entières. S'en est suivi différentes optimisations notamment avec les travaux [CMNT11] [CNT12] [CLT14] afin d'améliorer son utilisation pratique. Le principal problème de ces schémas est l'explosion de la taille de la clé publique à mesure que l'on souhaite augmenter le niveau de sécurité. Dans [Lep14] par exemple, pour un niveau de sécurité de 72 bits, les applications numériques montrent que la clé publique a une taille dépassant les 4 Go. La clé publique étant requise pour les opérations homomorphes, nous avons donc écarté ces schémas dans notre étude, bien qu'ils soient assez simples à manipuler (addition et multiplication d'entiers sur des millions de bits).

Le problème LWE et son application à l'homomorphe a été proposé dans [Reg09] à la même période que son homologue basé sur A-GCD. L'intérêt de ces schémas est que la clé publique ainsi que les chiffrés ont sensiblement la même taille (ou du moins le même ordre de grandeur). Il n'y a donc plus la nécessité d'un envoi de plusieurs gigas de données comme cela était le cas pour les versions du chiffrement homomorphe basées sur A-GCD. Du fait de son caractère plus pratique, cette branche de l'homomorphe a suscité beaucoup d'effervescence, amenant une succession d'optimisations aux schémas. Ceci a également eu comme impact l'existence de nombreux schémas [BV11][BGV12][Bra12][FV12][GSW13][KGV15] (également appelé SHIELD)[BV14]. En pratique, on utilise de préférence la version polynomiale (R-LWE), pour ses paramètres plus favorables asymptotiquement que la version initiale. En fonction du niveau de sécurité souhaité et de l'application cible, la clé publique ainsi qu'un chiffré peuvent aller de quelques centaines de milliers de bits, à plusieurs dizaines de millions de bits. Malgré de nombreux schémas existants, la majorité sont des améliorations des schémas précédents. [FV12] (nommé FV dans la suite du document) est une des versions les plus abouties. Les schémas de chiffrement qui ont suivi FV [GSW13][KGV15][BV14] (schémas dits de 3$^{\text{ème}}$ génération), ont eu comme optimisation de supprimer une étape lors de la multiplication homomorphe de FV, à savoir la relinéarisation. En quelques mots, après la multiplication homomorphe, le chiffré change de forme, impliquant une modification de la méthode de déchiffrement. L'opération de relinéarisation a pour but de redonner au chiffré sa forme initiale. Les schémas de 3$^{\text{ème}}$ génération n'ont pas besoin de relinéarisation car celle-ci est implicitement réalisée durant la multiplication homomorphe. De plus, ils ont un comportement asymptotique vis-à-vis du bruit plus favorable que FV. En contrepartie, ils demandent de manipuler des chiffrés de grande taille, même pour des algorithmes avec une profondeur multiplicative faible (avec des chiffrés de l'ordre de quelques millions de bits).

Il existe enfin une dernière famille de chiffrements homomorphes basés sur NTRU [LATV12] [BLLN13] [DS16]. Entre 2012 et 2016, YASHE' a notamment été un solide concurrent à FV, car pour des niveaux de sécurités semblables, YASHE' avait des chiffrés deux fois plus petits que FV. Cependant, en 2016 avec l'attaque proposée

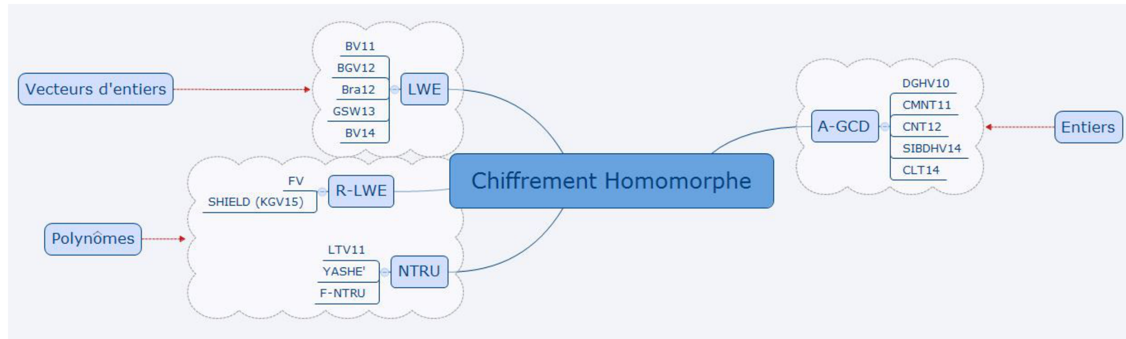


FIGURE 1.3 – Carte mentale représentant la diversité des chiffrements homomorphes.

dans [ABD16] (dit *sub-field attack*, YASHE' a été fortement impacté et n'est plus considéré suffisamment sécurisé depuis).

La figure 1.3 résume la diversité des chiffrements homomorphes à l'aide d'une représentation de type carte mentale.

En conclusion, selon l'état actuel du chiffrement homomorphe, FV semble le candidat le plus intéressant pour des petites applications. Pour des applications de plus grande envergure (grande profondeur multiplicative), des schémas de 3$^{\text{ème}}$ génération de type SHIELD semblent être des meilleurs candidats. Par la suite, nous nous intéresserons de plus près à FV, car il permet d'avoir des chiffrés de petite taille comparé aux schémas concurrents.

1.2.3 Utilisation du chiffrement homomorphe en pratique

Les schémas de chiffrements homomorphes permettent des opérations d'addition, de soustraction ou de multiplication sur des entiers modulo l'entier t. À l'heure actuelle, il n'est pas possible de pouvoir traiter séparément les différents bits de ces entiers. Ceci est particulièrement problématique lorsque l'on souhaite implémenter des opérateurs conditionnels qui demandent donc de manipuler les entiers au niveau bit. Pour la comparaison $a > b$ par exemple, l'algorithme standard est de calculer $a-b$ et de regarder le bit de signe du résultat. La solution proposée actuellement est d'utiliser le fait que les opérations sur les entiers se font modulo t. Pour $t = 2$, l'addition revient au calcul d'un OU exclusif binaire (porte XOR) et la multiplication d'un ET binaire (porte AND). A partir de ces deux opérateurs élémentaires, il est possible de construire toutes sortes d'opérateurs de plus haut-niveau. En contrepartie, si l'algorithme demande une opération d'addition par exemple, il faudra alors décrire cette opération en portes XOR et AND, augmentant significativement le nombre d'opérations à effectuer.

Pour illustrer notre propos, continuons sur l'exemple de la comparaison et analysons son implémentation dans le cas du chiffrement homomorphe. La représentation usuelle des nombres étant le complément à deux, la soustraction de deux nombres

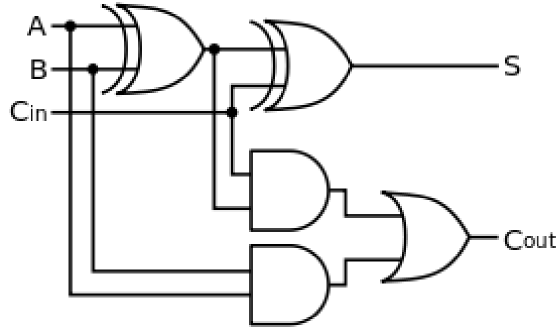


Figure 1.4 – Additionneur complet 1 bit.
Source : `https://fr.wikipedia.org/wiki/additionneur`

A et B se calcule :

$$A - B = A + \bar{B} + 1 \tag{1.1}$$

Avec $\bar{B}$ l'inverse binaire de B (appelé NOT par la suite). Il faut donc implémenter deux additions binaires, et regarder le bit de signe du résultat. La figure 1.4 décrit l'architecture d'un additionneur 1 bit, et la figure 1.5 décrit la mise en cascade d'additionneurs 1 bit pour créer un additionneur 4 bits. Sur la figure 1.4, A et B représentent deux nombres de 1 bit, C_{in} la retenue de l'addition précédente, S le résultat de l'addition et C_{out} la retenue de sortie. Pour calculer une addition sur plusieurs bits, il faut propager la retenue entre les bits, d'où la structure en cascade figure 1.5. Dans notre opérateur de comparaison, plusieurs adaptations sont à faire :

1) Il faut appliquer une inversion binaire sur chaque bit de B. Cette opération est obtenue en additionnant chaque bit par 1 ;
2) Pour ajouter le $+1$ de l'équation 1.1, il suffit juste de considérer que C_{in} du premier additionneur est à 1 ;
3) Il suffit de prendre la valeur du dernier C_{out}, donnant le signe de la soustraction.

Pour mener à bien une comparaison sur 8 bits, il faut donc calculer deux OU exclusif, deux ET et 2 inverses par étage d'additionneur, donnant donc size XOR, size AND et size NOT au total. De plus, on peut remarquer que les portes AND sont en cascade dans le circuit. Les portes AND étant des multiplications homomorphes, cette comparaison va demander une profondeur multiplicative de huit (ayant huit portes AND en cascade).

1.3 Cas d'étude : le chiffrement FV

Le schéma de chiffrement homomorphe FV fait partie des schémas les plus populaires à l'heure actuelle. Il est basé sur une application directe du schéma [Bra12] au problème R-LWE. Ses principaux atouts sont :

1) Le schéma est simple et facile à manipuler. En particulier, un chiffré est

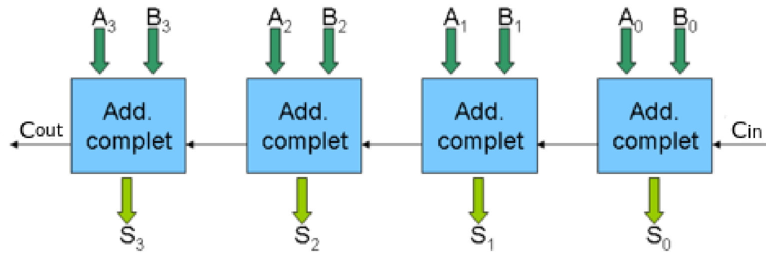


FIGURE 1.5 – Additionneur 4 bits.
Source : `https://fr.wikipedia.org/wiki/additionneur`

composé de seulement deux polynômes, l'addition homomorphe étant une simple addition terme-à-terme des polynômes des chiffrés, et la multiplication homomorphe (hors relinéarisation) est proche d'une multiplication naturelle des chiffrés.

2) La génération des clés ne demande pas de polynômes avec des propriétés particulières. Cela évite tout problème de rejet de polynômes, coûteux en temps de calcul, comme on pouvait le trouver sur les schémas basés sur NTRU comme YASHE'.
3) FV est capable d'adapter ses paramètres (degré et taille de ses coefficients) à toute profondeur multiplicative, permettant ainsi de minimiser sa complexité pour une application donnée.

1.3.1 R-LWE

Le problème LWE, proposé dans [Reg05] puis revisité en 2009 [Reg09], est une généralisation du problème « parity learning » [BKW03]. Le problème est simple : connaissant une séquence d'équations linéaires d'inconnue S avec l'ajout d'un bruit sur chaque équation, est-il possible de récupérer l'inconnue S ? Dans [Reg09], une réduction du problème LWE à certains problèmes sur les réseaux euclidiens considérés au pire cas est proposée, notamment les problèmes GapSVP et SIVP [LPR81]. Voici une définition plus formelle du problème R-LWE, variante de LWE appliquée au cas des polynômes (cas que nous étudierons en particulier par la suite) :

Definition 1. *(Distribution $L_{S,\chi}$ d'un problème R-LWE)*
Soit R un anneau de degré n sur $\mathbb{Z}$ (classiquement $R = \mathbb{Z}[x]/f(x)$, $f(x)$ un polynôme cyclotomique). Soit q un nombre entier positif, on note $R_q = R/qR$. Soient χ_{s,R_q}, χ_{r,R_q} deux lois de probabilité sur R_q. On définit la distribution $L_{S,\chi} = \{(A, [A \cdot S + E]_q)/A \in U_{R_q}, S \in \chi_{s,R_q}, E \in \chi_{e,R_q}\}$.

On remarque que la définition proposée contient bien une séquence d'équations linéaires bruitées par le facteur E, S étant l'inconnue et A étant connu. Pour résoudre le problème R-LWE, il faudrait en théorie récupérer S. Actuellement, et comme proposé dans [LPR10], seule la distinction du couple $(A, [A \cdot S + E]_q)$ d'une distribution uniforme est suffisante. Ce problème s'appelle Décision-R-LWE et s'exprime ainsi :

TABLE 1.1 – Valeur maximale pour $\log_2 q$ pour une dimension donnée n d'une instance R-LWE, avec λ le niveau de sécurité et $\sigma_{err} = 2\sqrt{n}$.

n	1024	2048	4096	8192
$\lambda = 80$ bits	48 bits	91 bits	177 bits	350 bits
$\lambda = 128$ bits	32 bits	60 bits	115 bits	227 bits

Definition 1. *(Décision Ring-LWE)*
Soit (A, C) un échantillon d'une distribution $L_{S,\chi}$. Distinguer si le couple (A, C) est un élément de $L_{S,\chi}$ ou un élément d'une distribution uniforme sur R_q est un problème difficile à résoudre.

Afin de paramétrer une instance R-LWE et assurer un certain niveau de sécurité, il est important de choisir avec précaution les lois de probabilités χ_{s,R_q} et χ_{r,R_q}, le degré du polynôme $f(x)$ ainsi que la taille du module q. Un sondage des différentes attaques sur R-LWE a été réalisé par Martin Albrecht dans [APS15]. Un outil en ligne pour la détermination de paramètres de sécurité est également joint à l'étude et disponible en ligne [Alb17]. Cette étude donne quelques recommandations pour le choix des paramètres de sécurité. Pour $R_q = \mathbb{Z}_q[x]/f(x)$ ($\deg(f) = n$), les auteurs de [APS15] proposent de choisir $\chi_{s,R_q} = B_R$ et $\chi_{r,R_q} = D_{R_q,\sigma}$ avec $\sigma = 2\sqrt{n}$. Ainsi, la clé est binaire au lieu de gaussienne, et la largeur de la gaussienne utilisée pour le bruit évolue en racine de n.

La table 1.1 liste les paramètres de sécurité d'une instance R-LWE avec les recommandations proposées dans [APS15] pour 80 bits et 128 bits de sécurité. Nous avons utilisé l'estimateur de Martin Albrecht, commit f59326c (13 juillet 2017). Ces paramètres seront repris afin de déterminer les paramètres de sécurité de FV.

1.3.2 FV

Intéressons-nous maintenant aux primitives du schéma FV. La figure 1.6 propose un organigramme du fonctionnement de FV avec le nom des fonctions associées que nous détaillerons par la suite.

Afin de paramétrer le schéma de chiffrement FV, il est nécessaire de définir trois constantes, t, ω et Δ :

1) La constante t, prise telle que $1 < t < q$, permet de définir l'arithmétique sur les messages. Ainsi, chaque opération homomorphe (addition ou multiplication) se traduira par une opération modulo t sur les messages.
2) ω est un paramètre qui modifie l'opération dite de relinéarisation. Cette opération, bien que facultative, est quasiment systématiquement effectuée après la multiplication homomorphe. En effet, après l'opération de multiplication, l'ordre du chiffré augmente, augmentant de ce fait la taille du chiffré et la complexité des futures opérations. La relinéarisation a donc pour seul but de redonner au chiffré sa forme initiale.

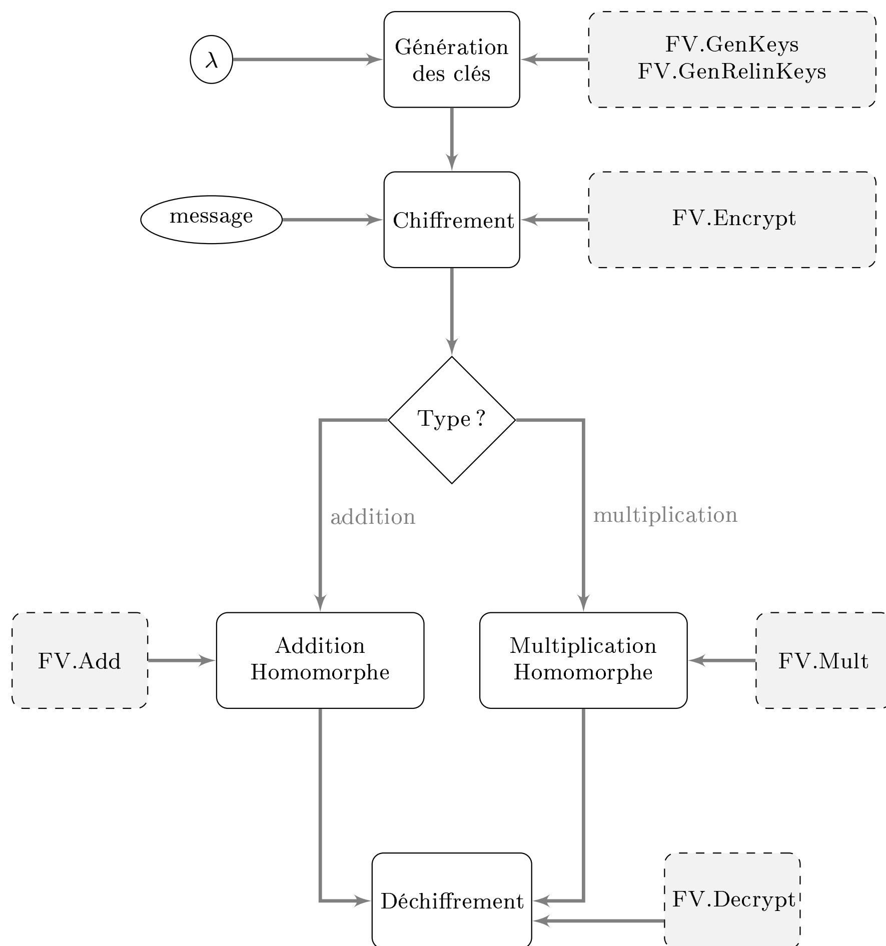


FIGURE 1.6 – Organigramme de l'utilisation du chiffrement FV avec les noms des fonctions associées.

3) $\Delta = \lfloor q/t \rfloor$, sera utilisé pour séparer le message du bruit interne du chiffré.

Avant de rentrer dans le détail de chaque opération, il est nécessaire d'introduire deux fonctions nommées FV.PowersOf$_{\omega,q}$ et FV.WordDecomp$_{\omega,q}$. Ces fonctions, qui dépendent du paramètre ω, seront utilisées notamment pour la relinéarisation. Elles font apparaître une constante $l_{\omega,q} = \lceil \log_2 q / \log_2 \omega \rceil$, et sont définies comme suit :

— FV.PowersOf$_{\omega,q}(A)$:
$\mathbf{A} \in R_q^{l_{\omega,q}}$
for $i = 0$ to $l_{\omega,q} - 1$
    $\mathbf{A}[i] = [A \cdot \omega^i]_q$
end for
return $\mathbf{A}$

— FV.WordDecomp$_{\omega,q}(A)$:
$\mathbf{A} \in R_q^{l_{\omega,q}}$
for $i = 0$ to $l_{\omega,q} - 1$
    $l_0 = i \cdot \log_2 \omega$
    $l_1 = (i+1) \cdot \log_2 \omega - 1$
    $\mathbf{A}[i] = A_{(l_0 \rightarrow l_1)}$
end for
return $\mathbf{A}$

FV.PowersOf$_{\omega,q}$ est donc une fonction qui, à un polynôme donné A, renvoie un

vecteur contenant A multiplié par des puissances de ω.

FV.WordDecomp$_{\omega,q}$ quant-à-elle, pour un polynôme donné A, renvoie un vecteur de polynômes dont chaque membre est une version segmentée (avec des segments de $\log_2 \omega$ bits) du polynôme A. Ces deux fonctions ont la propriété suivante qui sera utilisée pour mener à bien l'étape de relinéarisation :

$$\left\langle \text{FV.PowersOf}_{\omega,q}(A), \text{FV.WordDecomp}_{\omega,q}(B) \right\rangle \equiv A \cdot B \mod q$$

Nous pouvons maintenant présenter les primitives de FV :

$$\begin{aligned}
&\text{FV.GenKeys}(\lambda) : \\
&S \leftarrow D_{R_q,\sigma_{key}}, \ A \leftarrow U_{R_q}, \ E \leftarrow D_{R_q,\sigma_{err}} \\
&\mathbf{P_{key}} = ([-A \cdot S + E]_q, A) \\
&S_{key} = S \\
&\text{return } (\mathbf{P_{key}}, S_{key})
\end{aligned} \tag{1.2}$$

Pour la fonction de génération des clés de chiffrement FV.GenKeys, on remarque qu'il s'agit d'une utilisation directe du problème R-LWE (au signe près). Nous verrons que ce signe a de l'importance lors du déchiffrement.

$$\begin{aligned}
&\text{FV.GenRelinKeys}(\mathbf{P_{key}}, S_{key}) : \\
&\mathbf{A} \leftarrow U_{R_q}^{l_{\omega,q}}, \mathbf{E} \leftarrow D_{R_q,\sigma_{err}}^{l_{\omega,q}} \\
&\mathbf{\Gamma} = \left(\left[\text{FV.PowersOf}_{\omega,q}(S_{key}^2) - \left(\mathbf{A} \cdot S_{key} + \mathbf{E}\right) \right]_q, \mathbf{A} \right) \\
&\text{return } \mathbf{\Gamma}
\end{aligned} \tag{1.3}$$

La fonction FV.GenRelinKeys permet de générer les clés dites de relinéarisation qui seront exclusivement utilisées lors de la relinéarisation. L'idée est de cacher dans un problème R-LWE le carré de la clé secrète. Nous verrons par la suite son utilité.

$$\begin{aligned}
&\text{FV.Encrypt}(m, \mathbf{P_{key}}) : \\
&U \leftarrow D_{R_q,\sigma_{key}}, \ (E_1, E_2) \leftarrow D_{R_q,\sigma_{err}}^2 \\
&\mathbf{C} = \left(\left[\Delta m + \mathbf{P_{key}}[0] \cdot U + E_1 \right]_q, \left[\mathbf{P_{key}}[1] \cdot U + E_2 \right]_q \right) \\
&\text{return } \mathbf{C}
\end{aligned} \tag{1.4}$$

Pour la fonction de chiffrement FV.Encrypt, il est important de remarquer quelques points :

1) Nous pouvons observer ici la notion de bruit dans le chiffré. Une première partie du bruit est produite par la clé publique qui contient un élément bruité E, puis un second bruit est ajouté par les éléments E_1 et E_2. Ce bruit est choisi de telle sorte qu'il soit relativement faible par rapport au module q. Le message m, quant-à-lui, est multiplié par le facteur Δ, ce qui permet de positionner le message dans les poids forts des coefficients des polynômes du chiffré.
2) Comme $\mathbf{P_{key}}$ est supposée non distinguable d'une distribution uniforme selon la preuve de sécurité sur le problème R-LWE, alors un chiffré est construit comme un nouveau problème R-LWE, où U serait considéré comme le facteur secret.

$$
\begin{aligned}
&\text{FV.Decrypt}(\mathbf{C}, S_{key}) : \\
&\widetilde{M} = \Big[\mathbf{C}[0] + \mathbf{C}[1] \cdot S_{key}\Big]_q \\
&m = \left\lfloor \frac{t}{q} \widetilde{M}[0] \right\rceil \\
&\text{return } m
\end{aligned}
\tag{1.5}
$$

La fonction de déchiffrement FV.Decrypt permet de déchiffrer le message. Si on développe le chiffré d'après la définition de $\mathbf{P_{key}}$ fournie par l'équation 1.2, on obtient :

$$\Big(\Delta m - A \cdot S_{key} \cdot U + E \cdot U + E_1,\ A \cdot U + E_2\Big)$$

En multipliant par S_{key} le second membre et en additionnant les deux membres entre eux (comme effectué par la fonction FV.Decrypt), on obtient :

$$\Delta m + U \cdot E + E_1 + S_{key} \cdot E_2$$

Ainsi, tant que $||U \cdot E + E_1 + S_{key} \cdot E_2||_\infty < \frac{1}{2} \cdot \Delta m$, l'opération de déchiffrement est cohérente. Le facteur $\frac{1}{2}$ devant Δm est important car comme nous manipulons des bruits gaussiens centrés en zéro, il est donc tout à fait possible de trouver des résultats négatifs. Le facteur $||U \cdot E + E_1 + S_{key} \cdot E_2||_\infty$ est nommé *bruit initial*.

$$
\begin{aligned}
&\text{FV.Add}(\mathbf{C_A}, \mathbf{C_B}) : \\
&\mathbf{C}_+ = \Big(\Big[\mathbf{C_A}[0] + \mathbf{C_B}[0]\Big]_q, \Big[\mathbf{C_A}[1] + \mathbf{C_B}[1]\Big]_q\Big) \\
&\text{return } C_+
\end{aligned}
\tag{1.6}
$$

La fonction FV.Add représente la fonction d'addition homomorphe. Il est aisé de remarquer que cette opération donne bien un chiffré dont les messages sont ad-

ditionnés.

$$
\begin{aligned}
&\text{FV.Mult}(\mathbf{C_A}, \mathbf{C_B}, \mathbf{\Gamma}) : \\
&\widetilde{C}_0 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[0] \cdot \mathbf{C_B}[0] \right\rceil\right]_q \\
&\widetilde{C}_1 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[0] \cdot \mathbf{C_B}[1] + \frac{t}{q}\mathbf{C_A}[1] \cdot \mathbf{C_B}[0] \right\rceil\right]_q \\
&\widetilde{C}_2 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[1] \cdot \mathbf{C_B}[1] \right\rceil\right]_q \\
&\mathbf{C}_\times = \text{FV.Relin}(\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2, \mathbf{\Gamma}) \\
&\text{return } \mathbf{C}_\times
\end{aligned}
\tag{1.7}
$$

$$
\begin{aligned}
&\text{FV.Mult}(\mathbf{C_A}, \mathbf{C_B}) : \\
&\widetilde{C}_0 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[0] \cdot \mathbf{C_B}[0] \right\rceil\right]_q \\
&\widetilde{C}_1 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[0] \cdot \mathbf{C_B}[1] + \frac{t}{q}\mathbf{C_A}[1] \cdot \mathbf{C_B}[0] \right\rceil\right]_q \\
&\widetilde{C}_2 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[1] \cdot \mathbf{C_B}[1] \right\rceil\right]_q \\
&\text{return } (\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2)
\end{aligned}
\tag{1.8}
$$

La fonction FV.Mult définit l'opération de multiplication homomorphe. Il est important de formuler deux remarques à propos de cette opération : Premièrement, l'opération de multiplication homomorphe revient à multiplier entre eux les polynômes de chaque chiffré. Après la multiplication polynomiale, le message a la forme suivante :

$$\Delta^2 m_1 m_2$$

Il est donc nécessaire de diviser le résultat de cette opération par Δ pour avoir un chiffré avec la bonne forme. Ceci a un impact non négligeable sur les performances. En effet, il est nécessaire de réaliser une opération de multiplication sans réduction modulaire par q sinon nous perdons le message résultat. De plus, q pouvant avoir une forme quelconque, l'opération de division n'est pas forcément aisée, surtout en considérant une implémentation matérielle.

Deuxièmement, et comme nous l'avions précisé précédemment, après la multiplication homomorphe, l'ordre du chiffré augmente de 1, si bien que nous passons de deux polynômes à trois polynômes. Avec trois membres, il est tout à fait possible de déchiffrer, en appliquant la relation suivante :

$$\widetilde{C}_0 + \widetilde{C}_1 \cdot S_{key} + \widetilde{C}_2 \cdot S_{key}^2$$

Cependant, l'augmentation de la taille du chiffré amenant à plus de multiplications polynomiales, le but de l'opération de relinéarisation est de faire retrouver au chiffré une forme telle que l'opération de déchiffrement puisse s'exécuter comme dans FV.Decrypt, c'est-à-dire sans la nécessité de faire intervenir S_{key}. Il est donc intuitivement nécessaire de communiquer au serveur l'information de S_{key}^2 pour l'opération de relinéarisation. C'est à ce stade qu'interviennent les clés de relinéarisation.

$$\begin{aligned}
&\text{FV.Relin}(\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2, \mathbf{\Gamma}) : \\
&C_{R,0} = \left[\widetilde{C}_0 + \left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[0] \right\rangle\right]_q \\
&C_{R,1} = \left[\widetilde{C}_1 + \left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[1] \right\rangle\right]_q \\
&\mathbf{C_R} = (C_{R,0}, C_{R,1}) \\
&\text{return } \mathbf{C_R}
\end{aligned} \tag{1.9}$$

L'opération fondamentale de la relinéarisation FV.Relin est l'opération :

$$\left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[i] \right\rangle,\ i \in \{0,1\}$$

Pour alléger la notation, on pose $\text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2) = \mathbf{WD}$. Si on développe le calcul, on obtient :

$$\begin{aligned}
\left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[0] \right\rangle &= \left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \text{FV.PowersOf}_{\omega,q}(S_{key}^2) \right\rangle \\
&\quad - \left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{A} \cdot S_{key} + \mathbf{E} \right\rangle \\
&= \widetilde{C}_2 \cdot S_{key}^2 - \sum_{i=1}^{l_{\omega,q}} \mathbf{WD}[i] \cdot \big(\mathbf{A}[i] \cdot S_{key} + \mathbf{E}[i]\big)
\end{aligned}$$

Ainsi, après relinéarisation, le chiffré a la forme suivante :

$$\widetilde{C}_0 + \widetilde{C}_2 \cdot S_{key}^2 - \sum_{i=1}^{l_{\omega,q}} \mathbf{WD}[i] \cdot \big(\mathbf{A}[i] \cdot S_{key} + \mathbf{E}[i]\big),\ \widetilde{C}_1 + \sum_{i=1}^{l_{\omega,q}} \mathbf{WD}[i] \cdot \mathbf{A}[i]$$

En appliquant la formule de déchiffrement, on obtient :

$$\begin{aligned}
&\widetilde{C}_0 + \widetilde{C}_2 \cdot S_{key}^2 - \sum_{i=1}^{l_{\omega,q}} \big(\mathbf{WD}[i] \cdot \mathbf{A}[i] \cdot S_{key} + \mathbf{E}[i]\big) + \big(\widetilde{C}_1 + \sum_{i=1}^{l_{\omega,q}} \mathbf{WD}[i] \cdot \mathbf{A}[i]\big) \cdot S_{key} \\
&= \widetilde{C}_0 + \widetilde{C}_1 \cdot S_{key} + \widetilde{C}_2 \cdot S_{key}^2 - \sum_{i=1}^{l_{\omega,q}} \mathbf{WD}[i] \cdot \mathbf{E}[i]
\end{aligned}$$

On obtient alors un facteur $\widetilde{C}_0 + \widetilde{C}_1 \cdot S_{key} + \widetilde{C}_2 \cdot S_{key}^2$ qui correspond à l'opération de déchiffrement appliquée au chiffré avant relinéarisation, plus un facteur

$\sum_{i=1}^{l_{\omega,q}} \mathbf{WD}[i] \cdot \mathbf{E}[i]$ qui correspond à un bruit de relinéarisation. C'est ici qu'intervient le compromis bruit/performance selon le choix de ω. Par définition de l'opération FV.WordDecomp$_{\omega,q}$, on a $||\mathbf{WD}[i]||_\infty \leq \omega$. Ainsi, plus ω est petit, moins le bruit de relinéarisation est grand. En revanche, un ω faible implique un grand $l_{\omega,q} = \lceil \log_2 q / \log_2 \omega \rceil$, donc plus d'éléments dans le vecteur **WD** et plus de multiplications polynomiales lors de la relinéarisation. Il est également important de noter que plus l'évolution du bruit est faible, plus nous pouvons réduire la taille du module q et donc plus nous pouvons réduire la dimension n.

1.3.3 Choix des paramètres

Afin de déterminer les paramètres de sécurité pour FV, il est nécessaire de trouver un compromis entre le niveau de sécurité (qui tend à faire baisser la valeur de q pour un n fixé) et la profondeur multiplicative (qui à l'inverse tente de faire augmenter q pour un n donné). Il est important de constater que le paramètre ω aura un impact fort sur n et q, car la relinéarisation apporte un bruit proportionnel à ω. Ainsi, comme ω est généralement choisi entre 2^{32} et 2^{64}, la relinéarisation va apporter de 32 à 64 bits de bruit environ. Dans la pratique, on distingue deux types de bruits :

1) Le bruit initial : bruit lié à l'opération de chiffrement ;
2) Le bruit multiplicatif : bruit obtenu après multiplication homomorphe.

Avant de détailler les formules permettant d'obtenir ces différents bruits, il est important de pouvoir majorer le produit de deux polynômes. Dans la littérature, ce paramètre est nommé δ et défini par :

$$\delta = \left\{ \sup \frac{||A \cdot B||_\infty}{||A||_\infty \cdot ||B||_\infty}; (A, B) \in R_q \right\} = n \tag{1.10}$$

Comme A et B parcourent tout R_q, δ donne la valeur au pire cas de l'évolution de la norme d'un produit de polynômes. Par la suite, et pour deux polynômes A, B on utilisera le corollaire suivant :

$$\begin{aligned} ||A \cdot B||_\infty &\leq \delta \cdot ||A||_\infty \cdot ||B||_\infty \\ &= n \cdot ||A||_\infty \cdot ||B||_\infty \end{aligned} \tag{1.11}$$

1.3.3.1 Bruit initial

En reprenant la définition du bruit initial défini section 1.3.2 :

$$\begin{aligned} B_0 = ||U \cdot E + E_1 + S_{key} \cdot E_2||_\infty &\leq ||U \cdot E||_\infty + ||E_1||_\infty + ||S_{key} \cdot E_2||_\infty \\ &\leq B_{err} + 2n \cdot B_{err} \cdot B_{key} \\ &= B_{err} \cdot (1 + 2n \cdot B_{key}) \end{aligned} \tag{1.12}$$

Avec :

- $||E||_\infty = ||E_1||_\infty = ||E_2||_\infty = B_{err}$.

Table 1.2 – Valeur minimale de $\log_2 q$ après L multiplications homomorphes permettant une opération de déchiffrement cohérente dans le schéma de chiffrement FV, pour les paramètres suivants : $n = 4096$, $\omega = 2^{32}$, $t = 2$.

L	1	2	3	4	5	6
$\log_2 q$	57	83	109	135	161	186

- $||S_{key}||_\infty = ||U||_\infty = B_{key}$.
- $B_{key} = 1$, car U et S_{key} sont des polynômes binaires.
- $B_{err} = \beta_\epsilon \cdot \sigma_{err}$, avec $\beta_\epsilon = 9.2$ et $\sigma_{err} = 2 \cdot \sqrt{n}$ d'après [LPR10].

1.3.3.2 Bruit multiplicatif

L'extraction de l'évolution du bruit après multiplication n'est pas triviale car il faut appliquer l'opération de déchiffrement après multiplication. Nous avons suivi la dérivation proposée dans [LN14], qui permet de déduire le bruit après L multiplications :

$$\begin{aligned} B_L &= C_1^L \cdot B_0 + L \cdot C_1^{L-1} \cdot C_2 + B_{relin} < \frac{\Delta(1-t)}{2} - \frac{q}{2} \\ C_1 &= nt(4 + nB_{key}) \\ C_2 &= n^2 B_{key}(B_{key} + t^2) \\ B_{relin} &= n\omega l_{\omega,q} B_{err} \end{aligned} \tag{1.13}$$

1.3.3.3 Méthode de détermination des paramètres de sécurité

Montrons dans un cas simple comment sont déterminés les paramètres de sécurité de FV. Tout d'abord, il faut se fixer une valeur pour le degré du polynôme cyclotomique n. Prenons l'exemple de 4096. Pour cette valeur de n, la table 1.1 donne un module de taille maximale de 174 bits pour 80 bits de sécurité, ou 112 bits pour 128 bits de sécurité. Maintenant que le couple $(n, \log_2 q)$ est connu, il faut se fixer ω pour pouvoir déterminer le bruit de relinéarisation. Prenons par exemple $\omega = 2^{32}$. Maintenant, à partir de l'équation 1.13, il est possible de déterminer le nombre maximum de multiplications homomorphes successives L que l'on peut appliquer avant de rendre le déchiffrement incohérent. La table 1.2 donne la valeur minimale de $\log_2 q$ en fonction de L. Pour 80 bits de sécurité, comme $\log_2 q$ ne doit pas dépasser 174 bits pour des raisons de sécurité, la profondeur multiplicative est limitée à cinq. Pour 128 bits de sécurité, le $\log_2 q$ maximal étant de 112 bits, la profondeur multiplicative passe alors à trois.

Table 1.3 – Paramètres de sécurité pour le schéma FV.

	$\lambda \rightarrow$	80 bits			128 bits		
	$\omega \rightarrow$	27 bits	32 bits	64 bits	27 bits	32 bits	64 bits
Profondeur multiplicative	1	(1040, 49)	(1161, 54)	(1947, 87)	(1660, 50)	(1940, 55)	(3081, 88)
	2	(1639, 74)	(1758, 79)	(2553,112)	(2632, 76)	(2804, 81)	(4090,115)
	3	(2262,100)	(2374,105)	(3194,139)	(3648,103)	(3848,109)	(5096,143)
	4	(2885,127)	(3016,132)	(3834,166)	(4712,132)	(4912,137)	(6188,171)
	5	(3538,154)	(3664,159)	(4472,193)	(5754,160)	(5991,166)	(7239,200)
	6	(4179,182)	(4334,187)	(5120,221)	(6848,190)	(7044,195)	(8263,229)
	7	(4894,210)	(5002,215)	(5782,250)	(7976,220)	(8109,225)	(9419,259)
	8	(5579,239)	(5677,244)	(6491,279)	(9071,250)	(9247,255)	(10569,289)
	9	(6250,268)	(6376,273)	(7183,308)	(10213,280)	(10450,286)	(11661,320)
	10	(6957,298)	(7083,303)	(7850,337)	(11309,311)	(11675,317)	(12844,351)
	15	(10520,449)	(10643,454)	(11437,488)	(17259,470)	(17452,475)	(18722,509)
	20	(14156,606)	(14301,611)	(15127,645)	(23189,633)	(23417,639)	(24709,673)

1.3.3.4 Exemples de paramètres de sécurité

La table 1.3 propose des paramètres pour FV pour une gamme de profondeurs multiplicatives allant jusqu'à 20. La détermination est similaire à celle présentée dans la section 1.3.3.3, nous avons simplement raffiné le résultat en faisant varier n afin de minimiser le couple $(n, \log_2 q)$.

Nous pouvons faire plusieurs remarques intéressantes sur ces résultats :

1) L'impact de ω est non négligeable, et d'autant plus important que la profondeur multiplicative est faible. Comme le bruit de relinéarisation est additif, ce dernier va contribuer fortement au bruit pour la première multiplication homomorphe. Pour les suivantes, comme le bruit sera déjà bien supérieur au bruit de relinéarisation, son impact sera plus faible.
2) Le passage de 80 bits à 128 bits de sécurité a un lourd impact sur les paramètres de sécurité, car pour atteindre un tel niveau de sécurité, le module doit être bien inférieur au niveau de sécurité de 80 bits. Cela demande en pratique d'augmenter d'environ 50% la taille de n.
3) La profondeur multiplicative fait varier de manière très prononcée les paramètres de sécurité. Ainsi, il est nécessaire de bien dimensionner les paramètres de sécurité à notre application, sans quoi, le surcoût arithmétique peut devenir prohibitif.

1.3.4 La technique du *batching*

Le *batching* est une technique introduite dans [CCK+13], qui permet de « regrouper » plusieurs messages dans un seul chiffré. Il repose sur l'application du théorème chinois des restes aux polynômes.

Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe Vincent Migliore 2017

Théorème 1. *(Théorème chinois des restes appliqué aux polynômes)*
Soit $k \in \mathbb{Z}$ et $\mathbf{\Phi} \in \mathbb{Z}[x]^k$ un vecteur de k polynômes premiers entre eux.
On note $\Phi = \prod_{i=1}^{k} \mathbf{\Phi}[i]$.
Alors pour tout $\mathbf{A} \in \mathbb{Z}[x]^k$, il existe un unique polynôme P vérifiant $\deg(P) < \deg(\Phi)$ *tel que : $\forall i \in \{1, \cdots, k\}$, $P \equiv \mathbf{A}[i] \mod \mathbf{\Phi}[i]$*

On peut remarquer aisément que pour deux polynômes $P_a \equiv \mathbf{A}[i] \mod \mathbf{\Phi}[i]$ et $P_b \equiv \mathbf{B}[i] \mod \mathbf{\Phi}[i]$, nous avons les relations suivantes :

$$(P_a + P_b) \equiv \mathbf{A}[i] + \mathbf{B}[i] \mod \mathbf{\Phi}[i]$$

$$(P_a \cdot P_b) \equiv \mathbf{A}[i] \cdot \mathbf{B}[i] \mod \mathbf{\Phi}[i]$$

Ainsi (avec les notations du théorème), en prenant Φ comme polynôme cyclotomique de notre anneau R_q, chaque élément du vecteur $\mathbf{A}$ comme étant un message, il est possible de réaliser des opérations homomorphes sur chaque message en parallèle en manipulant uniquement le polynôme P. Il est cependant nécessaire de prendre quelques précautions. Notamment, il faut vérifier que notre polynôme cyclotomique, irréductible dans $\mathbb{Z}_q[x]$, est bien réductible dans l'espace des messages, c'est-à-dire réductible sur $\mathbb{Z}_t[x] = \mathbb{Z}_2[x]$.

Dans les implémentations actuelles du chiffrement homomorphe, le polynôme cyclotomique $X^n + 1$ (n étant une puissance de deux) est usuellement choisi car s'adapte il bien avec l'algorithme de multiplication NTT (algorithme présenté section 1.4.4). En pratique, le choix d'un tel polynôme n'est pas forcément judicieux car $X^n + 1$ n'est pas réductible dans $\mathbb{Z}_2[x]$. En somme, il n'est pas possible d'utiliser ce polynôme pour du *batching* sur des messages binaires. Le *batching* est cependant une technique très prometteuse car elle permet à la fois de réduire l'expansion du chiffré, c'est-à-dire le rapport entre la taille du chiffré et la taille du message en clair, et le nombre d'opérations homomorphes, car plusieurs opérations homomorphes sont faites en parallèle.

Nous avons donc choisi d'explorer dans notre étude le caractère pratique de l'implémentation du *batching*. D'après notre étude exploratoire sur l'existence et la complexité des polynômes cyclotomiques réductibles sur $\mathbb{Z}_2[x]$ présentée section 4.1, nous avons remarqué qu'il était aisé de trouver de tels polynômes, et ce pour tout degré et avec des polynômes possédant un nombre très faible de coefficients non nuls (dans la majorité des cas, inférieur à cinquante pour des polynômes permettant un niveau de *batching* inférieur ou égal à huit).

1.4 Multiplication de grands opérandes : état de l'art

La multiplication polynomiale est l'opération élémentaire particulièrement complexe dans le schéma de chiffrement homomorphe FV (et plus généralement, dans

les schémas basés sur R-LWE). Que ce soit pour la multiplication d'entiers ou de polynômes, il existe quatre principaux algorithmes de multiplication :

1) l'algorithme classique de complexité en $\mathcal{O}(n^2)$;
2) l'algorithme de Karatsuba-Ofman [KO62] de complexité en $\mathcal{O}(n^{1.58})$;
3) l'algorithme de Toom-Cook [Too63] de complexité en $\mathcal{O}(n^{1.465})$;
4) l'algorithme de la NTT [Pol71] de complexité en $\mathcal{O}(n \log n)$.

Ces complexités sont bien entendu asymptotiques, et la complexité réelle va dépendre de l'implémentation de l'algorithme lui-même. Ainsi, l'algorithme de la NTT, bien qu'ayant une complexité asymptotique bien plus favorable que les autres approches, pose quelques difficultés d'implémentation si bien que d'autres algorithmes peuvent être plus efficacement implémentés. Ces limitations font l'objet d'une explication plus détaillée section 1.5.2.

Pour la présentation de chaque algorithme, nous présenterons l'algorithme pour la multiplication entière et polynomiale. Pour la version entière, nous supposerons que nos entiers sont découpés par morceaux de β bits. Pour une multiplication $c = a \cdot b$, on définit donc :

$$\begin{aligned} a &= \sum_{i=0}^{l-1} a_i \cdot 2^{\beta i} \\ b &= \sum_{i=0}^{l-1} b_i \cdot 2^{\beta i} \\ l &= \lceil \log_2 a/\beta \rceil \end{aligned} \tag{1.14}$$

avec $l = \lceil \log_2 a/\beta \rceil$, $a_i = a_{(i \cdot \beta \to (i+1) \cdot \beta - 1)}$ et $b_i = b_{(i \cdot \beta \to (i+1) \cdot \beta - 1)}$. En particulier, a_i et b_i représentent les chiffres de a et b dans la base β.

Pour les versions polynomiales, on suppose que nos polynômes sont de degré n, donc pour une multiplication $C = A \cdot B = \sum_{i=0}^{2n} c_i X^i$, $A = \sum_{i=0}^{n} a_i X^i$ et $B = \sum_{i=0}^{n} b_i X^i$.

1.4.1 L'algorithme naïf

L'algorithme classique est l'algorithme de base pour la multiplication. C'est l'équivalent de l'opération de convolution. Pour la version sur les entiers, l'algorithme peut s'écrire :

$$c = \sum_{k=0}^{2 \cdot (l-1)} \sum_{i=0}^{k} a_i b_{k-i} \cdot 2^{k\beta} \tag{1.15}$$

On remarque que pour $\beta > 1$, le produit $a_i b_{k-i}$ est potentiellement plus grand que $2^{\beta} - 1$, ce qui implique la nécessité de gérer une propagation de retenues pour calculer c.

Pour le cas des polynômes, la formule devient :

$$C = \sum_{k=0}^{2\cdot n} \sum_{i=0}^{k} a_i \cdot b_{k-i} \cdot X^k \tag{1.16}$$

Cette fois-ci, aucune propagation de retenue n'est nécessaire.

1.4.2 L'algorithme de Karatsuba-Ofman [KO62]

L'algorithme de Karatsuba est une amélioration de l'algorithme de multiplication classique visant à réduire le nombre de multiplications élémentaires nécessaires.

Pour l'algorithme de Karatsuba, il est nécessaire de découper nos opérandes d'entrée en deux parties. Pour simplifier, on supposera qu'ils sont de même longueur (cela revient à fixer $l = 2$, donc $\beta = \lceil \log_2 a/2 \rceil$). On obtient alors $a = a_0 + a_1 \cdot 2^\beta$, et $b = b_0 + b_1 \cdot 2^\beta$. En utilisant l'algorithme de multiplication classique, on obtient :

$$c = a_0b_0 + (a_0b_1 + a_1b_0) \cdot 2^\beta + a_1b_1 \cdot 2^{2\beta} \tag{1.17}$$

On pose :

$$\begin{aligned} z_0 &= a_0b_0 \\ z_1 &= a_0b_1 + a_1b_0 \\ z_2 &= a_1b_1 \end{aligned}$$

On obtient alors $c = z_0 + z_1 \cdot 2^\beta + z_2 \cdot 2^{2\beta}$.

On remarque que pour mener à bien l'opération, il est nécessaire de réaliser quatre multiplications sur des entiers de β bits pour calculer c. L'optimisation proposée par Karatsuba consiste à remarquer que le terme z_1 peut s'écrire :

$$\begin{aligned} z_1 = a_0b_1 + a_1b_0 &= (a_0 + a_1) \cdot (b_0 + b_1) - a_0b_0 - a_1b_1 \\ &= (a_0 + a_1) \cdot (b_0 + b_1) - z_0 - z_2 \end{aligned} \tag{1.18}$$

Comme les facteurs z_0 et z_2 sont de toute façon calculés, le terme z_1 ne nécessite qu'une seule multiplication au lieu de deux avec l'algorithme classique, abaissant à trois le nombre de multiplications sur des entiers de β bits pour le calcul de c. L'algorithme de Karatsuba s'écrit donc :

$$c = a_0b_0 + \Big((a_0 + a_1) \cdot (b_0 + b_1) - a_0b_0 - a_1b_1 \Big) \cdot 2^\beta + a_1b_1 \cdot 2^{2\beta} \tag{1.19}$$

Afin de réduire davantage la complexité de la multiplication, l'algorithme de Karatsuba peut être appliqué aux sous-produits z_0, z_1 et z_2. Le nombre de fois où l'on applique l'algorithme de Karatsuba aux sous-produits est appelé le nombre de récursions.

Pour la version polynomiale, le fonctionnement reste très semblable. On divise les coefficients de nos polynômes en deux parties. On pose :

$$A = A_L + A_H \cdot X^{\lceil n/2 \rceil}$$
$$B = B_L + B_H \cdot X^{\lceil n/2 \rceil}$$

A_L (resp. B_L) sont les coefficients de poids faible du polynôme A (resp. B), et A_H (resp. B_H) les coefficients de poids fort. L'algorithme de Karatsuba peut s'écrire :

$$C = A_L B_L + \Big((A_L + A_H) \cdot (B_L + B_H) - A_L B_L - A_H B_H \Big) \cdot X^{\lceil n/2 \rceil} + A_H B_H \cdot X^{2\lceil n/2 \rceil} \tag{1.20}$$

1.4.3 L'algorithme de Toom-Cook [Too63]

Dans son écriture initiale, l'algorithme de Toom-Cook reprenait la démarche de l'algorithme de Karatsuba mais cette fois-ci en découpant les nombres à multiplier en trois parties au lieu de deux. Depuis, l'algorithme s'est généralisé et l'on parle de Toom-k pour une multiplication découpant les nombres à multiplier en k parties.

L'algorithme Toom-k est l'application du principe d'interpolation à la multiplication de polynômes. Le principe de l'interpolation polynomiale est simple : pour un polynôme de degré k, ce polynôme est défini de manière unique en connaissant $k+1$ évaluations de celui-ci. Dans un contexte de multiplication, pour des polynômes de degré k, il est nécessaire de connaitre $2 \cdot k - 1$ points.

Intéressons-nous plus particulièrement à Toom-3, le plus utilisé en pratique. Toom-3 demande l'évaluation d'un polynôme en $2 \cdot 3 - 1 = 5$ points (en pratique : 0, 1, -1, -2, ∞).

Posons :

$$a = a_0 + a_1 \cdot 2^{\beta} + a_2 \cdot 2^{2\beta}$$
$$b = b_0 + b_1 \cdot 2^{\beta} + b_2 \cdot 2^{2\beta}$$
$$P[X] = a_0 + a_1 \cdot X + a_2 \cdot X^2$$
$$Q[X] = b_0 + b_1 \cdot X + b_2 \cdot X^2$$

Alors, en appliquant le principe d'interpolation aux cinq points précédemment évoqués, on obtient :

$$
\begin{aligned}
P[0] &= a_0 & Q[0] &= b_0 \\
P[1] &= a_0 + a_1 + a_2 & Q[1] &= b_0 + b_1 + b_2 \\
P[-1] &= a_0 - a_1 + a_2 & Q[-1] &= b_0 - b_1 + b_2 \\
P[-2] &= a_0 - 2a_1 + 4a_2 & Q[-2] &= b_0 - 2b_1 + 4b_2 \\
P[\infty] &= a_2 & Q[\infty] &= b_2
\end{aligned}
$$

En multipliant les différents points d'interpolation termes-à-termes, nous obtenons un polynôme $R = \sum_{i=0}^{4} r_i \cdot X^i$ dont nous maîtrisons l'évaluation aux points d'interpolation. Pour retrouver les différents r_i, il suffit d'inverser le système d'équations. Par simplicité, on écrit généralement ces équations sous forme de matrice, ce qui donne dans notre cas :

$$
\begin{pmatrix} R[0] \\ R[1] \\ R[-1] \\ R[-2] \\ R[\infty] \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} \tag{1.21}
$$

Compte tenu des points d'interpolation choisis, cette matrice est inversible et est donnée par :

$$
\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} R[0] \\ R[1] \\ R[-1] \\ R[-2] \\ R[\infty] \end{pmatrix} \tag{1.22}
$$

Pour obtenir le résultat de la multiplication, il suffit de calculer $R[2^\beta]$.

La limitation principale de l'algorithme de multiplication Toom-3 est la présence de fractions dans la matrice inverse présentée équation 1.22, et notamment demande une division par trois. Pour utiliser Toom-3 à son meilleur potentiel, il est préférable que l'algorithme demande que le résultat final de la multiplication soit divisé par trois, afin que cette opération soit directement intégrée aux calculs.

L'application de Toom-3 à la multiplication polynomiale est directe.

1.4.4 L'algorithme de la NTT [Pol71]

L'algorithme de la NTT est une transformée de Fourier discrète appliquée à un corps fini. C'est l'algorithme qui demande le plus faible nombre de sous-produits asymptotiquement, au prix de pré- et post-calculs complexes (comparés aux précédents algorithmes) potentiellement coûteux à implémenter en pratique.

Son implémentation demande un paramétrage particulier :

1) Il faut découper les nombres à multiplier en n segments. On pose :

$$\begin{aligned} a &= \sum_{i=0}^{n-1} a_i \cdot 2^{i\beta} \\ b &= \sum_{i=0}^{n-1} b_i \cdot 2^{i\beta} \\ P[X] &= \sum_{i=0}^{n-1} a_i \cdot X^{i\beta} \\ Q[X] &= \sum_{i=0}^{n-1} b_i \cdot X^{i\beta} \end{aligned} \tag{1.23}$$

2) Le nombre de points de la NTT noté N doit satisfaire $N > 2 \cdot n$.
3) Les sous-produits entiers doivent être calculés modulo un entier p premier tel que $n \equiv 1 \mod p$.

Le point 3) permet de s'assurer que $\mathbb{Z}_p$ contient au moins une racine primitive $N^{\text{ème}}$ de l'unité w. On définit alors la NTT d'un polynôme A comme l'évaluation de ce polynôme aux points $(w^0, w^1, w^2, \cdots, w^{N-1})$. En d'autres termes :

$$\mathrm{NTT}_w^N(A) = \begin{pmatrix} w^0 & w^0 & w^0 & \cdots & w^0 \\ w^0 & w^1 & w^2 & \cdots & w^{N-1} \\ w^0 & w^2 & w^4 & \cdots & w^{2(N-1)} \\ w^0 & w^3 & w^6 & \cdots & w^{3(N-1)} \\ \vdots & & & & \vdots \\ w^0 & w^{N-1} & w^{2(N-1)} & \cdots & w^{(N-1)^2} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{1.24}$$

On peut remarquer que cette matrice est une matrice de Vandermonde :

$$\begin{pmatrix} \alpha_0^0 & \alpha_0^1 & \alpha_0^2 & \cdots & \alpha_0^{N-1} \\ \alpha_1^0 & \alpha_1^1 & \alpha_1^2 & \cdots & \alpha_1^{N-1} \\ \vdots & & & & \vdots \\ \alpha_{N-1}^0 & \alpha_{N-1}^1 & \alpha_{N-1}^2 & \cdots & \alpha_{N-1}^{N-1} \end{pmatrix} \tag{1.25}$$

avec $\alpha_i = w^i$. Cette matrice est donc inversible si et seulement si les α_i sont deux-à-deux distincts, ce qui est le cas car w est une racine primitive $N^{\text{ème}}$ de l'unité. On peut donc définir une NTT inverse nommée iNTT, et pouvant s'exprimer comme suit :

$$\text{iNTT}_w^N(A) = n^{-1} \begin{pmatrix} w^0 & w^0 & w^0 & \cdots & w^0 \\ w^0 & w^{-1} & w^{-2} & \cdots & w^{-(N-1)} \\ w^0 & w^{-2} & w^{-4} & \cdots & w^{-(2(N-1))} \\ w^0 & w^{-3} & w^{-6} & \cdots & w^{-(3(N-1))} \\ \vdots & & & & \vdots \\ w^0 & w^{-(N-1)} & w^{-2(N-1)} & \cdots & w^{-(N-1)^2} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \\ a_{n+1} \\ \vdots \\ a_{2n-1} \end{pmatrix} \tag{1.26}$$

Nous retrouvons également dans la littérature une écriture équivalente avec une somme :

$$\text{NTT}_w^n(A)[i] = \sum_{j=0}^{N-1} a_j \cdot w^{ij} \tag{1.27}$$

En reprenant les notations de l'équation 1.23, la multiplication des deux entiers a et b s'écrit :

$$\begin{aligned} R[X] &= \text{iNTT}_w^N\Big(\text{NTT}_w^N(P) \odot \text{NTT}_w^N(Q)\Big) \\ a \cdot b &= R[2^\beta] \end{aligned} \tag{1.28}$$

Il est tout à fait possible de rester dans le domaine de la NTT et continuer à faire des opérations sur les polynômes (additions/soustractions/multiplications).

Dans le cas de la multiplication de grands entiers, il faut faire bien attention au choix du module p. En effet, dans l'équation 1.28, le polynôme résultat $R[X]$ est défini dans $\mathbb{Z}_p[X]$. Ainsi, si p est trop petit, c'est-à-dire qu'un des coefficients de R est plus grand que p, celui-ci sera réduit modulo p et donc le résultat de la multiplication sera fausse. Pour éviter ce cas de figure, p doit respecter l'équation suivante :

$$\forall i \in \{0, 1, \cdots, N-1\}, \sum_{j=0}^{N-1} a_i \cdot b_{j-i} < p \tag{1.29}$$

Soit de manière équivalente :

$$\frac{N}{2}(2^\beta - 1)^2 < p \tag{1.30}$$

L'entier p doit être suffisamment grand pour limiter la taille de la NTT (c'est-à-dire N), mais suffisamment petit pour ne pas avoir à effectuer des multiplications entières complexes.

Prenons un exemple pour illustrer le paramétrage d'une NTT. On se fixe un nombre de points de la NTT à $2^{16} = 65536$. En reprenant l'équation 1.30 et en se fixant un module p tel que $\log_2 p = 63$ bits, il faut alors prendre $\beta = 2^{24}$. Ce paramétrage permettra de multiplier des entiers sur $\beta \cdot \frac{N}{2} = 24 \cdot 32\,768 = 786\,432$ bits.

En général, le calcul de la NTT (ainsi que la iNTT) est parallélisé, en utilisant le fait que w étant une racine primitive $N^{\text{ème}}$ de l'unité, il est possible de découper la somme définie dans l'équation 1.27 :

$$\begin{aligned}
\mathrm{NTT}_w^N(P)[i] &= \sum_{j=0}^{N} p_i \cdot w^{ij} \\
&= \sum_{j=0}^{N/2-1} p_{2j} \cdot w^{2ji} + \sum_{j=0}^{N/2-1} p_{2j+1} \cdot w^{(2j+1)i} \\
&= \sum_{j=0}^{N/2-1} p_{2j} \cdot w^{2ji} + w^i \cdot \sum_{j=0}^{N/2-1} p_{2j+1} \cdot w^{2ji} \\
&= \mathrm{NTT}_{w_{N/2}}^{N/2}(P_{pair})[i] + w^i \cdot \mathrm{NTT}_{w_{N/2}}^{N/2}(P_{impair})[i]
\end{aligned} \tag{1.31}$$

avec $w_{N/2}$ une racine $N/2^{\text{ème}}$ de l'unité, P_{pair} le polynôme contenant les coefficients pairs de P et P_{impair} celui contenant les coefficients impairs. De plus, comme les fonctions $\mathrm{NTT}_{w_{N/2}}^{N/2}$ sont $N/2$-périodiques, on peut écrire :

$$\begin{aligned}
\mathrm{NTT}_w^N(P)[i] &= \mathrm{NTT}_{w_{N/2}}^{N/2}(P_{pair})[i] + w^i \cdot \mathrm{NTT}_{w_{N/2}}^{N/2}(P_{impair})[i] \\
\mathrm{NTT}_w^N(P)[i+N/2] &= \mathrm{NTT}_{w_{N/2}}^{N/2}(P_{pair})[i] - w^i \cdot \mathrm{NTT}_{w_{N/2}}^{N/2}(P_{impair})[i]
\end{aligned} \tag{1.32}$$

Cette opération permet de diviser la taille des NTT par 2 à chaque décimation. De plus, l'exploitation de la périodicité de la NTT permet de diviser par 2 le nombre de multiplications entières à réaliser. Il est possible de réaliser ceci $\log_2 N$ fois, permettant de réduire au maximum la complexité de la NTT.

1.4.5 Optimiser la NTT pour l'homomorphe : l'utilisation de la NWC

Dans le cas de la multiplication polynomiale dans un contexte de chiffrement homomorphe, le résultat de la multiplication doit être calculé modulo un polynôme irréductible. En modifiant l'algorithme de la NTT pour le polynôme irréductible

particulier $X^n + 1$, il est alors possible de réaliser la réduction polynomiale pendant l'opération de multiplication terme-à-terme des deux NTT. Concrètement, réduire un polynôme par $X^n + 1$ revient à calculer l'opération suivante :

$$c_{(i)} = \sum_{j=0}^{i} a_{(j)} b_{(i-j)} - \sum_{j=i+1}^{n-1} a_{(j)} b_{(n+i-j)} \tag{1.33}$$

Cette opération peut facilement être adaptée à la NTT en trouvant $\phi \in \mathbb{Z}_p$ tel que $\phi^2 \equiv w \mod p$. L'adaptation de la multiplication est alors la suivante :

$$\begin{aligned} \bar{A} &= \overline{a_{(i)}} \cdot X^i, \quad \overline{a_{(i)}} = \phi^i \cdot a_{(i)} \\ \bar{B} &= \overline{b_{(i)}} \cdot X^i, \quad \overline{b_{(i)}} = \phi^i \cdot b_{(i)} \\ \bar{C} &= \mathrm{iNTT}_w^N(\mathrm{NTT}_w^N(\bar{A}) \odot \mathrm{NTT}_w^N(\bar{B})) \\ C &= c_{(i)} \cdot X^i, \quad c_{(i)} = \phi^{-i} \cdot \overline{c_{(i)}} \end{aligned} \tag{1.34}$$

Ceci permet de réduire par 2 la taille de la NTT requise pour la multiplication, et permet également de pouvoir rester dans le domaine de la NTT pour réaliser des opérations successives (additions/soustractions/multiplications). Cependant, cette technique est incompatible avec la technique du *batching* dans $\mathbb{Z}_2$, qui pour rappel permet de « regrouper » plusieurs messages dans un seul chiffré.

1.4.6 Quelques mots à propos du système RNS

Le système de représentation des nombres RNS permet de paralléliser des opérations sur de grands opérandes (entiers ou polynômes), en divisant ces derniers en sous-opérandes de plus petite taille. Pour fixer les idées, nous présenterons que la version sur des entiers. On définit l'application suivante :

$$\begin{aligned} \mathbb{Z}_q &\to \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}, \text{ avec } q = \prod_{i=1}^{k} q_i. \\ x &\to ([x]_{q_1}, [x]_{q_2}, \cdots, [x]_{q_k}) \end{aligned} \tag{1.35}$$

Avec $x = \left[\sum_{i=1}^{k} \left[[x]_{q_i} \cdot \frac{q_i}{q} \right]_{q_i} \cdot \frac{q}{q_i} \right]_q$.

On appelle les $[x]_{q_i}$ les résidus de x. Cette application permet de réaliser des opérations élémentaires (additions, soustractions, multiplications) dans le domaine des résidus au lieu de $\mathbb{Z}$. Ainsi, pour deux entiers a et b, faire l'addition, la soustraction ou la multiplication des deux nombres dans $\mathbb{Z}_q$ ou dans le domaine des résidus donne strictement le même résultat. Pour que RNS fonctionne, les modulis $(q_1, q_2, \cdots, q_k)$ doivent être premiers entre eux deux-à-deux. Comme dans le chiffrement homomorphe, les versions polynomiales peuvent avoir des coefficients de

plusieurs centaines de bits, le système RNS permet de paralléliser efficacement certains traitements et simplifie également l'implémentation de la NTT en réduisant la taille des coefficients. Cette optimisation est autant intéressante pour l'implémentation logicielle (utilisation de plusieurs cœurs, simplification de l'implémentation de la NTT), que matérielle (réduction de la taille des sous-opérations).

Le système RNS a cependant quelques limitations. Comme RNS est un système non-positionnel, la réduction modulaire ainsi que la comparaison ne sont pas possibles directement. De plus, la division par un entier r ne peut être effectuée que si elle est exacte ($x = q \cdot d + r$) et que $\text{pgcd}(r, q) = 1$.

1.5 Implémentations logicielles et/ou matérielles

Compte tenu de la variété des schémas et des paramètres de sécurité, réaliser une étude sur les implémentations existantes demande un lourd travail d'analyse et de tri. Compte tenu de l'intérêt de la communauté scientifique pour les schémas basés sur R-LWE, nous focaliserons notre étude sur ces derniers.

Une première remarque à propos des implémentations, quelles soient logicielles ou matérielles, est la forte prédominance de l'algorithme de la NTT pour la multiplication polynomiale. Étant asymptotiquement indéniablement plus performante comparé à ses homologues (complexité polynomiale), elle est un bon candidat *a priori*. Cependant, notamment lors de l'implémentation de la NTT en matériel, nous constaterons que sa complexité d'implémentation réduit sensiblement sa performance.

1.5.1 Les implémentations logicielles

Les bibliothèques logicielles existantes ont l'intérêt d'implémenter un schéma homomorphe complet, contrairement aux implémentations matérielles où souvent seul un sous-ensemble d'opérations est accéléré (généralement, la multiplication).

En ce qui concerne les bibliothèques pour les schémas basés sur R-LWE, les principales implémentations sont les suivantes :

- Helib [Hal], présentée dans [HS14], qui implémente [BGV12] ;
- SEAL [LCPa] de Microsoft qui implémente FV (à partir de la version 2.0) ;
- NFLlib [AMBG+], présentée dans [AMBG+16] qui est une boite à outils pour tout schéma reposant sur R-LWE (adaptation au schéma FV disponible) ;
- [BEHZ16] (FV-FULL-RNS) implémentant également FV (implémentation privée).

Les multiplications polynomiales sont implémentées avec l'algorithme de la NTT, améliorées par l'utilisation de RNS. La table 1.4 présente les temps de calculs pour le déchiffrement et la multiplication homomorphe pour les schémas implémentant FV, avec une profondeur multiplicative de 4. La génération de clés et le chiffrement

Table 1.4 – Temps de calcul des opérations de déchiffrement et de multiplication homomorphe pour différentes bibliothèques actuelles.

$n = 4096, \log_2 q \in [130, 160]$				
Bibliothèque	Déchiffrement	Multiplication homomorphe	Processeur	Source
SEAL v2.1	5.019 ms	48.12 ms	Intel i7-4910MQ	[LCPb]
FV-NFLlib	0.9 ms	17.20 ms	Intel Xeon E5-2666	[AMBG+16]
FV-FULL-RNS	0.326 ms	7.688 ms	Intel i7-4810MQ	[BEHZ16]

ne sont pas présentés car ils dépendent trop fortement de l'efficacité de la génération de nombres aléatoires.

Les deux bibliothèques les plus performantes à l'heure actuelle sont FV-NFLlib et FV-FULL-RNS, qui utilisent notamment la version NWC de la NTT. FV-NFLlib obtient des performances légèrement en dessous de FV-FULL-RNS, NFLlib étant plus généraliste donc plus flexible au prix de la performance. Ces implémentations ne supportent pas l'opération de *batching*.

SEAL quant-à-elle implémente également FV, mais est plus flexible que FV-FULL-RNS car implémente le *batching*. Cependant, le *batching* implémenté dans SEAL ne supporte pas le *batching* avec des messages binaires.

En ce qui concerne FV-FULL-RNS, elle obtient de telles performances car elle s'appuie sur une version modifiée de FV plus adaptée au système RNS.

La table 1.5 présente davantage de résultats sur le temps de calcul de la multiplication homomorphe pour la bibliothèque FV-FULL-RNS. Cette bibliothèque logicielle étant la plus performante à l'heure actuelle pour FV, nous utiliserons ces résultats par la suite comme outil de comparaison.

Il existe une dernière bibliothèque, plutôt prometteuse, basée sur le schéma de chiffrement SHIELD. Comme précisé dans la section 1.2.2, il s'agit d'un schéma de $3^{ème}$ génération reposant sur R-LWE, donc plus efficace asymptotiquement que les précédents schémas (notamment FV). Les auteurs de SHIELD présentent une implémentation de leur schéma sur GPU, permettant de réaliser une multiplication homomorphe en 3 ms pour une profondeur multiplicative de 10. Ceci est dû au fait que SHIELD demande la multiplication de nombreux polynômes ($(2 \cdot \log_2 q)^2$ pour être précis) mais de petite taille ($n = 1024$ et $\log_2 q = 31$ bits). Ceci est donc particulièrement bien adapté au GPU, qui possède de nombreuses matrices d'unité de calculs.

TABLE 1.5 – Temps de calcul des opérations de multiplication homomorphe pour FV-FULL-RNS présenté dans [BEHZ16].

$(n, \log_2 q)$	Multiplication Homomorphe de FV
(2048, 90)	2.71 ms
(4096,186)	7.69 ms
(8192,372)	37.74 ms
(16384,744)	206.51 ms
(32768,1550)	1,406.96 ms

1.5.2 Les implémentations matérielles

Les implémentations matérielles existantes sont souvent sources de confusion dans le cadre du chiffrement homomorphe. La majorité des implémentations se focalisent sur une opération élémentaire donnée, par exemple la multiplication polynomiale pour les schémas reposants sur R-LWE. Cela n'est pas forcément parlant pour le chiffrement homomorphe, car bien qu'il faille exécuter des multiplications polynomiales, elles doivent s'exécuter dans un contexte précis et demande donc d'adapter l'arithmétique. En particulier, dans FV, la multiplication homomorphe demande une opération de division et arrondi après celle-ci, qui en plus d'être potentiellement complexe à implémenter en matériel, demande que la multiplication polynomiale soit exécutée sans réduction modulaire des coefficients.

Certaines publications présentent néanmoins une architecture plus complète, et donc plus parlante. Si l'on se restreint aux implémentations proches du schéma FV, il existe trois implémentations proposant une multiplication homomorphe complète (avec relinéarisation). De plus, elles ont l'avantage de couvrir trois jeux de paramètres différents, et donc de pouvoir observer les différentes limitations matérielles lors de l'augmentation des paramètres de sécurité. Remarquons cependant que ces implémentations avaient été réalisées pour le schéma YASHE'. Cela n'est pas particulièrement bloquant en soit, les schémas YASHE' et FV ayant de nombreuses similitudes arithmétiques. Concrètement, là où FV demande trois multiplications polynomiales et une relinéarisation de deux polynômes, YASHE' ne demande qu'une multiplication polynomiale et une relinéarisation sur un polynôme (appelé explicitement « key switch »). Il est donc possible d'estimer le temps de traitement de FV à partir de ces implémentations. Pour être le plus précis possible, YASHE' demandait certes moins de polynômes par chiffré (un au lieu de deux pour FV), cependant les paramètres de l'époque imposaient un module q plus large. Ainsi, les implémentations, ramenées à FV, surestiment légèrement la valeur du module.

Ces implémentations permettent de couvrir les paramètres suivants :

- ($n = 4096, \log_2 q = 125$ bits, $\omega = 64$ bits) et ($n = 16384, \log_2 q = 512$ bits, $\omega = 64$ bits) pour [PG14] ;
- ($n = 32768, \log_2 q = 1228$ bits, $\omega = 64$ bits) pour [SRJV$^+$].

Dans le cadre de FV, ces implémentations permettent de réaliser des algorithmes avec respectivement : une profondeur multiplicative de 2 pour 113 bits de sécurité sans *batching*, une profondeur multiplicative de 15 pour 110 bits de sécurité sans *batching*, et une profondeur multiplicative supérieure à 15 pour 113 bits de sécurité (réduction progressive du niveau de sécurité pour les profondeurs suivantes) avec *batching*. Les architectures pour ($n = 4096, \log_2 q = 125$ bits / $n = 4096, \log_2 q = 512$ bits) et ($n = 16384, \log_2 q = 512$ bits) sont présentées respectivement figures 1.7 et 1.8. Concernant la figure 1.7, le *HomomorphicCore* est composé de deux unités, le *NttCore* qui est responsable du calcul des pré- et post-traitements de la NTT ainsi que les sous-produits, et du *NttMemMgr* qui gère la lecture/écriture des coefficients intermédiaires de la NTT avec les bancs mémoires. L'existence de deux modules q et q' est expliqué par les primitives FV.Mult de l'équation 1.8 et FV.Relin de l'équation 1.9. Pour FV.Mult, la multiplication polynomiale doit être effectuée sans réduction modulaire des coefficients, car on doit réaliser une opération de division et d'arrondi sur ceux-ci (donc on doit réaliser les opérations modulo un entier $q' > 2q$). Pour FV.Relin, les opérations se font modulo q, donc il est nécessaire d'implémenter l'arithmétique modulo q. La double limitation dans cette implémentation est l'interface avec de la mémoire externe, ainsi que la taille des multiplieurs entiers. Dans cet exemple, l'implémentation du système RNS aurait permis de réduire la taille des coefficients tout en augmentant le parallélisme.

L'architecture pour ($n = 32768, \log_2 q = 1228$ bits), compte tenu de la taille du module q, utilise le système RNS. De plus, les auteurs ont décidé de proposer une version avec *batching*, donc sans l'implémentation de la NWC. Quelques aménagements ont été nécessaires, notamment la réduction du résultat de multiplication par un polynôme cyclotomique (compatible avec le *batching*). Pour exploiter le parallélisme de RNS, les auteurs ont implémenté une matrice de petits crypto-processeurs possédant trois unités de calcul : La PAU pour la gestion de l'arithmétique sur les polynômes, la CRTU pour la gestion de transformations de bases RNS, et la DRU pour la gestion de la division et arrondi nécessaire dans FV. Ces unités sont représentées dans la figure 1.8. Là encore, la quantité de mémoire nécessaire ainsi que les temps d'accès à la mémoire pénalisent l'implémentation.

La table 1.6 donne les résultats d'implémentation des différentes architectures présentées, ainsi qu'une comparaison à l'implémentation logicielle FV-FULL-RNS. Quel que soit le paramétrage, on peut remarquer que la NTT demande un grand nombre de ressources mémoires (pour le stockage des coefficients intermédiaires notamment, ainsi que différents pré-calculs). Cette mémoire a également un impact sur les performances, réduisant la fréquence de fonctionnement du FPGA. La NTT est efficace lorsque le degré des polynômes est proche d'une puissance de 2. Plus on s'éloigne d'une puissance de 2, plus la NTT est surdimensionnée. Dans le chiffrement homomorphe FV, à cause de l'étalement des degrés selon la profondeur multiplicative, de nombreuses configurations sont défavorables à la NTT. En se référant à la table 1.3, pour ($\omega = 32$ bits, $\lambda = 80$ bits) et pour les 10 premières profondeurs multiplicatives, un surcoût supérieur à 30% survient dans 60% des cas (profondeurs 1, 3, 4, 6, 7 et 8), avec un pic pour les profondeurs multiplicatives 1, 3 et 6.

TABLE 1.6 – Présentation des résultats d'implémentations matérielles de la littérature.

		NWC [PNPM]		NTT [SRJV+]
$(n, \log_2 q)$		(4096, 125)	(16384, 512)	(32768,1228)
Resources	Cible	Stratix-V	Stratix-V	Virtex-7
	ALM/Slice LUTs	69 058	141 090	377,120
	Registres	144 747	391 773	323 120
	Bits de mémoire	8 031 568	17 626 400	26 247 168
	DSPs	144	577	1792
	f_{max}	100 MHz	66 MHz	143 MHz
FV × (*)		15.46 ms	122.30 ms	263.40 ms
Comparaison à l'état de l'art logiciel (FV-FULL-RNS)				
$(n, \log_2 q)$		(4096, 186)	(16384, 744)	(32768,1550)
FV ×		7.69 ms	206.51 ms	1,406.96 ms

(*) Estimation du temps de calcul de la multiplication homomorphe en considérant le calcul de 3 multiplications avec division et arrondi, et 2 relinéarisations, afin de faire correspondre le résultat au schéma de chiffrement FV.

En ce qui concerne la comparaison directe des implémentations logicielles et matérielles, on remarque une nette tendance favorable pour les implémentations matérielles pour les grandes profondeurs multiplicatives. Cela se traduit par un plus grand parallélisme proposé par les FPGA. Pour des degrés plus faibles cependant, la NTT reste un algorithme difficile à implémenter, notamment en matériel, et donc la puissance de calcul d'un CPU reste compétitif. L'étude reste cependant très parcellaire, car seule l'implémentation de la NTT est proposée. Ainsi, il n'est pas exclus que d'autres algorithmes, certes moins performants asymptotiquement, soient une bonne alternative pour l'implémentation de l'homomorphe, et en particulier pour des degrés de quelques milliers.

1.6 Conclusion

Comme nous avons pu le constater dans ce chapitre, sur le papier, le chiffrement homomorphe apporte une solution à de nombreux problèmes de protection des données personnelles (notamment depuis l'émergence des services *Cloud*).

Cependant, derrière les promesses d'une protection absolue des données privées, se cache une cryptographie complexe et qui possède de nombreux verrous scientifiques à relever. Les principaux freins à son implémentation sont les suivants :

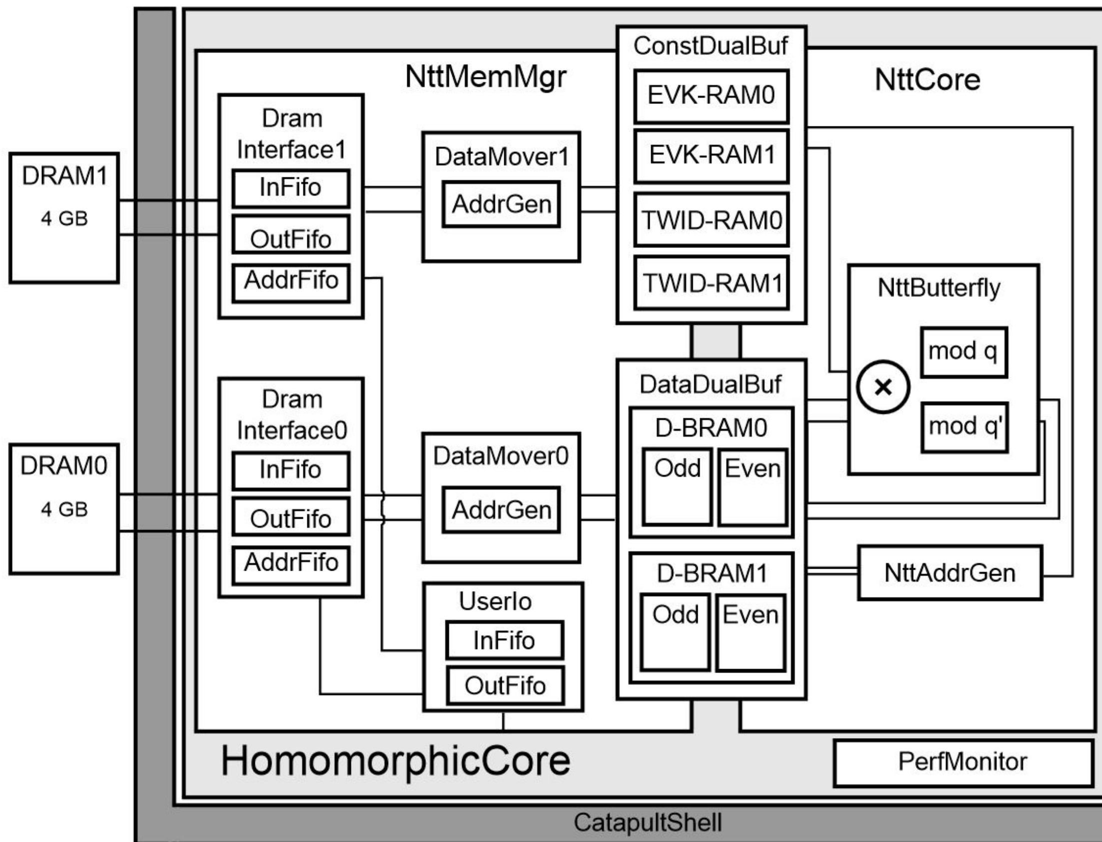


FIGURE 1.7 – Architecture de l'accélérateur matériel proposé dans [PG14].

1) Des données chiffrées de plusieurs millions de bits pour quelques bits (voir un seul bit) de message ;
2) L'obligation de transformer les opérations en circuit de portes AND et XOR ;
3) Le temps de traitement qui peut devenir rapidement prohibitif.

Il existe des solutions qui permettraient de rendre bien plus compétitif le chiffrement homomorphe. Notamment, la technique du *batching*, qui permettrait à la fois de réduire l'expansion du chiffré (donc moins de données chiffrées par bit de message) et les temps de calcul car plusieurs opérations seraient exécutées en parallèle.

Cependant, le *batching* étant difficilement compatible avec l'algorithme de la NTT (et d'autant plus la NWC), la plupart des implémentations font l'impasse dessus, ou n'implémentent pas la version sur les messages binaires qui pourtant est la seule version qui permet une grande flexibilité.

Dans tous les cas, le choix de l'algorithme (et donc du service) à implémenter sera crucial. Il semble plus raisonnable de s'intéresser à des algorithmes avec une profondeur multiplicative faible, afin de limiter la taille des chiffrés.

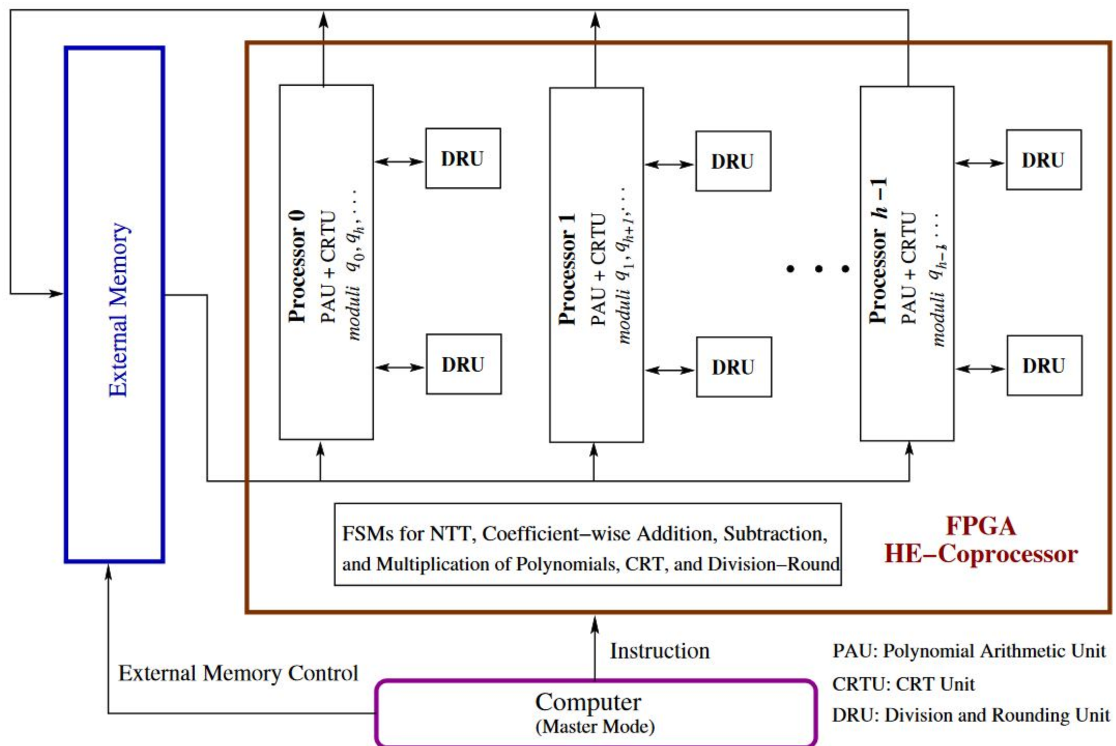


FIGURE 1.8 – Architecture de l'accélérateur matériel proposé dans [SRJV+].

1.7 Approche choisie

Compte tenu du bilan formulé à la section 1.6, nous nous sommes intéressés à l'accélération du chiffrement homomorphe pour des profondeurs multiplicatives raisonnables (typiquement jusqu'à 8) sans se donner de restriction sur le choix de l'algorithme de multiplication. En particulier, nous avons étudié l'algorithme de Karatsuba car il possède de nombreux avantages par rapport à la NTT :

- Son implémentation est simple, avec des pré- et post-traitements uniquement composés d'additions et de soustractions de polynômes ;
- Bien moins de restrictions sur le choix du module q et du degré n du polynôme cyclotomique (voir table 1.7 pour des exemples de degré n possibles) ;
- Naturellement compatible avec le *batching* car Karatsuba ne demande pas de changement d'espace comme le demande la NTT.

Nous verrons également au chapitre 3 que Karatsuba se prête bien à une application de type co-conception, avec un partage des calculs entre le logiciel et le composant matériel.

FV restant un schéma avec un grand nombre d'opérations, il nous a paru plus légitime de considérer une architecture comprenant un processeur généraliste (responsable des opérations de haut niveau) et un accélérateur matériel faisant office de co-processeur (pour l'accélération de certaines opérations moins performantes en logiciel qu'en matériel). Le choix du FPGA pour l'accélérateur matériel s'est avéré

TABLE 1.7 – Exemples de multiplications polynomiales possibles avec l'algorithme de Karatsuba.

Nombre de récursions	Degré des sous-produits	Degré de la multiplication polynomiale
9	3	2048
9	4	2560
9	5	3072
9	6	3584
10	3	4096
10	4	5120
10	5	6144
10	6	7168

naturel compte tenu de la variété des paramètres de sécurité pour le chiffrement homomorphe. Cette architecture permet également d'être plus flexible sur la répartition des tâches entre CPU et accélérateur matériel afin de répartir une opération donnée sur le composant le plus adapté.

La suite du document est organisée comme suit : Le chapitre 2 présente l'implémentation des opérations exécutées en logiciel. Le chapitre 3 décrit le fonctionnement de l'architecture co-conception logicielle/matérielle et l'adaptation de Karatsuba aux différentes fonctions du schéma de chiffrement FV. Le chapitre 4 présente et discute les résultats d'implémentation. Finalement, le chapitre 5 propose une conclusion à l'étude et les perspectives d'évolution.

Le travail d'implémentation a été mené en collaboration avec Cédric Seguin, ingénieur de recherche, recruté pour une durée de 9 mois dans le cadre du projet PEPS CNRS Homcrypt. Son travail a été de mettre en place une connexion rapide et efficace entre l'application logicielle et le composant matériel.

2

Implémentation logicielle des primitives de FV

L'objectif de ce chapitre est de présenter l'implémentation des différentes primitives de FV pour la partie logicielle uniquement. L'accélérateur matériel sera présenté chapitre 3. Ce chapitre couvre notamment :

- Une présentation des différentes bibliothèques logicielles existantes et leur limite dans notre architecture ;
- Le choix de la représentation des nombres et des polynômes ;
- Les détails de l'implémentation des opérations élémentaires sur les polynômes (et par extension FV), notamment l'addition, la soustraction, la réduction modulaire polynomiale ainsi que le *batching*.

2.1 Bibliothèques logicielles existantes

Afin d'exploiter au maximum les capacités de calcul des processeurs généralistes, nous nous sommes orientés vers la programmation en langage C. Permettant un accès à des fonctions de bas niveau, ce langage permet d'exploiter efficacement les ressources arithmétiques. De plus, la plupart des bibliothèques d'arithmétique sur les entiers/polynômes sont écrites en langage C, simplifiant l'interopérabilité avec notre approche.

Compte tenu que l'arithmétique du schéma FV est basée sur $\mathbb{Z}_q(X)/f(X)$, il était important de trouver une bibliothèque permettant d'implémenter efficacement cette arithmétique.

Compte tenu que toutes les opérations de multiplication entière seront effectuées sur le composant matériel, les opérations de la partie logicielle se résument à des calculs sur des entiers de quelques centaines de bits. Il reste néanmoins à implémenter les opérations d'addition et de soustraction d'entiers, de division et arrondie pour FV.Mult, ainsi que la réduction modulaire entière.

Ces opérations peuvent être implémentées avec GMP [Gra15] (GNU Multiple Precision Arithmetic Library). Cette bibliothèque permet d'accélérer les calculs sur

des nombres entiers (*mpz*), des nombres rationnels (*mpq*) ainsi que des nombres flottants (*mpf*) sans limite de précision (hormis la mémoire disponible sur la machine où GMP est exécuté). Elle est donc *a priori* bien adaptée à notre configuration. GMP tient ses performances du fait que certaines fonctions clés sont écrites en assembleur. De plus, la bibliothèque a été optimisée pour de nombreuses architectures de processeurs, permettant d'optimiser davantage les opérations.

En ce qui concerne les bibliothèques d'arithmétiques sur les polynômes, les plus performantes actuellement sont la NTL [Sho16] and FLINT [Har16]. Bien que non fondamentales dans notre architecture, ces bibliothèques ont l'avantage d'implémenter de nombreux algorithmes (factorisation d'entiers ou de polynômes, multiplication, réduction modulaire polynomiale...) et ont été d'une aide précieuse pour les différents tests et validations des accélérateurs, la détermination et l'exploration des polynômes cyclotomiques, la détermination des paramètres de *batching*, ainsi que le test fonctionnel des schémas de chiffrement homomorphe.

2.2 Approche retenue

Nous avons testé deux approches pour la conception de notre accélérateur. Nous avons premièrement effectué l'implémentation des différentes opérations à l'aide de bibliothèques standards (GMP et NTL). Bien que optimisées pour la performance, ces bibliothèques avaient une limitation importante dans notre application : la non correspondance de la taille des opérateurs arithmétiques entre le logiciel et le matériel. En effet, pour des processeurs récents, les unités arithmétiques sont de l'ordre de 32 ou 64 bits. Sur FPGA, la taille des opérandes varie d'un fabriquant à l'autre et d'une famille à une autre. Ainsi, de nombreuses conversions étaient nécessaires.

Les bibliothèques standards n'étant pas conçues dans cette optique, ces conversions devenaient coûteuses. Pour donner un ordre de grandeur, pour une profondeur multiplicative de 4 avec le schéma de chiffrement FV, les polynômes ont environ 3000 coefficients sur 130 bits. Après les pré-traitements de Karatsuba, cela revient à convertir environ 4 millions de bits de données. Nous nous sommes donc orientés vers une deuxième approche demandant la conception d'une bibliothèque optimisée prenant en compte les contraintes d'interface. Il aurait été également possible de déporter la conversion des données au niveau du composant matériel. Cette conversion amène cependant la conception d'une mémoire tampon pour amortir la différence de débit provoquée par cette conversion.

Afin d'être suffisamment flexible pour s'adapter aux contraintes matérielles, nous avons déterminé quelle représentation des nombres entiers était la plus adaptée pour utiliser au mieux les ressources logicielles, tout en simplifiant grandement la conversion des nombres vers le composant matériel.

2.3 Représentation des nombres

La représentation des nombres est une notion clé en arithmétique car elle va déterminer, en partie, les algorithmes pour les opérations, le parallélisme possible, ainsi que le domaine de validité des opérations. La représentation des nombres la plus répandue est la numération *simple de position* décrite dans [Mul89]. Nous recopions sa définition pour simplifier la lecture. Pour une base β, un nombre entier a peut s'exprimer :

$$a = \sum_{i=0}^{n-1} a_i \cdot \beta^i, \text{ avec } \begin{cases} a_i \text{ entier} \\ 0 \leq a_i < \beta \end{cases} \tag{2.1}$$

$(a_0, a_1, \cdots, a_{n-1})$ est appelée la décomposition de a en base β, et on la notera $(a_{n-1}a_{n-2}\cdots a_0)_\beta$. Les différents a_i sont également appelés chiffres du nombres a, et on gardera cette appellation par la suite.

Cette représentation est :

1) unique : une seule représentation par nombre.
2) complète : chaque nombre entier positif à une représentation.

Les machines actuelles basant leur arithmétique sur une représentation binaire des nombres, β est une puissance de deux ($(\log_2 \beta) - 1 = 32$ ou 64 bits sur les machines standards). Les opérateurs arithmétiques sont donc implémentés sur des mots de 32 ou 64 bits. Lorsque deux nombres a et b disposent de plusieurs chiffres, l'addition de a par b s'effectue par addition des chiffres terme-à-terme avec propagation de retenue. Cette propagation de retenue crée une dépendance de données obligeant à exécuter séquentiellement les calculs.

Afin de pouvoir paralléliser les opérations, il est nécessaire de casser cette dépendance de données. La notation classique permettant de faire cette séparation est nommée *notation à retenue conservée* [Mul89], et se définie comme suit :

$$a = \sum_{i=0}^{n-1} (a_i + c_i) \cdot \beta^i, \text{ avec } \begin{cases} a_i \text{ entier} \\ 0 \leq a_i < \beta \\ c_i \in \{0, 1\} \end{cases} \tag{2.2}$$

Cette représentation permet de faire des opérations d'addition en parallèle, car c_i garde l'information retenue. Cependant, le nombre d'additions successives reste faible (2 au pire cas). Afin de dépasser cette limite, cette notation se généralise en considérant que c_i peut prendre une taille quelconque. En plus de β, on va se fixer un entier α qui représente la valeur maximale que peut prendre un chiffre. Cette représentation se nomme numération *simple de position* avec redondance, et s'écrit

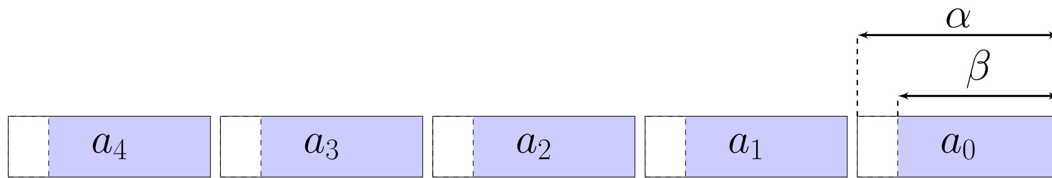


FIGURE 2.1 – Représentation d'un nombre entier a en base β avec une notation redondante. α correspond à la valeur maximale que peut prendre le chiffre a_i.

de manière plus formelle :

$$a = \sum_{i=0}^{n-1} a_i \cdot \beta^i, \text{ avec } \begin{cases} a_i \text{ entier} \\ 0 \leq a_i < \alpha \\ \alpha > \beta \end{cases} \tag{2.3}$$

Tant que chaque chiffre ne dépasse pas α pendant les opérations, cette représentation permet de garder l'information de retenue dans les bits de poids fort des chiffres, et donc retarder la nécessité de propager la retenue. Il est alors possible de faire des opérations successives et parallèles sur les chiffres. Cette représentation permet également une certaine flexibilité car il est possible de jouer sur les paramètres α et β. La figure 2.1 illustre cette représentation en donnant la représentation d'un entier a.

Cette flexibilité a été déterminante dans notre architecture. Compte tenu de la taille des opérateurs arithmétiques sur les FPGA, inférieurs aux opérateurs implémentés sur les processeurs, il a été possible de positionner β sur la taille des unités arithmétiques de l'accélérateur matériel et α sur les unités arithmétiques de la machine exécutant l'application logicielle. Cela a permis un double avantage, pouvoir paralléliser les opérations sur l'application logicielle, et s'adapter aux unités de calculs sur le composant matériel.

Pour simplifier les explications, nous nommerons *bits de garde* les bits situés entre les indices $\log_2 \beta$ et $\log_2 \alpha - 1$ dans la représentation binaire des chiffres.

Il reste un dernier point à présenter, à savoir la gestion des soustractions et donc des nombres négatifs. Pour ce faire, nous avons utilisé la représentation usuelle de la soustraction, à savoir la représentation en *complément à deux* [Mul89]. Pour passer d'un nombre positif à un nombre négatif, il suffit de calculer :

$$-a = \overline{a} + 1 \tag{2.4}$$

où $\overline{a}$ représente l'inversion bit à bit du nombre a. Avec cette représentation, les nombres négatifs sont représentés avec un bit de poids fort égal à 1. Ainsi, le nombre $(1001)_2$ est égal à $(-7)_{10}$. Par extension, la soustraction de deux nombres a et b s'écrit de la manière suivante :

$$a - b = a + (-b) = a + \overline{b} + 1 \tag{2.5}$$

Cette représentation a pour avantage de conserver le même opérateur pour les additions et les soustractions (donc diminution de la surface silicium utilisée), car la soustraction est transformée en addition.

2.4 Implémentation des opérations arithmétiques sur les entiers

Maintenant que nous avons présenté les différentes représentations des nombres, intéressons nous à leur implémentation. Nous nous intéresserons dans un premier temps aux opérations élémentaires d'addition et de soustraction, car elles seront réutilisées par la suite pour des opérations plus complexes (*batching*, réduction modulaire polynomiale, Karatsuba, ...).

2.4.1 L'addition de nombres dans la représentation en complément à deux

La figure 2.2 présente l'addition de deux nombres a et b et leur résultat c avec la représentation *simple de position* avec redondance. Les bits représentés en bleu représentent l'occupation réelle du chiffre à un instant donné, et la partie blanche les bits encore non utilisés. Dans cet exemple, les chiffres de a et b étant strictement inférieurs à β, ils ont une représentation équivalente à la numération *simple de position* décrite équation 2.1. Lors de l'opération d'addition, le résultat d'une sous-somme peut très bien être plus grand que β. Ainsi, le résultat final occupe davantage de bits que les opérandes initiales. Dans le pire cas, le chiffre résultant occupe 1 bit de plus que les opérandes d'entrée.

Au bout d'un certain nombre d'opérations successives, il est tout à fait possible que le chiffre se rapproche de la valeur limite α, et donc occupe tout l'espace alloué pour ce chiffre. Il est alors nécessaire de réaliser une opération de propagation de retenue afin de réinitialiser chaque chiffre à une valeur inférieure à β. La figure 2.3 présente l'algorithme de propagation de retenue. Le fonctionnement est simple, il suffit d'additionner les *bits de garde* d'un chiffre donné au chiffre suivant. Plusieurs remarques sont à formuler :

1) afin de réinitialiser tous les *bits de garde*, il est nécessaire de réaliser l'opération de propagation de retenue séquentiellement en commençant par les chiffres de plus petit rang ;
2) la propagation des retenues étant une opération d'addition, elle peut être dans un cas extrême à l'origine d'un débordement du chiffre par rapport à l'espace mémoire alloué (donc plus grand que α). Il sera alors important d'analyser les algorithmes utilisés pour ne pas se situer dans ce genre de cas ;
3) cette opération peut éventuellement faire déborder le dernier chiffre (a_4 dans notre exemple). Il existe donc une valeur maximale que peut prendre un

TABLE 2.1 – Influence des paramètres α et β de la représentation proposée équation 2.3 sur la taille maximale des nombres avec un nombre fixé de chiffres.

Nombre de chiffres	5				
$\log_2 \beta$	16	20	24	28	32
$\log_2 \alpha$	32	32	32	32	32
Taille maximale du nombre (bits)	96	112	128	144	160
Nombre de chiffres	10				
$\log_2 \beta$	16	20	24	28	32
$\log_2 \alpha$	32	32	32	32	32
Taille maximale du nombre (bits)	176	212	248	284	320

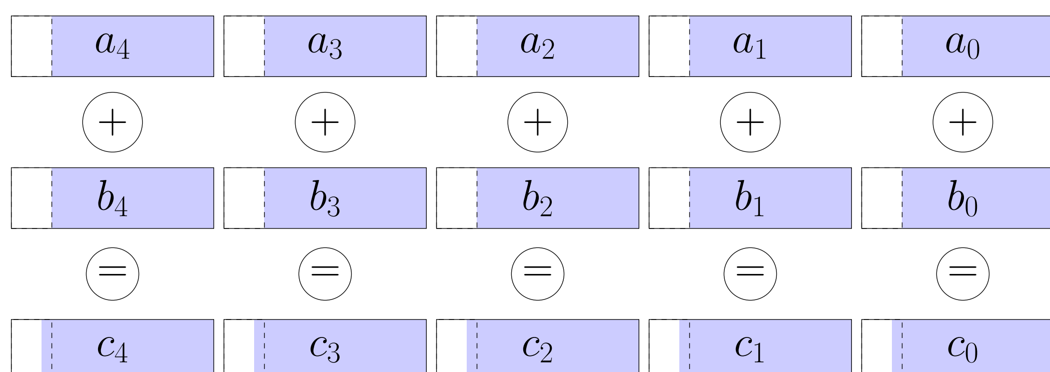


FIGURE 2.2 – Addition de nombres positifs.

nombre lorsque le nombre de chiffres est fixé. La table 2.1 donne un exemple de valeur maximale pour différentes configurations de α et β.

2.4.2 La soustraction de nombres dans la représentation en complément à deux

Alors que l'addition dans la représentation *simple de position* redondante ne pose pas de difficulté particulière, ce n'est pas le cas de la soustraction. Le problème vient du fait que pour la soustraction, il est nécessaire d'inverser les bits du second opérande. Dans la numération *simple de position*, cette inversion ne représente pas de difficulté particulière car les chiffres sont disjoints. Dans notre cas, les chiffres pouvant être plus grands que β, les *bits de garde* d'un chiffre débordent sur le chiffre suivant comme le montre la figure 2.4. Ainsi, la gestion naïve de ce problème serait de propager la retenue puis de faire l'inversion binaire. On perdrait alors le parallélisme que permet la représentation *simple de position* redondante. L'algorithme de

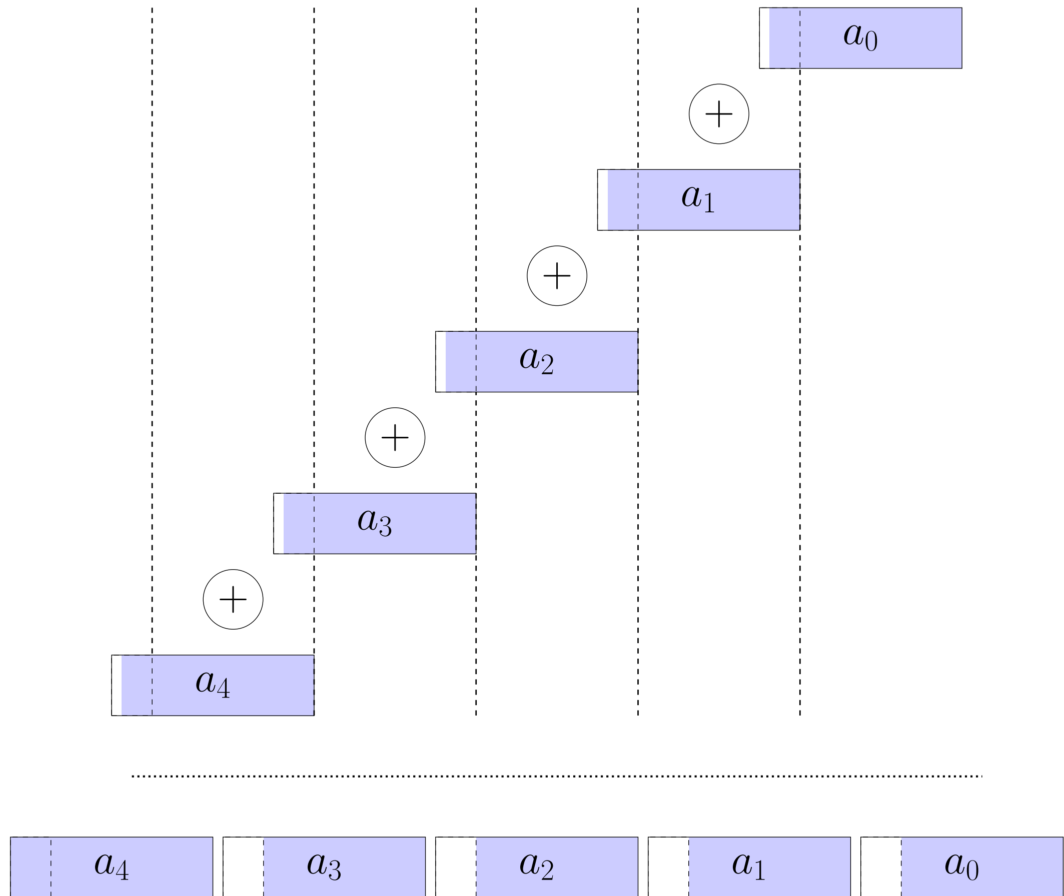


FIGURE 2.3 – Propagation de la retenue.

soustraction s'écrirait alors :

$$
\begin{aligned}
c_0 &= a_0 + \overline{b_0} + 1 \\
c_1 &= a_1 + \overline{b_1} \\
&\vdots \\
c_{n-1} &= a_{n-1} + \overline{b_{n-1}}
\end{aligned}
\tag{2.6}
$$

Cependant, il est tout à fait possible de séparer cette dépendance de données en utilisant les propriétés de la représentation en *complément à deux*. Notons m_{a_i} les $\log_2 \beta$ premiers bits d'un chiffre et r_{a_i} les bits suivants (qui correspondent aux *bits de garde*). L'écriture d'un nombre a revient donc à :

$$
a = \sum_{i=0}^{n-1} a_i \cdot \beta^i = \sum_{i=0}^{n-1} (m_{a_i} + \beta r_{a_i}) \cdot \beta^i \tag{2.7}
$$

Les parties qui se chevauchent figure 2.4 sont la combinaison entre r_{a_i} du chiffre d'indice i, et $m_{a_{i+1}}$ du chiffre $i+1$. Il suffit alors de déterminer comment inverser

$m_{a_{i+1}} + r_{a_i}$. Cette opération peut se simplifier de la manière suivante :

$$\begin{aligned}\overline{m_{a_{i+1}} + r_{a_i}} &= -m_{a_{i+1}} - r_{a_i} - 1 \\ &= -m_{a_{i+1}} - 1 - r_{a_i} - 1 + 1 \\ &= \overline{m_{a_{i+1}}} + \overline{r_{a_i}} + 1\end{aligned} \tag{2.8}$$

Cette relation est particulièrement intéressante car la dépendance de données entre les chiffres n'existe plus. Il est donc possible de calculer la soustraction de deux nombres avec l'algorithme suivant :

$$\begin{aligned}c_0 &= a_0 + \overline{b_0} + 1 \\ c_1 &= a_1 + \overline{b_1} + 1 \\ &\vdots \\ c_{n-1} &= a_{n-1} + \overline{b_{n-1}} + 1\end{aligned} \tag{2.9}$$

Cela revient à faire la soustraction des chiffres termes-à-termes.

2.4.3 Détermination du nombre d'opérations successives possibles avant propagation de retenue

L'intérêt de la représentation *simple de position* avec redondance est qu'elle permet de calculer successivement des opérations d'addition et de soustraction sans propagation de retenue. Il est donc important de bien maîtriser les moment où la propagation de retenues devient indispensable.

Il faut distinguer deux cas :

(1) l'addition de nombres positifs ;

(2) l'addition ou la soustraction de nombres négatifs.

Le cas (1) est un cas que l'on va retrouver principalement dans les pré-traitements de Karatsuba, car seules des additions de polynômes sont calculées. Comme il n'y a que des additions, il n'est pas nécessaire d'utiliser un bit par chiffre pour le stockage du signe, ce qui permet d'augmenter le nombre d'opérations successives. Afin de déterminer le nombre d'opérations successives avec propagation de retenue, il faut considérer que chaque chiffre peut prendre n'importe quelle valeur entre 0 et $\beta - 1$ et calculer l'évolution de la taille des chiffres pendant l'exécution de l'algorithme.

Pour le cas (2), l'approche est similaire, cependant il faut considérer quelques points :

(a) la présence d'un bit de signe pour chaque chiffre ;

(b) la représentation en *complément à deux* n'a pas une représentation symétrique des chiffres, c'est-à-dire que $-(\alpha - 1)$ n'est pas représentable ;

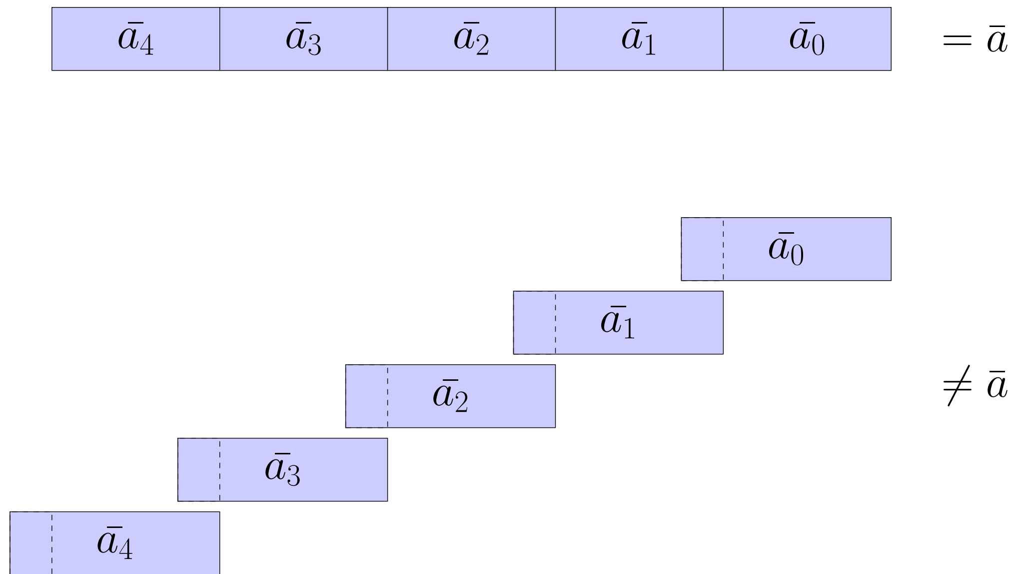


FIGURE 2.4 – Problème d'inversion dans la représentation en complément à deux.

(c) la présence d'un bit de signe demande de modifier légèrement l'opération de propagation de retenue.

Le point (a) va avoir comme impact de réduire le nombre d'opérations successives avant propagation de retenue.

Le point (b) va demander d'être vigilant pour que les chiffres n'atteignent jamais cette valeur durant les calculs.

Le point (c) va demander de recopier le bit de signe lorsque l'on va additionner les *bits de garde* d'un chiffre avec le chiffre suivant. Ainsi, si les *bits de garde* d'une chiffre sont $(1010)_2$, et que l'on souhaite additionner ces bits au chiffre suivant qui à la représentation $(1100100010)_2$, alors l'opération finale sera $(1111111010)_2+(1100100010)_2$.

2.5 Représentation des polynômes

Dans le schéma de chiffrement homomorphe FV, l'arithmétique des polynômes se fait modulo un entier q. Compte tenu du caractère aléatoire de la génération des polynômes, les coefficients ont une très grande probabilité d'occuper un espace mémoire équivalent à l'espace requis pour stocker le nombre $q-1$.

Nous avons donc décidé de représenter les polynômes comme une suite de coefficients de taille $q-1$ de manière contiguë comme présenté figure 2.5. Cette représentation permet également d'implémenter efficacement l'addition et la soustraction de polynômes comme nous le verrons section 2.6. Seule l'opération FV.Mult décrite équation 1.8 peut présenter un certain problème car elle demande une multiplication de polynômes sans réduction modulaire des coefficients. Cependant, cette opération sera implémentée sur le composant matériel et donc n'influence pas la représentation

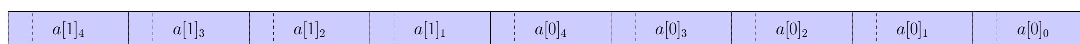


FIGURE 2.5 – Représentation des polynômes.

TABLE 2.2 – Exemples d'implémentations existantes d'opérateurs vectoriels sur processeurs.

Extension du jeu d'instructions	Processeur	Taille du vecteur	Taille des opérandes entiers
NEON	ARM	128 bits	8, 16, 32 bits + 64 bits à partir de ARMv8
SSE/AVX AVX2 AVX-512	Intel/AMD	128 bits 256 bits 512 bits	8, 16, 32, 64 bits

logicielle des polynômes.

2.6 Présentation de la programmation vectorielle

Comme nous l'avons vu, les polynômes sont représentés comme une suite de chiffres de manière contiguë. Compte tenu de la représentation des nombres adoptée, les opérations d'addition et de soustraction peuvent se faire de manière totalement parallèle. Cette représentation est très bien adaptée à la programmation vectorielle. Cette dernière permet en effet de réaliser des opérations sur un vecteur d'opérandes au lieu d'un opérande à la fois. Actuellement, selon la famille et la gamme du processeur, les vecteurs peuvent avoir une taille de 128, 256 ou 512 bits. La table 2.2 présente les différentes implémentations de la programmation vectorielle. Pour des processeurs récents grand public (depuis 2011), l'extension du jeu d'instructions le plus commun est l'AVX2. La programmation vectorielle permet de faire des opérations arithmétiques élémentaires (addition, soustraction, multiplication, comparaison, masquage,...) sur des nombres entiers. Un nombre plus restreint d'opérations est également possible sur des nombres à virgule flottante. Dans le cas des nombres entiers, il y a une certaine flexibilité dans le choix de la taille des opérandes comme en témoigne la table 2.2. Son avantage supplémentaire est d'être implémenté sur de nombreux types de processeurs, qu'ils soient pour ordinateurs généralistes (Intel/AMD), ou pour les systèmes embarqués (ARM).

La figure 2.6 donne un exemple d'addition de deux vecteurs de 128 bits contenant 4 entiers de 32 bits avec la programmation vectorielle. L'addition se fait terme-à-terme, en parallèle, et sans propagation de retenue entre les termes. Cette opération demande trois sous-opérations :

— Chargement des deux vecteurs dans deux registres depuis le cache.

— Addition des registres dans un troisième.
— Copie du registre résultant dans la mémoire.

Lorsque ce calcul est répété dans une boucle, ces sous-opérations peuvent s'exécuter en parallèle en utilisant le pipeline du processeur.

Nous pouvons formuler quelques remarques supplémentaires. Les données que l'on charge dans le vecteur doivent être contigües en mémoire. Ceci est à mettre en relation directe avec la gestion du cache d'un processeur fonctionnant par lignes contenant plusieurs données à la fois (typiquement 256 ou 512 bits). Pour augmenter l'efficacité des opérations, il est également conseillé (mais non requis) que l'adresse mémoire des données que l'on charge dans un vecteur soit alignée sur la taille des vecteurs eux-mêmes (donc alignée sur 256 bits pour l'AVX2). Le dernier point important, voire très important, est la non gestion de la propagation de retenue entre les éléments des vecteurs. Elle doit donc être gérée manuellement.

La figure 2.7 présente une version complète de la représentation des polynômes choisie et son utilisation dans la programmation vectorielle (jeu d'instructions AVX2). Le polynôme représenté est de degré 3071, avec des coefficients représentés sur cinq espaces mémoires de 32 bits. La base β utilisée ici est 2^{27} (base qui sera réutilisée dans notre implémentation car correspondent à la taille des unités arithmétiques qui seront implémentées sur le composant matériel), et permet dans cet exemple de représenter des nombres sur 140 bits. Pour l'exemple, le chiffre représenté n'occupe que 125 bits sur les 140 bits disponibles (d'où le fait que le dernier chiffre n'occupe que 17 bits).

Cette représentation permet de faire des opérations sur huit chiffres en parallèle grâce à l'extension du jeu d'instructions AVX2. Dans le cas où le nombre de chiffres de nos polynômes n'est pas un multiple de huit, il faut faire attention à la toute dernière opération et masquer les chiffres qui ne sont pas dans l'espace alloué au polynôme. Dans cet exemple, le problème n'est pas apparent car le nombre de chiffres est divisible par huit.

Pour information, cet exemple est extrait de la table 1.3 des paramètres de sécurité de FV, et correspond à une profondeur multiplicative de quatre, pour $\omega = 27$ bits et $\lambda = 80$ bits. Il correspond également à un cas étudié dans notre architecture, et nous aurons l'occasion d'en discuter au chapitre 3 lors de la présentation de l'accélérateur matériel.

Bien que présenté pour le jeu d'instructions AVX2, cette approche reste semblable pour les autres jeux d'instructions à quelques corrections près.

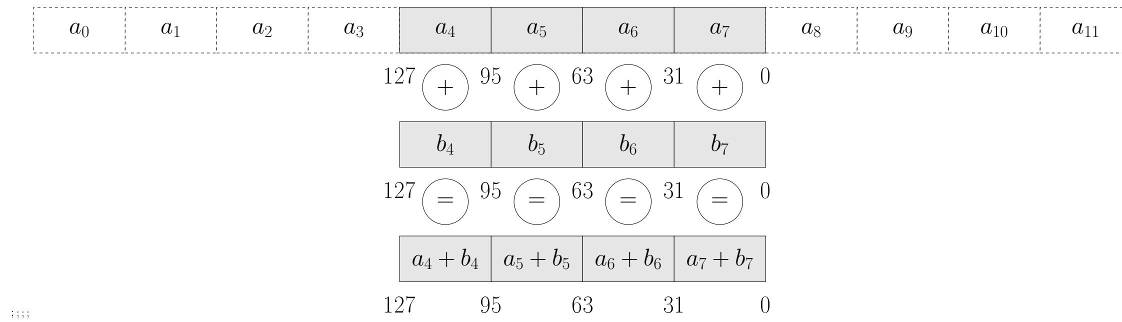


FIGURE 2.6 – Exemple d'une opération d'addition 32 bits de deux vecteurs de 128 bits avec la programmation vectorielle.

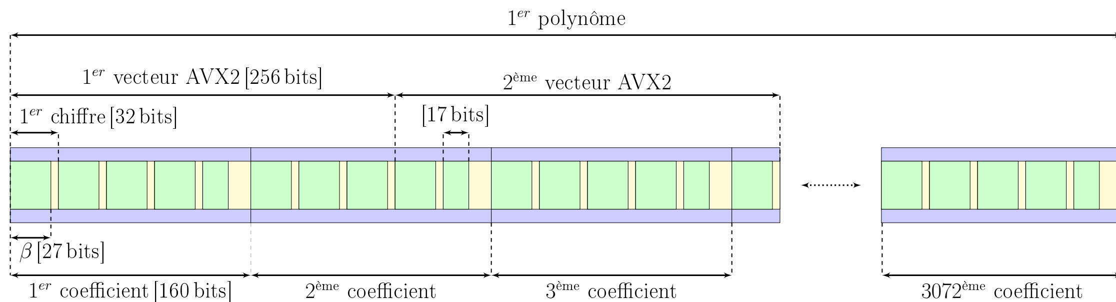


FIGURE 2.7 – Représentation d'un polynôme de 3072 coefficients sur 140 bits en mémoire, optimisé pour la programmation vectorielle.

2.7 Opérations arithmétiques

2.7.1 Addition/soustraction polynomiale

En reprenant l'étude sur la représentation des polynômes en mémoire, l'implémentation de l'addition et de la soustraction de polynômes est triviale. Il s'agit d'une boucle permettant le parcours de tout le polynôme. Le pas de la boucle dépend juste de la taille du vecteur de la programmation vectorielle, 4 pour un vecteur de 128 bits, 8 pour un vecteur de 256 bits et 16 pour un vecteur de 512 bits.

2.7.2 Réduction modulaire polynomiale

Contrairement à l'approche standard qui utilise l'algorithme de réduction de Barrett [Bar86], nous avons décidé d'utiliser une démarche plus exploratoire. Cette démarche est motivée par le fait que, comme la réduction modulaire polynomiale se fait sur des polynômes cyclotomiques, il est possible de tester différents candidats pour minimiser le temps de calcul de cette opération.

Nous ne détaillerons ici que la méthode employée pour le calcul de la réduction modulaire polynomiale. Des informations plus précises sur le travail exploratoire et

le choix des meilleurs polynômes cyclotomiques sont proposés section 4.1.

Prenons tout d'abord un cas simple, à savoir la réduction modulaire polynomiale par X^n+1. Pour se faire, il suffit de découper le polynôme A en deux sous polynômes A_L et A_H tels que $A = A_L + A_H \cdot X^n$ (A_L étant constitué des n premiers coefficients de A, et A_H les suivants).

En revenant à la définition de la réduction modulaire polynomiale, on obtient :

$$A = A_L + A_H \cdot X^n = B \cdot (X^n + 1) + R, \text{ avec } \deg(R) < n \tag{2.10}$$

Il suffit donc de résoudre le système d'équations :

$$\begin{cases} A_H = B \\ A_L = B + R \end{cases} \tag{2.11}$$

ce qui revient à calculer $A_L - A_H$.

Cette approche peut être généralisée pour n'importe quelle réduction modulaire polynomiale. Dans le cas des polynômes cyclotomiques, cette approche est considérablement simplifiée grâce à leur structure particulière. Ainsi, une vaste majorité de polynômes cyclotomiques ont les propriétés suivantes :

1) leurs coefficients appartiennent à $\{-1, 0, 1\}$;
2) ils possèdent un grand nombre de coefficients nuls.

Pour de tels polynômes cyclotomiques, leur écriture se résume alors à :

$$\Phi = \sum_{i=0}^{m} \alpha_i \cdot X^{\beta i}, \text{ avec } \begin{cases} \beta m = n \\ \alpha_i \in \{-1, 0, 1\} \end{cases} \tag{2.12}$$

Déterminons maintenant la version généralisée de la réduction modulaire polynomiale dans le cas des polynômes cyclotomiques. Soient A le polynôme que l'on souhaite réduire, B le polynôme quotient et R le polynôme reste. On note :

$$\begin{aligned} A &= \sum_{i=0}^{2 \cdot m-1} A_i \cdot X^{\beta i}, \text{ avec } \deg(A_i) < \beta \\ B &= \sum_{i=0}^{m-1} B_i \cdot X^{\beta i}, \text{ avec } \deg(B_i) < \beta \\ R &= \sum_{i=0}^{m-1} R_i \cdot X^{\beta i}, \text{ avec } \deg(R_i) < \beta \end{aligned} \tag{2.13}$$

La réduction modulaire polynomiale par un polynôme cyclotomique Φ s'écrit alors :

$$A = B \cdot \Phi + R, \text{ avec } \deg(R) < n \tag{2.14}$$

En utilisant l'algorithme de multiplication classique sur le produit $B \cdot \Phi$, on obtient :

$$\sum_{i=0}^{2 \cdot m-1} A_i \cdot X^{\beta i} = \sum_{i=0}^{2m} \sum_{j=0}^{i} B_j \cdot \alpha_{i-j} \cdot X^{\beta i} + R \tag{2.15}$$

En utilisant le fait que $\deg(R) < n$, on obtient le système d'équations suivant :

$$\begin{cases} \forall i \in \{m, 2m\},\ A_i = \sum_{j=0}^{i} B_j \cdot \alpha_{i-j} & \text{(a)} \\ \forall i \in \{0, m-1\},\ A_i = \sum_{j=0}^{i} B_j \cdot \alpha_{i-j} + R_i & \text{(b)} \end{cases} \tag{2.16}$$

Avec le système d'équations 2.16 (a), il est possible de déterminer les différents B_i, puis en les injectant dans le système d'équations 2.16 (b), nous pouvons calculer le polynôme R.

Finalement, le calcul de R revient à des opérations d'additions et de soustractions de polynômes de degré $\beta - 1$, pouvant efficacement être implémentées en logiciel. Pour simplifier le calcul des B_i, nous avons réalisé un script permettant l'inversion du système d'équations.

Cette réduction modulaire polynomiale est efficace lorsque le degré du polynôme A n'est pas trop grand devant Φ (typiquement $2n$). Pour des degrés de A supérieurs, le nombre de sous-additions et de sous-soustractions va considérablement augmenter rendant complexe ce calcul. Ce cas de figure peut arriver dans deux cas particuliers :

(1) après plusieurs multiplications successives de polynômes ;

(2) lorsque l'on souhaite récupérer le message d'un polynôme issu du *batching*.

Le point (1) est un cas très rare car la multiplication polynomiale étant un algorithme complexe, il est très certainement plus efficace de calculer la réduction modulaire polynomiale entre chaque opération de multiplication.

Pour le point (2), il faut en effet réduire le polynôme issu du *batching* par chacun des polynômes issus de la factorisation du polynôme cyclotomique dans $\mathbb{Z}_2$. Plus le nombre de *batches* est important, plus l'écart entre le polynôme à réduire et le polynôme réducteur est grand. Cependant, ceci n'est pas un problème dans notre cas car, comme les messages sont binaires, on sait que le résultat de la réduction modulaire polynomiale est binaire. Ainsi, seul le premier coefficient du polynôme réduit nous intéresse, et en plus, comme le résultat est calculé modulo 2, de nombreuses sous-additions et sous-soustractions ne sont pas à calculer.

2.7.3 *Batching*

Tout comme la réduction modulaire polynomiale, l'implémentation du *batching* doit être efficace pour que notre approche soit compétitive comparée aux versions actuelles optimisées (mais non compatibles avec le *batching*). Il est donc important

de déterminer quelles opérations peuvent être pré-calculées et quelles opérations sont réellement nécessaires afin d'accélérer le processus.

L'opération qui *batche* un vecteur de messages en un polynôme est en fait une opération qui peut très largement se simplifier.

Considérons Φ un polynôme cyclotomique, et $\boldsymbol{\varphi}$ un vecteur de polynômes contenant les facteurs de Φ dans $\mathbb{Z}_2[X]$ (k facteurs). Nous avons alors :

$$\prod_{i=1}^{k} \boldsymbol{\varphi}[i] \equiv \Phi \mod 2 \tag{2.17}$$

On note également $\boldsymbol{\Phi}$ le vecteur de polynômes vérifiant :

$$\boldsymbol{\Phi}[i] = \frac{\Phi}{\boldsymbol{\varphi}[i]} \tag{2.18}$$

Notons P le polynôme qui réalise le *batching* d'un vecteur de k messages binaires $\boldsymbol{M}$. Alors le calcul de P se résume par la formule :

$$P = \sum_{i=1}^{k} \boldsymbol{M}[i] \cdot \boldsymbol{S}[i] \cdot \boldsymbol{\Phi}[i] \mod \Phi \tag{2.19}$$

avec $\boldsymbol{S}$ un vecteur de polynômes vérifiant :

$$\boldsymbol{S}[i] \cdot \boldsymbol{\Phi}[i] \equiv 1 \mod \boldsymbol{\varphi}[i] \tag{2.20}$$

Dans l'équation 2.19, tous les polynômes à part $\boldsymbol{M}[i]$ sont des polynômes que l'on peut pré-calculer. Le vecteur $\boldsymbol{\varphi}$ (qui permet de calculer $\boldsymbol{\Phi}$) peut être pré-calculé grâce à l'algorithme de Cantor Zassenhaus [CZ81]. Le vecteur $\boldsymbol{S}$ quant-à-lui, peut être calculé grâce à l'équation 2.20. En pratique, il suffit de stocker le vecteur de polynômes $\boldsymbol{B}$, vecteur vérifiant $\boldsymbol{B}[i] = \boldsymbol{S}[i] \cdot \boldsymbol{\Phi}[i]$. L'équation de *batching* devient alors :

$$P = \sum_{i=1}^{k} \boldsymbol{M}[i] \cdot \boldsymbol{B}[i] \mod \Phi \tag{2.21}$$

Cette équation peut encore être simplifiée de la manière suivante :

(1) $\boldsymbol{M}[i]$ étant un nombre entier binaire, le produit $\boldsymbol{M}[i] \cdot \boldsymbol{B}[i]$ se réduit à $\boldsymbol{B}[i]$ si $\boldsymbol{M}[i] = 1$, 0 sinon ;

(2) la réduction modulaire polynomiale par Φ n'est pas nécessaire, car le degré du polynôme $\boldsymbol{M}[i] \cdot \boldsymbol{S}[i] \cdot \boldsymbol{\Phi}[i]$ est inférieur à Φ.

La démonstration du point (2) est simple :

$$\begin{aligned}\deg(\boldsymbol{M}[i]\cdot\boldsymbol{S}[i]\cdot\boldsymbol{\Phi}[i]) &= \deg\boldsymbol{M}[i]+\deg\boldsymbol{S}[i]+\deg\boldsymbol{\Phi}[i]\\&=\deg\boldsymbol{S}[i]+\deg\boldsymbol{\Phi}[i]\\&=\deg\boldsymbol{\varphi}[i]-1+\deg\boldsymbol{\Phi}[i]\\&=\deg\boldsymbol{\varphi}[i]-1+\deg\Phi-\deg\boldsymbol{\varphi}[i]\\&=\deg\Phi-1<\deg\Phi\end{aligned}\tag{2.22}$$

En conclusion, calculer le polynôme qui réalise le *batching* d'un vecteur de messages binaires revient à l'addition au maximum de k polynômes de degré $\deg\Phi-1$.

2.7.4 Le choix du module

Le choix du module q est un point important car il va influencer une bonne partie de l'arithmétique de FV. À l'heure actuelle, il n'existe aucune restriction particulière sur le module. Dans notre architecture les nombres étant représentés à l'aide d'une numération *simple de position* redondante, le choix d'un module comme une puissance de deux reste le plus pertinent. Ainsi, la réduction modulaire entière se réduit à une opération de troncature, celle-ci étant naturellement réalisée lorsque les nombres dépassent l'espace alloué.

2.8 Conclusion

Dans ce chapitre, nous avons proposé une représentation des polynômes de FV optimisée en logiciel. Celle-ci est basée sur une représentation *simple de position* redondante, et permet une implémentation efficace des additions et des soustractions de polynômes grâce à la programmation vectorielle. Nous avons également montré que les opérations de réduction modulaire polynomiale et de *batching*, potentiellement coûteuses, peuvent se réduire à des additions et des soustractions de polynômes, permettant également une implémentation efficace. Les résultats d'implémentation seront discutés chapitre 4.

Le prochain chapitre est consacré à présentation de notre approche de type co-conception logicielle/matérielle afin d'accélérer la multiplication polynomiale, FV.Mult, FV.Relin et FV.Encrypt.

3

Co-conception logicielle/matérielle pour l'accélération de l'opération de chiffrement et de multiplication homomorphe de FV

Dans ce chapitre, nous allons tout d'abord présenter une architecture de base pour l'accélération de la multiplication polynomiale. Cette accélération est basée sur l'algorithme de Karatsuba et suit une approche de type co-conception, où les calculs sont partagés entre l'application logicielle et un accélérateur matériel. Dans une seconde partie, nous adapterons cette architecture aux spécificités du schéma de chiffrement homomorphe FV, notamment les fonctions FV.Encrypt, FV.Mult et FV.Relin présentées section 1.3.2.

3.1 Conception d'un accélérateur de la multiplication polynomiale par l'utilisation de l'algorithme de Karatsuba

Dans cette section, nous présentons l'implémentation d'un accélérateur de la multiplication polynomiale utilisant l'algorithme de Karatsuba via une approche de co-conception. L'arithmétique du schéma de chiffrement homomorphe FV utilisant des polynômes dans $\mathbb{Z}_q(X)/f(X)$, il faudra adjoindre à cette multiplication polynomiale une réduction modulaire polynomiale comme présentée section 2.7.2. Pour l'ensemble de cette section, les notations suivantes sont utilisées :

- A et B représentent les polynômes à multiplier ;
- Le polynôme P représente indifféremment l'un des polynômes A ou B. En particulier, comme les pré-traitements de Karatsuba doivent s'effectuer indifféremment sur A et B, nous utiliserons P pour présenter les différents algorithmes.

3.1.1 Présentation de l'architecture

Nous avons vu dans le chapitre 2 que l'addition et la soustraction de polynômes peuvent être efficacement implémentées de manière logicielle grâce à la programmation vectorielle. Dans le cas particulier de l'algorithme de Karatsuba, cela permet en outre de calculer efficacement les pré- et post-récursions. En revanche, l'implémentation complète de Karatsuba en logiciel pose certaines limites. Avec p le nombre de récursions, le nombre de coefficients par sous-polynôme évolue en $1/2^p$ et le nombre de sous-polynômes en 3^p. On se retrouve ainsi à calculer des opérations sur de nombreux polynômes de degrés de plus en plus petits. Ces polynômes étant à des adresses mémoires différentes, le pipeline du processeur généraliste se trouve rompu régulièrement.

Une alternative possible consiste à faire calculer par un accélérateur matériel les différentes opérations de Karatsuba peu efficaces sur un processeur généraliste. À ce titre, notre implémentation se base sur une approche de co-conception, avec un accélérateur matériel accélérant spécifiquement certaines tâches de l'algorithme de Karatsuba. Ce dernier est en particulier responsable du calcul des dernières récursions de l'algorithme de Karatsuba ainsi que toutes les multiplications entières des sous-produits. Avec un bon découpage des opérations entre le logiciel et le composant matériel, cette approche permet d'accélérer significativement le temps de traitement de l'algorithme. Cependant, l'ajout d'un composant matériel en vu d'une accélération amène obligatoirement différentes contraintes :

(1) la présence d'un lien de communication pour le transfert des données entre l'application logicielle et le composant matériel (limitation du débit) ;
(2) le composant matériel lui-même a une limitation en fréquence de fonctionnement, qui peut être inférieure voire très inférieure au processeur généraliste selon le composant utilisé (il peut très bien y avoir un facteur dix entre les deux fréquences de fonctionnement).

La contrainte (2) est assez forte car elle demande que le parallélisme de l'algorithme soit suffisamment significatif pour contrebalancer les différences entre les fréquences de fonctionnement des deux composants. De plus, d'après la contrainte (1), le lien de communication transférant les données à l'accélérateur matériel de manière séquentielle, il faut que l'algorithme puisse être parallélisé efficacement après le transfert des données. Deux exemples d'opérations peu efficaces avec une telle architecture sont l'addition et la soustraction de polynômes. Du coté processeur, ce dernier est capable avec l'utilisation de l'AVX2 de calculer huit additions ou soustractions d'entiers de 32 bits en parallèle à une fréquence de l'ordre du GHz. Dans un cadre optimal, cette fréquence permet de traiter jusqu'à 89 Go de données par seconde. Comparé au bus PCI-E (bus de communication standard entre le processeur et les périphériques matériels) qui dans un cadre optimal permet une bande passante de 32 Go par seconde, le transfert lui-même serait déjà plus lent que le temps de calcul du processeur. Plus d'informations sur le bus PCI-E et son utilisation avec un périphérique matériel seront données section 3.1.4.

En conclusion, l'accélérateur matériel a été dédié à l'implémentation de la complexe opération de multiplication polynomiale. La figure 3.1 présente la stratégie d'implémentation de l'algorithme de multiplication basé sur Karatsuba dans une approche de co-conception. Le découpage des opérations, et notamment la répartition des pré- et post-traitements, sera étudiée plus en détail dans la section 3.1.6. L'exemple de la figure 3.1 présente l'accélération de la multiplication polynomiale de polynômes avec 3072 coefficients sur 127 bits chacun, mais l'approche reste la même pour des polynômes de tailles différentes. Pour améliorer la compréhension, l'évolution de la taille et du nombre de sous-polynômes entre les différentes étapes de l'algorithme de Karatsuba est présentée entre chaque bloc. 6 pré-récursions logicielles sont d'abord appliquées aux deux polynômes à multiplier A et B. Les sous-polynômes résultants sont ensuite combinés dans un unique espace mémoire avant de procéder à l'envoi de l'ensemble des sous-polynômes à l'accélérateur. Pendant le transfert des sous-polynômes, l'accélérateur matériel va procéder de manière pipelinée au calcul des dernières pré-récursions, à la multiplication des sous-produits et au calcul des premières post-récursions. Après les calculs, les données résultantes vont être envoyées à l'application logicielle, qui aura alors pour tâche de terminer les derniers post-traitements de Karatsuba, concluant ainsi l'opération de multiplication polynomiale.

3.1.2 Présentation de la plateforme retenue pour l'accélérateur matériel

Pour la conception d'un accélérateur matériel, il existe deux grandes familles : les FPGA et les ASIC.

Les FPGA s'appuient sur une architecture très flexible permettant une reconfiguration des connexions entre les éléments logiques et les éléments mémoires à volonté (sauf certains FPGA spécifiques, type anti-fusible, qui ne permettent qu'une seule configuration). Pour ce faire, les FPGA possèdent une matrice de configuration appelée routage qui permet de configurer toutes les connexions implémentées sur le composant. Cette flexibilité a un prix, car une grande partie des ressources est dédiée au routage exclusivement. De plus, l'utilisation finale du composant pour une opération donnée est inférieure aux ressources disponibles, car il faut être en mesure de pouvoir router les différents éléments entre eux, et ce à une fréquence de fonctionnement acceptable.

Les ASIC sont des composants dont l'architecture est fixe et optimisée. Comme l'architecture n'est pas flexible, l'utilisation des ressources est au plus près des besoins réels de l'application, et les connexions entre les éléments logiques et de mémorisation peuvent être fortement réduites par rapport aux FPGA, augmentant la fréquence de fonctionnement. Cependant, le cycle de conception d'un ASIC est long (des spécifications à la mise en fabrication), et très coûteux pour des petites séries de composants.

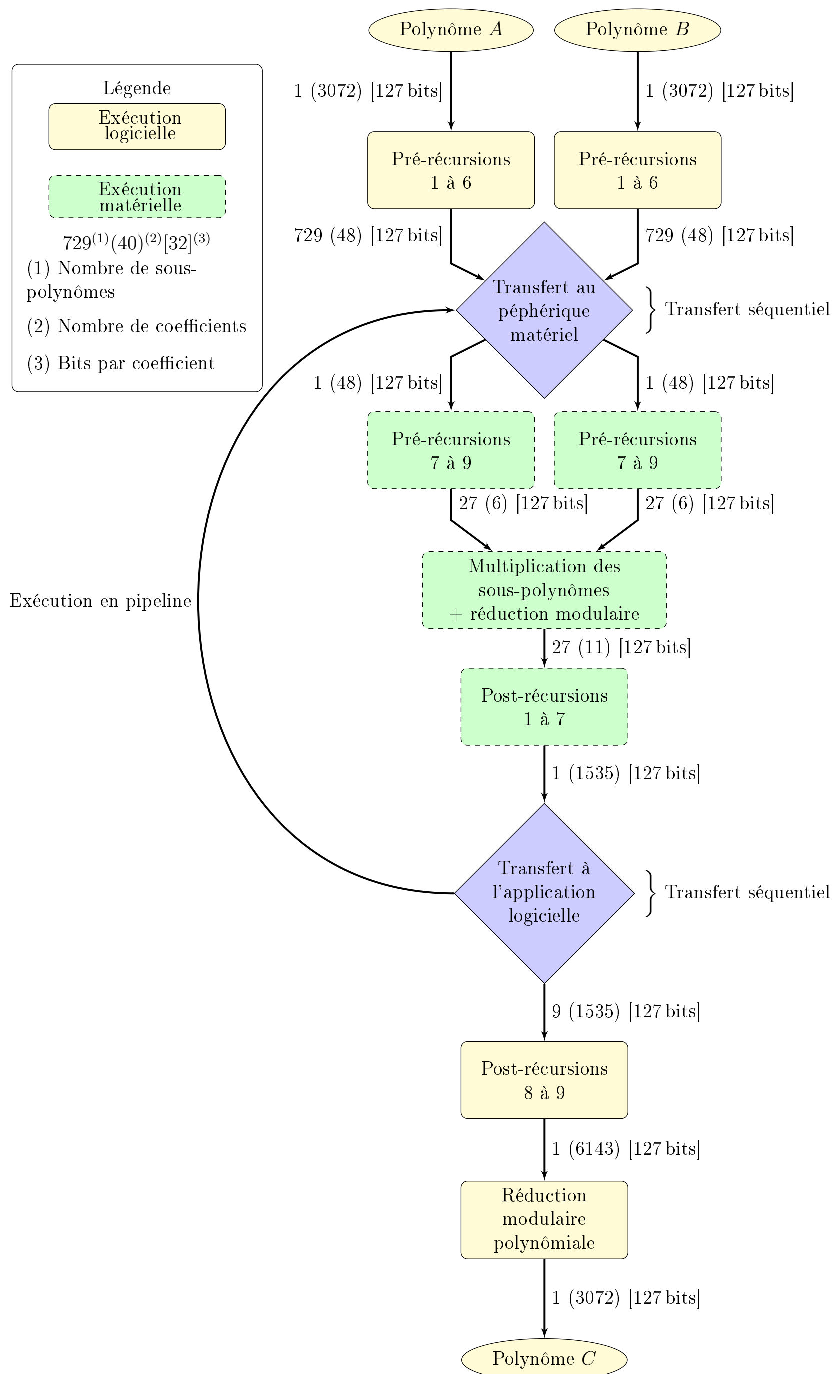


FIGURE 3.1 – Stratégie d'implémentation de l'accélération de la multiplication polynomiale avec Karatsuba dans une approche de co-conception. L'opération effectuée est $A \cdot B \mod \Phi = C$, où Φ est un polynôme cyclotomique. L'architecture est présentée pour des polynômes avec 3072 coefficients sur 127 bits.

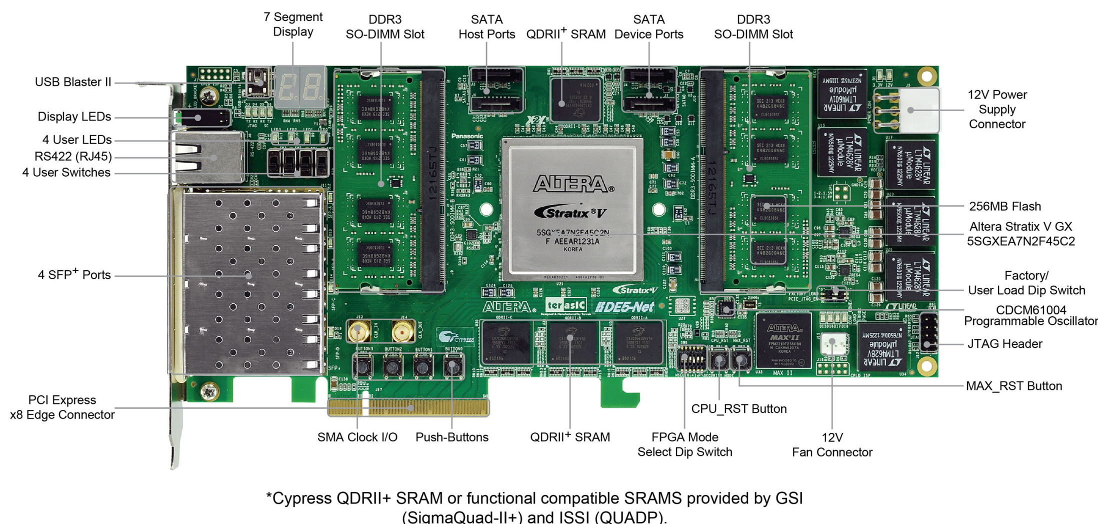


FIGURE 3.2 – Présentation de la plateforme DE5-net, possédant notamment un puissant FPGA Stratix V GX de chez Altera.
Source : `http://www.terasic.com.tw/`

C'est pourquoi nous nous sommes principalement intéressés aux FPGA, permettant un travail exploratoire plus simple (grâce à leur flexibilité) et rapide (grâce à un cycle de conception réduit), tout en ayant un coût financier bien inférieur aux ASIC.

Le chiffrement homomorphe possédant des opérandes pouvant atteindre le million de bits, nous nous sommes tournés vers des plateformes possédant des FPGA de grande capacité. Pour la qualité de son FPGA et sa connectique, nous avons effectué nos expérimentations sur la plateforme DE5-net de Terasic, présentée dans la figure 3.2. Elle embarque un FPGA Altera Stratix V GX pourvu de 622 000 éléments logiques, 6 Mo de mémoire embarquée, 8 Go de mémoire externe type DDR3 et 256 Mo de mémoire flash. Elle possède également des connectiques pour les bus RS422, SFP+ et PCI-E. Pour les opérations de multiplication, la plateforme possède 256 DSPs permettant la multiplication d'entiers 27×27 bits. Cette particularité sera exploitée pour l'implémentation de l'accélérateur.

3.1.3 Présentation du bus PCI-E

Comme nous l'avons vu dans la section 3.1.1, la communication entre le processeur et le périphérique matériel est un point essentiel dans une architecture de type co-conception. Dans notre application, pour des questions de débit, la liaison s'effectue par le biais du bus PCI-E. C'est l'interface standard dans les ordinateurs généralistes, qui regroupe un certain nombre de périphériques (carte graphique, carte réseau, carte son,...). La figure 3.3 présente l'architecture d'un bus PCI-E. Le *Root Complex* est une unité permettant d'interfacer la mémoire et le processeur (CPU) au

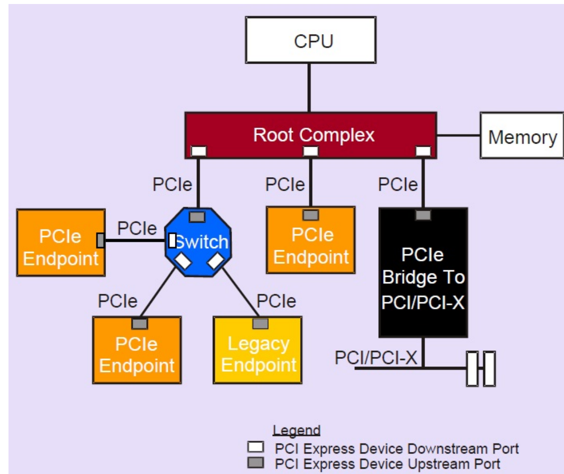


FIGURE 3.3 – Architecture du bus PCI-E comme implémenté dans un ordinateur généraliste.
Source : `https://e2e.ti.com/support/dsp/davinci_digital_media_processors/f/100/t/465737`

bus PCI-E. Certains périphériques, comme la carte graphique, possèdent un accès privilégié au *Root Complex* via une connexion directe. Pour les autres périphériques, un switch permet de partager l'accès au *Root Complex*. Il existe deux types d'unités sur le PCI-E :

1) les PCI-E *endpoints* : périphériques qui se connectent au PCI-E de manière standard. Ils peuvent initier une communication (*requester*), ou répondre à une communication (*completer*) ;
1) les PCI-E *legacy endpoint* : utilisent le bus de communication, mais ont plus de souplesse dans leur fonctionnement. Ils assurent notamment la compatibilité avec d'anciens périphériques PCI-E.

Le PCI-E est un bus FULL-DUPLEX (communication en émission et réception simultanément), fonctionnant sur plusieurs lignes en parallèle. Chaque ligne permet l'échange d'un bit d'information par coup d'horloge du bus. Il existe actuellement 6 variantes du nombre de lignes, $\times 1$, $\times 2$, $\times 4$, $\times 8$, $\times 16$ et $\times 32$. Plus le nombre de lignes est important, plus le débit maximal est élevé. Les cartes graphiques, demandant une grande bande passante, sont généralement implémentées sur seize lignes. Notre plateforme de test se positionne sur huit lignes à une fréquence de 8 GHz, permettant un débit théorique de 8 Go/s. Il existe actuellement trois générations de PCI-E, avec une quatrième version prévue pour 2017. Chaque génération a pour principale évolution l'augmentation du débit théorique. La table 3.1 présente les informations principales des différentes générations, ainsi que le débit théorique. Les FPGAs étant des composants par nature très flexibles, deux stratégies peuvent être adoptées pour gérer les données provenant du PCI-E : jouer sur la fréquence de fonctionnement du composant, ou jouer sur le parallélisme. La table 3.2 explicite les configurations implémentées sur les FPGA de chez Altera (famille Stratix V). Ces configurations sont proches chez les autres fabricants. On peut remarquer que certaines configurations

TABLE 3.1 – Comparaison des différentes générations de PCI-E.
Source : `http://pcisig.com/`

	PCI-E gen 1.X	PCI-E gen 2.X	PCI-E gen 3.X	PCI-E gen 4.X
Puissance Maximale délivrée (seize lignes)	75 W	75 W	75 W	à définir
Bande passante	2.5 GT/s	5 GT/s	8 GT/s	16 GT/s
Débit par ligne	250 Mo/s	500 Mo/s	984.6 Mo/s	1969.2 Mo/s
Débit pour huit lignes	2 Go/s	4 Go/s	7.88 Go/s	15.75 Go/s

permettent de choisir entre plus de parallélisme au prix d'une fréquence moindre, et moins de parallélisme mais avec une fréquence de fonctionnement plus élevée. Bien que en terme de débit ces deux possibilités soient équivalentes, il n'est pas forcément aisé d'atteindre une fréquence de 250 MHz dans une implémentation réelle. Ainsi, pouvoir réduire la fréquence de fonctionnement est utile. Nous pouvons remarquer que les configurations présentées table 3.2 permettent au maximum une utilisation d'accélérateurs sur une largeur de bus de 256 bits avec une fréquence de fonctionnement de 250 MHz. Il faut cependant garder à l'esprit qu'il est probable que le bus ne soit pas saturé par notre application et donc que les données soient transmises avec des latences. Ces latences sont d'ailleurs amplifiées par le fait que :

1) le PCI-E utilise un codage redondant pour l'envoi des données sur le bus ;
2) d'autres composants peuvent être connectés sur le bus PCI-E ;
3) l'horloge du bus doit se stabiliser avant les transferts sur celui-ci, ce qui peut prendre plusieurs dizaines de milliers de cycles.

Toutes ces limitations peuvent amener à une utilisation réelle inférieure, voire bien inférieure, à la théorie. Une étude plus complète de ces limitations sera proposée section 4.2.1.

3.1.4 Présentation de RIFFA

RIFFA (*Reusable Integration Framework for FPGA Accelerators*) [JRHK15], est un framework permettant l'utilisation du bus PCI-E dans des applications à base de FPGA. Il permet l'intégration de FPGA de chez Xilinx et Altera, et fonctionne à la fois sur Linux et Windows. Pour l'interface logicielle, RIFFA possède des portages vers le C/C++, python, Matlab et Java. Le framework RIFFA fourni :

- une API logicielle de haut niveau ;

Table 3.2 – Configurations possibles des FPGA de chez Altera (famille Stratix V) pour une interface avec le PCI-E.

Lignes	PCI-E gen 1.X	PCI-E gen 2.X	PCI-E gen 3.X
1	64 bits, 125 MHz	64 bits, 125 MHz	64 bits, 125 MHz
2	64 bits, 125 MHz	64 bits, 125 MHz	64 bits, 250 MHz 128 bits, 125 MHz
4	64 bits, 125 MHz	64 bits, 250 MHz 128 bits, 125 MHz	128 bits, 250 MHz 256 bits, 125 MHz
8	64 bits, 250 MHz 128 bits, 125 MHz	128 bits, 250 MHz 256 bits, 125 MHz	256 bits, 250 MHz

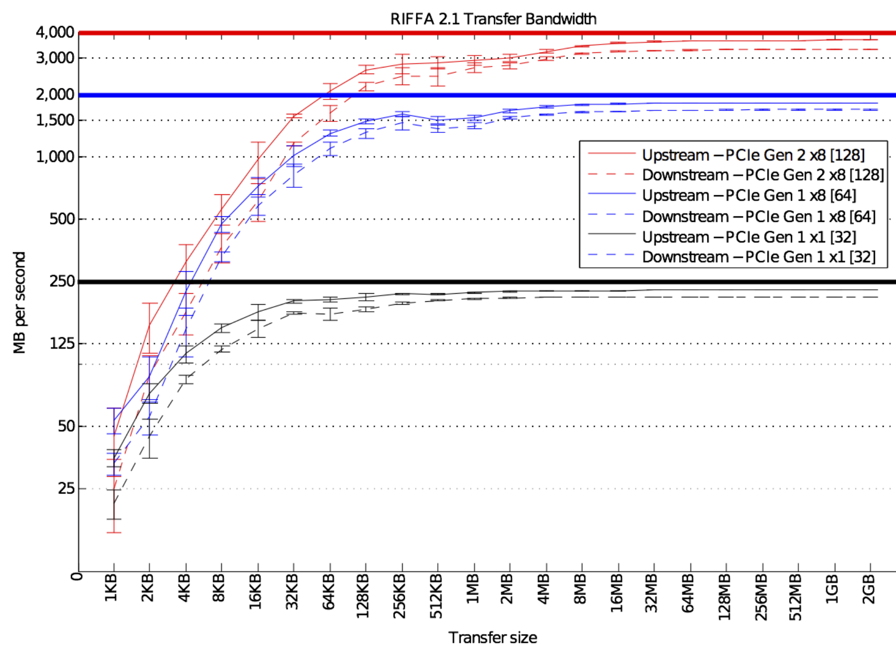


Figure 3.4 – Performances de RIFFA d'après la documentation [JRHK15].

- un *driver* PCI-E ;
- des fichiers sources pour son intégration sur le FPGA.

Il ne reste plus qu'au concepteur à implémenter son propre composant sur le FPGA et à l'interfacer avec RIFFA. Cette interface est constituée de deux bus de communications, un bus de lecture et un bus d'écriture (pour plus d'information, le lecteur peut se référer à [JRHK]).

RIFFA implémente un nombre restreint de configurations de l'interface PCI-E/FPGA (configurations présentées table 3.2), à savoir des transferts de 32, 64 et 128 bits à une fréquence de 250 MHz. En particulier, RIFFA n'implémente pas actuellement la configuration la plus performante, à savoir des transferts de 256 bits à 250 MHz.

La figure 3.4 présente les performances de RIFFA extraites de sa documentation [JRHK15]. Les auteurs ne fournissent pas d'informations détaillées sur la machine de test, à par le processeur (processeur six cœurs Intel I7 cadencés à 3.6 GHz). L'abscisse représente la taille du transfert à effectuer, l'ordonnée le débit sur le bus atteint, et les barres horizontales le débit théorique pour chaque configuration implémentée. Pour des grands volumes de données, RIFFA est capable de saturer le bus PCI-E.

Grâce à son intégration dans de nombreux FPGA et à sa simplicité apparente d'utilisation, nous avons utilisé RIFFA pour interfacer notre accélérateur matériel à l'application logicielle.

3.1.5 Implémentation logicielle de Karatsuba

L'implémentation logicielle des primitives de l'algorithme de Karatsuba se résume au calcul des pré- et post-traitements.

3.1.5.1 Les pré-traitements

Pour rappel, les pré-traitements de Karatsuba sont constitués de simples additions de polynômes. Pour simplifier la lecture, nous recopions ci-dessous l'équation 1.20 de Karatsuba, en encadrant les opérations à effectuer en pré-traitement :

$$\begin{aligned} A &= A_L + X^{n/2} \cdot A_H, B = B_L + X^{n/2} \cdot B_H \\ C &= A_L B_L \\ &\quad + \big((A_L \boxed{+} A_H) \cdot (B_L \boxed{+} B_H) - A_L B_L - A_H B_H\big) \cdot X^{\lceil n/2 \rceil} \\ &\quad + A_H B_H \cdot X^{2\lceil n/2 \rceil} \end{aligned} \tag{3.1}$$

Chaque pré-récursion de Karatsuba demande le calcul de $A_L + A_H$ et $B_L + B_H$, les polynômes A_L, A_H, B_L et B_H n'étant que des copies d'une sous-partie des polynômes A et B. La figure 3.5 présente l'arborescence des opérations dans l'algorithme de

Karatsuba pour trois pré-récursions. Les polynômes sont représentés par un couple d'indices i et j ($P_{i,j}$), où i représente le niveau de récursion et j un indice unique identifiant les sous-polynômes pour une récursion donnée. Pour les différentes branches, la branche L (resp. H) correspond au sous-polynôme constitué des coefficients de poids faibles (resp. de poids forts) du polynôme complet. La branche $L + H$ correspond à la somme des deux sous-polynômes des branches L et H. Le polynôme sans indice est le polynôme initial. Les sous-polynômes qui représentent une sous-partie d'un autre polynôme sont entourés d'un cercle en pointillé. À partir de cette représentation, il est relativement aisé de définir une stratégie d'implémentation des pré-traitements.

Tout d'abord, la structure en arbre de l'algorithme incite à utiliser une fonction récursive, cette dernière appelant trois instances d'elle-même pour les trois branches de sous-polynômes.

Ensuite, seuls les polynômes des branches $L + H$ sont des nouveaux polynômes, les branches L et H étant des copies du polynôme complet. Ainsi, une grande partie de l'allocation mémoire peut être sauvée en passant en paramètre une référence au polynôme complet plutôt que de faire une copie. Les sous-polynômes des branches $L + H$ sont stockés en mémoire pendant toute la durée des pré-traitements car ils seront réutilisés par les pré-récursions suivantes. Pour donner un exemple, le polynôme $P_{1,2}$ est réutilisé pour le calcul de $P_{2,6}$, $P_{2,7}$, $P_{2,8}$, $P_{3,18}$, $P_{3,19}$, $P_{3,20}$,$P_{3,21}$, $P_{3,22}$, $P_{3,23}$.

Pour des raisons d'efficacité, chaque sous-polynôme demandant un stockage est pré-alloué, et l'espace mémoire sera réutilisé pour les pré-traitements de futurs polynômes. Au final, pour des polynômes de n coefficients, il faut stocker un polynôme de $n/2$ coefficients pour la première récursion, trois polynômes de $n/4$ coefficients pour la seconde récursion, et ainsi de suite. Pour p récursions, le nombre total de coefficients à stocker peut alors s'écrire :

$$\begin{aligned}\sum_{i=1}^{p} \frac{n}{2^i} \cdot 3^{i-1} &= \sum_{i=1}^{p} \frac{n}{3} \cdot \left(\frac{3}{2}\right)^i \\ &= \frac{n}{3} \cdot \frac{1-\left(\frac{3}{2}\right)^{p+1}}{1-\frac{3}{2}} - \frac{n}{3} \\ &= \left(\left(\frac{3}{2}\right)^p - 1\right) \cdot n\end{aligned} \tag{3.2}$$

Il est intéressant de noter que tous les sous-polynômes de la dernière pré-récursion sont pré-alloués de manière contigüe sous forme d'une mémoire tampon pour faire l'interface entre le *driver* et le bus PCI-E. Le stockage de tous ces sous-polynômes demande $\left(\frac{3}{2}\right)^p \cdot n$ coefficients, soit presque autant que ce qui est nécessaire pour l'algorithme.

De la même façon, nous pouvons déterminer le nombre d'opérations d'addition de coefficients à réaliser pour obtenir ce résultat. Il est en fait égal au nombre

Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe Vincent Migliore 2017

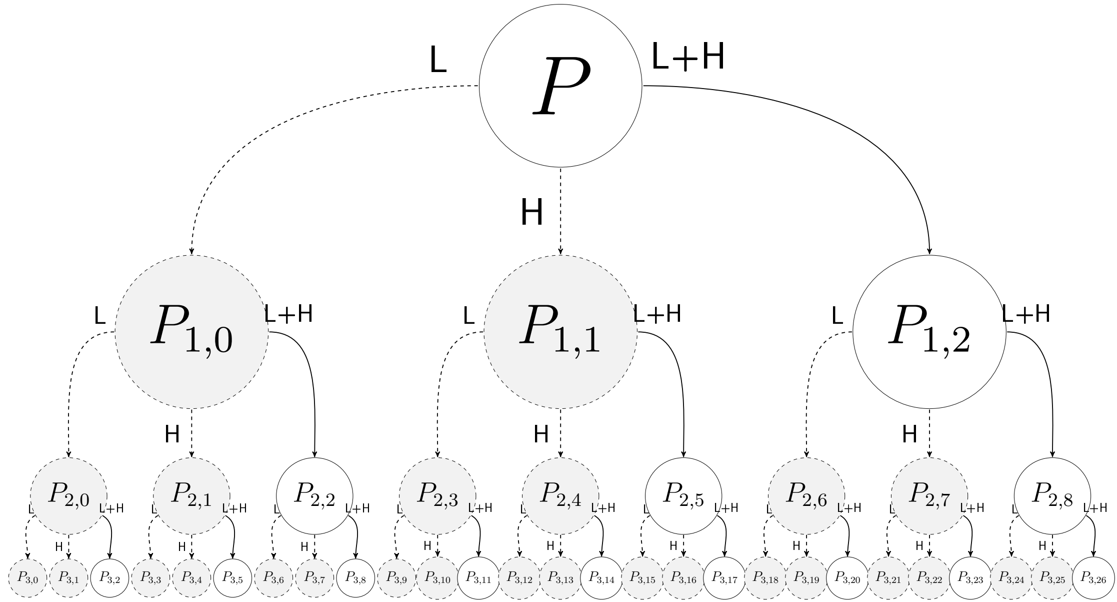


FIGURE 3.5 – Présentation de l'arborescence des opérations dans l'algorithme de Karatsuba pour trois pré-récursions.

de coefficients alloués, à savoir le résultat fourni par l'équation 3.2, car seuls les polynômes issus d'additions sont stockés.

Pour l'addition des sous-polynômes, il faut garder en mémoire le fait que le choix de la représentation des nombres (section 2.7.1) demande une propagation de retenue au bout d'un nombre donné d'additions. Toujours d'après la figure 3.5, on remarque que c'est la branche de droite (branche des chemins de type $L + H$) qui contient le plus d'additions successives. De plus, cette addition se fait toujours au pire cas car comme on additionne une sous-partie du polynôme avec une autre sous-partie du même polynôme, le nombre de bits de gardes consommés pour chaque coefficient est au pire cas le même. Cela induit, pour chaque pré-récursion, la consommation d'un *bit de garde* au pire cas. Ainsi, après p récursions, la consommation est de p bits de garde au pire cas. Cependant, la propagation de retenue est simple car elle n'est à effectuer que sur un nombre restreint de sous-polynômes.

3.1.5.2 Les post-traitements

De la même manière que les pré-traitements, nous recopions ci-dessous l'équation 1.20 de Karatsuba, en encadrant les opérations à effectuer en post-traitement :

$$\begin{aligned}
&A = A_L + X^{n/2} \cdot A_H, B = B_L + X^{n/2} \cdot B_H \\
&C = A_L B_L \\
&\quad \boxed{+}\big((A_L + A_H) \cdot (B_L + B_H)\boxed{-}A_L B_L\boxed{-}A_H B_H\big) \cdot X^{\lceil n/2 \rceil} \\
&\quad \boxed{+}A_H B_H \cdot X^{2\lceil n/2 \rceil}
\end{aligned} \tag{3.3}$$

Pour bien comprendre la particularité des post-traitements, la figure 3.6 présente de manière schématique les opérations à effectuer. Le résultat final est obtenu en additionnant les coefficients de manière verticale. Une post-récursion est tout d'abord composée du calcul du sous-polynôme central $(A_L + A_H) \cdot (B_L + B_H) - A_L \cdot B_L - A_H \cdot B_H$, qui demande donc deux soustractions, puis est composé de deux additions de sous-polynômes à cause de la superposition des sous-polynômes $A_L \cdot B_L$ et $A_H \cdot B_H$ avec le sous-polynôme central. L'algorithme demande le pré-calcul des différents indices pour réaliser le bon alignement des coefficients des sous-polynômes. Pour l'implémentation en tant que telle, les post-traitements suivant une arborescence similaire aux pré-traitements, l'utilisation de fonctions récursives a également été adoptée.

En ce qui concerne le choix du moment le plus judicieux pour effectuer la propagation de retenue, les post-traitements de Karatsuba demandent plus d'attention. Ceci est accentué en particulier car un coefficient peut être le résultat d'une ou plusieurs additions ou soustractions, avec des coefficients possédant un nombre de *bits de garde* variable. De plus, l'opération de soustraction réduit de 1 bit le nombre de *bits de garde*. Pour simplifier cette recherche, il est tout à fait possible de calculer explicitement le nombre de *bits de garde* utilisés au fur et à mesure des opérations. Il suffit de lancer l'algorithme en mettant tous les coefficients des sous-polynômes en bas du feuillage de l'algorithme de Karatsuba à 1 avant de lancer le calcul des post-traitements. Pour chaque chiffre, le nombre de *bits de garde* consommés sera donné par le nombre de bits du chiffre, moins 1.

Le stockage des sous-polynômes ne pouvant pas être optimisé simplement, l'ensemble des polynômes a été pré-alloué. Le nombre total de coefficients à stocker s'écrit alors :

$$\sum_{i=1}^{p}(2 \cdot \frac{n}{2^i} - 1) \cdot 3^i = \left(\left(\frac{3}{2}\right)^p - 1\right) \cdot 6 \cdot n - (3^p - 1) \cdot 6 \tag{3.4}$$

3.1.6 Implémentation matérielle de Karatsuba

Maintenant que nous avons vu l'implémentation logicielle de l'algorithme de Karatsuba, intéressons-nous à la partie implémentation matérielle. Pour rappel, le composant matériel est responsable du calcul des dernières pré-récursions de Karatsuba, de la multiplication des sous-polynômes résultants terme-à-terme, puis du calcul des premières post-récursions.

L'opération qui demande une attention plus particulière est la multiplication des sous-polynômes, car celle-ci peut devenir le goulot d'étranglement de l'architecture. Deux approches sont possibles :

1) une approche itérative, qui calcule la multiplication polynomiale en étalant les calculs sur plusieurs itérations en réutilisant les mêmes unités arithmétiques.

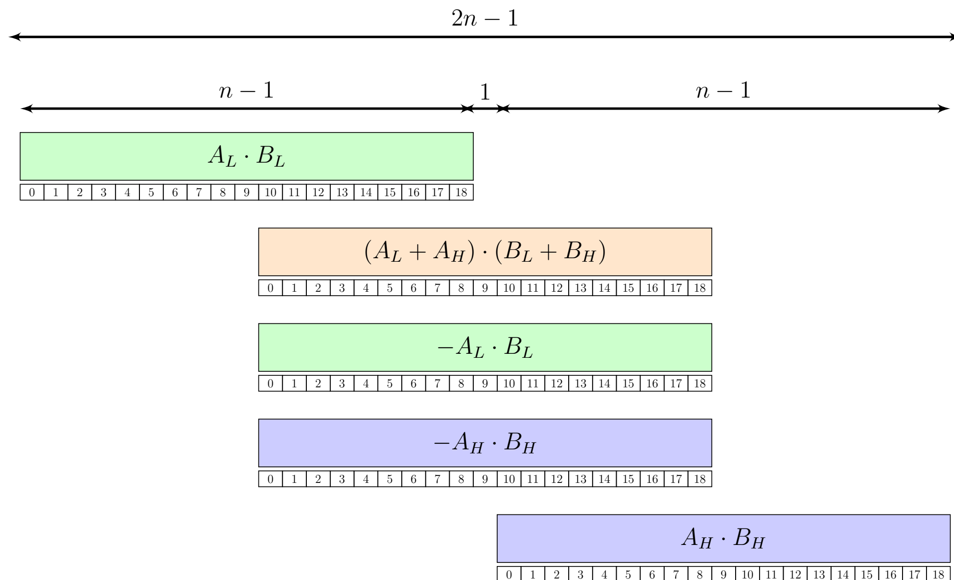


FIGURE 3.6 – Présentation schématique des post-traitements de Karatsuba. Chaque rectangle représente un polynôme et les indices représentés en dessous les indices des coefficients.

Cette approche introduit des latences entre chaque multiplication polynomiale ;

2) une approche parallèle, qui utilise plus d'unités arithmétiques mais qui peut calculer la multiplication polynomiale sans itération.

Pour des raisons d'optimisation, nous avons décidé de concevoir un accélérateur capable de faire des calculs pendant le transfert des sous-polynômes de l'application logicielle vers notre composant matériel. Si nous maîtrisions les latences sur le bus PCI-E, alors une approche série de la multiplication polynomiale aurait permis de compenser ces latences. En pratique, ce n'est pas le cas car elles dépendent de nombreux paramètres (qualité de la carte mère, composants qui utilisent le bus à un instant donné, ...) et donc nous nous sommes orientés vers une implémentation complètement parallèle.

La figure 3.7 présente l'architecture de l'accélérateur matériel. Elle est composée de cinq unités principales :

1) unité de pré-traitement : cette unité a pour rôle l'implémentation des dernières pré-récursions de Karatsuba ;
2) unité pré-crossbar : cette unité va permettre de réordonnancer les polynômes provenant de l'unité de pré-traitement pour réduire les ressources matérielles utilisées ;
3) unité de multiplication polynomiale : multiplie terme-à-terme les polynômes sortant des pré-crossbars ;
4) unité post-crossbar : réalise le réordonnancement inverse des polynômes par rapport au pré-crossbar ;
5) unité de post-traitement : cette unité a pour rôle l'implémentation des pre-

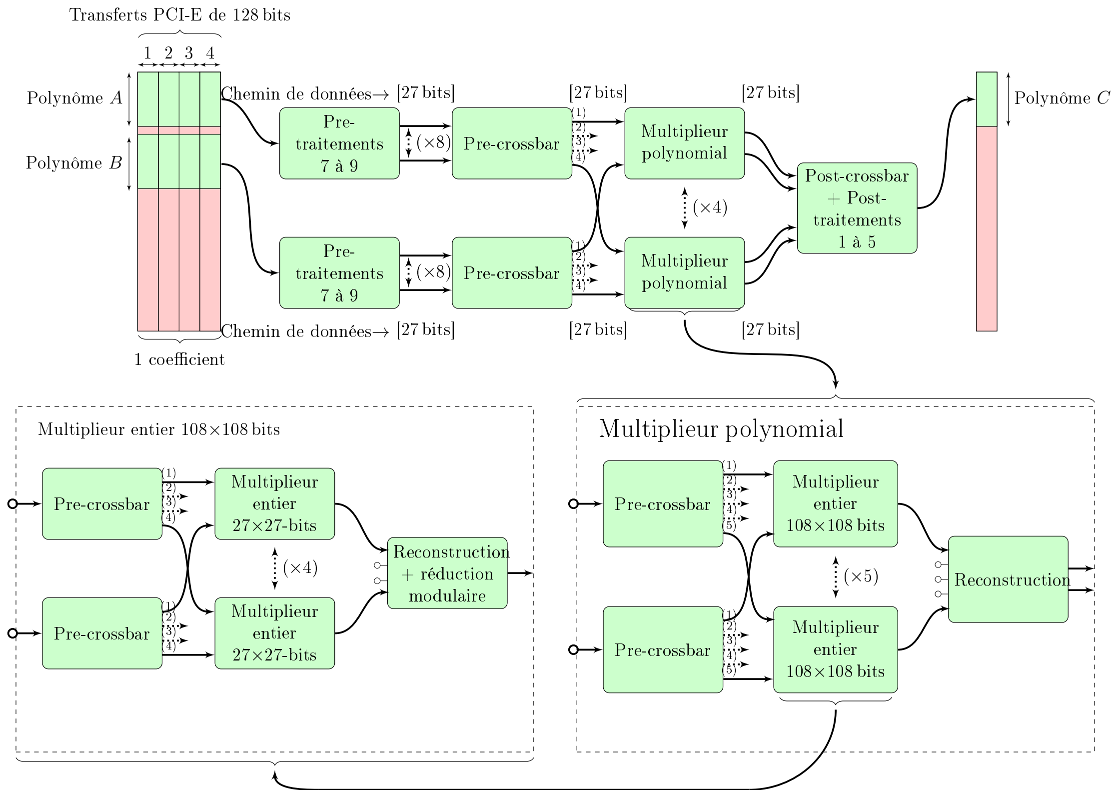


FIGURE 3.7 – Architecture retenue pour l'accélérateur matériel de Karatsuba pour une multiplication polynomiale de polynômes de 2560 coefficients, codés sur 108 bits.

mières post-récursions de Karatsuba.

Nous détaillerons par la suite le fonctionnement de chaque sous-système. L'architecture, illustrée par la figure 3.7, propose une configuration particulière de l'accélérateur afin de fixer les idées. Cette architecture permet une multiplication polynomiale avec des polynômes possédant 2560 coefficients, codés sur 108 bits. Le choix du nombre de pré-récursions sera discuté dans la section 3.1.6.1, et le nombre de post-récursions dans la section 3.1.6.4. Le chemin de données a été défini sur 27 bits, c'est-à-dire que chaque addition, soustraction et multiplication s'effectue sur des opérandes de 27 bits. Ce choix est la conséquence de l'implémentation des multiplieurs entiers câblés (DSP) sur notre plateforme FPGA (plateforme DE5-net de Terasic utilisant un FPGA Stratix V de chez Altera) qui multiplient des opérandes entières de 27×27 bits au maximum. Par conséquent, chaque coefficient est découpé par chiffres de taille de 27 bits. Contrairement à la partie logicielle, l'accélérateur n'a pas été implémenté avec une représentation redondante des chiffres, ce qui aurait permis de traiter plusieurs chiffres en parallèle. En pratique, cela se serait traduit par une utilisation plus importante de la bande passante du PCI-E et un composant utilisant d'autant plus de surface qu'il y a de chiffres à traiter en parallèle. Cette solution n'a pas été retenue car l'adaptation de l'accélérateur à FV.Relin demande l'envoie des clés de relinéarisation. Ainsi, dans tous les cas, le bus PCI-E est déjà fortement sollicité.

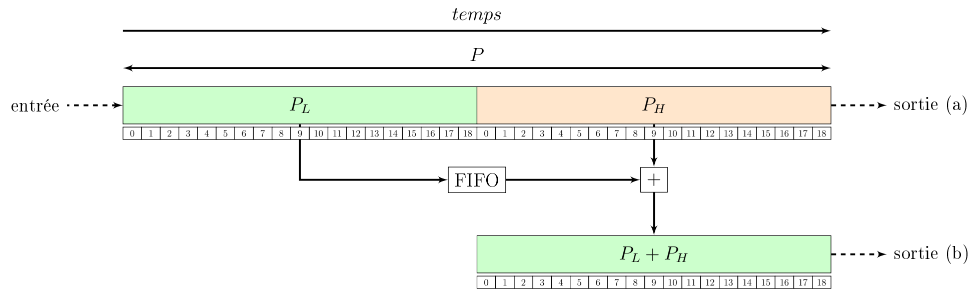


FIGURE 3.8 – Présentation schématique des pré-traitements de Karatsuba dans le composant matériel. Chaque rectangle représente un polynôme et les indices représentés en dessous les indices des coefficients.

3.1.6.1 L'unité pré-traitements

L'unité de pré-traitements est l'unité permettant de donner du parallélisme à l'architecture. Il s'agit d'un résultat assez naturel car les pré-récursions de Karatsuba multiplient par 1.5 le nombre de coefficients à traiter par récursion supplémentaire. La figure 3.8 présente l'architecture matérielle de l'implémentation d'une pré-récursion dite élémentaire. L'échelle des temps permet de donner l'ordre d'arrivée des coefficients des sous-polynômes depuis l'entrée du composant. Pour rappel, les chemins de données étant sur 27 bits, la réception complète d'un coefficient va demander plusieurs cycles.

Afin d'additionner le polynôme P_L avec le polynôme P_H, il est nécessaire de stocker temporairement les coefficients de P_L à l'aide d'une FIFO. Cette architecture crée deux chemins de données, un chemin pour les polynômes P_L et P_H, et un second pour le polynôme $P_L + P_H$. Afin d'implémenter une seconde récursion, une opération similaire est à réaliser sur chacun des polynômes de sortie. En terme d'architecture, cela se traduit simplement par la duplication de l'architecture élémentaire sur les chemins de données (a) et (b). Il faut cependant adapter légèrement la FIFO car les sous-polynômes ont maintenant deux fois moins de coefficients. Cette structure peut être réitérée autant de fois que nécessaire afin de réaliser le nombre requis de récursions. Pour chaque récursion supplémentaire, le nombre de chemins de données est doublé. Ainsi, pour trois pré-traitements, on obtient huit chemins de données. Cela justifie *a posteriori* la présence de huit chemins de données sur la figure 3.7. Pour maximiser la fréquence de fonctionnement du FPGA, un étage de pipeline a été ajouté à la suite de l'opération d'addition. Pour éviter un désalignement des coefficients entre les différents chemins, un registre de pipeline a été également inséré sur l'autre branche de sortie. Cependant, remarquons que cette architecture crée de nombreux chemins de données avec un nombre faible de sous-polynômes comme le montre la figure 3.9a. Cela va se traduire par une sous-utilisation des multiplieurs de sous-polynômes. C'est pour cette raison qu'une unité intermédiaire est ajoutée, afin de réordonnancer les sous-polynômes. Cette unité est appelée pré-crossbar.

3.1.6.2 L'unité pré-crossbar

Comme nous l'avons vu dans la section précédente, l'unité de pré-crossbar va permettre d'optimiser l'utilisation des chemins de données. La figure 3.9a présente l'ordonnancement des sous-polynômes à la sortie du pré-traitement (trois pré-récursions). On remarque bien qu'en sortie du pré-traitement, chaque ligne de sous-polynômes a un taux de remplissage variable. Dans le cas présenté, ce remplissage est de 42%. La figure 3.9b présente un exemple d'ordonnancement des sous-polynômes dans le cas d'un envoi continu de sous-polynômes. Dans l'exemple présenté, le réordonnancement réduit le nombre de lignes par deux, permettant un remplissage des lignes de 84%. Lorsqu'il existe des délais entre les sous-polynômes, il est également possible de réduire davantage le nombre de lignes en utilisant l'espace laissé vacant entre les sous-polynômes.

La table 3.3 synthétise les résultats obtenus concernant la réduction du nombre de lignes pour les dix premières pré-récursions, dans le cas d'absence de délais entre les sous-polynômes. On remarque que l'augmentation du nombre de pré-récursions permet une plus grande flexibilité dans l'optimisation de l'utilisation des lignes. A partir de cinq récursions, le taux de remplissage est proche de 95%, permettant un remplissage quasi optimal. Cependant, l'augmentation du nombre de récursions augmente également le nombre de lignes à réordonner, complexifiant l'implémentation de l'unité de pré-crossbar. Pour trois récursions, elle reste assez simple car seules quatre lignes de sous-polynômes sont à réordonnancer. La $4^{\text{ème}}$ récursion n'apporte pas d'amélioration du taux de remplissage par rapport à la $3^{\text{ème}}$ récursion. Ainsi, afin de limiter l'utilisation des ressources matérielles tout en gardant un taux de remplissage suffisant, nous avons implémenté le réordonnancement pour trois récursions, comme présenté figure 3.9b.

3.1.6.3 L'unité multiplication polynomiale

À la fin du processus de pré-traitement et de pré-crossbar, les sous-polynômes ainsi calculés sont de petit degré (typiquement, entre 3 et 6 comme présenté dans la table 1.7). Il faut maintenant déterminer l'algorithme de multiplication à utiliser. Pour de si petits degrés, l'algorithme de Karatsuba possède quelques limitations qui proviennent :

(1) du nombre faible de pré-récursions restantes (une à deux pré-récursions) ;
(2) du degré des sous-polynômes qui n'est pas nécessairement divisible par deux, amenant dans la majorité des cas à une impossibilité de diviser chaque sous-polynôme en deux polynômes avec le même nombre de coefficients.

La limitation (1) fait qu'il est complexe d'obtenir un réordonnancement efficace des sous-polynômes. Pour donner un exemple, le réordonnancement de deux pré-récursions conduit à une utilisation des multiplieurs de 75% d'après la table 3.3, conduisant à une réduction réelle de 25% de sous-produits comparé à l'algorithme classique (au lieu de 43.75%). À cela il faut considérer l'ajout de nombreuses unités, à savoir une unité de pré-traitement, de pré-crossbar, de post-crossbar et de post-

Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe Vincent Migliore 2017

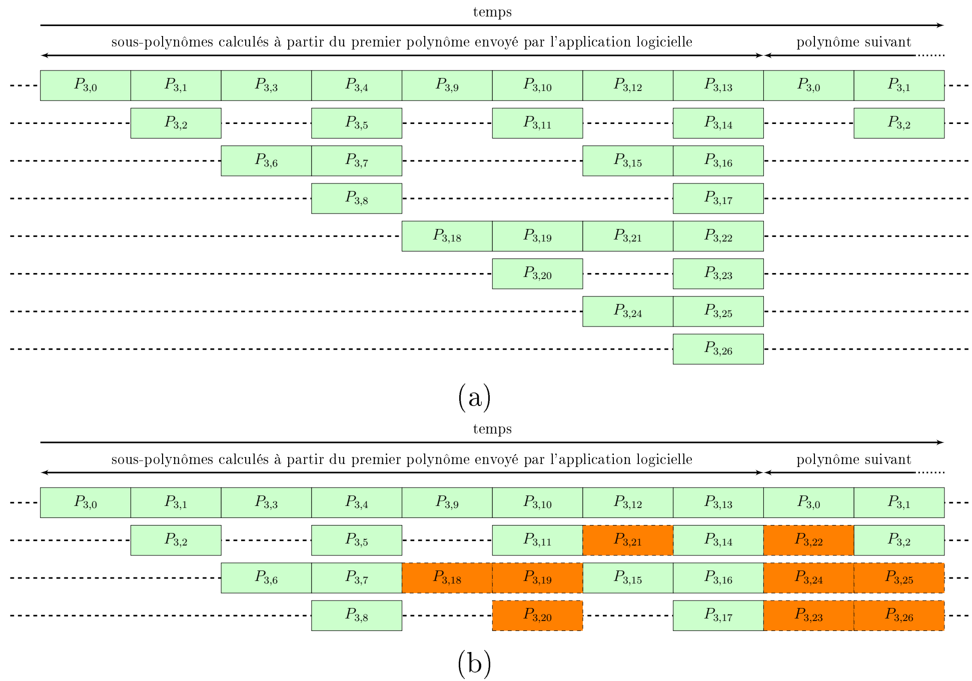


FIGURE 3.9 – Représentation de l'ordonnancement des sous-polynômes en sortie du pré-traitement (a) et du pré-crossbar (b). Chaque ligne en pointillé représente un chemin de données sur 27 bits.

traitement.

La limitation (2) complexifie également l'implémentation de Karatsuba. Le fait que de nombreux cas amènent au fait que le nombre de coefficients d'un polynôme n'est pas divisible par deux demande de gérer deux tailles de sous-polynômes.

Ainsi, comparé à l'algorithme classique, Karatsuba ne permet de réduire le nombre de multiplieurs câblés que de 25% dans le meilleur des cas, au prix d'une architecture complexe. Il nous a donc semblé plus raisonnable d'implémenter l'algorithme classique. De plus, ce choix permet une modification aisée du degré des sous-polynômes et du nombre de chiffres par coefficient comme nous le verrons par la suite.

Le choix de l'algorithme se pose également pour la multiplication entière des coefficients. Là encore, compte tenu de la taille relativement faible des coefficients devant la taille des chiffres, une centaine de bits pour les coefficients pour des chiffres de 27 bits, cela revient à avoir des coefficients sur cinq à deuf chiffres selon la profondeur multiplicative choisie. On se retrouve donc dans un cas similaire à la multiplication de polynômes avec un petit degré, c'est-à-dire que le nombre de chiffres par coefficient est faible. Nous avons donc également privilégié l'algorithme classique pour la multiplication des coefficients.

La figure 3.10 présente le fonctionnement du multiplieur de sous-polynômes. Afin de réaliser l'opération de multiplication polynomiale, un ordonnancement différent

TABLE 3.3 – Présentation de la réduction maximale du nombre de lignes de sous-polynômes avec l'aide d'un pré-crossbar.

	Sans réordonnancement		Avec réordonnancement	
Récursions	Taux d'utilisation des multiplieurs	Lignes de sous-polynômes	Taux d'utilisation des multiplieurs	Lignes de sous-polynômes
1	75 %	2	75 %	2
2	56.25%	4	75 %	3
3	42.19%	8	84.38%	4
4	31.64%	16	84.38%	6
5	23.73%	32	94.92%	8
6	17.8 %	64	94.92%	12
7	13.25%	128	94.92%	18
8	10.01%	256	98.57%	26
9	7.51%	512	98.57%	39
10	5.63%	1024	99.42%	58

est appliqué par opérande de la multiplication polynomiale. Pour le premier polynôme, ses coefficients sont distribués un à un sur chaque ligne de multiplieurs et stockés durant toute la durée des calculs. Il est à noté que comme chaque coefficient possède plusieurs chiffres, nous avons implémenté un registre cyclique sur chaque ligne. Nous appellerons type (a) ce réordonnancement par la suite. Le second polynôme quant à lui, est retardé d'un coefficient d'une ligne à l'autre. Cela est réalisé par l'implémentation d'une FIFO par ligne de multiplieurs de coefficients. Nous appellerons type (b) ce réordonnancement. Les coefficients sont ensuite multipliés terme-à-terme puis sommés de manière horizontale pour reconstruire chaque coefficient de sortie. Lorsque deux multiplications polynomiales sont exécutées séquentiellement, chaque ligne de multiplieurs est remplie à 100%, maximisant l'utilisation des multiplieurs embarqués. Il faut cependant faire attention à l'opération de reconstruction des coefficients de sortie car sommer toutes les lignes horizontalement additionnerait des coefficients de polynômes différents.

Chaque ligne de coefficients a donc été distribuée sur deux sous-lignes avant reconstruction. Finalement, la sortie d'un multiplieur de polynômes se fait sur deux lignes distinctes. En théorie, ce même phénomène se retrouve au niveau du multiplieur de coefficients car le même algorithme est utilisé. Il y aurait donc deux sorties par multiplieur de coefficients, puis par extension 4 sorties au niveau du multiplieur de sous-polynômes. Pour réduire le nombre de lignes et ainsi simplifier les post-traitements, nous avons décidé d'implémenter la réduction modulaire entière au niveau du multiplieur de coefficient.

Au niveau de la consommation de ressources, le nombre de multiplieurs 27×27 bits utilisés est proportionnel :

- Au nombre de multiplieurs polynomiaux en parallèle (quatre dans notre architecture) ;
- Au degré des sous-polynômes ($p+1$ multiplieurs pour des sous-polynômes de degré p) ;
- Au nombre de chiffres par coefficient.

L'adaptation de l'architecture à différents degrés est trivial, il suffit de garder la même stratégie en adaptant le nombre de lignes de multiplieurs au degré.

3.1.6.4 L'unité post-traitements

L'implémentation des post-traitements de Karatsuba en matériel est relativement aisée. La figure 3.11 propose l'architecture utilisée pour les post-traitements. L'ordre d'arrivée des sous-polynômes reprend ici leur ordre naturel suite à un pré-traitement sans réordonnancement (en reprenant la figure 3.8, d'abord P_L est généré en sortie (a), puis P_H et P_L+P_H sont générés en même temps). Dans cette architecture, deux FIFOs sont nécessaires :

1) FIFO (a) : permet de stocker temporairement $A_L \cdot B_L$ pour la reconstruction du terme central de Karatsuba ;
2) FIFO (b) : permet de retarder le sous-polynôme $A_H \cdot B_H$ pour ensuite l'additionner au reste du sous-polynôme de sortie.

On peut remarquer ici que les retards générés par les FIFOs permettent de multiplier virtuellement les sous-polynômes par X^t (t dépendant du retard choisi). Il suffit ensuite d'implémenter deux soustracteurs 27 bits pour le calcul du terme central, et deux additionneurs 27 bits pour prendre en compte la superposition des coefficients entre sous-polynômes.

En réalité, l'architecture réelle du post-traitement est légèrement plus complexe. On retrouve ici un problème similaire à la multiplication polynomiale. Lorsque plusieurs post-traitements sont réalisés successivement, il y a recouvrement entre les sous-polynômes de sortie. Cela est dû au fait que le polynôme de sortie est plus grand que les polynômes d'entrée. Afin de palier ce problème, une distribution des sous-polynômes entre deux unités distinctes peut être réalisée. Dans ce cas particulier, afin d'économiser des ressources matérielles, il est tout à fait possible de faire cette redistribution après les FIFOs, afin de ne pas avoir à doubler l'espace mémoire requis. Dans ce cas, les FIFOs doivent permettre une lecture/écriture des données en mémoire de manière simultanée, contrairement au premier cas où les polynômes ont suffisamment d'écart entre eux pour que la lecture et l'écriture soient disjointes. Cette seconde option a été implémentée.

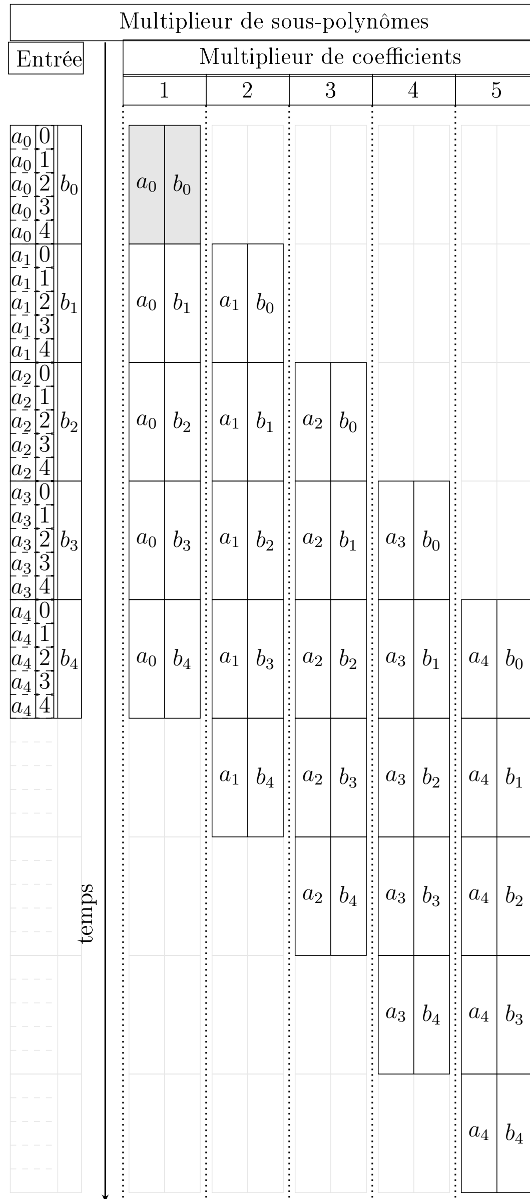


FIGURE 3.10 – Ordonnancement du multiplieur de sous-polynômes en utilisant l'algorithme de multiplication classique. Les sous-polynômes sont de degré 4 et les coefficients sont représentés sur cinq chiffres.

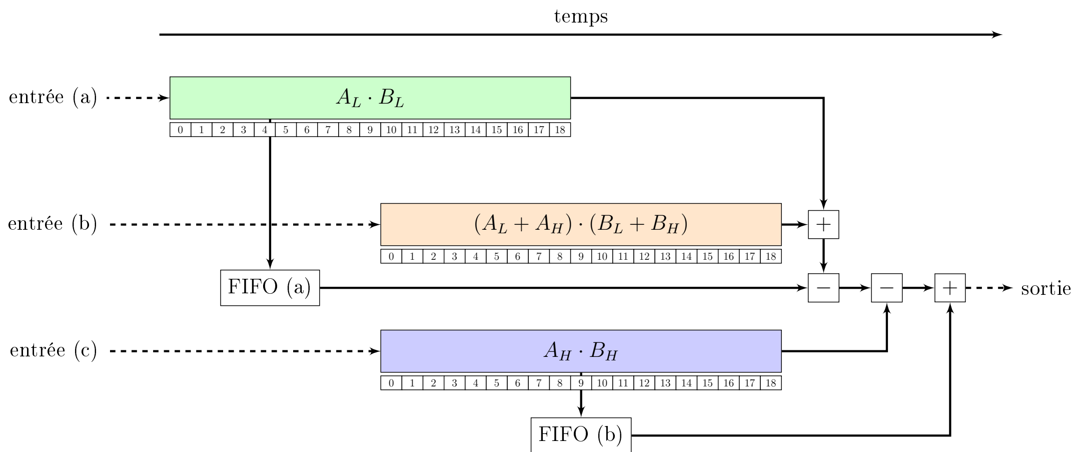


FIGURE 3.11 – Présentation schématique de l'implémentation des post-traitements de Karatsuba dans le composant matériel. Chaque rectangle représente un polynôme et les indices mentionnés en dessous représentent les indices des coefficients.

3.1.6.5 L'unité post-crossbar

Le réordonnancement introduit par l'unité de pré-crossbar nécessite une unité permettant le réordonnancement inverse après la multiplication des sous-polynômes. Celle-ci peut être réalisée :

(1) en une seule fois, comme pour les pré-traitements ;

(2) de manière étagée en l'entrecoupant d'étages de post-traitements.

L'approche (1) est l'approche la plus naturelle car elle est similaire à la stratégie employée pour les pré-traitements.

L'approche (2) permet de sauver de nombreuses ressources mémoires car un minimum de sous-polynômes sont réordonnancés à chaque étage.

La figure 3.12 présente l'ordonnancement des sous-polynômes en sortie de l'unité de multiplication polynomiale. Dans cet exemple, quasiment l'intégralité des sous-polynômes sont bien ordonnancés pour une première post-récursion, sauf le sous-polynôme $A_{3,21} \cdot B_{3,21}$ qui est en avance devant $A_{3,22} \cdot B_{3,22}$ et $A_{3,23} \cdot B_{3,23}$. Si nous avions implémenté une unité de post-crossbar unique, il aurait fallu aligner tous les sous-polynômes par rapport au sous-polynôme le plus retardé, impliquant une unité de stockage par ligne de sous-polynômes.

Compte tenu que l'unité de pré-crossbar réordonnance les sous-polynômes pour les trois dernières pré-récursions, après trois post-récursions réordonnancées, il n'est pas nécessaire de réordonnancer les sous-polynômes pour les récursions suivantes.

Au bout de trois post-récursions, les sous-polynômes de sortie sont disposés sur deux lignes distinctes. Ceci est due à la multiplication polynomiale de polynômes de degré p qui génère un polynôme de degré $2p$ (donc deux fois plus grand à un coefficient près). En calculant des post-récursions supplémentaires, il est possible de réduire le nombre de lignes de sous-polynômes à un. Comme le montre la figure 3.13,

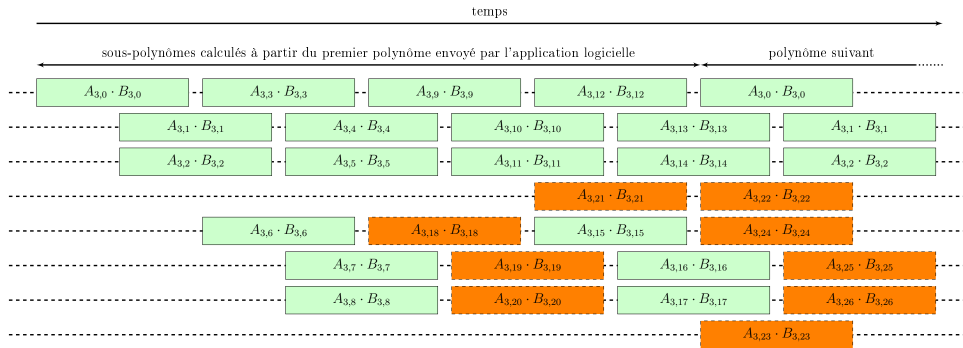


FIGURE 3.12 – Ordonnancement des sous-polynômes après multiplication des sous-produits de Karatsuba.

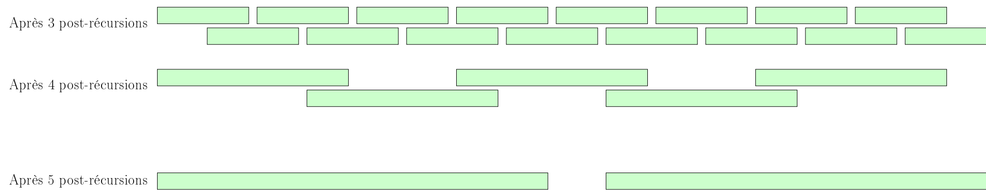


FIGURE 3.13 – Ordonnancement des sous-polynômes après les post-récursions 3, 4 et 5.

ce résultat est atteint à partir de deux post-récursions supplémentaires (soit la 5ème post-récursion).

3.1.6.6 Détermination du nombre optimum de post-récursions à implémenter

Le nombre de post-récursions à implémenter est un choix important dans l'architecture à cause du bus PCI-E. Lors de l'initiation d'une communication sur le bus, l'horloge doit se synchroniser entre les deux périphériques qui communiquent. Cette phase, ajoutée au principe d'arbitrage des communications sur le bus, conduit à l'introduction d'une latence avant le début du transfert. Durant nos expérimentations, nous avons pu constater que cette latence pouvait aller de 8 000 cycles à plus de 40 000 cycles avec notre matériel. Cette limitation à un impact non négligeable sur la stratégie de transfert car cela rend très délicat l'envoi de résultats partiels à l'application logicielle pendant que l'accélérateur matériel effectue les calculs. Cela impliquerait l'ajout d'une FIFO pour stocker les nouveaux coefficients qui sortent de l'accélérateur pour absorber ces latences. Ces latences se produisant par transfert, la stratégie que nous avons adoptée a été de stocker dans une FIFO tous les coefficients de sortie puis d'initier un unique transfert vers l'application logicielle. Afin de minimiser l'espace mémoire requis, nous pouvons jouer sur le nombre de post-récursions implémentées sur le composant matériel. Afin d'évaluer le nombre optimum de post-récursions minimisant la quantité de mémoire requise, nous avons évalué théoriquement la taille de la FIFO de sortie pour p post-récursions implémen-

tées. Cette quantité de mémoire est tout simplement le produit du nombre de sous-polynômes à la post-récursion p (3^p), du nombre de coefficients par sous-polynôme ($\frac{n}{2^{p-1}} - 1$), du nombre de chiffres par coefficient (noté l) et du nombre de bits par chiffres (27 bits). Le calcul complet donne :

$$\mathrm{F}_a(n,p,l) = \left(\frac{n}{2^{p-1}} - 1\right) \cdot 3^p \cdot l \cdot 27 \tag{3.5}$$

Cependant, l'implémentation de post-traitement a un coût en mémoire comme nous l'avons vu section 3.1.6.4. Comme le montre la figure 3.13, à partir de la post-récursion trois les sous-polynômes sont calculés séquentiellement. Ainsi, il faut considérer le stockage des trois sous-polynômes d'une post-récursion donnée avant de l'envoyer à l'unité de post-traitements. Cela demande donc un stockage en tout de cinq sous-polynômes pour une récursion donnée (2 étant nécessaires durant le post-traitement lui-même). Pour simplifier, nous avons considéré les post-récursions au dessus de trois. Pour la récursion p, l'espace mémoire requis est donné par :

$$\mathrm{F}_b(n,p,l) = \sum_{i=3}^{p} 5 \cdot \left(\frac{n}{2^{i-1}} - 1\right) \cdot l \cdot 27 \tag{3.6}$$

L'objectif est donc de minimiser $\mathrm{F}_a(n,p,l) + \mathrm{F}_b(n,p,l)$, pour n et l fixés. La figure 3.14 présente la quantité de mémoire requise en fonction du nombre de post-récursions implémentées sur le composant matériel. Les résultats sont donnés pour quatre configurations différentes représentant les configurations de FV pour les profondeurs multiplicatives 3, 4, 7 et 8 (λ = 80 bits, ω=27 bits). On remarque qu'il existe un nombre de post-récursions p minimisant l'utilisation de la mémoire. Pour les configurations (a) et (b), p=6 est l'optimal, suivi de près par p=7. Pour les configurations (c) et (d), p=7 est l'optimal suivi de près par p=8. Compte tenu du faible écart entre les deux choix possibles pour chaque configuration, prendre la valeur de p la plus grande permet de réduire le temps de transfert (transfert plus petit) et également le temps de traitement logiciel (moins de post-récursions à calculer).

3.2 Adaptation de l'accélérateur de Karatsuba pour la multiplication homomorphe

3.2.1 Nouvelle architecture de haut-niveau

Dans cette section, nous allons présenter l'adaptation de l'accélérateur matériel pour réaliser les opérations FV.Mult et FV.Relin. La figure 3.15 présente la modification effectuée sur l'architecture. Les modifications principales sont les suivantes :

1) l'ajout des pré-traitements pour les clés de relinéarisation ;
2) l'implémentation de l'opération de division et arrondi pour FV.Mult ;
3) la modification de l'architecture du multiplieur polynomial pour FV.Relin.

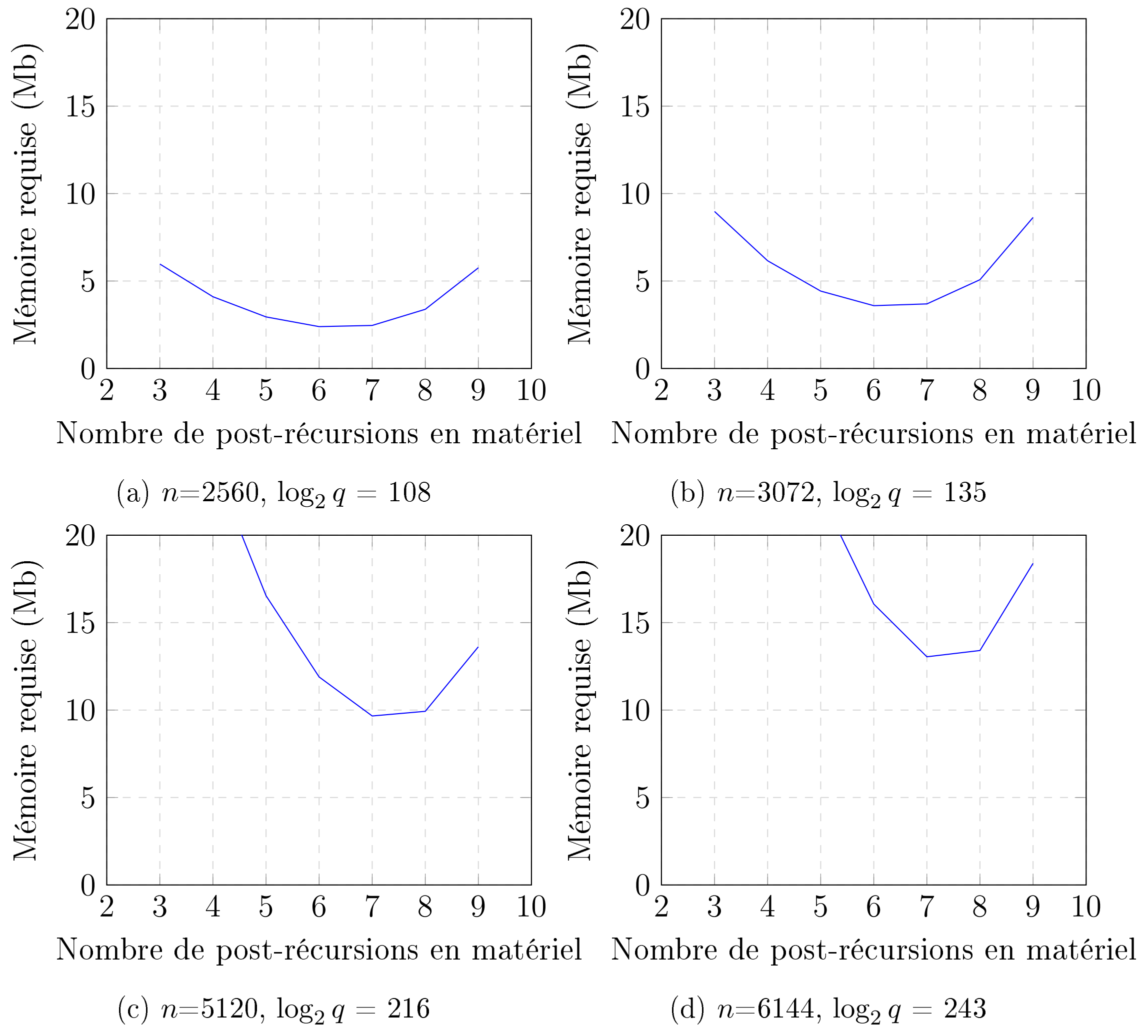


FIGURE 3.14 – Évaluation de la quantité de mémoire requise en fonction du nombre de post-récursions implémentées sur le composant matériel.

3.2.2 Adaptation de FV.Mult

3.2.2.1 Démarche

Pour simplifier la lecture, la fonction FV.Mult est recopiée ci-dessous :

$$
\begin{aligned}
&\text{FV.Mult}(\mathbf{C_A}, \mathbf{C_B}, \mathbf{\Gamma}) : \\
&\widetilde{C}_0 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[0] \cdot \mathbf{C_B}[0] \right\rceil\right]_q \\
&\widetilde{C}_1 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[0] \cdot \mathbf{C_B}[1] + \frac{t}{q}\mathbf{C_A}[1] \cdot \mathbf{C_B}[0] \right\rceil\right]_q \\
&\widetilde{C}_2 = \left[\left\lfloor \frac{t}{q}\mathbf{C_A}[1] \cdot \mathbf{C_B}[1] \right\rceil\right]_q \\
&\mathbf{C}_\times = \text{FV.Relin}(\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2, \mathbf{\Gamma}) \\
&\text{return } \mathbf{C}_\times
\end{aligned}
$$

Le point bloquant dans cette écriture est le fait que l'opération de division et arrondi doit être effectuée après l'opération de multiplication polynomiale. Cela implique dans notre architecture de déporter la réduction modulaire entière (qui pour rappel est implémentée en sortie des multiplieurs de coefficients) après la multiplication polynomiale. Cela revient donc à doubler la complexité des post-traitements (matérielle et logicielle), ainsi que la taille du transfert de l'accélérateur matériel vers l'application logicielle.

Afin de simplifier les calculs de cette étape, nous avons effectué quelques modifications du schéma FV. Compte tenu du fait que FV (et les schémas reposant sur R-LWE en général) jouent sur la présence d'un bruit qui s'intensifie au fur et à mesure des opérations, nous avons effectué des approximations sur l'opération d'arrondi au prix d'une augmentation du niveau de bruit. Les deux approximations proposées sont les suivantes :

Approximation (1) : transformation de l'opération d'arrondi au plus proche en opération d'arrondi à l'entier inférieur ;

Approximation (2) : opération de division calculée juste après les multiplieurs de coefficients (donc au même moment que l'implémentation de la réduction modulaire entière de l'accélérateur présentée section 3.1).

Avant de calculer la contribution de notre approximation au bruit dans le cas de notre architecture, déterminons tout d'abord ce bruit pour un algorithme de multiplication quelconque. Nous appliquerons ensuite le résultat à l'algorithme de Karatsuba. Considérons A et B deux polynômes et $C = A \cdot B$ leur produit. Quel que soit l'algorithme utilisé, chaque coefficient de C est le résultat d'une accumulation de l sous-produits des coefficients issus de A et B. Si on note c un coefficient quelconque de C, a_i et b_i certains coefficients des polynômes A et B (qui sont fonction du

coefficient c et de l'algorithme de multiplication choisi), alors c peut s'exprimer par :

$$c = \sum_{i=1}^{l} a_i \cdot b_i \tag{3.7}$$

Calculons maintenant l'impact des deux approximations proposées sur le résultat de $\left\lfloor \frac{q}{t} c \right\rceil$:

$$\begin{aligned} \left\lfloor \frac{t}{q} c \right\rceil &= \left\lfloor \frac{t}{q} \sum_{i=1}^{l} a_i \cdot b_i \right\rceil \\ &= \left\lfloor \frac{t}{q} \sum_{i=1}^{l} a_i \cdot b_i \right\rfloor + \epsilon_1 \text{ (approximation (1))} \\ &= \left\lfloor \sum_{i=1}^{l} \frac{t}{q} \cdot a_i \cdot b_i \right\rfloor + \epsilon_1 \\ &= \sum_{i=1}^{l} \left\lfloor \frac{t}{q} \cdot a_i \cdot b_i \right\rfloor + \epsilon_1 + \epsilon_2 \text{ (approximation (2))} \end{aligned} \tag{3.8}$$

Pour l'approximation (1), comme l'erreur entre l'arrondi à l'entier le plus proche et l'arrondi à l'entier inférieur est au pire cas de 1, nous avons $|\epsilon_1| \leq 1$.

Pour l'approximation (2), au pire cas, chaque élément de la somme contribue à une erreur de 1, nous avons donc $|\epsilon_2| \leq l$. En conclusion, cette approximation contribue au pire cas à $\lceil \log_2(l+1) \rceil$ bits de bruit. Compte tenu du fait que la relinéarisation contribue elle-même au bruit lors de la multiplication homomorphe, notre approximation est intéressante si la contribution au bruit est négligeable devant le bruit de relinéarisation. Le bruit de relinéarisation est égal à ω, qui usuellement est fixé à 32 ou 64 bits. Ainsi, il faut donc que l soit négligeable devant ω pour que les approximations n'impactent pas significativement le niveau de bruit.

Intéressons-nous maintenant au cas de l'algorithme de Karatsuba et calculons le nombre de sous-produits au pire cas. Pour rappel, l'équation de l'algorithme de Karatsuba est la suivante :

$$\begin{aligned} &A = A_L + X^{n/2} \cdot A_H, B = B_L + X^{n/2} \cdot B_H \\ &C = A_L B_L \\ &\quad + \big((A_L + A_H) \cdot (B_L + B_H) - A_L B_L - A_H B_H\big) \cdot X^{\lceil n/2 \rceil} \\ &\quad + A_H B_H \cdot X^{2\lceil n/2 \rceil} \\ &C = A_L B_L \\ &\quad + \big(A_L B_L + A_L B_H + A_H B_L + A_H B_H - A_L B_L - A_H B_H\big) \cdot X^{\lceil n/2 \rceil} \\ &\quad + A_H B_H \cdot X^{2\lceil n/2 \rceil} \end{aligned}$$

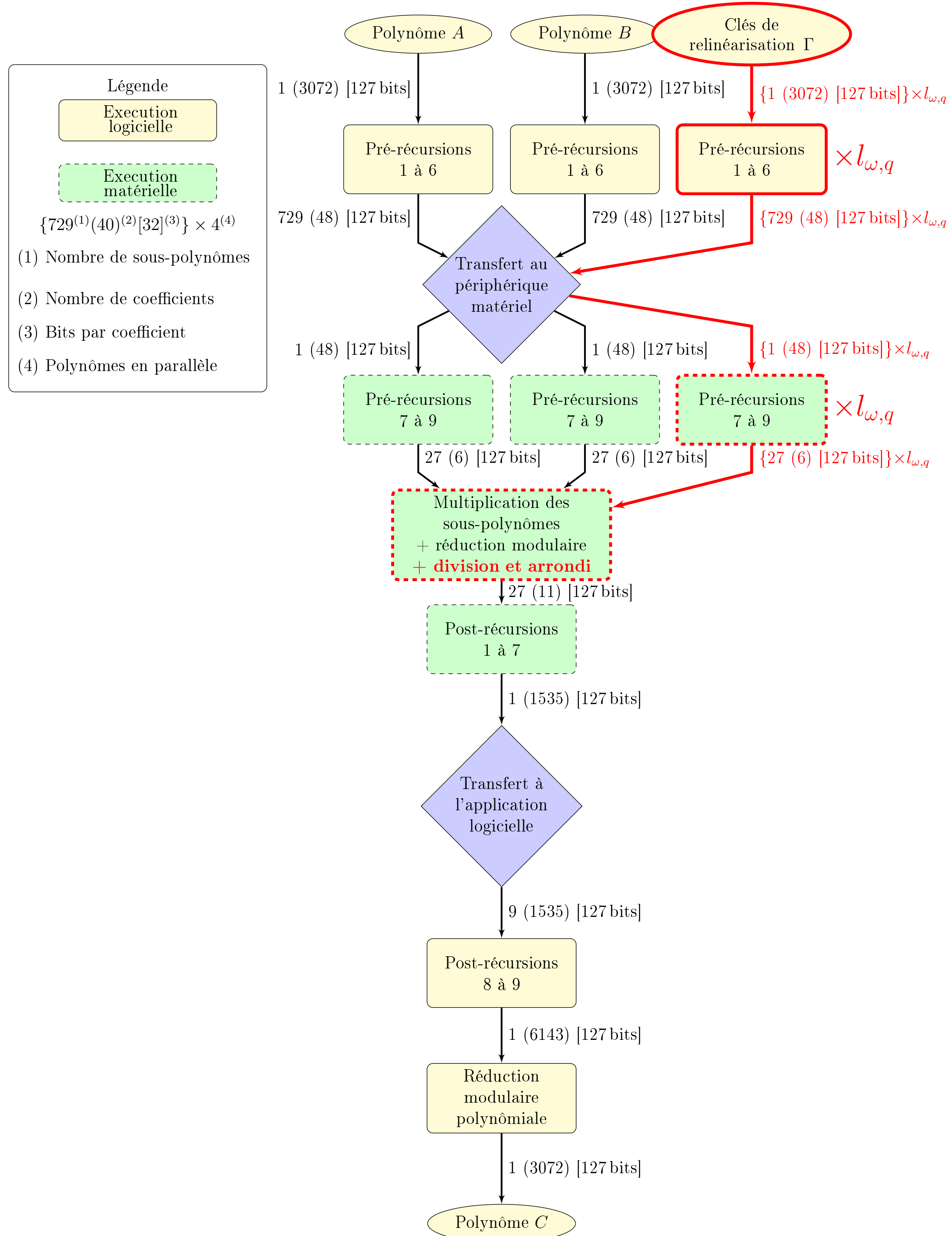


FIGURE 3.15 – Stratégie d'implémentation de l'accélération de FV.mult et FV.relin avec Karatsuba dans une approche de co-conception. L'architecture peut effectuer deux opérations : $\left\lfloor A \cdot B \cdot \frac{2}{q} \right\rceil$ (pour FV.mult) et une variante de $\left\langle \text{FV.WordDecomp}_{\omega,q}(A), \mathbf{\Gamma} \right\rangle$.

Tout d'abord remarquons que $\left\lfloor \frac{t}{q}(A_L B_L - A_L B_L) \right\rceil = \left\lfloor \frac{t}{q} A_L B_L \right\rceil - \left\lfloor \frac{t}{q} A_L B_L \right\rceil (= 0)$.

TABLE 3.4 – Impact des approximations (1) et (2) introduites dans FV sur le niveau de bruit pour l'algorithme de Karatsuba.

n	p	Contribution au bruit des approximations (1) et (2) dans FV.Mult
2048	9	19 bits
2560	9	19 bits
3072	9	20 bits
3584	9	20 bits
4096	10	20 bits
5120	10	21 bits
6144	10	22 bits
7168	10	22 bits

Ainsi, seuls les sous-produits $A_L B_H + A_H B_L$ contribuent à l'erreur d'approximation pour le terme central.

Ainsi, pour chaque récursion de Karatsuba, l'approximation se fait sur trois produits-accumulations de sous-polynômes (et non pas quatre car $A_L B_L$ et $A_H B_H \cdot X^{2\lceil n/2 \rceil}$ n'ont pas d'indices de coefficients en commun). À la fin des récursions, l'algorithme classique est utilisé, et comme les sous-polynômes ont $\frac{n}{2^p}$ coefficients, il y a $\left(\frac{n}{2^p}\right)^2$ sous-produits. Finalement, le nombre de sous-produits au total est donné par :

$$3^p \cdot \left(\frac{n}{2^p}\right)^2 \tag{3.9}$$

La table 3.4 donne une application numérique du niveau de bruit ajouté par les approximations (1) et (2) pour différentes configurations de Karatsuba. On remarque que la contribution au bruit est comprise entre 19 et 22 bits pour des polynômes allant de 2048 à 7168 coefficients. La contribution est non négligeable, mais reste modeste devant la contribution de la relinéarisation.

3.2.2.2 Modification de l'architecture

Bien que l'architecture de l'accélérateur soit globalement inchangée, il reste néanmoins à implémenter l'opération de division et arrondi à l'entier inférieur en matériel. Cette opération est grandement simplifiée par le choix d'un module q qui est une puissance de 2, et le fait que l'espace des messages est binaire (donc $t = 2$). Ainsi, l'opération se réduit par un simple décalage de $\log_2(q) - 1$ bits à droite.

3.2.3 Adaptation de FV.Relin

3.2.3.1 Démarche

Pour simplifier la lecture, les fonctions FV.Relin, FV.PowersOf$_{\omega,q}$ et FV.WordDecomp$_{\omega,q}$ sont recopiées ci-dessous :

— FV.PowersOf$_{\omega,q}(A)$:
$\mathbf{A} \in R_q^{l_{\omega,q}}$
for $i = 0$ to $l_{\omega,q} - 1$
  $\mathbf{A}[i] = [A \cdot \omega^i]_q$
end for
return $\mathbf{A}$

— FV.WordDecomp$_{\omega,q}(A)$:
$\mathbf{A} \in R_q^{l_{\omega,q}}$
for $i = 0$ to $l_{\omega,q} - 1$
  $l_0 = i \cdot \log_2 \omega$
  $l_1 = (i+1) \cdot \log_2 \omega - 1$
  $\mathbf{A}[i] = A_{(l_0 \to l_1)}$
end for
return $\mathbf{A}$

FV.Relin$(\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2, \mathbf{\Gamma})$:

$$C_{R,0} = \left[\widetilde{C}_0 + \left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[0] \right\rangle\right]_q$$

$$C_{R,1} = \left[\widetilde{C}_1 + \left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[1] \right\rangle\right]_q$$

$$\mathbf{C_R} = (C_{R,0}, C_{R,1})$$

return $\mathbf{C_R}$

Selon le choix de ω, les calculs de $C_{R,0}$ et $C_{R,1}$ vont demander $l_{\omega,q} = \left\lceil \frac{\log_2 q}{\omega} \right\rceil$ multiplications et $l_{\omega,q}$ additions de polynômes. L'architecture de l'accélérateur présentée dans la section 3.1 traitant une multiplication polynomiale à la fois, cela se traduirait par $l_{\omega,q}$ multiplications exécutées séquentiellement. Cependant, les coefficients des polynômes à multiplier ont ici une forme particulière comparé au cas général. En effet, avec l'opération FV.WordDecomp$_{\omega,q}$, le polynôme est découpé en $l_{\omega,q}$ sous-polynômes avec des coefficients sur $\log_2 \omega$ bits. Par extension, les sous-produits de coefficients se feront sur $\omega \times \log_2 q$ bits au lieu de $\log_2 q \times \log_2 q$ bits. Reste à exploiter cette propriété dans le cadre de l'algorithme de Karatsuba.

Nous avons fait reposé notre optimisation sur deux propriétés que nous démontrerons ensuite :

(1) le produit/accumulation des sous-polynômes peut s'effectuer au niveau des coefficients directement dans le cadre de la multiplication classique ;
(2) le produit/accumulation des polynômes de la relinéarisation peut être effectué au niveau des sous-polynômes de Karatsuba.

Les deux optimisations utilisées conjointement permettent de déporter le produit/accumulation des polynômes de la relinéarisation au niveau des coefficients dans notre architecture. En utilisant le fait que les multiplications des coefficients

sont sur $\omega \times \log_2 q$ bits (donc utilisent moins de multiplieurs 27×27 bits que le cas classique), il est possible de réutiliser les multiplieurs 27×27 bits non utilisés pour calculer plusieurs sous-produits en parallèle. De plus, en effectuant l'accumulation des polynômes en sortie de la multiplication des coefficients (optimisation (1)) permet d'avoir à calculer les post-traitements pour un unique polynôme au lieu de plusieurs. De plus, cela permet également de réduire le besoin en bande passante sur le PCI-E.

Intéressons-nous maintenant à la démonstration des propriétés (1) et (2). Tout d'abord, afin de simplifier la lecture, nous allons renommer FV.WordDecomp$_{\omega,q}$ par WD. Pour le polynôme A, notons $\mathrm{WD}(A) = \mathbf{A}$. Notons C le résultat de $\left\langle \mathbf{A}, B \right\rangle$, c'est-à-dire la partie de l'opération de relinéarisation que l'on souhaite optimiser. En appliquant la définition de la multiplication classique à C, on obtient que :

$$\begin{aligned} C &= \left\langle \mathbf{A}, B \right\rangle \\ &= \sum_{i=1}^{l_{\omega,q}} \mathbf{A}[i] \cdot B \\ &= \sum_{i=1}^{l_{\omega,q}} \underbrace{\sum_{j=0}^{k} \mathbf{a}[i]_j b_{k-j}}_{\text{multiplication classique}} \\ &= \sum_{j=0}^{k} \underbrace{\sum_{i=1}^{l_{\omega,q}} \mathbf{a}[i]_j b_{k-j}}_{\substack{\text{produit/accumulation} \\ \text{des coefficients}}} \end{aligned} \tag{3.10}$$

Ainsi, l'inversion de la somme permet de déporter le produit/accumulation de polynômes au niveau des coefficients. La multiplication classique étant effectuée dans notre architecture au niveau des sous-polynômes de Karatsuba, il reste à démontrer que l'on peut effectuer le produit/accumulation au niveau des sous-polynômes de

Karatsuba (propriété (2)). En reprenant la définition de l'algorithme de Karatsuba :

$$\begin{aligned} C &= \sum_{i=1}^{l_{\omega,q}} \mathbf{A}[i]B \\ &= \sum_{i=1}^{l_{\omega,q}} \Bigg[\mathbf{A}_L[i]B_L + \mathbf{A}_H[i]B_H x^n \\ &+ \Big((\mathbf{A}_L[i] + \mathbf{A}_H[i])(B_L + B_H) - \mathbf{A}_L[i]B_L - \mathbf{A}_H[i]B_H\Big)x^{n/2}\Bigg] \\ &= \sum_{i=1}^{l_{\omega,q}} \mathbf{A}_L[i]B_L + x^n \sum_{i=1}^{l_{\omega,q}} \mathbf{A}_H[i]B_H + x^{n/2} \sum_{i=1}^{l_{\omega,q}} (\mathbf{A}_L[i] + \mathbf{A}_H[i])(B_L + B_H) \\ &- x^{n/2} \sum_{i=1}^{l_{\omega,q}} \mathbf{A}_L[i]B_L - x^{n/2} \sum_{i=1}^{l_{\omega,q}} \mathbf{A}_H[i]B_H \end{aligned}$$

On remarque ici que l'opération de somme se distribue sur chacun des sous-polynômes de Karatsuba de la $1^{\text{ème}}$ récursion. Ceci se vérifie également pour les récursions suivantes par récursion immédiate. Ainsi, le produit/accumulation peut s'effectuer au niveau des sous-polynômes de n'importe quelle récursion de Karatsuba.

Il existe un dernier point bloquant à l'accélération de la relinéarisation. Dans le calcul de $\left\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[i] \right\rangle$, il est normalement nécessaire de calculer les pré-traitements de chaque élément du vecteur $\text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2)$ de manière séparée. Les coefficients de chaque membre de $\text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2)$ étant initialement sur $\log_2 \omega$ bits, après p pré-récursions, ils seront sur $\log_2 \omega + p$ bits. Le choix de ω est donc crucial. Deux stratégies peuvent être adoptées, avec différents impacts sur l'architecture :

(1) on prend $\log_2 \omega + p = 27$ bits. Dans ce cas, après les pré-traitements, les coefficients de chaque membre de $\text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2)$ ont la bonne taille pour les multiplieurs embarqués. En contrepartie, le choix d'un tel ω implique :
   (a) un nombre plus important de clés de relinéarisation comparé au cas classique $\log_2 \omega = 32$ bits. En particulier, pour 9 pré-récursions, il faudrait prendre $\log_2 \omega = 18$ bits, impliquant une augmentation du nombre de clés par approximativement $32/18 = 1.77$. Chaque clé étant un polynôme dont chaque coefficient est sur $\log_2 q$ bits, l'impact est non négligeable car le besoin en bande passante sur le PCI-E en est d'autant augmenté ;
   (b) la limitation (a) implique également un nombre plus important de multiplications polynomiales à effectuer car proportionnel au nombre de clés de relinéarisation ;
   (c) l'opération $\text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2)$ n'est pas triviale car il faut découper chaque coefficient du polynôme $\widetilde{C}_2$ sur des coefficients de $\log_2 \omega$ bits.

(2) prendre $\log_2 \omega + p > 27$ bits. Dans ce cas, le nombre de clés de relinéarisation est plus faible, et le choix de $\log_2 \omega = 27$ bits est plus naturel car l'opération

de FV.WordDecomp$_{\omega,q}(\widetilde{C}_2)$ est déjà faite par construction car les chiffres de nos polynômes sont sur 27 bits. En contrepartie :

(a) en choisissant $\log_2 \omega = 27$ bits et en considérant qu'il y aura au moins 6 pré-récursions en logiciel, nous avons $\log_2 \omega + p_{\text{logiciel}} \geq 33$ bits. L'espace alloué pour les chiffres en logiciel étant de 32 bits, à la fin des pré-traitements, les coefficients de chaque membre de FV.WordDecomp$_{\omega,q}(\widetilde{C}_2)$ seront sur 2 chiffres. Cela revient à doubler l'espace mémoire requis pour les pré-traitements logiciels et à approximativement doubler le temps de traitement associé. De plus, cela demande d'adapter l'accélérateur matériel pour gérer ces multiplications particulières.

Pour le choix final, nous avons décidé d'adapter l'opération de relinéarisation afin de ne pas avoir à traiter les différentes limitations des stratégies (1) et (2). En reprenant la définition des clés de relinéarisation, l'équation de relinéarisation devient :

$$\begin{aligned}&\Big\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[0]\Big\rangle \\ &= \Big\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \text{FV.PowersOf}_{\omega,q}(S_{key}^2) - \big(\mathbf{A}\cdot S_{key} + \mathbf{E}\big)\Big\rangle \\ &= \underbrace{\sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \big(\omega^{i-1}\cdot S_{key}^2\big)}_{(a)} - \underbrace{\sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \big(\mathbf{A}[i]\cdot S_{key} + \mathbf{E}[i]\big)}_{(b)}\end{aligned} \tag{3.11}$$

$$\begin{aligned}&\Big\langle \text{FV.WordDecomp}_{\omega,q}(\widetilde{C}_2), \mathbf{\Gamma}[1]\Big\rangle \\ &= \sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \mathbf{A}[i]\end{aligned} \tag{3.12}$$

Pour rappel, le terme (a) donne comme résultat $\widetilde{C}_2 \cdot S_{key}^2$. Cela se comprend bien car le terme FV.WordDecomp$_{\omega,q}(\widetilde{C}_2)$ revient à faire l'extraction binaire de chaque coefficient par des morceaux de $\log_2 \omega$ bits, que l'on vient multiplier par les puissances de ω (car le second terme est $\omega^{i-1} \cdot S_{key}^2$) :

$$\begin{aligned}\sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \big(\omega^{i-1}\cdot S_{key}^2\big) &= S_{key}^2 \cdot \underbrace{\sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \omega^{i-1}}_{\widetilde{C}_2} \\ &= \widetilde{C}_2 \cdot S_{key}^2\end{aligned} \tag{3.13}$$

Pendant les pré-traitements de Karatsuba, chaque terme de $\widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)}$ va grandir et potentiellement dépasser ω. Cependant, comme ce dernier est multiplié

à la fin par une puissance de ω, il est tout à fait possible de faire une opération de propagation de retenue en considérant les chiffres sur $\log_2 \omega$ bits. De manière plus formelle, en considérant une récursion donnée de Karatsuba, avec P un sous-polynôme de $\widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)}$, P' le polynôme contenant les bits de P ne dépassant pas $\log_2 \omega$, et Q le sous-polynôme de S_{key}, alors au moment de la multiplication des sous-polynômes de l'algorithme de Karatsuba, on obtient :

$$
\begin{aligned}
&\big(P_{0\to\omega-1} && +\omega\cdot P'_0\big)\cdot\big(\omega^0\cdot Q\big)\\
+&\big(P_{\omega\to 2\cdot\omega-1} && +\omega\cdot P'_1\big)\cdot\big(\omega^1\cdot Q\big)\\
+&\big(P_{2\cdot\omega\to 3\cdot\omega-1} && +\omega\cdot P'_2\big)\cdot\big(\omega^2\cdot Q\big)\\
+&\cdots\\
=&\big(P_{0\to\omega-1} && +0\big)\cdot\big(\omega^0\cdot Q\big)\\
+&\big(P_{\omega\to 2\cdot\omega-1} && +P'_0\big)\cdot\big(\omega^1\cdot Q\big)\\
+&\big(P_{2\cdot\omega\to 3\cdot\omega-1} && +P'_1\big)\cdot\big(\omega^2\cdot Q\big)\\
+&\cdots
\end{aligned}
\tag{3.14}
$$

En conclusion, en prenant $\log_2 \omega = 27$ bits, il est tout à fait possible de calculer les pré-traitements sur $\widetilde{C}_2$ directement, de propager la retenue avant multiplication des sous-produits et finalement obtenir le même résultat que si nous avions séparé les pré-traitements des polynômes du vecteur FV.WordDecomp$_{\omega,q}(\widetilde{C}_2)$. Cette optimisation permet de se passer de toutes les limitations exposées précédemment et de ne demander qu'une modification mineure de notre architecture. Il reste néanmoins à s'assurer qu'il n'y a pas d'impact sur le membre (b) de l'équation 3.11. Pour cela, il faut bien comprendre la philosophie derrière ce terme. Il faut en effet que le terme $-\sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \big(\mathbf{A}[i]\cdot S_{key} + \mathbf{E}[i]\big)$ de l'équation 3.11 se simplifie avec $\sum_{i=1}^{l_{\omega,q}} \widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)} \cdot \mathbf{A}[i]$ de l'équation 3.12 après déchiffrement. En fait, le facteur $\widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)}$ ne doit pas avoir une forme particulière pour le déchiffrement, il faut juste qu'il coïncide terme-à-terme entre chaque membre du chiffré après relinéarisation. Ainsi, notre optimisation ne donnera pas exactement le terme $\widetilde{C}_{2((i-1)\cdot\omega\to i\cdot\omega)}$, mais comme il reste identique sur chacun des deux membres, notre optimisation n'impacte pas le déchiffrement.

3.2.3.2 Modification de l'architecture

Suite aux optimisations présentées précédemment, l'architecture finale va demander certaines modifications. Contrairement à FV.Mult, l'adaptation de l'accélérateur pour FV.Relin nécessite plusieurs étape, notamment :

(1) le calcul des pré-traitements de Karatsuba de $\widetilde{C}_2$ ainsi que des clés de relinéarisation en logiciel ;

(2) l'envoie à l'accélérateur matériel des sous-polynômes issus de $\widetilde{C}_2$ et des clés de relinéarisation ;
(3) de calculer les dernières pré-récursions de Karatsuba (coté composant matériel) pour les clés de relinéarisation et pour le polynôme à relinéariser ;
(4) de s'assurer que l'on a bien fait l'opération de propagation de retenue sur des chiffres de $\log_2 \omega$ bits avant le calcul des sous-produits (obtenu par construction en prenant $\log_2 \omega = 27$ bits ;
(5) d'adapter l'unité matérielle de multiplication polynomiale.

L'opération (1) est simple et ne demande pas de modification particulière. On peut cependant préciser que comme les clés de relinéarisation seront réutilisées pour toutes les opérations de multiplication homomorphe, il n'est nécessaire de calculer leurs pré-traitements qu'une seule fois. Le point (2) en revanche peut devenir rapidement le goulot d'étranglement de l'architecture très limitant si la bande passante du PCI-E est insuffisante comparée aux nombres de clés de relinéarisation. Avec les implémentations actuelles du PCI-E, la configuration la plus performante permet de traiter 256 bits de données en parallèle à 250 MHz (configuration 8 lignes, PCI-E gen 3.X de la table 3.2). Il faut donc faire tenir sur 256 bits le polynôme à relinéariser plus les différentes clés de relinéarisation. Comme les polynômes sont envoyés chiffres par chiffres eux-mêmes sur 27 bits, il est possible d'envoyer au mieux neuf polynômes en parallèle. Pour effectuer l'opération de relinéarisation avec un seul transfert, il faut donc au plus huit clés de relinéarisation. Il faut donc que le module q ne dépasse pas $8 \cdot 27 = 216$ bits. Ainsi, pour la profondeur multiplicative huit et pour 80 bits de sécurité (demandant $\log_2 q = 239$ bits), la bande passante du PCI-E devient trop faible pour exécuter la relinéarisation avec un seul transfert. Il faut donc soit réaliser deux transferts successifs, soit stocker une clé de relinéarisation sur la mémoire embarquée du composant matériel. Cette limitation arrive plus tôt si, bien entendu, le PCI-E fonctionne avec une taille de données de transfert plus faible. Ceci est notamment le cas avec notre implémentation car RIFFA supporte au mieux la configuration 128 bits/250 MHz.

L'opération (3) demande d'ajouter au sein du composant matériel, une unité de pré-traitement et une unité pré-crossbar par clé de relinéarisation.

Enfin l'opération (5) demande d'adapter l'unité de multiplication de polynômes et l'unité de multiplication des coefficients. Pour l'unité de multiplication de polynômes, rappelons que deux réordonnancements différents sont implémentés, un par sous-polynôme. Il faut donc appliquer un premier réordonnancement au sous-polynôme provenant du polynôme à relinéariser, et ajouter un second réordonnancement par sous-polynôme provenant des clés de relinéarisation. Pour le multiplieur de coefficients, le choix du bon ordonnancement est capital pour obtenir le bon résultat. En notant a un coefficient d'un sous-polynôme du polynôme à relinéariser (et a_i son $i^{\text{ème}}$ chiffre), et $\gamma[i]$ un coefficient d'un sous-polynôme de la $i^{\text{ème}}$ clé de relinéarisation, alors il faut calculer $a_i \cdot \gamma[i]$. Il faut donc répartir les différents a_i sur des lignes de multiplieur distinctes, et de multiplier chaque ligne avec le $\gamma[i]$ associé. Ainsi, les coefficients issus des clés de relinéarisation n'ont pas d'ordonnancement particulier, il faut juste s'assurer du bon alignement du coefficient avec le chiffre a_i

associé.

3.3 Adaptation de l'accélérateur de Karatsuba pour le chiffrement

3.3.1 Nouvelle architecture de haut-niveau

L'adaptation de l'architecture de l'accélérateur à FV.Encrypt est présentée figure 3.16. Les éléments en rouge correspondent à des adaptations qui ont été nécessaires par rapport à l'architecture de base présentée à la figure 3.1. Les modifications effectuées sont les suivantes :

1) Modification du chemin de données pour s'adapter à la taille du polynôme U.
2) Implémentation de plusieurs chiffrements en parallèle.

3.3.2 Adaptation de FV.Encrypt

Pour simplifier la lecture, la fonction FV.Encrypt est recopiée ci-dessous :

$$\begin{aligned}
&\text{FV.Encrypt}(m, \mathbf{P_{key}}) : \\
&U \leftarrow D_{R_q,\sigma_{key}},\ (E_1, E_2) \leftarrow D^2_{R_q,\sigma_{err}} \\
&\mathbf{C} = \left(\left[\Delta m + \mathbf{P_{key}}[0]\cdot U + E_1\right]_q, \left[\mathbf{P_{key}}[1]\cdot U + E_2\right]_q\right) \\
&\text{return } \mathbf{C}
\end{aligned}$$

Là encore, la partie complexe du calcul est l'opération de multiplication $\mathbf{P_{key}}[0] \cdot U$ et $\mathbf{P_{key}}[1] \cdot U$. Comme il s'agit d'une multiplication polynomiale simple, l'accélérateur principal présenté section 3.1 convient. Cependant, compte tenu de la forme particulière du polynôme U, utiliser l'accélérateur matériel directement sans modification implique une sous-utilisation des ressources. En effet, le polynôme U étant actuellement choisi avec des coefficients binaires, il en résulte une sous-utilisation importante des multiplieurs embarqués. Deux approches sont possibles afin d'optimiser cette opération :

(1) Réduire les ressources matérielles utilisées (et notamment le nombre de multiplieurs) ;
(2) Calculer plusieurs chiffrés en parallèle (avec potentiellement davantage de ressources matérielles utilisées) ;

Dans notre étude, il nous a semblé plus pertinent d'explorer le chiffrement de plusieurs messages en parallèle afin de gagner en parallélisme. En effet, les messages non chiffrés étant binaires, il est fort probable qu'une application utilisant le chiffrement homomorphe demande le chiffrement de plusieurs messages.

3.3.3 Stratégie d'implémentation

L'accélération du chiffrement demande les adaptations suivantes de notre architecture :

(1) Calculer les pré-traitements logiciels de la clé publique ainsi que de chaque U (un polynôme U par chiffrement) ;

(2) Envoyer l'ensemble des sous-polynômes à l'accélérateur matériel (provenant de la clé publique et des différents U) ;

(3) Calculer les derniers pré-traitements sur le composant matériel pour les sous-polynômes de la clé publique ainsi que de chaque U ;

(4) Adapter l'unité de multiplication polynomiale pour calculer quatre multiplications de sous-polynômes en parallèle ;

(5) Implémenter un post-traitement en matériel par chiffrement.

(6) Terminer les post-traitements sur l'application logicielle.

Pour le point (1), il est possible d'exploiter la petite taille des coefficients du polynôme U pour réduire le nombre d'opérations lors des pré-traitements de Karatsuba comparé à un polynôme de taille standard.

Pour le point (2), il faut distinguer deux types de coefficients :

1) Les coefficients issus de la clé publique et qui sont sur $\log_2 q$ bits ;

2) Les coefficients issus du polynôme U et qui sont sur 1 bit.

Les coefficients issus de la clé publique sont envoyés de manière classique à l'accélérateur, en découpant les coefficients en chiffres de 27 bits. En outre, pour un coefficient sur 135 bits, il faudra cinq coups d'horloge pour que l'accélérateur reçoive le coefficient complet. Les coefficients provenant du polynôme U étant bien plus petits, il est possible d'envoyer plusieurs coefficients pendant le temps de transfert d'un coefficient provenant de la clé publique. Afin de limiter la taille du chemin de données de l'accélérateur, nous nous sommes restreint à un coefficient provenant de U par coup d'horloge. Ainsi, pour $\log_2 q = 135$ bits, il est possible d'envoyer cinq coefficients pendant le transfert d'un coefficient provenant de la clé publique.

Cette restriction du parallélisme permet également de limiter la complexité du transfert des polynômes calculés après post-récursion matérielle. Il faut en effet prendre en compte que plus il y a de chiffrements en parallèle, plus il y aura de sous-polynômes générés en parallèle à la sortie de l'accélérateur matériel. Selon la taille de l'interface implémentée avec le PCI-E (128 bits ou 256 bits), il existe un nombre limite de chiffrements en parallèle avant de dépasser la largeur du bus. En particulier, par coup d'horloge, l'accélérateur produit un chiffre de 27 bits par chiffrement en parallèle. Pour quatre chiffrements, $4 \cdot 27 = 108$ bits de données sont produits par coup d'horloge. Pour cinq chiffrements, cette valeur passe à $5 \cdot 27 = 135$ bits. Ainsi, selon la taille de l'interface avec le PCI-E, le nombre limite de chiffrements en parallèle maximal est différent. Il est bien entendu possible de dépasser cette limite en séquentialisant les données reçues avant transfert sur le PCI-E, cependant nous avons fait le choix de simplifier l'architecture en limitant tout simplement le nombre de chiffrements en parallèle (coïncidant par la même occasion à l'approche réalisée pour diminuer le chemin de données des pré-traitements).

Les points (3), (4) et (5) demandent une modification mineure de l'accélérateur matériel et seront discutés par la suite.

Le point (6) a été implémenté en calculant séquentiellement les post-traitements pour chaque chiffrement.

3.3.4 Modification de l'application logicielle

3.3.4.1 Adaptation des pré-traitements

Puisque le polynôme U possède des coefficients sur un bit, il est possible de simplifier fortement le calcul des pré-traitements. La figure 3.18 présente la représentation optimisée des coefficients de U. Cette représentation permet de stocker jusqu'à quatre coefficients par segment mémoire de 32 bits en gardant sept bits de garde. Notre implémentation de Karatsuba demandant de six à sept pré-récursions en logiciel, cette représentation permet d'enchainer toutes les pré-récusions sans besoin d'une quelconque gestion de propagation de retenue. De plus, cette représentation est parfaitement compatible avec la programmation vectorielle, permettant d'additionner $8 \cdot 4 = 32$ coefficients en parallèle.

3.3.4.2 Adaptation des post-traitements

Comme précisé précédemment, les post-traitements n'ont pas besoin d'une adaptation particulière, il suffit d'appliquer l'approche classique pour chaque chiffrement.

3.3.5 Modification de l'accélérateur matériel

3.3.5.1 Adaptation de l'unité de pré-traitements

Comme précisé dans la section 3.3.3, un coefficient d'un sous-polynôme de U est envoyé par coup d'horloge à l'unité de pré-traitement. Compte tenu de la faible taille des coefficients comparé à un coefficient complet, le chemin de données a été adapté en conséquence. Les coefficients provenant de U faisant 1 bit initialement, après p récursions, ils sont définis sur $p + 1$ bits. Nous avons donc fixé la taille du chemin de données à $p + 1$ bits pour à la fois réduire au maximum le chemin de données, et garder un chiffre par coefficient. Ce choix permet également de se soustraire de la gestion d'une quelconque propagation de retenue, simplifiant l'architecture très légèrement. Le nombre de pré-récursions étant compris entre neuf et dix, selon les paramètres du schéma FV, le chemin de données sera fixé sur 10 ou 11 bits.

3.3.5.2 Adaptation de l'unité pré-crossbar

Pour la ligne des sous-polynômes de U, l'unité de pré-crossbar reste identique mais avec un chemin de données plus petit.

3.3.5.3 Adaptation de l'unité de multiplication polynomiale et de coefficients

L'unité de multiplication polynomiale reste globalement inchangée, seule l'unité de multiplication de coefficients a dû être adaptée. La figure 3.17 présente l'ordonnancement des coefficients du multiplieur de coefficients. En considérant que les différents polynômes U sont représentés par un vecteur $\mathbf{U}$, alors $a_i[j]$ représente un coefficient quelconque provenant du polynôme $\mathbf{U}[j]$. b_i représente quant à lui un coefficient provenant de la clé publique. Dans cet exemple, 4 sous-produits sont calculés en parallèle. Chaque coefficient provenant des différents $\mathbf{U}[j]$ est distribué sur une ligne de multiplieurs 27×27 bits puis multiplié par le coefficient provenant de la clé publique. Après cette multiplication, chaque ligne va ensuite être traitée séparément pour les post-traitements.

3.4 Conclusion

Dans ce chapitre, nous avons présenté une architecture de type co-conception basée sur l'algorithme de Karatsuba, ainsi que son adaptation à trois grandes primitives de FV, à savoir FV.Encrypt, FV.Mult et FV.Relin. Compte tenu de la complexité de ces opérations, la simplicité et la flexibilité de l'algorithme de Karatsuba ont été des atouts majeurs dans l'implémentation. Il reste néanmoins le fait que ces adaptations ne sont pas triviales et ont demandé un travail d'exploration et d'analyse important.

Le chapitre suivant est consacré à la présentation des résultats d'implémentation.

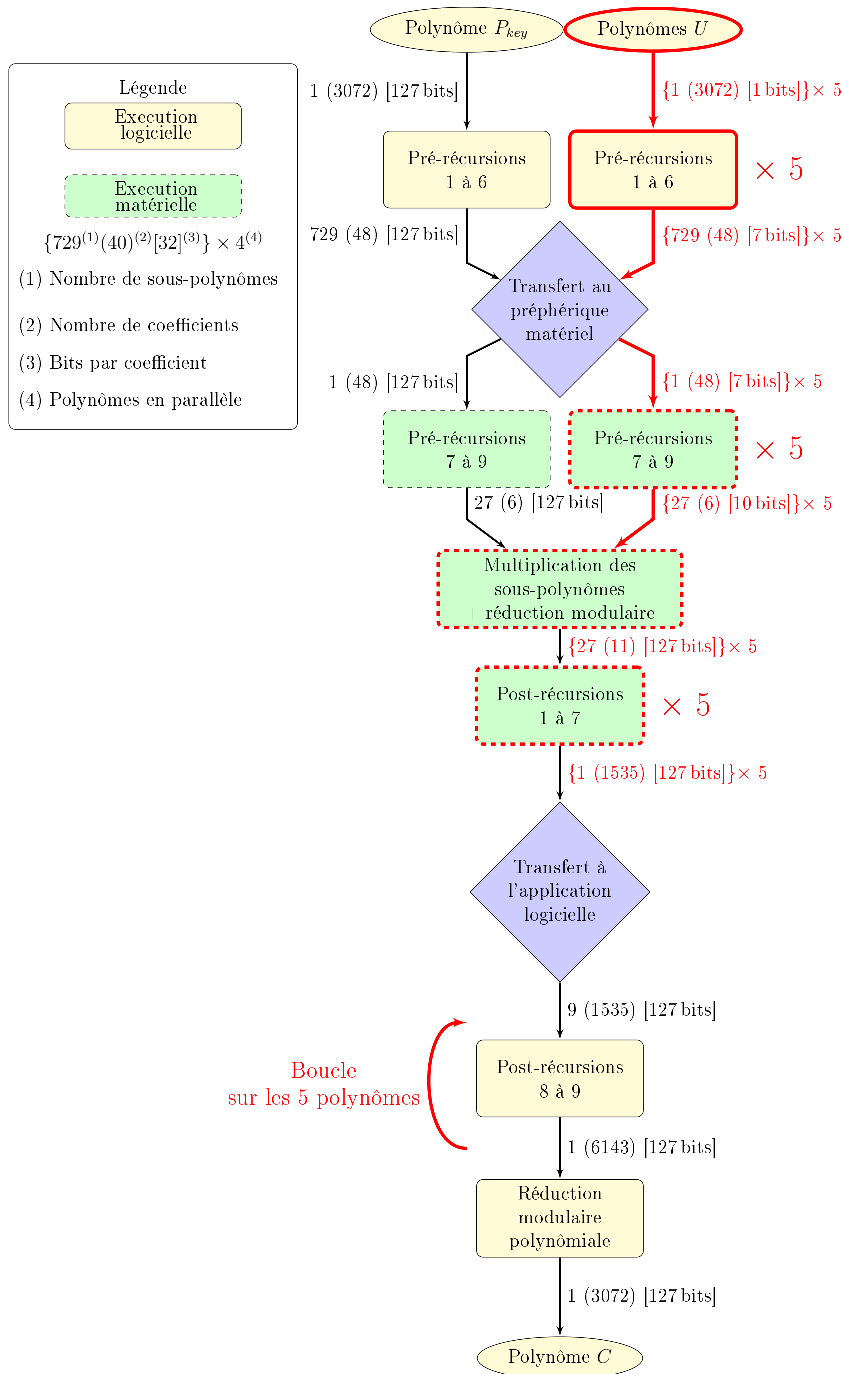


FIGURE 3.16 – Stratégie d'implémentation de l'accélération FV.encrypt avec Karatsuba dans une approche de co-conception. L'opération effectuée est $P_{key} \cdot U \bmod \Phi = C$, où P_{key} est un des membres de la clé publique, U un polynôme binaire aléatoire et Φ un polynôme cyclotomique. L'architecture est présentée pour des polynômes ayant 3072 coefficients codés sur 127 bits, avec jusqu'à cinq chiffrements en parallèle. Les modifications par rapport à l'architecture de base sont précisées en rouge.

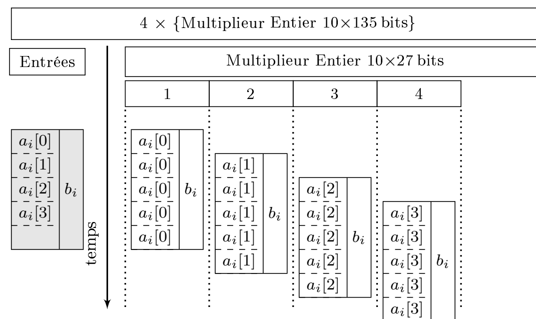


FIGURE 3.17 – Ordonnancement de 4×{Multiplieur Entier 10×135 bits}. $a_i[j]$ représente le i^{eme} coefficient du j^{eme} sous-polynôme provenant de U, et b_i le i^{eme} coefficient d'un sous-polynôme provenant de P_{key}.

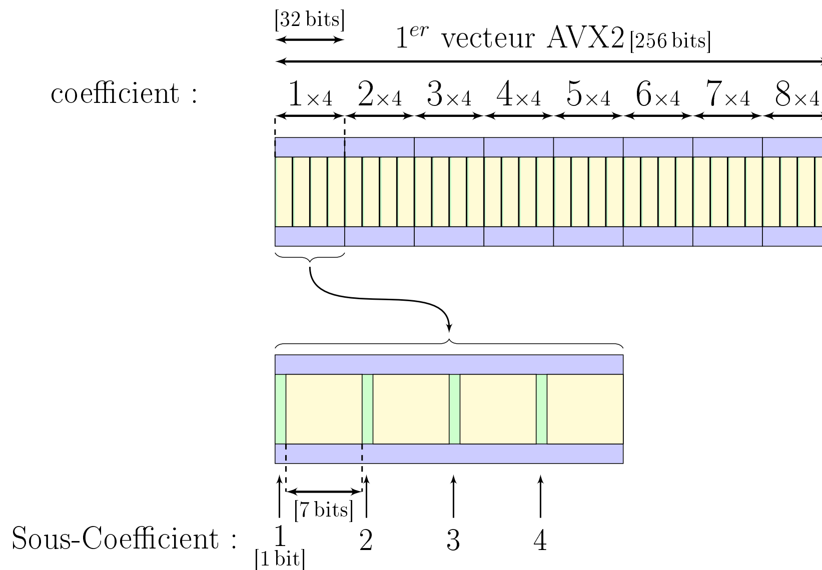


FIGURE 3.18 – Représentation en mémoire du polynôme U.

4

Résultats d'implémentation

Dans ce chapitre, nous allons présenter les résultats d'implémentation des principales primitives du schéma FV.

Nous présenterons notamment l'accélération :

1) de la réduction polynomiale et du *batching* ;
2) de la multiplication homomorphe FV.Mult et FV.Relin ;
3) du chiffrement FV.Encrypt.

Pour simplifier l'étude des résultats d'implémentation, nous nous sommes restreints à 4 profondeurs multiplicatives, à savoir les profondeurs trois, quatre, sept et huit (λ=80 bits, ω=27 bits). Ces configurations ont été choisies pour plusieurs raisons :

1) pour des profondeurs multiplicatives inférieures à trois, il existe des alternatives à FV, comme le schéma [BGN05].
2) Pour les profondeurs multiplicatives supérieures à huit, augmenter la profondeur multiplicative a un coût très significatif. Concrètement, il est nécessaire d'augmenter la taille des chiffrés de plus de 600 000 bits par profondeur multiplicative supplémentaire. Ainsi, la taille des chiffrés dépasse les 3 millions de bits pour la profondeur multiplicative neuf et 4 millions de bits pour la multiplicative dix. Nous avons fixé une limite à huit pour la profondeur multiplicative afin de ne pas avoir des chiffrés de taille déraisonnable, tout en permettant une certaine souplesse dans l'implémentation d'opérations homomorphes.

4.1 Réduction modulaire polynomiale et *batching*

Le *batching* et la réduction polynomiale sont des opérations extrêmement liées car le nombre de *batches* (nombre maximum de messages que l'on peut stocker dans un chiffré) dépend du choix du polynôme cyclotomique.

L'intérêt de notre approche basée sur Karatsuba est le fait que la multiplication polynomiale se fait sans réduction modulaire polynomiale et sans changement d'espace (à l'inverse de la NTT). Karatsuba est donc naturellement adapté à l'implémentation du *batching*. Ainsi, l'intérêt de notre approche est en partie déterminée

par l'efficacité de l'implémentation du *batching*.

Nous avons fait un travail important d'exploration afin de déterminer les meilleurs polynômes cyclotomiques permettant à la fois un nombre important de *batches* et une implémentation de la réduction modulaire polynomiale efficace. Cette exploration s'est faite en 3 étapes :

(1) nous avons créé une base de données contenant les paramètres d'un grand nombre de polynômes cyclotomiques. Cette base de données contient quatre informations essentielles par polynôme :
  a) son degré ;
  b) son indice d'Euler (pour pouvoir l'identifier de manière unique) ;
  c) le niveau de *batching* qu'il permet ;
  d) son nombre de coefficients non nuls H.

(2) Ensuite, pour une profondeur multiplicative donnée, nous déterminons la plage possible pour le degré du polynôme cyclotomique. Cette plage est déterminée de la manière suivante :
  a) la borne inférieure est positionnée sur le degré minimal que doit avoir le polynôme cyclotomique pour permettre cette profondeur multiplicative (d'après la table 1.3 des paramètres de FV) ;
  b) la borne supérieure est positionnée sur le degré maximal de la multiplication polynomiale que notre implémentation de l'algorithme Karatsuba permet (d'après la table 1.7).

(3) Nous procédons ensuite à des requêtes successives en augmentant peu à peu le niveau de *batching* possible en sélectionnant à chaque fois le polynôme avec un nombre de coefficients non nuls le plus faible.

La table 4.1 présente le résultat de ce travail d'exploration. Pour chaque profondeur multiplicative, nous avons tout d'abord extrait quatre polynômes cyclotomiques qui minimisent le nombre de coefficients non nuls du polynôme tout en maximisant le nombre de *batches*. Ensuite, nous avons ajouté un dernier polynôme cyclotomique qui permet un nombre de *batches* supérieur ou égal à quarante.

Nous pouvons remarquer qu'il existe de nombreuses configurations possibles en termes de nombre de *batches* pour chaque profondeur multiplicative. Cette variété est favorisée par le fait que l'exploration des polynômes cyclotomiques se fait sur une plage assez large de degrés. Pour la profondeur multiplicative trois, le domaine d'exploration s'étale des degrés 2317 à 2560.

Les temps d'exécutions présentés table 4.1 sont issus d'une implémentation en C++ avec l'utilisation de la programmation vectorielle. Plusieurs remarques sont à formuler à propos de ces résultats :

(1) les temps de traitements sont de l'ordre de la centaine de μs pour la très grande majorité des configurations ;

2) le nombre de coefficients non nuls H reste un estimateur fiable du temps de calcul dans la majorité des cas, bien que certaines configurations soient plus efficacement implémentées que d'autres configurations avec un H plus faible.

Pour le point (1), nous verrons dans la section 4.2.2 que le temps d'exécution de la

réduction modulaire polynomiale présentée ici est nettement inférieure à celui de la multiplication polynomiale. Cela justifie *a posteriori* l'efficacité de notre approche comparée à l'approche classique qui utilise l'algorithme de réduction de Barrett. Pour rappel, l'algorithme de réduction de Barrett demande l'équivalent de deux multiplications polynomiales pour la réduction modulaire.

Pour le point (2), le fait d'utiliser le nombre de coefficients non nuls H comme seul paramètre déterminant pour le temps d'exécution n'est pas toujours valable. Ce résultat peu intuitif est illustré en comparant les polynômes d'indices d'Euler 6615 et 3875 (pour la profondeur multiplicative quatre). Pour $m = 6615$, le temps de calcul est de $188\,\mu$s pour un H de 33, alors que pour $m = 3875$ qui a un H de 49, le temps d'exécution est de $85\,\mu$s. Pour comprendre ce phénomène, il faut retourner à la construction de notre algorithme de réduction modulaire polynomiale décrit section 2.7.2. Après avoir résolu le système d'équations, pour calculer la réduction polynomiale P' d'un polynôme P, le polynôme P' est calculé par parties, chaque partie étant calculée avec des additions et soustractions de sous-parties du polynôme P. Ces additions/soustractions de polynômes sont sujettes à la limitation décrite section 2.7.1, à savoir qu'il y a une limite au nombre d'opérations successives que l'on peut faire avant de devoir faire une propagation de retenue. Cette opération crée une dépendance de données et dégrade l'efficacité de l'implémentation. On retrouve typiquement ce cas dans l'exemple cité. Pour $m = 3875$, le cas où l'implémentation est la plus efficace, il n'y a aucune nécessité de propagation de retenue pendant les calculs intermédiaires, contrairement au cas $m = 6615$.

En conclusion, ces résultats d'implémentation montrent que la réduction modulaire de polynômes compatibles avec le *batching* peut être efficacement calculée. Avec un léger surcoût arithmétique (qui reste faible devant l'opération de multiplication polynomiale), il est possible d'obtenir de deux à soixante *batches*.

Pour la suite de l'étude, nous nous sommes intéressés à quatre polynômes cyclotomiques particuliers qui ont la particularité d'offrir un nombre de *batches* supérieur à huit (un octet), tout en ayant un temps de calcul compétitif. Ces polynômes correspondent aux lignes en bleu dans la table 4.1, et serviront de référence pour les temps de calcul des primitives de FV.

4.2 Accélération matérielle

Le gros avantage de notre architecture est de proposer une solution clé en main pour accélérer les calculs du chiffrement homomorphe. Notre approche est utilisable sur des machines standards ou des serveurs, et ne nécessite que l'installation de la plateforme FPGA pour fonctionner. Ce n'est malheureusement pas le cas pour les implémentations matérielles basées sur la NTT ([PNPM] et [SRJV$^+$]). En effet, leur approche ne prend pas en compte les transferts entre la plateforme FPGA et la machine, point pourtant essentiel dans une utilisation réelle de l'accélérateur.

Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe Vincent Migliore 2017

TABLE 4.1 – Exemples de polynômes cyclotomiques et leur temps de calcul en utilisant la démarché présentée section 1.3.4 (1000 essais), où L est la profondeur multiplicative associée pour le schéma FV, n le degré du polynôme, m son indice d'Euler, *batching* le niveau de *batching* possible et H le nombre de coefficients non nuls du polynôme.

L	n	m	*batching*	H	Temps de calcul
3	2500	6250	0	5	50 μs
	2420	3993	2	15	59 μs
	2352	5145	4	33	146 μs
	2420	2783	22	41	110 μs
	2400	4575	40	145	341 μs
4	3072	9216	0	3	54 μs
	3000	5625	2	7	68 μs
	3024	6615	12	33	188 μs
	3000	3875	30	49	85 μs
	3024	3577	48	209	157 μs
7	5000	12500	0	5	185 μs
	5000	9375	2	7	231 μs
	5000	6875	10	17	276 μs
	5040	11025	12	33	632 μs
	5000	6275	50	401	358 μs
8	5832	17496	0	3	201 μs
	5832	10935	2	7	247 μs
	5832	9477	6	17	252 μs
	5832	11025	18	49	558 μs
	6000	11625	60	73	1062 μs

4.2.1 Efficacité du lien PCI-E

L'efficacité du lien PCI-E fait également partie des points indispensables à une implémentation de type co-conception. Nous avons donc cherché à évaluer les performances de ce lien par rapport au temps de calcul des opérations matérielles (pour les profondeurs multiplicatives étudiées).

La figure 4.1 présente les résultats des temps de transfert entre l'application logicielle et le composant matériel. Trois différents résultats sont présentés :

1) « Transfert optimal » : c'est le temps de transfert qui correspond parfaitement au temps de traitement de l'accélération matérielle ;
2) « Transfert optimal RIFFA » : le temps de transfert obtenu en se référant à la documentation de RIFFA ;
3) « Transfert réel » : le temps de transfert réel mesuré sur notre machine.

Les mesures sont basées sur la version la plus performante de RIFFA, qui implémente côté FPGA, des envois de 128 bits à une fréquence de 250 MHz.

On peut remarquer que selon la documentation de RIFFA et compte tenu du volume de données à envoyer, le bus PCI-E devrait être en pratique utilisé de manière optimale. Dans nos mesures, nous avons cependant remarqué un écart assez significatif avec les bandes passantes proposées par RIFFA. Ces disparités peuvent être expliquées par plusieurs facteurs :

1) un processeur plus performant (six cœurs Intel I7 cadencés à 3.6 GHz dans [JRHK15] comparé à quatre cœurs Intel I7 cadencés à 3.2 GHz pour notre machine) ;
2) des composants de meilleure qualité (notamment une carte mère avec des meilleurs connecteurs, une meilleur compatibilité electro-magnétique, ...).

Afin d'évaluer les performances que l'on peut espérer avec notre approche de co-conception, nous avons ajouté aux résultats d'implémentation une estimation du temps de calcul en considérant que nous arriverons à atteindre les débits du PCI-E présentés dans la documentation de RIFFA [JRHK15]. Nous appellerons cette estimation (a).

La deuxième limitation de RIFFA qui va potentiellement impacter l'efficacité de notre implémentation est le fait que RIFFA n'implémente pas la configuration permettant des envois sur le FPGA de 256 bits à 250 MHz. Cette limitation va principalement jouer sur le temps de calcul de la relinéarisation, car les besoins en bande passante sont très élevés (il faut envoyer les clés de relinéarisation pendant le transfert du polynôme à relinéariser). Nous présenterons également les temps de calcul estimés en simulant cette configuration. Nous noterons cette estimation (b).

4.2.2 FV.Mult et FV.Relin

En terme de temps de calcul uniquement, l'efficacité de l'implémentation de FV.Mult et FV.Relin va dépendre de manière significative du nombre d'opérations implémentées sur le composant matériel. Implémenter davantage d'opérations sur le

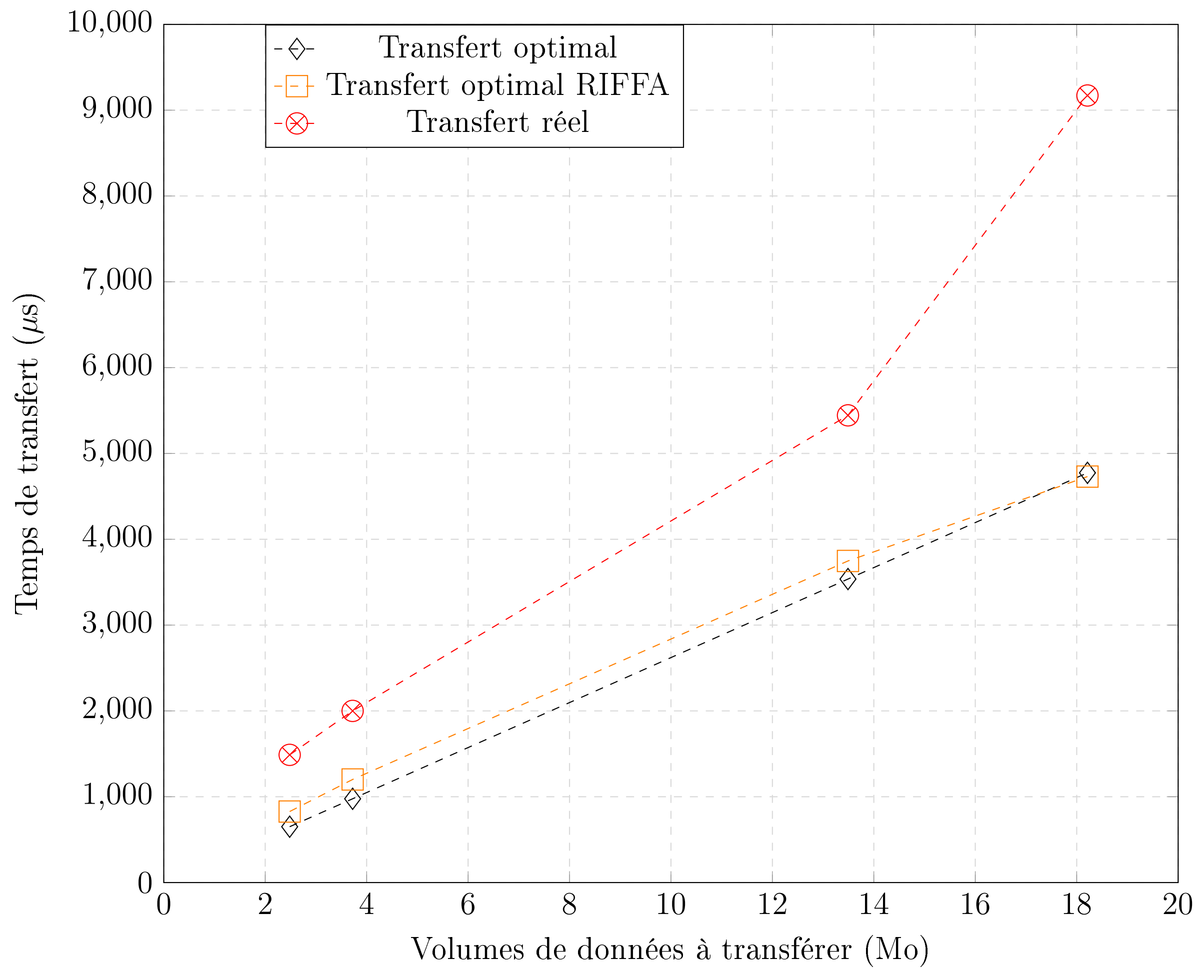


FIGURE 4.1 – Comparaison du temps de transfert annoncé dans la documentation de RIFFA, et le temps de transfert mesuré sur notre machine.

composant matériel va en effet réduire à la fois la quantité de calculs exécutés par le processeur et la tailles des transferts.

Dans notre approche, nous avons cherché à obtenir un résultat équilibré entre l'utilisation efficace des ressources matérielles, et un nombre suffisant d'opérations sur l'accélérateur matériel pour obtenir des temps de calcul compétitifs.

La table 4.2 résume les paramètres retenus pour l'implémentation des quatre profondeurs multiplicatives étudiées. Ces configurations ont l'avantage de couvrir des polynômes avec des degrés de 2317 à 5620, pour des coefficients allant de 100 bits à 239 bits (soit une taille de polynômes de 231 700 bits à 1 343 180 bits). Cela permet de montrer la flexibilité de notre accélérateur.

Les paramètres variables permettant de modifier l'arithmétique de notre accélérateur sont les suivants :

(1) le nombre de pré- et post-récursions ;
(2) le degré des sous-produits de polynômes (taille des polynômes en bas du feuillage des opérations de Karatsuba) ;
(3) le découpage logiciel/matériel des pré-traitements ;
(4) le découpage logiciel/matériel des post-traitements ;
(5) le nombre de chiffres par nombre ;
(6) le nombre de clés de relinéarisation.

Les points (1) et (2) sont déterminés à l'aide de la table 1.7, qui énumère les différentes combinaisons possibles du nombre de récursions et du degré des sous-produits de polynômes, en donnant la taille de la multiplication associée. Il suffit de prendre la combinaison qui se rapproche le plus du degré réel des polynômes à multiplier.

Le point (3) est simple à déterminer, car pour simplifier l'implémentation matérielle des opérations de Karatsuba, le nombre de pré-récursions exécutées en matériel a été fixée à trois (voir section 3.1.6.2 pour plus d'informations).

Le point (4) est déterminé en reprenant l'étude faite section 3.1.6.6, qui détermine le nombre optimal de post-récursions à implémenter sur le composant matériel pour minimiser le coût mémoire. Notons que cette étude se base sur les limitations du PCI-E, qui nous oblige à stocker temporairement les résultats des calculs de l'accélérateur matériel avant de les envoyer. Pour d'autres liens de communication qui ne demanderaient pas une telle restriction, un travail exploratoire serait nécessaire pour déterminer le découpage optimal.

Pour le point (5), il s'agit d'un simple calcul. Comme les unités arithmétiques matérielles implémentées sont définies sur 27 bits (pour utiliser efficacement les multiplieurs embarqués qui fonctionnent sur 27 bits), le nombre de chiffres est obtenu en divisant le nombre de bits d'un coefficient ($\log_2 q$) par 27.

Pour le point (6), il s'agit également d'un simple calcul. Comme nous avons fixé le paramètre ω du schéma FV sur 27 bits d'après l'étude proposée section 3.2.3, le

Table 4.2 – Configuration de l'accélérateur matériel pour les profondeurs multiplicatives 3, 4, 7 et 8. (tailles et volumes représentés en bits)

L		3	4	7	8
Nombre de récursions		9		10	
Nombre de pré-récursions	logicielles	6		7	
	matérielles	3			
Nombre de post-récursions	logicielles	2			
	matérielles	7		8	
Degré des sous-polynômes		4	5	4	5
Chiffres par nombre		4	5	8	9
Clés de relinéarisation		4	5	8	9
Degré maximal de la multiplication polynomiale		2559	3071	5119	6143
Taille maximale des coefficients		108	135	216	243
Nombre de sous-polynômes après pré-traitements logiciels		729		2187	
Nombre de coefficients par sous-polynômes après pré-traitements logiciels		40	48	40	48
Volume de données à envoyer à l'accélérateur matériel		14 929 920	22 394 880	89 579 520	120 932 352
Nombre de sous-polynômes après post-traitements matériels		9			
Nombre de coefficients par sous-polynômes après post-traitements matériels		1279	1535	2559	3071
Volume de données à envoyer à l'application logicielle		5 893 632	8 841 600	23 583 744	31 840 128

Cybersécurite matérielle et conception de composants dédiés au calcul homomorphe Vincent Migliore 2017

nombre de clés de relinéarisation $l_{\omega,q}$ est fixé par :

$$l_{\omega,q} = \left\lceil \frac{\log_2 q}{\log_2 \omega} \right\rceil = \left\lceil \frac{\log_2 q}{27} \right\rceil$$

La figure 4.2a présente graphiquement les temps de calcul de la multiplication homomorphe pour les quatre profondeurs multiplicatives étudiées, ainsi qu'une estimation des temps de calcul avec les estimations (a) et (b) présentées section 4.2.1 (sans *batching*), et la figure 4.2b présente ces mêmes résultats mais en considérant huit *batches*. Les tables 4.3, 4.4, 4.5 et 4.6 présentent quant à elles les ressources matérielles utilisées pour chaque profondeur multiplicative et la valeur numérique du temps de calcul de la multiplication homomorphe. Enfin, la table 4.7 présente l'ensemble des résultats d'implémentations (cette étude + état de l'art).

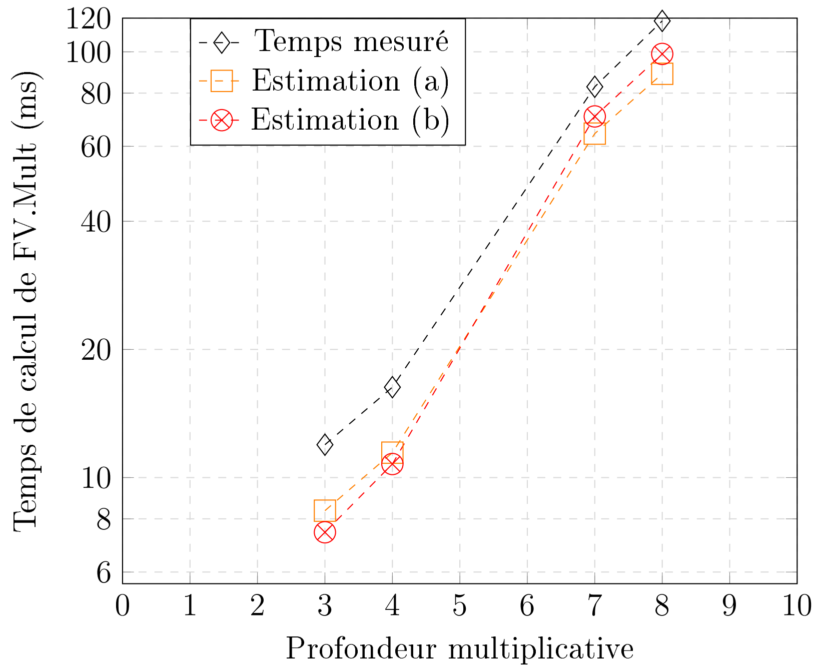
Nous avons eu quelques difficultés à comparer nos résultats d'implémentation avec l'état de l'art pour plusieurs raisons :

(1) il existe un nombre assez faible d'implémentations matérielles accélérant la multiplication homomorphe de manière complète ;
(2) les implémentations matérielles ne couvrent pas assez de profondeurs multiplicatives (donc des paramètres) différentes ;
(3) les implémentations matérielles sont basées sur le schéma de chiffrement YASHE' (mais une correspondance est possible avec FV) ;
(4) notre implémentation est compatible avec l'utilisation du *batching*, ce qui n'est pas le cas de la quasi totalité des implémentations (matérielles ou logicielles).

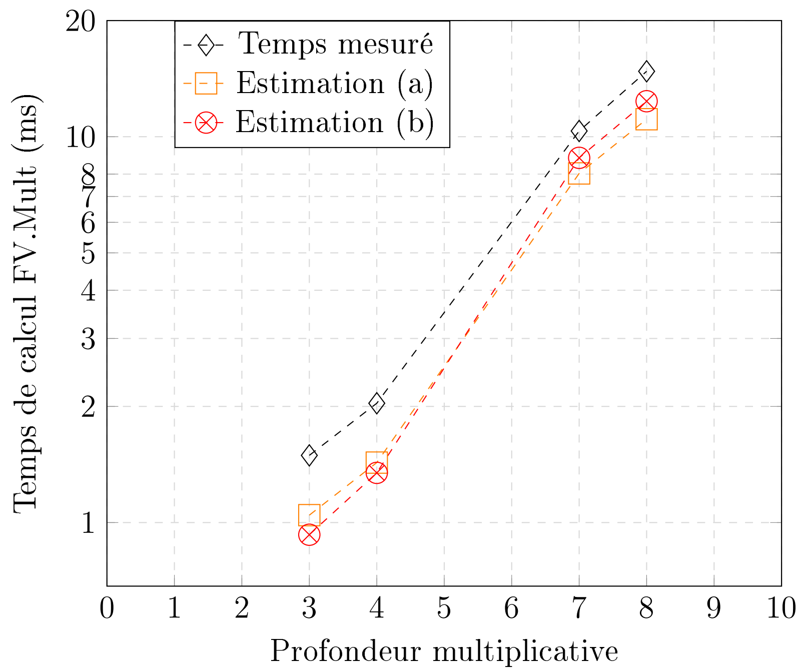
Le point le plus décisif dans notre implémentation étant le *batching*, le choix du critère de comparaison avec les autres implémentations non compatibles avec cette fonctionnalité était décisif. Ainsi, diviser le temps de calcul de la multiplication homomorphe par le nombre de *batches* puis comparer le résultat avec les implémentations existantes (sans *batching*) est-il pertinent ? La réponse n'est pas forcément oui car bien que le nombre de *batches* peut effectivement paralléliser un certain nombre d'opérations homomorphes, encore faut-il avoir besoin d'une telle parallélisation. Pour avoir une idée plus raisonnable de l'efficacité de notre accélérateur vis-à-vis du *batching*, nous avons donc décidé de donner le temps de calcul par multiplication homomorphe en considérant huit *batches* (soit un octet).

Afin d'obtenir une analyse plus fine des ressources matérielles utilisées, les contributions de trois unités sont présentées :

1) « RIFFA + Interface PCI-E » : contient le composant d'interface d'Altera qui permet de communiquer avec le PCI-E, plus l'implémentation de RIFFA qui permet de transformer les trames PCI-E en communication série de 128 bits à 250 MHz.
2) « Interface RIFFA/Karatsuba » : contient les différentes mémoires tampon afin d'amortir les problèmes de débit sur le bus PCI-E.
3) « Karatsuba » : l'accélérateur matériel.



(a) Temps de calcul sans considérer de *batching*.



(b) Temps de calcul en considèrant un *batching* de 8.

FIGURE 4.2 – Temps de calcul de la multiplication homomorphe de FV avec notre accélérateur basé sur Karatsuba. Les temps de calculs avec les estimations (a) et (b) (estimations présentées section 4.2.1) sont également ajoutés pour information. La figure 4.2a présente les résultats sans considérer de *batching*, tandis que la figure 4.2b présente les résultats en supposant huit *batches*.

Table 4.3 – Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (2560, 108)$. Les temps de calcul sont donnés pour 1000 essais.

$(n, \log_2 q)$	(2560, 108)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	TOTAL
ALM	8 148	186	31 302	39 801
Registres	11 350	191	76 964	88 644
Mémoire (bits)	697 720	1 992 704	868 130	3 558 554
DSPs	0	0	80	80
f_{max}	250 MHz (limitation du PCI-E)			
Temps de calcul				
FV.Mult	11.95 ms			
FV.Mult (estimation (a))	8.36 ms			
FV.Mult (estimation (b))	7.44 ms			

Table 4.4 – Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (3072, 135)$. Les temps de calcul sont donnés pour 1000 essais.

Setup $(n, \log_2 q)$	(3072, 135)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	Total
ALM	8 156	230	41 387	49 888
Registres	11 367	206	103 656	115 368
Mémoire (bits)	697 520	2 215 936	1 904 478	4 818 134
DSPs	0	0	120	120
f_{max}	250 MHz (limitation du PCI-E)			
Temps de calcul				
FV.Mult	16.30 ms			
FV.Mult (estimation (a))	11.44 ms			
FV.Mult (estimation (b))	10.76 ms			

Table 4.5 – Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (5120, 216)$. Les temps de calcul sont donnés pour 1000 essais.

Setup $(n, \log_2 q)$	(5120, 216)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	Total
ALM	8,100	234	57,439	65,938
Registres	11 364	209	136 727	148 438
Mémoire (bits)	697 720	3 985 408	7 244 062	11 927 190
DSPs	0	0	160	160
f_{max}	250 MHz (limitation du PCI-E)			
Temps de calcul				
FV.Mult	82.78 ms			
FV.Mult (estimation (a))	64.26 ms			
FV.Mult (estimation (b))	70.53 ms			

Table 4.6 – Résultats d'implémentation de la multiplication homomorphe pour la configuration $(n, \log_2 q) = (6144, 243)$. Les temps de calcul sont donnés pour 1000 essais.

Setup $(n, \log_2 q)$	(6144, 243)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	Total
ALM	6 513	471	73 430	80 495
Registres	11 330	233	177 384	189,086
Mémoire (bits)	697 720	14 602 240	3 862 906	19 162 866
DSPs	0	0	216	216
f_{max}	250 MHz (limitation du PCI-E)			
Temps de calcul				
FV.Mult	118.20 ms			
FV.Mult (estimation (a))	88.80 ms			
FV.Mult (estimation (b))	98.87 ms			

4.2.2.1 Comparaison des résultats d'implémentation avec les implémentations matérielles basées sur la NTT

Un des avantages essentiels de notre architecture comparée aux implémentations matérielles [PNPM] et [SRJV+] basées sur la NTT est le fait que notre accélérateur ne nécessite aucune mémoire externe câblée sur la plateforme FPGA. Cet avantage permet à la fois de simplifier considérablement l'architecture du composant et de supprimer un des goulots d'étranglement majeur des implémentations [PNPM] et [SRJV+]. Il faut donc garder à l'esprit que notre approche utilise une quantité de mémoire bien inférieure aux implémentations basées sur la NTT, et que les quantités de mémoire exposées dans les tables de résultats ne représentent que la mémoire utilisée dans le FPGA.

Le principal problème que nous avons rencontré pour la comparaison de notre accélérateur avec l'état de l'art est le fait que les couples de paramètres $(n, \log_2 q)$ ne correspondent pas systématiquement aux couples de paramètres implémentés dans les solutions à base de NTT. En particulier, les implémentations [PNPM] et [SRJV+] couvrent les jeux de paramètre (4096, 125), (16384, 512) et (32768, 1228), tandis que notre approche couvre (2560, 108), (3072, 135), (5120, 216) et (6144, 243).

La raison principale de ces choix est le fait que la NTT oblige à fixer le degré de la multiplication polynomiale à une puissance de deux. Ces implémentations se sont donc calées sur des profondeurs multiplicatives requérant un degré proche d'une puissance de deux.

Seul un jeu de paramètres coïncide parfaitement avec une configuration de notre accélérateur, à savoir (4096, 125) implémenté dans [PNPM] qui correspond fonctionnellement à la configuration (3075, 135). Nous nous sommes donc basés sur cette implémentation pour réaliser notre principale comparaison.

Pour les temps de calcul, nous pouvons remarquer que notre approche obtient des résultats proches de l'implémentation de la NTT (16.30 ms pour notre approche contre 15.46 ms pour la NTT). Cependant, rappelons que le temps de transfert entre la machine et l'accélérateur n'est pas inclus dans le cas de la NTT. Ainsi, nous pouvons donc supposer que notre accélérateur obtiendra des résultats réels sensiblement plus performants que la NTT.

En revanche, pour les ressources matérielles, notre implémentation est très nettement plus performante que la NTT. En plus de ne nécessiter aucune mémoire externe câblée sur la plateforme FPGA, notre approche réduit de 28% les ressources logiques (ALM), de 20% l'utilisation de registres, de 40% l'utilisation de la mémoire embarquée, de 17% l'utilisation des DSP, et avec une fréquence de fonctionnement atteignant 250 MHz.

De plus, notre approche permet l'utilisation du *batching*, ce qui améliore d'autant plus les résultats d'implémentation. Ainsi, pour huit *batches*, notre accélérateur est capable de calculer la multiplication homomorphe de FV en 2 ms, donnant un facteur d'accélération de 7.8 par rapport à l'approche NTT.

En ce qui concerne le comportement asymptotique de notre solution, on peut remarquer que la NTT rattrape rapidement Karatsuba. En effet, la configuration (6144, 243) de notre accélérateur et la configuration (16384, 512) obtiennent des temps de calcul similaires alors que le module q est bien inférieur dans notre cas. Cependant, même pour ce jeu de paramètres, Karatsuba reste une alternative crédible car il permet l'implémentation du *batching* et utilise très nettement moins de ressources que la NTT (pas de mémoire externe câblée notamment). En revanche, pour le jeu de paramètres (32768, 1228) implémenté dans [SRJV+], la tendance est clairement à l'avantage de la NTT, qui propose un temps de calcul de 263.40 ms pour la multiplication homomorphe, qui risque d'être difficilement atteignable pour l'algorithme de Karatsuba.

4.2.2.2 Comparaison des résultats d'implémentation avec l'implémentation logicielle FV-FULL-RNS

La bibliothèque logicielle FV-FULL-RNS [BEHZ16], basée sur le système de représentation des nombres RNS, est un compétiteur très sérieux de notre accélérateur. Alors que notre accélérateur obtient de très bonnes performances comparées aux implémentations matérielles de la NTT pour des degrés plus petits que 4096, FV-FULL-RNS obtient des résultats impressionnants (multiplication homomorphe en 7.69 ms pour le jeu de paramètres (4096,186)) pour ces degrés. Cependant, son incompatibilité avec le *batching* rend notre approche plus performante en utilisant cette fonctionnalité.

Pour la profondeur multiplicative quatre (configuration de notre accélérateur (3072, 135)), notre accélérateur devient compétitif dès l'implémentation de trois *batches*. Pour huit *batches*, notre implémentation permet un facteur d'accélération de 3.7. Ce facteur passe à 5.7 en considérant l'estimation (b).

Pour les profondeurs multiplicatives sept et huit, la comparaison est moins immédiate car il n'existe pas de résultats d'implémentation pour ces profondeurs. Notamment, la NTT à implémenter a une taille de 8192, et pour une telle taille de NTT dans les résultats d'implémentation de FV-FULL-RNS, le module associé est sur 372 bits. Afin d'avoir une comparaison plus fiable avec notre approche, nous avons estimé le temps de calcul pour un module plus petit. Comme FV-FULL-RNS implémente RNS (la multiplication d'entiers se fait par multiplication terme-à-terme des moduli), il suffit de faire une mise à l'échelle en prenant le nombre de moduli adéquat. FV-FULL-RNS utilise des modulis sur 62 bits (soit six modulis pour un module q sur 372 bits), il faut donc multiplier le temps de calcul par $4/6$ pour avoir une estimation du temps de calcul avec un module q sur 248 bits (ce qui donne un temps de calcul de 25.16 ms pour la multiplication homomorphe de FV-FULL-RNS).

Pour une profondeur multiplicative de sept, il est nécessaire d'implémenter quatre *batches* avec notre accélérateur pour dépasser les performances de FV-FULL-RNS. Pour huit *batches*, le facteur d'accélération est de 2.43. En considérant l'estimation (a), ce facteur passe à 3.13.

Pour une profondeur multiplicative de huit, le nombre de *batches* passe à cinq pour dépasser les performances de FV-FULL-RNS. Pour huit *batches*, le facteur d'accélération est de 1.7. En considérant l'estimation (a), ce facteur passe à 2.26.

Ces résultats peuvent être améliorés si on implémente davantage de *batches* dans notre solution.

4.2.2.3 Remarque sur les estimations (a) et (b)

À propos des estimations (a) et (b), nous avons remarqué que l'estimation (b), qui est censée être une optimisation de l'estimation (a), ne donne pas toujours des meilleurs résultats. En particulier, une inversion de leur performance est à noter pour les profondeurs multiplicatives sept et huit. Il faut garder en mémoire que l'application logicielle doit préparer la mémoire tampon avant d'envoyer le tout à l'accélérateur matériel. Ainsi, augmenter la taille de cette mémoire va devenir contre-productif pour des polynômes de grande taille car l'application aura à remplir une mémoire tampon s'étalant sur un grand espace mémoire. Comme la même taille de mémoire tampon est utilisée pour FV.Mult et FV.Relin, les calculs de FV.Mult pénalisent le calcul complet de la multiplication homomorphe.

4.2.3 FV.Encrypt

La comparaison de notre accélération de FV.Encrypt avec l'état de l'art a été également complexe. Cela peut s'expliquer pour plusieurs raisons :

1) le cœur de bataille des implémentations actuelles est principalement tourné vers l'accélération de FV.Mult et FV.Relin ;
2) pour un chiffrement complet, cela demande la génération de polynômes dont les coefficients sont distribués selon une gaussienne discrète, ce qui peut être un frein à l'implémentation.

De plus, pour avoir une comparaison fiable, il fallait également se comparer à une solution compatible avec le *batching*. Pour ce faire, nous nous sommes tournés vers NFLlib [AMBG+16], qui est suffisamment flexible pour implémenter des opérations avec *batching*. Plus précisément, comme notre implémentation accélère le calcul du produit/accumulation utilisé pendant l'opération FV.Encrypt ($P_{key} \cdot U + E$), nous avons implémenté cette opération avec NFLlib. L'algorithme 1 présente cette implémentation.

Dans notre approche, on considère que la clé publique P_{key} est déjà dans la bonne forme (RNS + NTT). Cela se comprend bien en considérant qu'une clé publique peut servir pour différentes applications homomorphes (elle peut donc être stockée sous cette forme pour une réutilisation future). Pour le chiffrement, après avoir généré les polynômes U et E, il faut calculer les restes RNS des polynômes U et E, ainsi que leur NTT. Ensuite le calcul du produit/accumulation peut être effectué dans le domaine de la NTT. Pour calculer le résultat final, il faut appliquer une opération

TABLE 4.7 – Résumé des résultats d'implémentation pour la multiplication homomorphe de FV.

Source	$(n, \log_2 q)$	Algorithme	RNS	*Batching*	FPGA	ALM Slice LUTs	Registres	Bits de mémoire	DSPs	Temps de calcul
Cette étude	(2560,108)	Karatsuba	non	oui	Stratix-V	39 801	88 644	3 558 554	80	11.95 ms
	(3072,135)	Karatsuba	non	oui	Stratix-V	49 888	115 368	4 818 134	120	16.30 ms
	(5120,216)	Karatsuba	non	oui	Stratix-V	65 938	148 438	11 927 190	160	82.78 ms
	(6144,243)	Karatsuba	non	oui	Stratix-V	80 495	189 086	19 162 866	216	118.20 ms
[PNPM]	(4096,125)	NTT NWC	non	non	Stratix-V	69 058	144 747	8 031 568	144	15.46 ms
	(16384,512)	NTT NWC	non	non	Stratix-V	141 090	391 773	17 626 400	577	122.30 ms
[SRJV+]	(32768,1228)	NTT	oui	oui	Virtex-7	377 120	323 120	26 247 168	1792	263.40 ms
FV-FULL-RNS [BEHZ16]	(2048,90)	NTT NWC	oui	non						2.71 ms
	(4096,186)	NTT NWC	oui	non						7.69 ms
	(8192,372)	NTT NWC	oui	non						37.2 ms
	(16384,744)	NTT NWC	oui	non						206.51 ms
	(32768,1550)	NTT NWC	oui	non						1 406.96 ms

Algorithm 1 Algorithme de calcul du produit/accumulation utilisé pendant l'opération FV.Encrypt en utilisant NFLlib

Require: P_{key} (sous forme RNS + NTT), U et E
1: $\widetilde{U} \leftarrow \mathrm{NTT}(\mathrm{RNS}(U))$
2: $\widetilde{E} \leftarrow \mathrm{NTT}(\mathrm{RNS}(E))$
3: $\widetilde{R} \leftarrow P_{key} \cdot \widetilde{U} + \widetilde{E}$
4: $R \leftarrow \mathrm{invRNS}(\mathrm{invNTT}(\widetilde{R}))$
5: **return** R

TABLE 4.8 – Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (2560, 108)$. Les temps de calcul sont donnés pour 1000 essais.

$(n, \log_2 q)$	(2560, 108)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/Karatsuba	Karatsuba	TOTAL
ALM	8 148	283	57 248	65 769
Registres	11 325	192	80 763	92 390
Mémoire (bits)	697 720	8 464 384	2 689 852	11 851 956
DSPs	0	0	80	80
f_{max}	250 MHz (Limitation du PCI-E)			
Temps de calcul				
quatre produits/accumulations	2.482 ms			

de NTT inverse et de RNS inverse.

La figure 4.3 compare graphiquement les temps de calcul du produit/accumulation de NFLlib et de notre accélérateur, pour les quatre profondeurs multiplicatives étudiées. Les tables 4.8, 4.9, 4.10 et 4.11 présentent quant à elles les ressources matérielles utilisées pour chaque profondeur multiplicative et la valeur numérique du temps de calcul de quatre produits/accumulations nécessaires pendant FV.Encrypt. Enfin, la table 4.12 présente l'ensemble des résultats d'implémentations.

Comme pour la présentation des résultats d'implémentation de FV.Mult et FV.Relin, les contributions des unités « RIFFA + Interface PCI-E », « Interface RIFFA/Karatsuba » et « Karatsuba » sont séparées. La différence principale avec l'accélération de la multiplication homomorphe est le fait que maintenant, les post-traitements se font sur quatre polynômes en parallèle. Les ressources mémoires pour les post-traitements sont ainsi multipliées par quatre par rapport à l'accélérateur initial. Ce point a été particulièrement limitant pour la profondeur multiplicative sept (resp. huit), qui a nécessité de découper le transfert en trois (resp. neuf) sous-

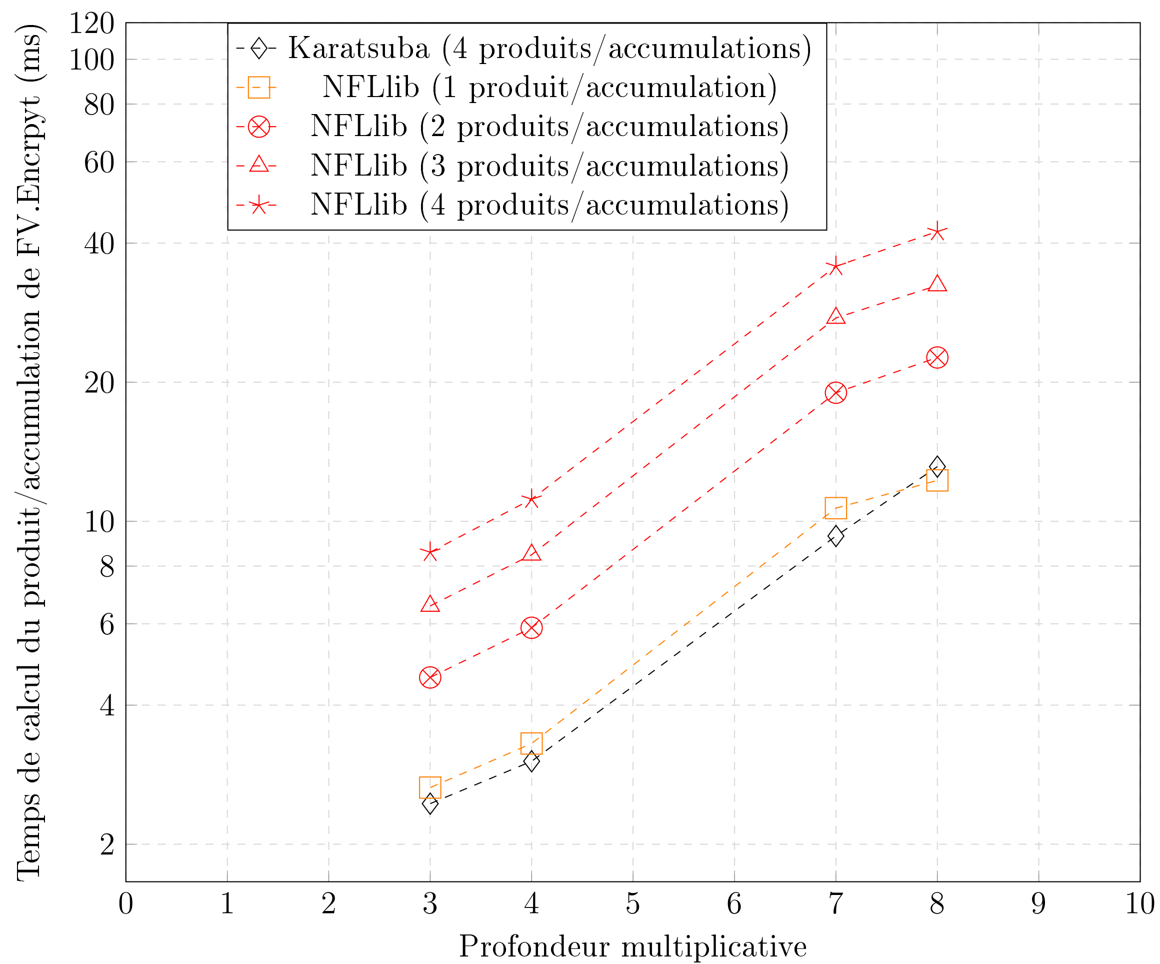


FIGURE 4.3 – Temps de calcul de un à quatre produits/accumulations successifs dans FV.Encrypt.

Table 4.9 – Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (3072, 135)$. Les temps de calcul sont donnés pour 1000 essais.

$(n, \log_2 q)$	(3072, 135)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	Total
ALM	8 149	281	72 848	81 372
Registres	11 325	199	92 469	104 104
Mémoire (bits)	697 720	8 540 160	6 277 356	15 152 236
DSPs	0	0	96	96
f_{max}	250 MHz (Limitation du PCI-E)			
Temps de calcul				
quatre produits/accumulations	3.024 ms			

Table 4.10 – Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (5120, 216)$. Les temps de calcul sont donnés pour 1000 essais.

$(n, \log_2 q)$	(5120, 216)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	Total
ALM	8 149	285	93 762	102 285
Registres	11 325	198	88 036	99 669
Mémoire (bits)	697 728	8 540 160	26 738 676	35 976 556
DSPs	0	0	160	160
f_{max}	250 MHz (Limitation du PCI-E)			
Temps de calcul				
quatre produits/accumulations	9.292 ms			

TABLE 4.11 – Résultats d'implémentation de la configuration de FV $(n, \log_2 q) = (6144, 243)$. Les temps de calcul sont donnés pour 1000 essais.

$(n, \log_2 q)$	(6144, 243)			
Ressources matérielles				
	RIFFA + Interface PCI-E	Interface RIFFA/ Karatsuba	Karatsuba	Total
ALM	8 149	224	109 731	118 186
Registres	11 325	199	96 819	108 453
Mémoire (bits)	697 720	570 856	12 541 252	17 584 828
DSPs	0	0	216	216
f_{max}	250 MHz (Limitation du PCI-E)			
Temps de calcul				
quatre produits/accumulations	9.292 ms			

transferts afin de réduire la mémoire tampon nécessaire en sortie de l'accélérateur matériel. L'utilisation d'une mémoire externe était possible, cependant à la vue des limitations rencontrées par les accélérateurs utilisant l'algorithme de la NTT sur ce point, nous avons écarté cette possibilité.

En ce qui concerne les temps de calcul (d'après la figure 4.3), on remarque que notre approche permet de calculer quatre fois plus de produits/accumulations que l'approche basée sur la NTT, et ce pour les quatre profondeurs multiplicatives étudiées. Ceci démontre une nouvelle fois que l'algorithme de Karatsuba, même pour des polynômes sur quelques millions de bits, reste un concurrent sérieux à la NTT.

On note cependant une inflexion de cette tendance qui débute très légèrement à partir de la profondeur multiplicative huit. Ainsi, la NTT reste l'algorithme qui asymptotiquement devient plus efficace. Nous n'avons cependant pas eu l'occasion de déterminer plus finement cette inflexion, car il n'a pas été possible avec notre approche (accélérateur flexible) d'aller tester des profondeurs multiplicatives supérieures (limite de ressources mémoires embarquées sur le FPGA), et aurait demandé des optimisations supplémentaires pour y parvenir.

4.3 Conclusion

Dans ce chapitre, nous avons présenté les résultats d'implémentation de notre accélérateur de type co-conception basé sur l'algorithme de Karatsuba. Nous avons démontré que l'opération du chiffrement homomorphe compatible avec le *batching* peut efficacement être implémentée et accélérée à l'aide de l'algorithme de Karat-

Table 4.12 – Résumé des résultats de l'implémentation du produit/accumulation nécessaire lors du chiffrement de FV (quatre produits/accumulations).

Source	(n,$\log_2 q$)	Algorithme	RNS	*Batching*	FPGA	ALM Slice LUTs	Registres	Bits de mémoire	DSPs	Temps de calcul
Cette étude	(2560,108)	Karatsuba	non	oui	Stratix-V	65 769	92 769	11 851 956	80	2.48 ms
	(3072,135)	Karatsuba	non	oui	Stratix-V	81 372	104 104	15 152 236	96	3.02 ms
	(5120,216)	Karatsuba	non	oui	Stratix-V	102 285	99 669	35 976 556	160	9.29 ms
	(6144,243)	Karatsuba	non	oui	Stratix-V	118 186	108 453	17 584 828	216	13.13 ms
NFLlib [AMBG+]	(2048,90)	NTT NWC	oui	non						8.57 ms
	(4096,186)	NTT NWC	oui	non						11.17 ms
	(8192,372)	NTT NWC	oui	non						35.65 ms
	(16384,744)	NTT NWC	oui	non						42.38 ms
	(32768,1550)	NTT NWC	oui	non						1 406.96 ms

suba.

Pour la multiplication homomorphe, grâce à l'utilisation du *batching*, notre accélérateur permet, pour les profondeurs multiplicatives étudiées (trois, quatre, sept et huit) d'être plus performant que les différentes implémentations existantes (performance dépendante du nombre de *batches* implémentés).

Pour l'accélération du chiffrement, la NTT souffre principalement du double coût du calcul de restes RNS et de la NTT, rendant notre accélérateur quatre fois plus rapide.

Nous avons cependant remarqué que, pour les profondeurs multiplicatives supérieures à huit, l'algorithme de la NTT semble prendre l'ascendant sur l'algorithme de Karatsuba, aidé par sa plus faible complexité asymptotique.

5 Conclusion de l'étude et perspectives

5.1 Conclusion

Nous avons vu au cours de notre étude que le chiffrement homomorphe possède encore aujourd'hui de nombreuses barrières technologiques à son utilisation pratique. Les chiffrés peuvent atteindre quelques millions de bits par bit de message, impliquant des volumes importants à transférer sur le réseau et des temps de calcul pour la multiplication homomorphe pouvant aller de quelques ms pour des petites profondeurs multiplicatives à quelques centaines de ms pour des profondeurs multiplicatives plus grandes. De plus, le nombre d'opérations élémentaires (additions, soustractions et multiplications homomorphes) est important car pour des opérateurs non-élémentaires (comparaison, seuil,..), il est nécessaire de représenter les données au niveau bit puis de reconstruire ces opérateurs avec des opérations élémentaires NOT, XOR et AND. Ainsi, pour une simple opération de tri par ordre croissant de dix valeurs sur quatre bits, 1620 additions (XOR) et 1350 multiplications (AND) homomorphes sont requises d'après [FSF+13].

Dans notre étude, nous nous sommes intéressés plus particulièrement au *batching* car il permet d'inclure plusieurs messages binaires dans un même chiffré et faire que les calculs s'exécutent en parallèle sur chacun des messages. Cela permet d'apporter une double réduction, celle de l'expansion du chiffré (c'est-à-dire le rapport entre la taille du chiffré et la taille du message), et des temps de calcul. Par extension, cela permet également d'optimiser l'utilisation de la bande passante du réseau.

En étudiant plus précisément l'état de l'art du chiffrement homomorphe, nous avons remarqué cependant que les implémentations actuelles possédaient quelques limitations vis-à-vis du *batching*, notamment à cause de l'utilisation de l'algorithme de la NTT. Cet algorithme possède en effet deux importantes limitations :

1) la NTT s'implémente efficacement dans une configuration qui ne permet pas le *batching* sur des messages binaires (c-à-d la NTT NWC) ;
2) lors de la multiplication polynomiale, la NTT est très souvent surdimensionnée car elle implémente la multiplication pour des polynômes de degré d'une puissance de deux, alors que les degrés réels des polynômes sont variés (voir la table 1.3).

Dû à ces limitations, nous nous sommes intéressés à un algorithme concurrent, à savoir l'algorithme de Karatsuba. Cet algorithme à l'avantage d'implémenter des multiplications polynomiales plus proches des degrés réels des polynômes, et comme la multiplication polynomiale se fait sans changer d'espace (contrairement à la NTT), il est naturellement compatible avec le *batching*. Nous avons donc évalué l'intérêt de Karatsuba pour différentes configurations de degrés et de taille de coefficients comparé à la NTT.

Notre étude a abouti à l'implémentation de Karatsuba dans une architecture de type co-conception. Ce choix s'est révélé très pertinent car les résultats de nos implémentations chapitre 4 montrent que pour des profondeurs multiplicatives inférieures ou égales à huit, l'algorithme de Karatsuba permet d'obtenir de meilleurs temps de calcul pour le chiffrement et pour la multiplication homomorphe que les implémentations basées sur la NTT. Le *batching* peut être implémenté sans surcoût important, et permet des facteurs d'accélération des temps de calcul qui deviennent de plus en plus significatifs à mesure que l'on augmente le nombre de *batches* utilisés, tout en optimisant l'utilisation de la bande passante du réseau.

En comparaison aux implémentations matérielles existantes utilisant la NTT, pour des polynômes de 3072 coefficients codés sur 135 bits, nous obtenons des temps de calcul pour la multiplication homomorphe proches de la NTT (16 ms), mais avec une réduction de 28% du nombre d'ALM, de 20% du nombre de registres, de 40% de mémoire embarquée et de 17% de DSP ; tout en ne nécessitant pas l'utilisation de mémoire externe câblée au niveau de l'accélérateur matériel, et en permettant l'utilisation du *batching*. Pour des polynômes plus grands (avec plus de 6144 coefficients), la comparaison est plus difficile à cause du manque d'implémentations matérielles. Cependant, nous pouvons remarquer que la NTT reprend légèrement l'ascendant sur notre approche en termes de temps de calcul, mais notre approche reste une bonne alternative à la NTT grâce à l'implémentation du *batching*.

En comparaison aux implémentations logicielles existantes utilisant la NTT et en comparaison à la bibliothèque la plus performante actuellement ([BEHZ16]), notre approche permet d'obtenir des facteurs d'accélération de 3.7 pour le jeu de paramètres $(n, \log_2 q) = (3072{,}135)$, de 2.43 pour le jeu de paramètres (5120,216) et de 1.7 pour le jeu de paramètres (6144,243), et ce en n'implémentant qu'un *batching* de huit.

5.2 Perspectives

Il est bien entendu toujours possible d'améliorer, ou de raffiner l'accélérateur matériel. Cependant, les excellents temps de calcul de la multiplication homomorphe de FV dans [BEHZ16], qui pour rappel propose une implémentation purement logicielle et très optimisée du chiffrement homomorphe FV basée sur RNS et la NTT NWC, ouvre de nombreuses portes d'exploration. Ainsi, il serait très intéressant d'évaluer l'intérêt de RNS pour des implémentations matérielles, que ce soit dans le cas de

Karatsuba que de la NTT. RNS permet en effet de paralléliser les additions, les soustractions mais surtout les multiplications. Comme le chiffrement homomorphe demande des tailles de coefficients sur plusieurs centaines de bits, RNS permettrait de pouvoir très efficacement paralléliser les calculs de multiplication des coefficients.

Bien que notre accélérateur soit optimisé pour l'accélération du schéma de chiffrement FV, compte tenu d'une arithmétique proche entre les schémas de chiffrements basés sur R-LWE, celui-ci peut être adapté à d'autres schémas. En particulier, l'arithmétique nécessaire au calcul de la relinéarisation va se retrouver sur d'autres schémas, et notamment les schémas de 3$^{\text{ème}}$ génération. Ainsi, l'étude de l'intérêt de notre accélérateur pour d'autres schémas est une bonne perspective de recherche.

Il reste également un dernier point d'ouverture à présenter, à savoir l'implémentation du chiffrement homomorphe dans le cas du transchiffrement [LN14]. L'objectif est d'envoyer les messages au serveur de calcul non pas chiffrés avec le chiffrement homomorphe mais avec un algorithme de chiffrement symétrique. Le serveur de son côté va déchiffrer les messages dans le domaine homomorphe avec la clé secrète du chiffrement symétrique (qui est elle-même chiffrée avec le chiffrement homomorphe), puis procéder au calcul sur les données qui sont maintenant dans le domaine homomorphe. Cette stratégie permet de réduire très fortement l'expansion du chiffré (elle devient équivalente à celle du chiffrement symétrique, qui en général à une expansion très faible). En contrepartie, il est nécessaire d'exécuter un algorithme de cryptographie à clé symétrique en homomorphe, ce qui implique un surcoût en temps de calcul et en terme de profondeur multiplicative. Cette solution à l'énorme avantage de réduire drastiquement la bande passante du réseau que l'on utilise. Lorsque chaque bit de message correspond à plusieurs millions de bits de données chiffrées à transférer sur le réseau, le coût du transfert devient rapidement prohibitif lorsqu'il y a un nombre important de messages. Il serait donc intéressant de comparer au niveau applicatif notre approche basée sur le *batching* (qui permet de réduire également l'expansion du chiffré, mais à un niveau inférieur à celui du transchiffrement) et l'approche transchiffrement.

Liste des publications

Journaux

1) Vincent Migliore, Cédric Seguin, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat and Russell Tessier, *A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm*, in Proc. of CODES+ISSS and apprears as part of the ESWEEK-ACM TECS special issue, 2017.
2) Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine and Guy Gogniat, *Hardware/Software co-Design of an Accelerator for FV Homomorphic Encryption Scheme using Karatsuba Algorithm*, IEEE Transactions on Computers, 2017.
3) Vincent Migliore, Guillaume Bonnoron and Caroline Fontaine, *Guidelines to Practical Parameters for Somewhat Homomorphic Encryption (SHE) Schemes*, IEEE Transactions on Computers, 2017. **(sourmis)**

Conférences internationales avec acte

1) (ordre alphabetique) Guillaume Bonnoron, Caroline Fontaine, Guy Gogniat, Vincent Herbert, Vianney Lapotre, Vincent Migliore and Adeline Roux-Langlois, *Somewhat/Fully Homomorphic Encryption : Implementation Progresses and Challenges*, in Proc. of C2SI 2017.
2) Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine and Guy Gogniat, *Fast Polynomial Arithmetic for Somewhat Homomorphic Encryption Operations in Hardware with Karatsuba Algorithm*, in Proc. of FPT 2016.
3) Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine and Guy Gogniat, *Exploration of Polynomial Multiplication Algorithms for Homomorphic Encryption Schemes*, in Proc. of Reconfig 2015.

Conférences nationales

1) Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine and Guy Gogniat, *Cryptographie Complètement Homomorphe : Quels Candidats et Quelles Attentes à Avoir pour l'Accélération Matérielle de ces Schémas de Chiffrement ?*, JNRDM 2015.
2) Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine et Guy Gogniat, *Algorithmes pour le Chiffrement Homomorphe*, COMPAS 2016.

Communications orales dans des séminaires

1) Vincent Migliore, *On Realistic Speed-up and Possible Homomorphic Operations of Somewhat Homomorphic Encryption Schemes in Hardware*, Cryptarchi 2016.
2) Vincent Migliore, *Somewhat Homomorphic Encryption Schemes : Which Candidates and Which Expectations to Have With These Encryption Schemes ?*, Cryptarchi 2015.
3) Vincent Migliore, *Calcul sur les Données Chiffrées, de la Théorie à la Pratique*, GDR SoC-SiP 2015.

Communication par poster dans des séminaires

1) Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine and Guy Gogniat, *Exploration of Polynomial Multiplication Algorithms for Homomorphic Encryption Schemes*, CHES 2015.

Bibliographie

[ABD16] Martin Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched ntru assumptions : Cryptanalysis of some fhe and graded encoding schemes. Cryptology ePrint Archive, Report 2016/127, 2016.

[Ajt96] M. Ajtai. Generating Hard Instances of Lattice Problems. In *Proc. of STOC*, 1996.

[Alb17] Martin Albrecht. R-lwe estimator. `https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py`, 2017. [Online ; accessed 31-May-2017].

[AMBG+] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, and Lepoint Tancrède Killijian, MArc-Olivier. Nfllib. `https://github.com/quarkslab/NFLlib`. [Online ; accessed 31-May-2017].

[AMBG+16] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, and Lepoint Tancrède Killijian, MArc-Olivier. Nfllib : Ntt-based fast lattice library. pages 138–154, 2016.

[AMGH10] Carlos Aguilar Melchor, Philippe Gaborit, and Javier Herranz. Additively Homomorphic Encryption with d-Operand Multiplications. In *Proc. of CRYPTO*, 2010.

[APS15] Martin Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Accepted to Journal of Mathematical Cryptology*, 2015.

[Bar86] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Proc. of CRYPTO*, 1986.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. In *Proc. of Selected Areas in Cryptography - SAC*, 2016.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In *Proc. of TCC*, 2005.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proc. of ITCS*, 2012.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. In *ACM*, 2003.

[BLLN13] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Proc. of IMACC 2013*, 2013.

[Bra12] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Proc. of CRYPTO*, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In *Proc. of CRYPTO*, 2011.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based fhe as secure as pke. In *Proc. of ITCS 2014*, 2014.

[CCK+13] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch Fully Homomorphic Encryption over the Integers. In *Proc. of EUROCRYPT*, 2013.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption : Bootstrapping in less than 0.1 seconds. In *Proc. of ASIACRYPT*, 2016.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-Invariant Fully Homomorphic Encryption over the Integers. In *Proc. of PKC*, 2014.

[CMNT11] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully Homomorphic Encryption Over the Integers with Shorter Public Keys. In *Proc. of CRYPTO*, 2011.

[CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption Over the Integers. In *Proc. of EUROCRYPT*, 2012.

[CZ81] David G. Cantor and Hans Zassenhaus. A New Algorithm for Factoring Polynomials Over Finite Fields. *Mathematics of Computation*, 1981.

[DGHV10] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In *Proc. of EUROCRYPT*, 2010.

[DR01] Joan Daemen and Vincent Rijmen. The Design of Rijndael. *AES - The Advanced Encryption Standard*, 2001.

[DS16] Yarkin Doröz and Berk Sunar. Flattening NTRU for Evaluation Key Free Homomorphic Encryption. Cryptology ePrint Archive, Report 2016/315, 2016.

[FSF+13] Simon Fau, Renaud Sirdey, Caroline Fontaine, Carlos Aguilar-Melchor, and Guy Gogniat. Towards practical program execution over fully homomorphic encryption schemes. In *Proc. of P2P*, 2013.

[FV12] Junfeng Fan and Frederick Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[Gam85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4) :469–472, 1985.

[Gen09] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[Gra15] Torbjörn Granlund. GMP library. https://gmplib.org/, 2015. [Online ; accessed 31-May-2016].

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors : Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Proc. of CRYPTO*, 2013.

[Hal] Shai Halevi. Helib. https://github.com/shaih/HElib. [Online ; accessed 31-May-2017].

[Har16] William Hart. FLINT library. http://www.flintlib.org/, 2016. [Online ; accessed 31-May-2016].

[HG01] Nick Howgrave-Graham. Approximate Integer Common Divisors. In *Proc. of CaLC*, 2001.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU : A Ring-Based Public Key Cryptosystem. In *Proc. of ANTS*, 1998.

[HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. 2014.

[JRHK] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner. Riffa. http://riffa.ucsd.edu/. [Online ; accessed 31-May-2017].

[JRHK15] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner. RIFFA 2.1 : A Reusable Integration Framework for FPGA Accelerators, 2015.

[KGV15] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. SHIELD : Scalable Homomorphic Implementation of Encrypted Data-Classifiers. *Accepted to IEEE Transactions on Computers*, 2015.

[KO62] A. Karatsuba and Y. Ofman. Multiplication of multi-digit numbers on automata (in russian). *Doklady Akad. Nauk SSSR*, 145(2) :293–294, 1962. Translation in Soviet Physics-Doklady, 44(7), 1963, pp. 595-596.

[LATV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proc. of STOC*, 2012.

[LCPa] Kim Laine, Hao Chen, and Rachel Player. Simple encrypted arithmetic library - seal (v2.1). https://sealcrypto.codeplex.com/. [Online ; accessed 31-May-2017].

[LCPb] Kim Laine, Hao Chen, and Rachel Player. Simple encrypted arithmetic library - seal (v2.1). https://sealcrypto.codeplex.com/. [Online ; accessed 31-May-2017].

[Lep14] Tancrède Lepoint. *Design and Implementation of Lattice-Based Cryptography*. PhD thesis, École Normale Supérieure de Paris, 2014.

[LN14] Tancrede Lepoint and Michael Naehrig. A Comparison of the Homomorphic Encryption Schemes FV and YASHE. In *Proc. of AFRICACRYPT*, 2014.

[LPR81] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. Another NP-Complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice. In *Technical Report 8104, University of Amsterdam*, 1981.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In *Proc. of EUROCRYPT*, 2010.

[Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In *Proc. of CRYPTO*, 1985.

[Mul89] Jean-Michel Muller. Arithmétique des Ordianteurs. 1989.

[NIS] NIST : cryptographie post-quantique. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/`.

[Pai99] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proc. of Advances in Cryptology — EUROCRYPT 1999*, 1999.

[PG14] Thomas Pöppelmann and Tim Güneysu. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Proc. of SAC 2014*, 2014.

[PNPM] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware. In *Proc. of Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 143–163. Springer.

[Pol71] J.M. Pollard. The Fast Fourier Transform in a Finite Field. In *Mathematics of Computation*, volume 25, pages 365–374. 1971.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proc. of STOC*, 2005.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *ACM*, 2009.

[RSA78] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2) :120–126, February 1978.

[SHA] Spécifications SHA512. `http://csrc.nist.gov/groups/ST/toolkit/`.

[Sho97] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 1997.

[Sho16] Viktor Shoup. NTL library. `http://www.shoup.net/ntl/`, 2016. [Online ; accessed 31-May-2016].

[SRJV+] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular hardware architecture for

somewhat homomorphic function evaluation. In *Proc. of Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 164–184. Springer.

[Too63] A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics-Doklady*, 7 :714–716, 1963.