Microcontrôleur à flux chiffré d'instructions et de données

Thomas Hiscock

université PARIS-SACLAY

NTT : 2017SACLV074

Thèse de doctorat

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Microcontrôleur à flux chiffré d'instructions et de données

Thèse de doctorat préparée au sein du **CEA LETI** et de **L'université de Versailles Saint-Quentin en Yvelines**

École doctorale n°580, sciences et technologies de l'information et de la communication (STIC)
Spécialité du doctorat : Informatique

Thèse présentée à **Grenoble** le **7 décembre 2017**, par

Thomas Hiscock

Composition du jury :

M. Arnaud Tisserand Directeur de recherche, CNRS, Laboratoire Lab-STICC	Rapporteur (Président)
M. Régis Leveugle Professeur, Grenoble INP - Phelma	Rapporteur
Mme. Aline Gouget Ingénieur chercheur, Gemalto	Examinatrice
M. Sylvain Guilley Professeur, Télécom ParisTech	Examinateur
Mme. Vanessa Vitse Maître de conférences, Université Grenoble-Alpes	Examinatrice
M. Olivier Savry Ingénieur chercheur, CEA LETI	Encadrant
M. Louis Goubin Professeur,UVSQ	Directeur de thèse

Remerciements

Je tiens à remercier Arnaud Tisserand, Regis Leveugle, Aline Gouget, Vanessa Vitse et Sylvain Guilley d'avoir accepté de faire partie de mon jury de thèse et pris le temps de considérer mes travaux. Leurs commentaires, remarques et les échanges que nous avons eus le jour de la soutenance ont contribué à améliorer ce mémoire.

Je remercie Olivier Savry, d'une part pour avoir proposé ce sujet de thèse, mais aussi pour le temps et les efforts qu'il a consacrés à l'encadrement de cette thèse. Je remercie également Louis Goubin, mon directeur de thèse, pour les nombreuses interactions enrichissantes que nous avons pu avoir et qui, malgré la distance géographique, a fait plusieurs fois le déplacement à Grenoble pour suivre l'avancement de mes travaux. Je remercie chaleureusement tous mes collègues du laboratoire LSOC et LSOSP. Travailler à vos côtés est un vrai plaisir, que j'ai la chance de pouvoir prolonger pendant quelques années encore ! Un grand merci aux personnes qui ont soutenu ma candidature et rendu cela possible. Je remercie également tous les autres collègues que j'ai pu côtoyer durant ces trois années au CEA.

Enfin, j'adresse des remerciements tout particuliers à mes amis, ma famille, mes parents, mes frères, mes grands-parents, mes oncles et tantes, cousins et cousines : partager des moments avec vous est toujours un vrai bonheur et parfois une bouffée d'air ! Enfin, merci Julie, d'avoir été à mes côtés, de m'avoir soutenu et supporté pendant ces trois années, même dans les périodes difficiles et mes moments « grincheux».

Table des matières

Table des figures

Liste des tableaux

1 Contexte et motivations

Sommaire

1.1. Introduction

1.1.1. Processeurs et sécurité

Les processeurs sont omniprésents dans les technologies qui nous entourent. Par dizaines, ils composent et orchestrent le fonctionnement des systèmes. Les processeurs grand public, qui équipent nos ordinateurs, ne sont finalement qu'une partie très restreinte de l'ensemble des processeurs utilisés. Les processeurs embarqués, plus petits, généralement moins puissants, mais beaucoup plus efficaces en terme énergétique sont en réalité bien plus répandus.

La sécurité d'un système numérique est directement liée à la sécurité du ou des processeurs qui le composent. Ces processeurs ont en effet la responsabilité de collecter, stocker et traiter des données parfois sensibles. Mais ils ont également un rôle actif en effectuant des actions visibles : donner une autorisation après une vérification de mot de passe, échanger des données, activer des moteurs, ... Les propriétés les plus communes attendues pour un système sont :

— La confidentialité, le système est capable de maintenir le secret des données sensibles qui lui sont confiées.

— L'intégrité, le système est capable de détecter une modification de ses données ou de son état.
— L'authenticité, le système est capable de vérifier l'origine des données.
— À mi-chemin entre la sécurité et la sûreté de fonctionnement, la robustesse, c'est-à-dire que le système doit être capable de continuer à fonctionner, même en présence de dysfonctionnements ou de perturbations.

Dans cette thèse nous nous concentrerons sur les mécanismes permettant d'assurer la propriété de confidentialité pour un programme sur un processeur embarqué donné.

1.1.2. Confidentialité d'un programme

Un programme est composé d'un ensemble d'instructions et de données qui spécifient les calculs à effectuer par le processeur. On distinguera les données statiques, qui sont présentes dans le programme, des données dynamiques, calculées et connues seulement à l'exécution.

Historiquement, la motivation pour maintenir la confidentialité d'un programme était économique, pour préserver la propriété intellectuelle de codes ou d'algorithmes qui auraient nécessité beaucoup de temps et d'argent à développer.

Mais aujourd'hui, la confidentialité est une propriété essentielle pour la sécurité. Les programmes sont en effet amenés à manipuler des données sensibles, dont le secret doit être maintenu à tout prix.

Pour des systèmes embarqués, la confidentialité du code permet de rendre plus complexe la rétro-conception du système. La rétro-conception consiste en l'étude et l'analyse d'un système pour comprendre son fonctionnement. Il s'agit en général de la première opération effectuée par un attaquant, dans laquelle en plus de la compréhension, il va chercher des failles et des points de vulnérabilités dans le système. Sans cette étape, il est très difficile de concevoir une attaque.

Enfin, le développement des objets connectés amène un besoin de distribution de code sécurisée. La confidentialité peut ici être intéressante lors de la découverte d'une faille critique dans un code, pour transmettre un programme correctif sans révéler la faille elle-même qui pourrait rendre les systèmes vulnérables à court terme (vulnérabilité *Zero Day*).

1.2. Quelles menaces ?

Nous avons mis en valeur les enjeux de la protection d'un code. Nous nous penchons maintenant sur les menaces qui peuvent permettre d'extraire des informations du programme présent sur un système. Les processeurs embarqués, par leur proximité physique avec l'utilisateur, offrent malheureusement une large surface d'attaque. Un attaquant peut en effet librement expérimenter et interagir avec le système.

1.2.1. Extraction et espionnage direct des composants

Un premier vecteur d'attaque possible est l'accès direct au programme par extraction ou lecture des composants mémoires. Ceci est particulièrement envisageable lorsque ces composants sont externes sur le circuit imprimé. Il est alors aisé d'accéder à ces composants, soit pour espionner les interfaces avec un analyseur logique, ou encore les extraire du support pour les analyser à l'aide de matériel spécifique.

Non seulement les mémoires persistantes (EEPROM, FLASH, ...) sont concernées, mais également les mémoires dynamiques. La rémanence des données peut en effet être augmentée à plusieurs dizaines de minutes en refroidissant la mémoire, par exemple à l'aide d'azote liquide, on parle d'attaque par démarrage à froid (dites *cold-boot*) [HSH+09].

Ces attaques concernent un grand nombre de systèmes. Cependant, la tendance aujourd'hui est d'intégrer de plus en plus les technologies, même différentes sur la même puce (*System on Chip)*, ou du moins dans le même boîtier *(System in Package)*. Dans de tels systèmes, l'accès direct aux composants mémoire est très difficile, voire impossible.

1.2.2. Attaques logiques

Cependant, les entrées sorties du système resteront toujours accessibles à un utilisateur. Ces ports peuvent présenter un point d'accès pour un attaquant au système. L'attaque de Boileau [Boi06] illustre bien ce problème. L'auteur a réussi à partir d'un port externe de type FireWire à prendre le contrôle d'un composant de gestion de transfert mémoire (DMA). Dès lors, la mémoire principale du processeur peut être accédée en lecture et écriture, depuis ce port externe.

La logique de test et vérification du système (interface de déboggage, JTAG), lorsqu'elle n'est pas désactivée offre un autre point de vulnérabilité.

1.2.3. Fuites par partage d'éléments d'architecture

Les attaques présentées précédemment supposent un accès physique au système. On pourrait penser que ces attaques disparaissent pour un système inaccessible physiquement à l'utilisateur. Pourtant, le simple fait qu'un programme partage des ressources matérielles avec un autre programme potentiellement corrompu peut induire des fuites d'informations. On parle d'attaques par canaux auxiliaires logicielles. Les éléments d'architecture les plus dangereux à partager sont ceux qui influencent fortement les temps d'exécution, comme les prédicteurs de branchements, les mémoires caches ou encore les tables de traduction d'adresses virtuelles (TLB).

Aussi, on trouve un grand nombre d'attaques, les plus communes étant basées sur le partage de caches [Ber05]. Les auteurs arrivent en invalidant des lignes de cache et en analysant les latences d'accès à retrouver des informations sensibles d'un autre processus concurrent.

Les mémoires DRAM sont également des éléments dangereux à partager comme l'illustre la vulnérabilité DRAMA [PGM+16]. La fuite ici est induite par le partage de lignes en mémoire DRAM. À partir de variations observées dans les temps accès, les auteurs parviennent à extraire des parties de la mémoire d'un autre processus, alors même que des mécanismes de protection mémoire (mémoire virtuelle, système d'exploitation) devraient les interdire.

1.3. Modèle de sécurité

Ainsi, il est clair que le stockage sur les mémoires externes ne peut pas être supposé fiable. De manière plus générale, dans un contexte de systèmes avec plusieurs applications concurrentes, le partage de n'importe quel élément d'architecture constitue potentiellement une fuite d'information.

Nous essayons maintenant de formaliser le modèle de sécurité considéré dans cette thèse. La figure 1.1 représente l'architecture d'un système sur puce relativement simple. On y retrouve un processeur qui est connecté à divers périphériques par un bus. Le processeur possède ou non des mémoires caches. Il est relié à divers périphériques. À l'extérieur de la puce sont présentes une mémoire FLASH pour permettre du stockage non volatile, ainsi qu'une mémoire dynamique (DRAM) pour le stockage des données à l'exécution.

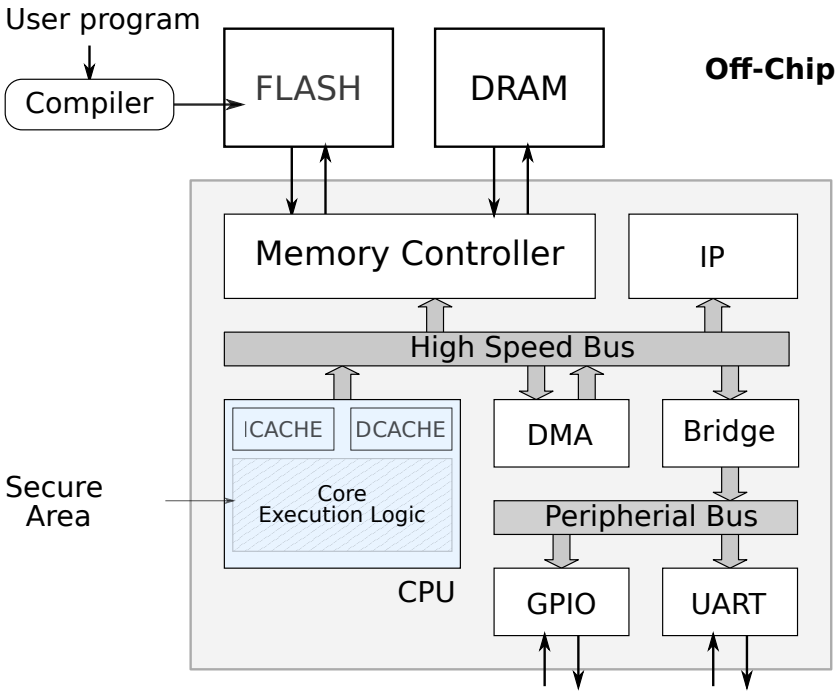


FIGURE 1.1. – Architecture d'un système sur puce

L'objectif est de protéger la confidentialité d'un programme et de ses données. Le programme et les données initiales sont stockés dans la mémoire externe. Dans

ce contexte, nous supposons que la logique interne au processeur est une zone de confiance, dans laquelle les données sont supposées inaccessibles et non modifiables par un attaquant. Cette zone inclut les registres et les divers éléments de calculs comme les unités arithmétiques et logiques, éventuellement les unités de calcul flottant. En revanche, cette zone de confiance s'arrête aux interfaces avec la mémoire, c'est-à-dire aux signaux de communication avec les caches, ou directement à un bus.

Ainsi, dans ce cadre, un attaquant peut accéder en lecture à toute donnée sortante de la zone de confiance (la logique d'exécution du processeur). Autrement dit, il peut :

— Lire à n'importe quel instant le code stocké en mémoire externe. Ceci modélise les menaces liées à l'observation directe ou indirecte des mémoires.
— Observer les données échangées sur le bus. Ce qui modélise l'espionnage par un composant corrompu, par exemple un autre processeur ou un DMA
— Accéder à n'importe quelle donnée présente dans les caches d'instructions ou de données s'ils existent. Cette capacité est donnée dans le but de modéliser les fuites de données directes ou indirectes par le cache.

1.4. Objectifs de la thèse et organisation du manuscrit

1.4.1. Questions abordées

Nous cherchons donc à assurer la confidentialité d'un programme sur un processeur en supposant l'existence d'une zone de confiance.

La première question abordée dans cette thèse est : comment peut-on maintenir la confidentialité jusqu'à cette zone de confiance ? Si cela est possible, nous chercherons évidemment à le faire le plus efficacement possible.

Les processeurs modernes utilisent une architecture dite de « Harvard», où les instructions et les données sont lues par des ports physiquement séparés. D'un coté, les accès à la mémoire d'instructions sont relativement séquentiels. De l'autre, les accès mémoire sont nettement moins structurés. Peut-on adresser plus efficacement la protection des instructions ou des données par des mécanismes dédiés à chaque domaine ?

Enfin, jusqu'à quelle limite peut-on maintenir les données confidentielles avant et pendant leur manipulation ? Peut-on, même les garder chiffrées dans cette zone de confiance ?

1.4.2. Structure du document

Nous commençons par dresser un état de l'art des solutions pour la confidentialité d'un programme dans le chapitre 2.

Nous présentons ensuite dans le chapitre 3 une méthode de chiffrement des instructions d'un programme permettant de déchiffrer à la frontière de la zone de confiance.

Le chapitre 4 présente une solution basée sur les algorithmes de chiffrement homomorphe pour maintenir les données chiffrées en mémoire et également durant les

manipulations, c'est-à-dire à l'intérieur même de la zone de confiance. Le chapitre 5 présente la conception matérielle d'un cœur de calcul pour l'évaluation efficace des opérations homomorphes.

L'évaluation de ces propositions sur des composants logiques programmables est présentée dans le chapitre 6.

Enfin le chapitre 7 conclut ce manuscrit et synthétise les réponses apportées au cours de cette thèse.

1.4.3. Contributions

Les travaux présentés dans le chapitre 3 ont fait l'objet d'une présentation à la conférence DSD 2017 [HSG17]. Deux brevets ont été déposés sur cette thématique, un premier sur la méthode elle-même [PPSH17] et un second sur la combinaison de ce chiffrement avec des programmes auto modifiants (ce point n'est pas abordé dans ce manuscrit).

Une version préliminaire des travaux présentés dans les chapitres 4 et 5 a été présentée aux sessions doctorants de RESSI 2017 [1]. Un papier décrivant la solution complète est écrit et actuellement en cours de soumission. Un brevet a également été déposé sur ces travaux [HS17].

1. « Chiffrement homomorphe pour la protection de calculs sur processeur embarqué», https://ressi2017.sciencesconf.org/133764/document

Méthodes pour la protection de la confidentialité d'un programme

2

Sommaire

Nous avons vu dans le chapitre précédent les différentes sources d'attaques qui peuvent compromettre la confidentialité d'un programme et de ses données. Nous effectuons dans ce chapitre un état de l'art des différentes solutions applicables à ce problème.

Nous commencerons par les techniques d'obfuscation, qui sera l'occasion de formaliser la problématique de la protection d'un programme. Puis, nous présenterons les différentes techniques utilisées dans les processeurs sécurisés. Dans un premier temps, les architectures qui introduisent du non-déterminisme d'exécution en vue de limiter les fuites d'informations par des canaux auxiliaires. Puis, par la suite, les architectures qui intègrent du chiffrement mémoire. Enfin, nous terminerons ce chapitre par une synthèse où seront identifiés les points sur lesquels les travaux de cette thèse amènent des améliorations.

2.1. Obfuscation, approche formelle au problème

L'obfuscation, est un anglicisme provenant du verbe *obfuscate* qui signifie obscurcir, embrouiller. Dans le monde de l'informatique, l'obfuscation réfère au procédé de transformation d'un programme visant à le rendre inintelligible. Par exemple, la figure 2.1 montre un programme Perl obfusqué qui affiche seulement le message « Just

```
@P=split//,".URRUU\c8R";
@d=split//,"\nkcah lreP rehtona tsuJ";
sub p{@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;
($q*=2)+=$f=!fork;map{$P=$P[$f^ord($p{$_})&6];
$p{$_}=/ ^$P/ix?$P:close$_}keys%p}p;p;p;p;p;
map{$p{$_}=~/^[P.]/ && close$_}%p;wait until$?;
map{/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)if/\S/;print
```

FIGURE 2.1. – Exemple de programme obfusqué, tiré de Wikipedia (https://en.wikipedia.org/wiki/Obfuscation_(software))

another Perl hack». Cette fonctionnalité très simple s'exprime en très peu de lignes de code dans la plupart des langages de programmation. Pourtant, il est extrêmement difficile seulement à partir du code brouillé de la figure 2.1 de deviner le comportement ou des données internes du programme original.

L'obfuscation permet non seulement d'assurer la confidentialité des programmes, mais également de prévenir toute forme rétro-conception ou la réutilisation d'une partie du programme. Ceci permet de protéger la propriété intellectuelle d'un code ou de gérer des droits numériques (DRM). À des fins malveillantes, l'obfuscation est monnaie courante dans les *malwares* et est utilisée pour enfouir la partie dangereuse du programme, réduisant ainsi ses chances de détection. Enfin, à titre plus ludique, le concours *International Obfuscated C Code Contest* (http://www.ioccc.org/) récompense chaque année le code C le plus obscur.

Le programme de la figure 2.1 est très probablement écrit manuellement. On comprend que ce procédé peut-être assez long et ne permet donc pas d'obfusquer des programmes complexes. Heureusement, un nombre important de méthodes heuristiques permettent de brouiller automatiquement un programme. Par exemple, l'aplatissement du flot de contrôle, l'utilisation de prédicats opaques ou encore l'évaluation du programme dans plusieurs niveaux machines virtuelles imbriquées. Cependant, si des obfuscateurs complexes existent, il existe également des méthodes d'attaque et d'analyse de programmes obfusqués très performantes.

2.1.1. L'obfuscation au sens théorique

L'obfuscation bénéficie d'un traitement théorique rigoureux [BGI+01], à mi-chemin entre la cryptologie et la théorie de la complexité. Formellement, un obfuscateur $\mathcal{O}$ prend un programme $\mathcal{P}$ en entrée et produit un programme obfusqué $\mathcal{O}(\mathcal{P})$. Deux propriétés fondamentales sont attendues, quelle que soit le type d'obfuscation considérée :

— L'équivalence fonctionnelle : pour toutes les entrées, le programme obfusqué doit produire le même résultat que la version originale du programme.
— L'efficacité : la taille et le temps d'exécution du programme obfusqué doivent être comparables à l'original. Plus précisément, dans les deux cas un facteur au

plus polynomial est toléré.

Un obfuscateur doit également offrir des propriétés de sécurité quant au programme obfusqué. La protection la plus forte est l'obfuscation en boîte noire dite *Virtual Black Box* (VBB). On impose alors que toute l'information obtenue sur la version obfusquée d'un programme soit simulable à partir d'un accès en boîte noire à ce programme.

La contrainte VBB est très forte et l'existence d'une telle obfuscation aurait des applications impressionnantes. Par exemple créer un schéma de cryptographie asymétrique à partir d'un schéma symétrique ou encore créer un schéma complètement homomorphe efficace. Il est malheureusement impossible de construire un obfuscateur générique au sens VBB pour toutes les fonctions existantes. Ce résultat très important du domaine a été démontré par Barak et. al dans [BGI+01] en construisant une famille de fonctions impossible à obfusquer.

Pourtant, ce résultat est loin d'avoir mis un terme à la recherche en obfuscation, en particulier un certain nombre de points restent ouverts :

— L'obfsucation au sens VBB est une contrainte très forte, qu'en est-il de modèles plus restreints ?

— La famille de fonctions impossible à obfusquer construite dans [BGI+01] ne constitue qu'un contre-exemple particulier. Le résultat d'impossibilité est tout de même applicable pour toute classe contenant TC^0 (*threshold circuits*). Il s'agit de la famille des circuits de profondeur constante constitués d'un nombre polynomial de portes à vote majoritaire. Cependant, il reste un grand nombre de fonctions utiles pour lesquelles l'existence d'un obfuscateur n'est pas contredite par [BGI+01].

2.1.1.1. Obfuscation indistinguable

Une définition plus relâchée proposée dans [BGI+01] est l'*Indistinguishability Obfuscation* (*iO*). Pour deux circuits C_1, C_2 de même taille et qui calculent exactement la même fonction, la propriété d'iO impose qu'il soit impossible de distinguer $\mathcal{O}(C_1)$ de $\mathcal{O}(C_2)$.

Une construction générique d'un tel obfuscateur a été proposée dans [GGH+13], sous l'hypothèse d'existence des *multilinear-maps*. Cependant, la sécurité apportée par ce modèle plus restreint est moins intuitive que la propriété VBB.

2.1.1.2. Obfuscation de familles de fonctions particulières

Un premier cas intéressant d'obfuscation sont les fonctions à point (*point functions*) qui retournent faux pour toutes les entrées sauf une bien précise (voir figure 2.2). Ces fonctions sont utilisées dans la vérification de mot de passe.

Un domaine fortement lié à l'obfuscation est la *whitebox cryptography*. Il s'agit de protéger la clé d'une primitive cryptographique dans une implémentation logicielle complètement observable et modifiable (modèle VBB). L'objectif ici n'est pas de protéger la fonctionnalité, mais la clé. Le nom *whitebox cryptography* fait opposition à la cryptographie en boîte grise (*greybox)* qui réfère à la cryptographie sur les systèmes

```
bool check_password(string password) {
    if (password == "abcdefgh") {
        return true;
    }
    return false;
}
```

FIGURE 2.2. – Exemple de fonction à point

physiques où l'attaquant a accès à des informations partielles (e.g., canaux auxiliaires) et à la cryptographie théorique où dans les preuves, les primitives sont manipulées comme des boîtes noires (*blackbox)*.

Un challenge a été lancé dans le cadre de la conférence CHES 2017 pour proposer une implémentation *whitebox* d'un AES 128 avec une clé fixée. Il est intéressant d'observer qu'aucune proposition n'a résisté. Ceci illustre à la fois la difficulté du problème de l'obfuscation, même d'un programme relativement simple, mais également la puissance des techniques d'analyse. Dans ce domaine, les méthodes les plus efficaces sont une adaptation logicielle des attaques par canaux auxiliaires, appelées *Differential Computation Analysis* (DCA) et introduites dans [BHMT16].

2.2. Techniques pour la protection des implémentations matérielles

L'obfuscation se place dans un modèle de menaces très fort : une observabilité et un contrôle complet de toutes les valeurs intermédiaires d'un calcul. En présence d'un circuit matériel, l'observabilité est bien plus réduite. Il est très difficile, voire impossible d'obtenir l'état exact du système à un instant donné. Par contre, les fuites par des canaux auxiliaires comme la consommation, le rayonnement électromagnétique ou encore le bruit acoustique sont des éléments bien plus faciles à observer.

2.2.1. Masquage

Un modèle d'observabilité classique est le *d-probing* de Ishai et. al [ISW03], où un attaquant obtient à chaque instant (une étape de calcul ou un cycle d'horloge) les valeurs de d fils de son choix.

Le masquage est une contre-mesure très répandue contre ces fuites partielles d'information. Pour une fonction donnée utilisant une variable sensible (clé secrète par exemple), un schéma de masquage à l'ordre d rend n'importe quel d-uplet de variables intermédiaires statistiquement indépendant de la donnée sensible. Le masquage booléen additif est le schéma le plus courant. Pour chaque valeur sensible x, d valeurs aléatoires $(x_1, \ldots, x_d)$ sont générées, puis la valeur masquée est définie comme

$$x_0 = x \oplus x_1 \oplus \cdots \oplus x_d.$$

Le démasquage consiste à calculer la somme de tous les x_i, ce qui annule les masques présents dans x_0, $x = x_0 \oplus x_1 \oplus \cdots \oplus x_d$. Il faut dès lors définir les opérations sur cette représentation masquée des variables. Pour un masquage additif, les opérations linaires sont triviales. En effet, pour une fonction f linéaire, on a :

$$f(x_0) = f(x \oplus x_1 \oplus \cdots \oplus x_d) = f(x) \oplus f(x_1) \oplus \cdots \oplus f(x_d).$$

Ainsi, le $d+1$-uplet $(f(x_0), f(x1), \ldots, f(x_d))$ est bien un masquage de $f(x)$. En revanche, les opérations non linéaires (l'opération « AND » par exemple) sont loin d'être aussi faciles à traduire. Par exemple le schéma proposé dans [ISW03] pour masquer une porte « AND» nécessite $(d+1)^2$ portes « AND» logiques classiques et $2d(d+1)$ portes « XOR ».

Un des points forts du masquage est qu'il s'agit d'une contre-mesure prouvable, dans [CJRR99] les auteurs prouvent que le nombre d'observations nécessaires pour distinguer une variable sensible augmente exponentiellement avec l'ordre du masquage. Ce résultat fut généralisé à un schéma de masquage complet dans [PR13].

2.2.2. Processeurs non déterministes

Les techniques de masquage s'appliquent à un algorithme ou un circuit particulier. Pour offrir une protection contre les attaques par canaux auxiliaires indépendamment de l'algorithme, une autre approche est de protéger le support d'exécution, autrement dit le processeur. Les travaux dans cette direction ont été initiés par May et. al, qui ont proposé un ordonnancement aléatoire des instructions à l'exécution, qu'ils nomment *instruction descheduling* [MMS01a]. Les mêmes auteurs proposent par la suite d'utiliser du renommage de registres aléatoire à l'exécution dans [MMS01b]. L'injection de fausses instructions, sans effet notable sur le programme est proposée dans [ARP07], pour introduire des variations temporelles dans l'exécution.

Dans [AAF+16] les auteurs proposent d'alterner aléatoirement l'exécution de plusieurs *threads* matériels. Seuls les éléments d'états comme les registres sont dupliqués. Il est ainsi possible d'entrelacer l'exécution d'un AES sur les données réelles puis en parallèle, un AES avec une clé et un message aléatoires.

Dans [BBT+15] les auteurs comparent l'effet de différentes techniques : l'insertion d'instructions NOPs à des moments aléatoires dans le but d'ajouter des variations temporelles (*hiding*) et un masquage du premier ordre d'une partie du chemin de données. Ces contre-mesures sont intégrées dans un processeur *SecretBlaze*. Elles augmentent la surface d'environ 50% et réduisent la fréquence maximum de 19%. L'attaque considérée qui se base sur le rayonnement électromagnétique échoue lorsque ces contre-mesures sont déployées (avec un nombre de traces allant jusqu'à 200K).

Dans [GJM+16], les auteurs présentent le masquage d'un chemin de données protégé à l'aide d'un masquage DOM (*Domain Oriented Masking*) dans un processeur RISCV. La surface de l'ALU augmente de manière quadratique avec l'ordre du masquage (à cause du masquage des portes «AND»), cependant les auteurs évaluent un masquage allant jusqu'à l'ordre 4. Dans ce cas, la surface est approximativement multipliée par

trois, pour une fréquence maximale diminuée de 8%. La sécurité apportée est validée à l'aide d'un test t de Welch.

Le test t de Welch est un test statistique. Dans le cadre des attaques par canaux auxiliaires, il est souvent utilisé pour comparer les traces obtenues lorsque les données sensibles de l'algorithme sont complètement aléatoires et les traces pour une donnée sensible fixée. Ce test permet de révéler d'éventuels points de fuites d'informations dans le temps. Cependant, ce test est purement indicatif, il ne permet pas de déterminer comment exploiter la fuite.

2.3. Architectures de processeurs sécurisées

Les premières architectures de processeurs visant à améliorer la sécurité apparaissent au début des années 80 avec une série de brevets et publications par Best [Bes79, Bes81, Bes80]. L'objectif est de protéger les éventuelles fuites du programme et des données placées en mémoire. Ces travaux introduisent pour la première fois des primitives de chiffrement au sein d'un processeur, aux interfaces mémoires (chiffrement de bus). Les primitives de chiffrement utilisées sont bien moins robustes que celles actuelles. Les données sont chiffrées, mais également les adresses émises ce qui rend l'exécution très complexe à analyser. Ces travaux ont constitué le fondement des processeurs sécurisés Dallas de la famille DS5000 [Sem]. On les retrouve dans de nombreux systèmes réels comme les terminaux de paiement ou encore les consoles de jeux.

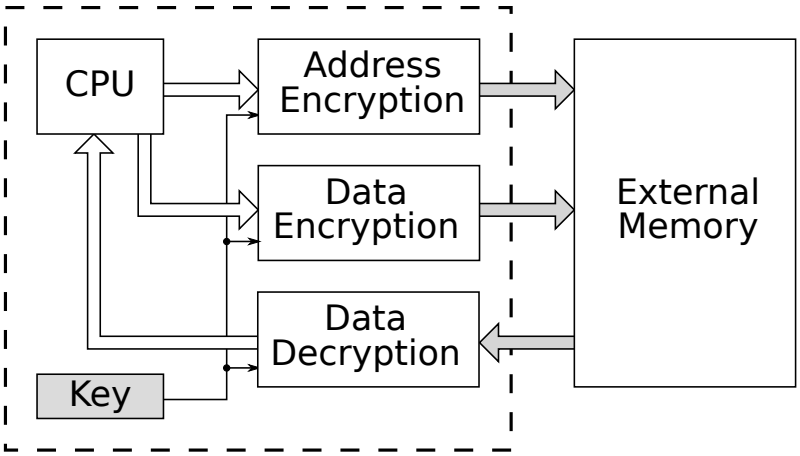


FIGURE 2.3. – Vue schématique des processeurs de la famille DS5000

La figure 2.3 représente les différents blocs de chiffrement insérés aux interfaces avec la mémoire externe dans cette famille de processeurs. On y retrouve un bloc pour chiffrer les données sortantes vers la mémoire, un bloc qui déchiffre les données entrantes et enfin un bloc qui chiffre l'adresse émise par le processeur. L'ensemble de ces blocs utilisent une clé secrète interne au processeur, qui est gardée sur une mémoire volatile. Cette clé est produite par un générateur aléatoire interne au processeur

lors de la procédure de chargement d'un programme. À aucun moment la clé ne sort du processeur, même le développeur n'y a pas accès. Le système est également équipé de capteurs d'intrusions capables de couper l'alimentation du processeur en cas d'anomalie, détruisant ainsi la clé et rendant le système inutilisable.

Malgré la présence de ces différents mécanismes de sécurité, une célèbre attaque fut publiée par Kuhn [Kuh98] sur les premières versions de ce processeur. Il a exploité le fait que l'on pouvait librement injecter des instructions, car aucune vérification d'intégrité n'était présente. Ainsi, en observant la réaction des sorties du système en fonction des instructions injectées, il a pu identifier un certain nombre d'instructions utiles et construire un programme extrayant toute la mémoire. Cette attaque est en partie réalisable grâce au fait que les instructions sont encodées sur 8 bits et qu'en conséquence, une énumération complète est très rapide. Le simple fait d'utiliser des instructions de 32 bits réduit fortement la faisabilité de l'attaque.

2.3.1. Les *Oblivious RAMs*

Goldreich et. al [GO96] ont formalisé un modèle de menaces proche de celui présenté dans le chapitre précédent : le stockage en mémoire externe est une surface d'attaque. Le modèle *Oblivious RAMs* (ORAM) fait l'hypothèse que le processeur est complètement opaque, une boîte noire dont aucune donnée interne n'est accessible. Seuls les accès mémoire sont visibles (adresses et données) et modifiables librement par un attaquant. Ce modèle de menaces fait complètement sens dans les systèmes embarqués où le processeur est fortement intégré sur la puce et plus difficile d'accès qu'une mémoire externe, dont les entrées sorties sont physiquement accessibles. Ce modèle a connu un renouveau pour le stockage sécurisé de données sur le *cloud*. Dans ce cas, le système de calcul (smartphone, ordinateur, ...) est en possession de l'utilisateur et isolé physiquement du fournisseur par le réseau et un canal de transmission sécurisé. Le fournisseur *cloud* quant à lui a accès aux données de l'utilisateur et peut les modifier à sa guise. Il observe également l'équivalent des adresses (index ou n'importe quel élément identifiant les données à accéder).

Dans le modèle ORAM, il faut en premier lieu préserver la confidentialité et l'intégrité des données. De plus, les motifs d'accès mémoire, mêmes à des données chiffrées peuvent révéler des informations sur les données sous-jacentes [IKK12]. Une approche naïve est de lire toute la mémoire à chaque accès en écriture, ainsi les accès mémoires sont indépendants des adresses accédées (propriété *oblivious*). Dans [GO96] Goldreich et. al proposent plusieurs protocoles beaucoup plus efficaces pour sécuriser le stockage.

2.3.2. Architectures à chiffrement mémoire

De nombreux travaux s'inscrivent dans la lignée du Dallas et l'améliorent sur différents aspects. Kuhn propose dans [Mar97] l'architecture *TrustNo 1* qui étend les mécanismes de chiffrement de mémoire à des systèmes multi-tâches. Chaque tâche a une clé associée pour effectuer les opérations cryptographiques. Il propose un mécanisme de changement de contexte sécurisé et également les fonctionnalités de gestion

de clés :

— le transfert d'une nouvelle clé, nécessaire à l'envoi d'un programme sur le processeur cible,
— la révocation, pour empêcher l'exécution d'un processus.

L'architecture XOM

L'architecture XOM [LTM+00] vise à obtenir un code *execute-only* c'est-à-dire accessible seulement en exécution et pas en lecture ou écriture par le processus. Ceci est assuré en chiffrant la mémoire d'instructions. L'architecture permet à plusieurs tâches de cohabiter de manière isolée en associant des clés de session à chacune des tâches (comme dans *TrustNo 1*). L'architecture n'est pas implémentée ni simulée dans ce travail (une preuve formelle est néanmoins fournie dans [LMTH03]), les auteurs ont estimé un ralentissement pouvant s'élever dans le pire des cas à 50% en supposant l'utilisation d'un algorithme DES pour le chiffrement et déchiffrement.

L'architecture HIDE

L'architecture HIDE [ZZP04] intègre d'une part les mécanismes de chiffrement et d'intégrité mémoire connus et en plus cherche à protéger les fuites d'informations par le bus d'adresse. Les auteurs proposent un cache de niveau 2 modifié, chargé de permuter la mémoire pour cacher les motifs d'accès mémoire. La mémoire est permutée par blocs de taille configurable (les tailles de bloc évaluées sont 16KB, 32KB, 64KB). L'architecture a un fonctionnement similaire à une machine ORAM mais par blocs. Autrement dit, la fuite d'information est due seulement aux accès entre blocs. Les ralentissements observés en simulation sur la suite de benchmark SPEC2000 sont très faibles 1.5% en moyenne.

L'architecture AEGIS

L'architecture AEGIS [SOD05, SOSD05] est une architecture complète fournissant deux modes de fonctionnement :

1. *Tamper Evident*, où seule l'intégrité de la mémoire est vérifiée.
2. *Private Tamper Resistant* (PTR), où la mémoire est également chiffrée.

Ceci permet d'améliorer les performances pour les zones de code moins critiques en termes de confidentialité.

Ce travail propose des améliorations sur les mécanismes de chiffrement et d'intégrité mémoire. Les auteurs utilisent un AES en mode compteur pour le chiffrement et ajoutent des *timestamps* pour éviter les attaques par rejeu. L'intégrité est vérifiée à l'aide d'un arbre de Merkle [Mer88], dont les valeurs de hachage intermédiaires sont mises en cache et supposées fiables pour accélérer le temps de vérification. L'utilisation d'un hachage permet de prévenir toute modification ou tout déplacement d'un bloc mémoire.

Cette architecture a été intégrée dans un processeur OpenRISC. Ce circuit a même été fondu sur silicium en technologie TSMC $18\mu m$. La version sécurisée du processeur est environ deux fois supérieure en taille par rapport au processeur de base. C'est le module de chiffrement qui contribue principalement à cette logique supplémentaire

(environ 30% de la surface totale). Pour ce qui est des performances, en mode PTR pour un cache de taille moyenne 32KB les ralentissements sont imperceptibles. En revanche pour une taille de cache plus faible (4KB), des ralentissements allants jusqu'à 73% sont mesurés.

CryptoPage

L'architecture CryptoPage [Duc07] proposée par Guillaume Duc dans sa thèse combine les avancées en méthodes de chiffrement et d'intégrité avec les techniques de protection de fuite d'information sur le bus d'adresse de HIDE. L'architecture propose en plus toute l'infrastructure logicielle permettant l'exécution sécurisée. L'évaluation est faite à l'aide du simulateur SimpleScalar, les ralentissements observés vont jusqu'à 25%, mais sont réduits jusqu'à 3% en utilisant une architecture plus spéculative (superposition de la vérification d'intégrité avec l'exécution des instructions).

Address Independent Seed Encryption (AISE)

Les architectures présentées jusqu'à présent permettent pour la plupart de faire cohabiter des processus indépendants. En revanche, aucune architecture ne permet de partager efficacement des données entre processus. En effet, ceci est dû à l'utilisation de l'adresse virtuelle comme composant du vecteur d'initialisation pour le chiffrement. Dans [RCPS07, CRSP09] les auteurs introduisent un schéma de choix de vecteurs d'initialisation basé non plus sur l'adresse virtuelle, mais sur un numéro de page logique (pour garantir l'unicité spatiale) et un compteur d'accès (pour prévenir le rejeu). Le papier introduit également une variante pour la vérification d'intégrité, en ne construisant l'arbre de hachage que sur les compteurs (*Bonsai Merkle Tree*).

2.3.3. Permutation aléatoire du jeu d'instructions

Une forme plus légère de chiffrement de code est utilisée dans les techniques d'*Instruction Set Randomisation* (ISR) ou encore *Randomised Instruction Set Execution* (RISE). Ces méthodes visent à limiter la rétro-conception du code, mais également prévenir l'injection de code (en particulier la conception d'*exploits)*. Le mécanisme de base utilisé est de permuter le jeu d'instruction aléatoirement (par exemple les *opcodes*, les identificateurs de registres). Ainsi, un attaquant qui n'a pas accès à cette permutation aura bien plus de difficultés à injecter du code.

La détection d'une injection de code est basée sur les états invalides du processeur (voir figure 2.4). Lorsque du code aléatoire est exécuté, on comprend qu'une erreur arrive avec une forte probabilité. Une injection est réussie sont lorsqu'un effet extérieur est produit (accès à un périphérique, sortie d'une donnée) ou que le flot de contrôle est dérivé.

Les techniques d'ISR ont été introduites dans [BAP+03] et supportées à l'aide d'une machine virtuelle, rendant plus simple le déploiement du mécanisme. Pour chiffrer un code, une séquence aléatoire unique de la même taille que le programme est générée, puis, chaque instruction est chiffrée et déchiffrée à l'aide d'un «XOR». Il est essentiel que le déchiffrement soit rapide, car ses performances impactent directement le débit d'instructions exécutées.

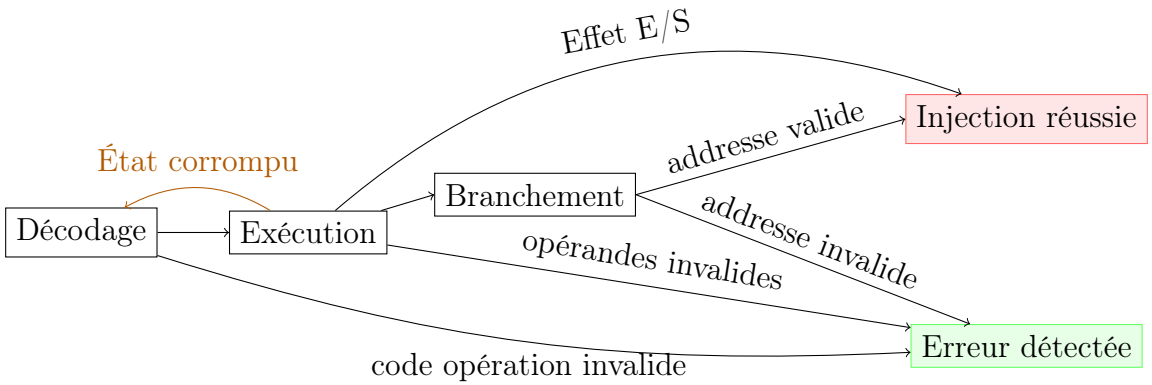


FIGURE 2.4. – Représentation des états du processeur lors de l'exécution de code aléatoire

Plus récemment, les auteurs de [PLPI13] ont montré comment effectuer cette permutation de manière matérielle, ce qui permet d'offrir une sécurité similaire aux architectures de processeur sécurisées pour le flux d'instructions. La permutation est faite soit par un vrai chiffrement par bloc placé avant un cache, ou après un cache en effectuant un simple « XOR» avec une constante.

Dans [DGP14], les auteurs proposent un mécanisme d'ISR basé sur un chiffrement par flux (Trivium) et l'évaluent sur un processeur LEON 3. Le déchiffrement est assez rapide pour être placé après le cache. La permutation du jeu d'instruction est faite à l'aide d'une instruction supplémentaire `rev`, qui permet de changer une partie du vecteur d'initialisation du chiffrement par flux.

En se basant sur un mécanisme d'ISR, l'architecture SOFIA [DCDKC+16] cherche à assurer l'intégrité du flot de contrôle d'un programme. L'idée est de rendre le déchiffrement dépendant d'informations relatives au chemin d'exécution (par exemple adresse courante, adresse précédente). Ici, le déchiffrement est effectué à l'échelle de l'instruction à l'aide d'un chiffrement par blocs, RECTANGLE, en mode compteur. La solution est intégrée elle aussi dans un processeur LEON 3. L'augmentation en taille de circuit est de 30 % (ce qui inclut le mécanisme d'intégrité et de chiffrement) et le temps d'exécution est doublé.

2.4. Comparaison et limites actuelles

Les techniques d'obfuscation offrent une solution purement logicielle pour assurer la confidentialité d'un programme. L'avantage indéniable est la facilité d'intégration. En effet, elles sont applicables sur n'importe quel système capable d'exécuter un programme. En revanche, la sécurité qu'elles apportent est très difficile à prouver et les coûts en taille de programme et temps d'exécution sont très importants (facteur 1000 au moins).

Une autre approche consiste à encapsuler un certain nombre de fonctionnalités dans

une zone quasiment inviolable physiquement. La sécurité apportée par cette approche est très forte. Cependant, les fonctionnalités proposées sont réduites, il en résulte un manque de flexibilité par rapport aux applications supportées. Si ces solutions se prêtent parfaitement à l'implémentation d'API cryptographiques (e.g., TPM), il est plus difficile d'envisager leur utilisation pour la protection de programmes arbitraires.

Enfin un grand nombre de travaux se sont orientés vers une option alternative : considérer le processeur comme une zone fiable, inaccessible à un attaquant. Dès lors, on cherche à protéger les interactions avec la mémoire externe qui est la zone vulnérable du système. Nous avons vu qu'un nombre important de techniques permettant d'assurer la confidentialité, l'intégrité ou encore prévenir le rejeu. Nous pouvons raisonnablement conclure que les techniques disponibles aujourd'hui sont bien établies.

Cependant, les contre-mesures sont applicables pour des processeurs de taille relativement importante et se basent principalement sur des mémoires cache pour masquer les latences. Dans le cas de l'architecture AEGIS par exemple, de forts ralentissements sont observés lorsque les caches utilisés sont plus petits. Aussi, peut-on adapter ces protections à des systèmes plus contraints en ressource ou n'ayant pas de mémoire cache ?

Il peut être souhaitable d'effectuer le chiffrement sans supposer la présence d'un cache. À la fois pour protéger les systèmes qui n'en n'ont pas, mais aussi pour prévenir une éventuelle fuite d'information par les mémoires caches.

2.5. Conclusion

Nous avons présenté dans ce chapitre plusieurs méthodes applicables pour la protection de la confidentialité d'un code. Les architectures de processeurs qui utilisent du chiffrement mémoire offrent une forte sécurité. L'hypothèse faite en termes de sécurité est l'existence d'une zone de confiance où les opérations de chiffrement, de déchiffrement et également les manipulations de données peuvent être effectuées en toute sécurité. Cependant, le chiffrement mémoire augmente significativement la taille du circuit et la latence d'accès à la mémoire. Sur des systèmes de taille déjà importante, l'effet reste modéré. De plus, les latences sont réduites en effectuant le déchiffrement avant de mettre dans une mémoire cache. Aussi, ces mécanismes semblent plus difficiles à intégrer dans les systèmes plus petits et plus contraints en ressources.

Les solutions proposées dans cette thèse visent à maintenir le chiffrement d'un programme et de ses données le plus longtemps possible au fil de l'exécution. Nous proposons tout d'abord un mécanisme léger de chiffrement de code à la granularité de l'instruction pouvant adresser les systèmes contraints ou souhaitant déchiffrer après la mémoire cache, c'est-à-dire juste avant de décoder l'instruction. Puis, dans un second temps, nous évaluons la faisabilité des algorithmes de chiffrement homomorphe pour maintenir le chiffrement non seulement durant le stockage, mais également durant les manipulations.

Chiffrement d'un flot d'instructions 3

Sommaire

3.1. Introduction

Nous décrivons dans cette partie une méthode de chiffrement d'un programme capable de déchiffrer au fil de l'exécution les instructions une par une. Supporter efficacement un tel mécanisme présente plusieurs avantages. D'un point de vue temporel, le déchiffrement est effectué à l'exécution de chaque instruction, ce qui limite les problématiques de type *time of verification, time of execution* où le programme pourrait être accédé en clair dans diverses mémoires temporaires. De plus, le chiffrement est maintenu au plus proche du processeur dans la hiérarchie mémoire, rendant l'accès au code en clair plus difficile physiquement. Enfin, le déchiffrement à l'échelle de l'instruction est une nécessité pour intégrer le chiffrement d'instructions dans des systèmes très contraints qui n'auraient pas de mémoire cache.

Il est essentiel que le chiffrement soit léger pour adresser des systèmes contraints. Il doit également être efficace. En effet, le déchiffrement est sur le chemin critique

d'exécution du processeur, il faut idéalement être capable de déchiffrer à la même vitesse que la lecture d'instructions pour ne pas ralentir l'exécution. Les chiffrements par flots apparaissent comme de bons candidats pour cette tâche, ils sont très légers et efficaces par rapport à la majorité des chiffrements par blocs et permettent d'effectuer le déchiffrement de manière itérative (symbole par symbole). Ceci permet d'exploiter la nature séquentielle des accès à la mémoire d'instructions.

Cependant, l'utilisation de chiffrements par flots dans un tel contexte n'est pas immédiate. Nous étudierons les différents problèmes soulevés et les solutions possibles. Enfin, cette solution peut être vue de manière plus générale comme une méthode de chiffrement basée sur le flot de contrôle du programme et peut être utilisée avec la plupart des primitives de chiffrement.

3.2. Chiffrements par flots

Nous prenons comme définition d'un chiffrement par flots une approche au chiffrement où l'on génère une séquence pseudo-aléatoire utilisée comme un masque additif pour produire un message chiffré. En opposition, un chiffrement par blocs se base sur une permutation pseudo-aléatoire à taille fixe et donc produit des chiffrés par blocs. Un chiffrement par flots peut quant à lui chiffrer un symbole à la fois. Ceux destinés à une implémentation matérielle sont généralement capables d'effectuer le chiffrement bit par bit.

Un chiffrement par flots est constitué d'un générateur pseudo-aléatoire, dont on peut initialiser l'état à l'aide d'une clé secrète et d'un vecteur d'initialisation (IV). Formellement, deux fonctions déterministes le définissent :

— `init` : $\mathcal{K} \times \mathcal{IV} \to \mathcal{S}$, qui à partir d'une clé secrète et d'un vecteur d'initialisation génère un état initial $state_0$.
— `genBits` : $\mathcal{S} \to \mathcal{S} \times \mathcal{O}$, qui à partir de l'état courant $state_t$, génère l'état suivant $state_{t+1}$ et une sortie pseudo-aléatoire.

Pour chiffrer un message avec une clé donnée, un vecteur d'initialisation est tout d'abord sélectionné et transmis comme entête au chiffré. Ensuite, l'état initial est généré en calculant $state_0 = \texttt{init}(key, IV)$. Puis, la fonction `genBits` est itérée et le flot aléatoire obtenu est combiné à l'aide d'un « XOR » avec le message original. Ce processus est représenté sur la figure 3.1. Le déchiffrement consiste à générer la même suite aléatoire à partir de l'IV présent en tête du message chiffré et de retirer le masque en effectuant un « XOR ».

De manière générale, la fonction `genBits` est une opération très efficace, de sorte que chiffrer de longues séquences soit avantageux. À l'opposé, la fonction `init` est souvent plus coûteuse. En effet, son rôle est de générer un état le plus aléatoire possible à partir d'une clé et d'un IV.

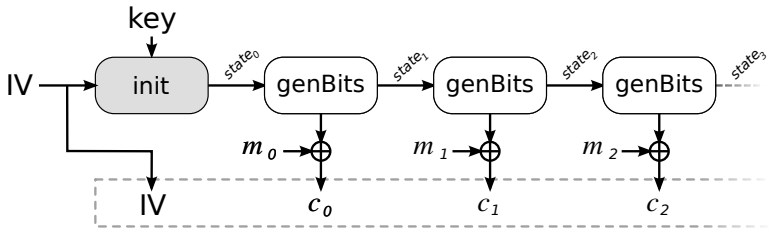


FIGURE 3.1. – Représentation graphique d'un chiffrement par flots

3.2.1. Le chiffrement Trivium

La compétition eStream [Bil08] qui s'est achevée en 2008 a sélectionné plusieurs candidats en séparant les profils matériels et logiciels. Pour le profil matériel, trois candidats ont été retenus, Grain, Mickey et Trivium.

Le chiffrement Trivium a été retenu pour l'évaluation de la solution. Son circuit est d'une simplicité remarquable et aucune attaque n'est connue sur la version originale (seulement sur des versions réduites). Selon l'étude effectuée dans [Rog07], c'est également celui qui offre le meilleur débit par rapport à la surface sur tous les candidats de la compétition eStream. Il est en effet capable de générer jusqu'à 64 bits par cycles d'horloges, là où Grain (également très performant) ne peut générer que 16 bits au plus.

Le chiffrement Trivium supporte une taille de clé et de vecteur d'initialisation de 80 bits. Son état interne est constitué de 288 bits. L'algorithme 3.1 spécifie la fonction `genBits` de Trivium. Nous décrivons une version où le nombre de bits de sortie par itération est explicite (nous parlerons aussi de niveau de parallélisme). La spécification officielle du schéma [DCP08] représentée sur la figure 3.3 correspond au cas $p = 1$. On voit que Trivium est constitué de trois registres à décalage qui apparaissent des lignes 13 à 15 de l'algorithme. Ils commencent aux indices 0, 93 et 177 de l'état. Pour déterminer le niveau de parallélisme maximal, il faut s'assurer que les fenêtres accédées sur l'état s sont simplement des translations de l'état précédent. La contrainte la plus forte en ce sens est la ligne 2 de l'algorithme qui limite le niveau de parallélisme à $p \leq 66$ (car il faut $w = p - 1 \leq 65$).

Il s'agit ici d'un parallélisme réel, spécifique à Trivium, le nombre de bits en sortie par itération est augmenté, mais pas la profondeur du circuit. Il est toujours possible, avec n'importe quel chiffrement par flots de chaîner des itérations, pour augmenter le nombre de bits de sortie par itération. En revanche, la profondeur de circuit sera en général multipliée par le degré de parallélisme souhaité. Le processus est représenté sur la figure 3.2. En réalité, la version parallèle présentée dans l'algorithme 3.3 est obtenue exactement par cette transformation générique. Pour Trivium, il apparaît que tant que $p \leq 66$, il n'y a pas d'augmentation de profondeur. Au vu de la faible profondeur du circuit original (1 porte « ET» et 2 portes « XOR »), une version 128 bits du chiffrement, où l'on chaîne deux itérations de largeur 64 bits, est parfaitement

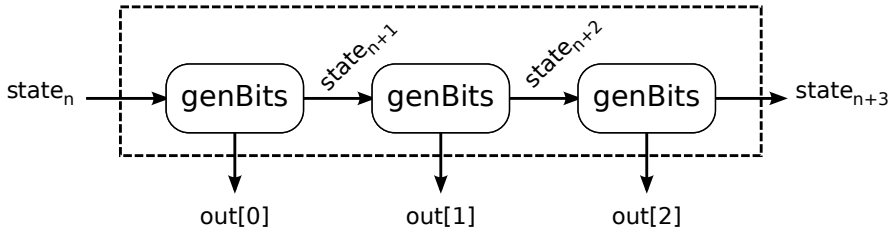


FIGURE 3.2. – Augmentation générique du nombre de bits générés par itération d'un chiffrement par flots, dans cet exemple, le débit est multiplié par trois.

envisageable.

Algorithme 3.1 Fonction `genBits` de Trivium

Entrée : s l'état courant, $p \geq 1$, le niveau de parallélisme

1: $w \leftarrow p - 1$
2: $t_1 \leftarrow s[65 - w,\ 65] \oplus s[92 - w,\ 92]$
3: $t_2 \leftarrow s[161 - w,\ 161] \oplus s[176 - w,\ 176]$
4: $t_3 \leftarrow s[242 - w,\ 242] \oplus s[287 - w,\ 287]$
5:
6: $u_1 \leftarrow s[90 - w,\ 90] \wedge s[91 - w,\ 91]$
7: $u_2 \leftarrow s[174 - w,\ 174] \wedge s[175 - w,\ 175]$
8: $u_3 \leftarrow s[285 - w,\ 285] \wedge s[286 - w,\ 286]$
9: $w_1 \leftarrow t_1 \oplus u_1 \oplus s[170 - w,\ 170]$
10: $w_2 \leftarrow t_2 \oplus u_2 \oplus s[263 - w,\ 263]$
11: $w_3 \leftarrow t_3 \oplus u_3 \oplus s[68 - w,\ 68]$
12:
13: $s_{next}[0,\ 92] \leftarrow w_1 \| s[0,\ 92 - p]$
14: $s_{next}[93,\ 176] \leftarrow w_2 \| s[93,\ 176 - p]$
15: $s_{next}[177,\ 287] \leftarrow w_3 \| s[177,\ 287 - p]$
16: **return** $s_{next},\ t_1 \oplus t_2 \oplus t_3$

La fonction d'initialisation de Trivium de l'algorithme 3.2 est construite à partir de la fonction `genBits`. Après placement de la clé et du vecteur d'initialisation dans l'état initial, il faut effectuer 1152 itérations à vide. Il est tout à fait possible d'utiliser une version parallèle de la fonction `genBits` comme celle de l'algorithme 3.1 pour réduire le nombre d'itérations. En particulier, si le niveau de parallélisme p divise 1152, le nombre de cycles d'initialisation est réduit à $1152/p$. Par exemple, 18 cycles pour un parallélisme $p = 64$, ou encore 9 cycles pour $p = 128$.

Aucune cryptanalyse réalisable en temps raisonnable sur Trivium n'est connue à ce jour. L'attaque considérée en général consiste à retrouver la clé à partir du flux de sortie, dans l'hypothèse d'un IV choisi librement par l'attaquant. Une approche par énumération complète de toutes les clés est évidemment toujours possible. De ce point de vue, la taille de 80 bits est assez faible par rapport aux 128 bits actuelle-

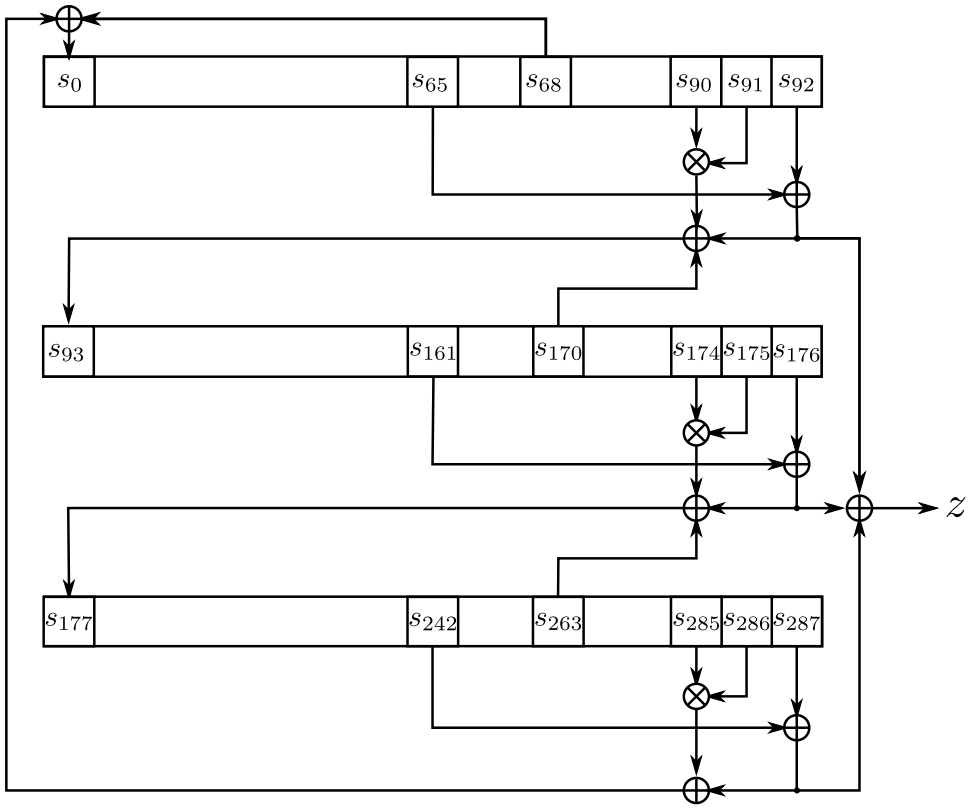


FIGURE 3.3. – Circuit du chiffrement par flots Trivium

Algorithme 3.2 Fonction `init` de Trivium

Entrée : k une clé secrète, IV un vecteur d'initialisation

1: $s \leftarrow 0 \dots 0$
2: $s[0,\ 79] \leftarrow k$
3: $s[92,\ 176] \leftarrow IV$
4: $s[285,\ 287] \leftarrow 1,\ 1,\ 1$
5: **for** $i = 0$ to $4 \cdot 288$ **do**
6: $\quad s, \_ \leftarrow genBits(s)$ ▷ ignorer la sortie
7: **end for**
8: **return** s

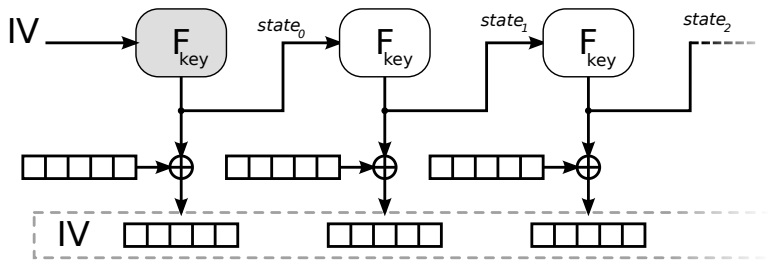


FIGURE 3.4. – Chiffrement par bloc en mode *Output Feedback*

ment recommandés pour les années à venir. Les seules attaques plus efficaces connues concernent des versions du schéma où le nombre de cycles d'initialisation est réduit. L'attaque présentée dans [KMN12] permet de distinguer un flot de Trivium d'un flot aléatoire pour une classe de 2^{26} clés avec 961 cycles d'initialisation. L'attaque de [FV14] retrouve intégralement n'importe quelle clé pour 799 cycles d'initialisation, mais nécessite $O(2^{62})$ calculs. La meilleure cryptanalyse en un temps raisonnable est effectuée sur un Trivium avec 645 cycles d'initialisation et nécessite $O(2^{42.2})$ calculs [QW14] pour retrouver la clé.

3.2.2. Construction à partir d'un chiffrement par bloc

Trivium est un bon exemple de chiffrement par flots extrêmement efficace. Mais, il est également possible de construire un chiffrement par flots à partir d'un chiffrement par blocs. Par exemple en utilisant le mode de chiffrement *Output Feedback* (OFB). Notons F_{key} la fonction de chiffrement par blocs, spécialisée pour une clé secrète *key*. L'état comprend la clé secrète ainsi qu'un bloc (e.g, 128 bits). Les fonctions qui caractérisent un chiffrement par flots s'expriment de la manière suivante :

— `init`$(key, IV) = F_{key}(IV)$
— `genBits`$(s) = (F_{key}(s), F_{key}(s))$

Un chiffrement par blocs en mode compteur peut également être vu comme un chiffrement par flots, l'état est constitué de la clé et de la valeur de compteur courante. En revanche le mode de chiffrement *Cipher Bloc Chaining* (CBC) ne peut pas réellement être considéré comme un chiffrement par flots, car le message à chiffrer serait utilisé comme entrée de la fonction `genBits`.

3.3. Chiffrement d'un programme

Si l'on observe les méthodes de chiffrement de code utilisées dans la littérature (présentées section 2.3), on remarque qu'elles n'ont pas, voire très peu, d'état interne :

— avec un chiffrement basé sur un mode compteur, l'adresse (la valeur de compteur) seule est suffisante avec la clé pour déchiffrer directement l'instruction courante.

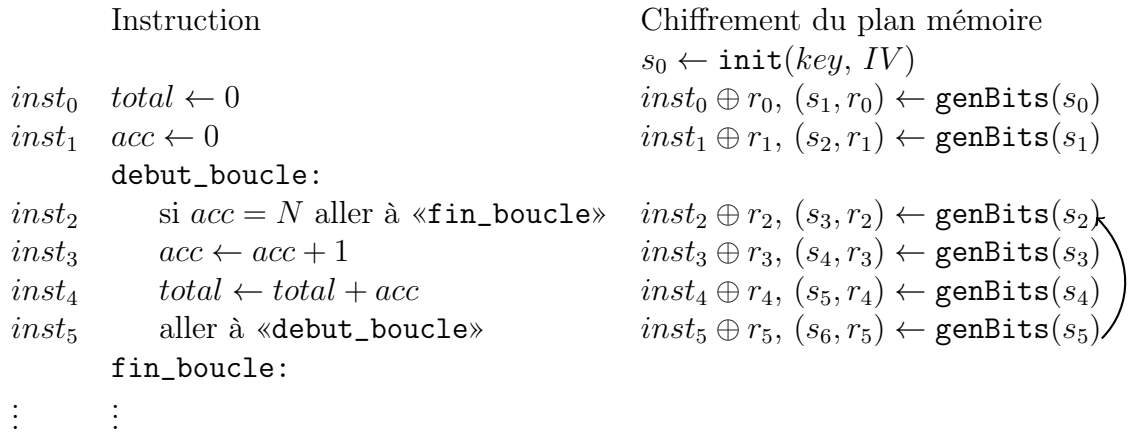


FIGURE 3.5. – Chiffrement du plan mémoire avec une unique suite chiffrante

— Quand les données sont déchiffrées par blocs de taille fixe, l'état de la primitive de chiffrement est local au bloc seulement.

À l'opposé, les chiffrements par flots sont intrinsèquement basés sur un état pour générer leur flot aléatoire. En conséquence, il n'est pas possible de reconstruire simplement un état donné, autrement dit déchiffrer directement le milieu d'un message chiffré. La seule manière générique de procéder pour construire un état donné est de rejouer l'exécution du chiffrement par flots depuis l'initialisation jusqu'à l'état souhaité.

Ceci pose une difficulté immédiate pour chiffrer un programme : lorsqu'un branchement est effectué par le programme, il est alors nécessaire d'obtenir l'état utilisé pour chiffrer la destination du branchement pour continuer à déchiffrer. Par exemple, sur la figure 3.5, le programme a été chiffré avec une unique suite chiffrante. Quand le branchement est effectué depuis l'instruction $inst_5$ vers $inst_2$, l'état courant vaut s_6, il faut alors réinitialiser l'état du chiffrement par flots à s_2 pour continuer à déchiffrer correctement. Pour se faire, il faut donc réinitialiser le chiffrement par flots au début de la suite chiffrante ($inst_0$) et l'exécuter à vide jusqu'à l'instruction cible ($inst_2$). Pour des questions de performances, il n'est clairement pas envisageable d'effectuer ce rejeu depuis le début du programme à chaque branchement. Cette observation suggère donc l'utilisation d'une structure plus fine que le programme entier pour le chiffrement.

3.3.1. Chiffrement basé sur le flot de contrôle

Définition 1 (Bloc de base)**.** Un bloc de base est une suite d'instructions ayant :

1. un unique point d'entrée (la première instruction),
2. un seul branchement sortant (la dernière instruction).

Cette structure est fondamentale dans la construction de compilateurs. On parlera de graphe de flot de contrôle (CFG) pour désigner la décomposition d'un programme sous forme de blocs de base.

Un bloc de base se prête parfaitement à un chiffrement avec une unique suite chiffrante. Le point d'entrée est par définition la première instruction, le problème de restaurer des états arbitraires n'apparaît plus. En effet, tous les états à réinitialiser sont connus statiquement et correspondent à la première instruction des blocs de base. C'est également le début de la suite chiffrante, dont l'état est généré par la fonction `init` du chiffrement par flots.

L'algorithme de chiffrement d'un programme basé sur le flot de contrôle procède donc de la manière suivante :

1. Construire le graphe de flot de contrôle du programme.
2. Pour chaque bloc de base, sélectionner un vecteur d'initialisation (nous discuterons des alternatives possibles pour ce choix dans la section 3.3.4).
3. Chiffrer les différents blocs avec la clé secrète associée au programme.

Pour déchiffrer à l'exécution, il faut réinitialiser le chiffrement par flots à chaque branchement.

3.3.2. Restauration des branchements entre blocs successifs

Le fait d'associer une suite chiffrante par bloc de base pour le chiffrement de code implique que l'on puisse détecter tous les changements de bloc pour réinitialiser correctement l'état du chiffrement par flots. D'après la définition 1, chaque branchement effectué marque un changement de bloc de base. Cependant, la réciproque n'est pas vraie, certains blocs sont atteints sans branchements.

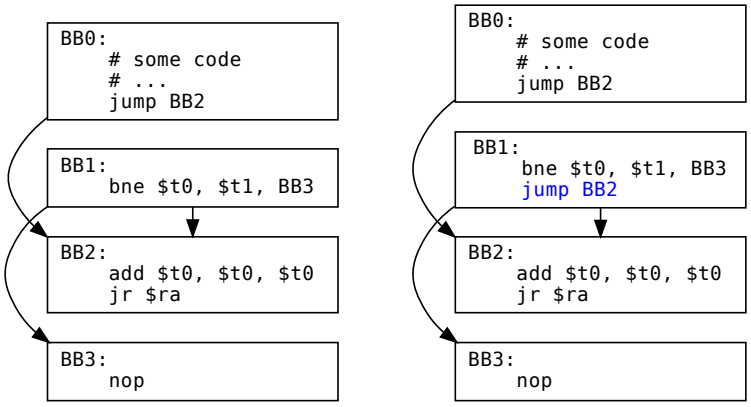
Ce cas très précis est dû à une optimisation effectuée par les compilateurs : lorsque deux blocs de bases sont consécutifs dans le code, il est inutile d'avoir un branchement entre les deux. Cette optimisation fait parfaitement sens en temps normal, mais pose problème pour le chiffrement de code où il faut obligatoirement changer de suite chiffrante. Ainsi, cette optimisation doit être désactivée pour permettre l'exécution correcte de code chiffré.

La figure 3.6 illustre cette situation sur un programme réel. Sur la version compilée du programme, le branchement du bloc de base BB1 à BB2 est supprimé. Or, si le code est chiffré, il faut impérativement un branchement explicite entre ces deux blocs de base.

3.3.3. Blocs de base étendus

Si un bloc de base est une structure adaptée au chiffrement, ils ont souvent une taille très petite. D'un point de vue des performances, il est intéressant de maximiser la taille des blocs de base pour minimiser les appels coûteux à la fonction `init` du chiffrement par flots.

On remarque que pour chiffrer de manière correcte, seule la première condition de la définition d'un bloc de base est nécessaire. Autrement dit, des séquences avec un nombre arbitraire de branchements sortants peuvent être chiffrées. Ceci nous amène à la définition suivante, d'un bloc de base étendu pour le chiffrement.



(a) Branchement de BB1 à BB2 optimisé

(b) Branchement de BB1 à BB2 explicite

FIGURE 3.6. – Situation nécessitant l'ajout d'un branchement explicite

Définition 2 (Bloc de base chiffrable)**.** Un bloc de base chiffrable est une suite d'instruction :

1. ayant un unique point d'entrée (la première instruction),
2. contenant un nombre arbitraire de branchements sortants,
3. dont la dernière instruction est un branchement inconditionnel [1].

3.3.3.1. Fusion statique

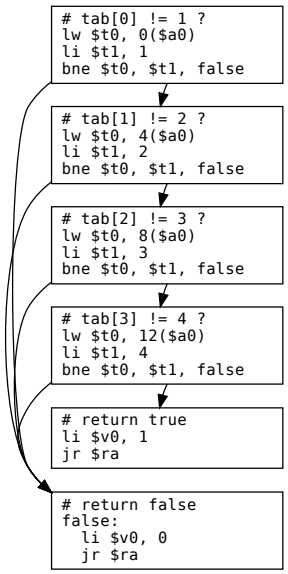
Une manière très simple d'augmenter la taille des blocs de base est d'appliquer une stratégie de fusion des blocs consécutifs. Pour deux blocs de base consécutifs dans le code, si le second n'est atteignable que depuis le premier, alors les deux blocs peuvent être fusionnés. Cette fusion deux à deux est répétée tant qu'un point fixe n'est pas atteint.

Cette optimisation s'applique principalement aux successions de structures conditionnelles. La figure 3.7 montre un programme qui vérifie qu'un tableau commence par un préfixe donné, ici : 1, 2, 3, 4. Pour cela, une une séquence de tests conditionnels est utilisée. On retrouve souvent de telles séquences de tests dans les programmes lorsque les fonctions vérifient des pré-conditions sur leurs paramètres ou des post-conditions sur leurs sorties. Le graphe de flot de contrôle original de la figure 3.7 est composé de 6 blocs de base. L'exécution chiffrée d'un tel programme serait donc très lente. Il faudrait dans le pire des cas réinitialiser 5 fois le chiffrement par flots. Il s'agit ici du cas typique où la fusion proposée s'applique. Ce graphe possède en réalité seulement deux blocs de base chiffrables au sens de la définition 2.

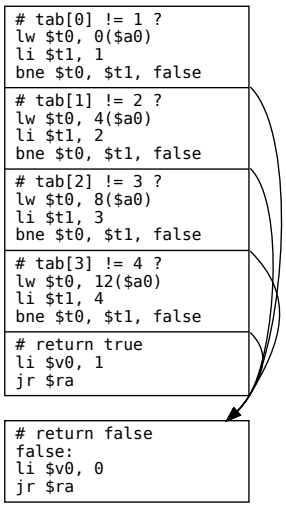
1. Cette contrainte est nécessaire pour prendre en compte la problématique discutée dans la section 3.3.2.

```
bool f(int tab[]) {
    if (tab[0] != 1)
        return false;
    if (tab[1] != 2)
        return false;
    if (tab[2] != 3)
        return false;
    if (tab[3] != 4)
        return false;
    return true;
}
```

(a) Programme source



(b) CFG original



(c) CFG après fusion

FIGURE 3.7. – Fusion statique de structures conditionnelles pour le chiffrement de code

3.3.3.2. Techniques de fusion avancées

Si l'on ne considère que les deux premiers points de la définition 2, il s'agit là encore d'une structure standard en compilation connue sous le nom de *superblock* [HMC+93]. Ces blocs de base étendus sont utilisés principalement dans la compilation pour les processeurs *Very Long Instruction Word* (VLIW). Pour ces architectures, il faut augmenter statiquement au maximum le parallélisme d'instructions, pour exploiter le fort potentiel d'exécution parallèle. Un bloc de base avec plus d'instructions, présente naturellement plus de choix quant à l'ordonnancement des instructions.

Pour augmenter la taille des *superblocks,* un mécanisme central est la *tail-duplication*, qui consiste à dupliquer un bloc de base ayant plusieurs prédécesseurs. D'autres optimisations comme le déroulement de boucles, la suppression de structures conditionnelles par *if-conversion* [MLC+92] permettent également d'augmenter la taille des blocs.

En revanche, ces méthodes nécessitent une trace d'exécution caractéristique du programme, qui permet d'identifier les chemins critiques dans le flot de contrôle à optimiser. En effet, ces optimisations se basent principalement sur la duplication de blocs, la taille du programme risque de devenir bien trop grande si elles ne sont pas ciblées précisément.

3.3.4. Choix des vecteurs d'initialisation

Nous n'avons pas encore précisé la méthode de choix des vecteurs d'initialisation associés à chaque bloc de base étendu. Un vecteur d'initialisation est une donnée publique, néanmoins, il est impératif pour garantir la sécurité du chiffrement qu'ils soient uniques à travers tout le programme.

3.3.4.1. Vecteur complètement aléatoire

Une première option est de tirer pour chaque bloc de base un vecteur d'initialisation complètement aléatoire. Il faut ensuite spécifier un format de stockage, par exemple en début de bloc de base. Cette approche a un fort effet négatif sur la taille de code : il faut stocker autant d'IVs que de blocs de base chiffrables dans le programme.

3.3.4.2. Calculs à partir d'éléments d'adresse

Une seconde possibilité est de calculer l'IV à partir de l'adresse de la première instruction du bloc de base courant. En comparaison à la précédente, cette solution a très peu d'effet sur la taille du programme, l'IV n'est pas stocké dans le code.

De plus, utiliser l'adresse comme partie du vecteur d'initialisation a un avantage, cela permet de prévenir le déplacement de parties du code. En effet, si une partie de code est déplacée, alors le vecteur d'initialisation calculé à l'exécution sera incorrect. En conséquence, du code arbitraire sera exécuté menant avec forte probabilité à une erreur détectable (voir figure 2.4). Dans [DCDKC+16] les auteurs suggèrent de calculer l'IV à partir de l'adresse courante et également l'adresse du bloc précédent. Ceci

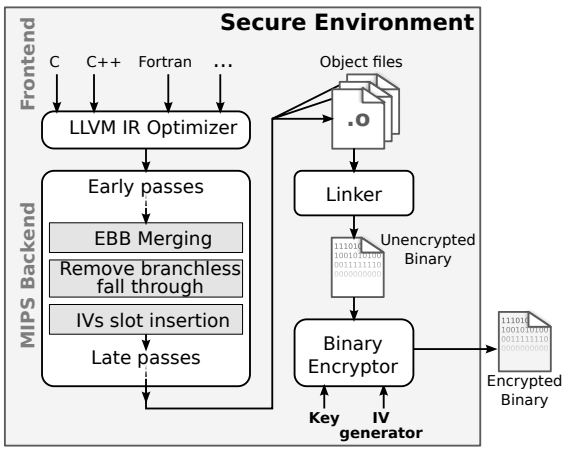


FIGURE 3.8. – Flot de compilation modifié pour inclure le chiffrement

permet de forcer certains chemins d'exécution précis (intégrité de flot de contrôle). On peut parfaitement baser cette application intéressante sur la méthode de chiffrement proposée.

L'utilisation de l'adresse comme partie de l'IV impose une position fixe au code. Ce qui élimine, ou du moins oblige à repenser la gestion du partage de code. Le problème est le même que dans [CRSP09], le chiffrement est incompatible avec la mémoire virtuelle, où une même zone physique peut-être accédée par des adresses virtuelles différentes.

Enfin, il est toujours possible de combiner les deux approches. C'est-à-dire de construire un IV constitué d'une partie aléatoire stockée dans le code et d'une partie implicite calculée à partir de l'adresse du bloc.

3.3.5. Intégration dans une chaîne de compilation

La compilation d'un programme incluant le chiffrement procède en deux étapes et est représentée figure 3.8. Tout d'abord, les pré-traitements et optimisations décrites dans cette section ont été intégrés dans une chaîne de compilation LLVM [LA04]. Puis, le chiffrement pour une clé secrète donnée est effectué dans une seconde étape sur le programme final.

3.3.5.1. Le projet LLVM

LLVM était à l'origine l'acronyme de *Low-Level Virtual Machine*, une «machine virtuelle bas niveau» permettant d'effectuer l'optimisation des programmes indépendamment de l'architecture cible. Le projet LLVM a depuis bien évolué, c'est à ce jour une suite d'outils de construction de compilateurs sous la forme de bibliothèques écrites en langage C++. L'architecture logicielle est d'une qualité remarquable et extrêmement modulaire, ce qui contribue au succès toujours grandissant du projet.

La chaîne de compilation est divisée en trois modules. Le *front-end*, spécifique à chaque langage (C, C++, Fortran, ...) génère à partir d'un programme une représentation intermédiaire (IR). Il s'agit d'un assembleur extrêmement générique, pour une machine fictive. Un grand nombre d'optimisations sont effectuées indépendamment du processeur cible sur cette représentation intermédiaire. Enfin, c'est le *backend,* spécifique à chaque architecture (x86, ARM, MIPS, ...), qui effectue les dernières optimisations ainsi que la génération de code. Grâce à cette structure très modulaire, il est aisé d'accéder et de manipuler le graphe de flot de contrôle. On peut ainsi inclure des traitements sur la représentation intermédiaire au moment de l'optimisation ou sur une représentation plus bas niveau au moment de la génération de code.

3.3.5.2. Avantages par rapport à un traitement sur le binaire

Utiliser une suite comme LLVM présente un coût de prise en main non négligeable. Cependant, les efforts investis sont largement compensés :

— Les traitements sont intégrés de manière uniforme dans le pipeline de compilation. De nombreuses informations additionnelles sont accessibles pour l'optimisation, lesquelles sont loin d'être triviales à extraire depuis un fichier binaire brut. De plus, les traitements peuvent s'intégrer entre des optimisations déjà existantes pour en bénéficier.

— Les transformations sont effectuées sur une représentation abstraite du programme. Des opérations comme l'émission de code, le calcul des adresses sont complètement transparentes.

En travaillant directement sur un fichier binaire, les transformations de code sont beaucoup plus complexes. Par exemple, pour une simple insertion d'instruction, il faut s'assurer que le plan mémoire est toujours cohérent et mettre à jour si nécessaire les adresses calculées statiquement.

3.3.5.3. Pré-traitements

Les pré-traitements et optimisations d'un programme pour le chiffrement prennent la forme d'un ensemble de passes [2] additionnelles. Ces passes sont toutes effectuées dans la couche de génération de code de l'architecture MIPS (l'architecture utilisée pour la validation).

L'intégration d'une passe sur la représentation intermédiaire serait beaucoup plus aisée. Il est en effet possible de développer et de compiler de telles passes en dehors du projet LLVM et de les charger dynamiquement. Malheureusement, les pré-traitements nécessitent que le placement des blocs de base ait été effectué. Ce placement est effectué en début de génération de code, ce qui oblige donc à travailler après cette passe. En revanche, les éventuelles optimisations d'élargissement de blocs de base peuvent être effectuées sur la représentation intermédiaire.

Trois passes sont incluses, tout d'abord la fusion des blocs de base (section 3.3.3.1), puis l'ajout des branchements manquants entre blocs successifs (section 3.3.2). Enfin,

2. La terminologie utilisée pour décrire une étape de transformation.

une dernière étape, optionnelle selon la politique de choix des IVs, est chargée d'insérer dans chaque bloc de base la place nécessaire au stockage des vecteurs d'initialisation. La modularité de LLVM permet de bénéficier des dernières passes d'optimisation, en particulier le remplissage des *delay-slots* (l'instruction après un branchement) qui permet de compenser l'ajout de branchements supplémentaires.

3.3.5.4. Chiffrement complet du programme

La dernière étape de la chaîne de compilation effectue le chiffrement complet du programme. Il est important que cette étape soit effectuée sur le programme complet, après l'étape d'édition de liens. Ceci permet de faire le choix des vecteurs d'initialisation et de s'assurer de l'unicité à travers tout le programme.

L'outil de chiffrement, pour un binaire donné, reconstruit dans un premier temps le graphe de flot de contrôle, puis utilise ensuite une implémentation logicielle de la primitive de chiffrement (Trivium par exemple) pour chiffrer chaque bloc de base.

3.4. Support matériel

Pour supporter l'exécution de code chiffré, le processeur doit évidemment être modifié. Il faut d'une part ajouter le circuit de déchiffrement. De plus, les branchements doivent déclencher un changement de vecteur d'initialisation.

La figure 3.9 représente une intégration possible dans un processeur RISC classique. Le déchiffrement est placé après la lecture d'une instruction depuis la mémoire et avant son décodage. La profondeur de circuit additionnelle correspond simplement à un «XOR». Le vecteur d'initialisation est calculé par une fonction f prenant en entrée l'adresse courante et la donnée lue depuis la mémoire. Ceci permet de lire les vecteurs d'initialisation stockés dans le code. Enfin, le module de déchiffrement expose deux signaux pour la logique de contrôle du processeur. Un premier destiné à bloquer l'exécution lorsque la sortie du chiffrement par flots n'est pas encore prête. Ce signal est activé juste après un branchement, pendant la réinitialisation du chiffrement. Un second signal est utilisé pour ignorer l'instruction courante. Ce dernier sert à lire les vecteurs d'initialisations depuis la mémoire. Comme il ne s'agit pas d'instructions, mais de données brutes, cette zone ne doit pas être exécutée.

3.4.1. Activation et désactivation du chiffrement

Il est essentiel de pouvoir désactiver le chiffrement, en particulier pour exécuter des routines partagées, des fonctions de la libraire standard C par exemple. Pour gérer son état, le bloc de chiffrement prend en entrée un signal d'activation. C'est seulement au moment de la réinitialisation du chiffrement, après un branchement, que ce signal est pris en compte. Ce signal d'activation est accessible en lecture écriture par un registre de contrôle supplémentaire. Cette approche est bien plus flexible qu'une instruction qui désactiverait instantanément le chiffrement.

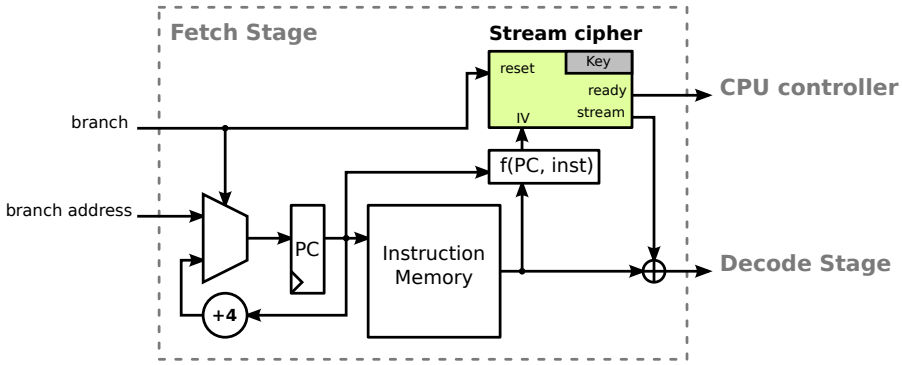


FIGURE 3.9. – Intégration du déchiffrement dans la lecture d'instruction du processeur

Lorsque le chiffrement est désactivé, le bloc émet un flot constant de zéros, l'opération « XOR» n'a donc aucun effet sur l'instruction lue depuis la mémoire.

3.4.2. Gestion des exceptions

Une exception correspond à un évènement non prévu dans l'exécution de code. Les exemples les plus courants d'exceptions sont les divisions par zéro, les accès mémoire invalides, les interruptions extérieures et bien d'autres. Lorsqu'un tel évènement se produit, une routine de traitement de l'exception est exécutée.

La façon de gérer les exceptions est propre à chaque processeur, nous supposons une gestion relativement standard. Concrètement, lorsqu'une exception se produit, le registre d'adresse PC est sauvegardé dans un registre spécial que nous notons EPC, puis un branchement est effectué vers un traitant d'exception et le processeur passe dans un mode spécial (mode exception). Une instruction spéciale du processeur permet de reprendre l'exécution en mode normal à l'adresse contenue dans le registre EPC.

Sortir d'une suite d'instructions chiffrées pour exécuter le traitant ne pose aucun problème par rapport à la définition 2 de bloc de base chiffrable. En revanche, le réel problème se produit au retour de l'interruption. Il y a alors un risque d'effectuer un branchement au milieu d'un bloc de base, ce qui entre en conflit avec la définition 2.

Encore une fois, le problème sous-jacent est la restauration de l'état du chiffrement par flots à l'instruction de retour. Sauvegarder et restaurer l'état complet du chiffrement par flots n'est pas une solution envisageable, d'une part à cause du stockage important nécessaire, mais aussi par souci de sécurité. Contrairement à un vecteur d'initialisation, l'état permet de générer toute la suite chiffrante à venir et donc déchiffrer tous les messages suivants.

Nous proposons un mécanisme permettant de restaurer l'état du chiffrement par flots au retour d'une exception. Pour ce faire, nous ajoutons deux registres supplémentaires :

— PC_{prev}, dans l'étage de lecture d'instructions (figure 3.9), qui sauvegarde l'adresse

de destination du dernier branchement effectué. Autrement dit, l'adresse de début du bloc de base courant.
— EPC_{prev}, dans les registres de contrôle et statut du processeur. Il sauvegarde la valeur du registre PC_{prev} lorsqu'une exception arrive.

La procédure de retour d'exception est formalisée dans l'algorithme 3.3. Au lieu de retourner directement à l'instruction de branchement, le retour s'effectue au début du dernier bloc de base (adresse PC_{prev}). Les instructions sont alors ignorées jusqu'à ce que $PC = EPC$, autrement dit que l'on soit revenu à l'instruction où est survenue l'exception. Ceci permet de restaurer l'état du chiffrement par flots en rejouant l'exécution depuis le début du bloc de base (les instructions étant ignorées pendant ce temps).

Algorithme 3.3 Procédure de retour d'exception

Entrée : EPC_{prev}, EPC
1: $PC \leftarrow EPC_{prev}$
2: $PC_{prev} \leftarrow EPC_{prev}$
3: Réinitialiser le chiffrement par flots
4: **while** $PC \neq EPC$ **do**
5: Ignorer l'instruction courante
6: $PC \leftarrow PC + 4$
7: **end while**
8: Continuer l'exécution normalement

Un des inconvénients de cette approche est le temps de latence variable au retour d'une exception, en particulier si l'exception se produit à la fin d'un bloc de base de taille importante. Ainsi, pour les interruptions extérieures, il peut être avantageux dans certains cas d'attendre un changement de bloc de base pour les traiter.

3.4.3. Changement de contexte

Dès lors que les exceptions sont supportées, il est aisé de supporter un changement de contexte. Ceci permet de faire cohabiter plusieurs tâches qui exécutent du code chiffré. Pour cela, il faut rendre accessible au programmeur le registre EPC_{prev} en lecture et écriture. Le contexte d'une tâche, qui contient tous les registres généraux et certains de contrôle et de statuts doit alors inclure le registre additionnel PC_{prev}. Il faut qu'il soit sauvegardé et restauré par les routines de changement de contexte. Dès lors, c'est la logique de gestion des exceptions décrite dans la section précédente qui se chargera de restaurer l'état du chiffrement.

Comme pour les interruptions, il est possible d'attendre la fin d'un bloc de base pour effectuer les changements de contexte. On peut alors se passer du mécanisme de sauvegarde / restauration décrit précédemment. Cependant, il faut impérativement garantir que toutes les sources d'exceptions qui peuvent mener à un changement de contexte ne se situent pas au milieu d'un bloc. Une telle contrainte peut être difficile à garantir dans un cas général d'utilisation.

3.5. Conclusion

Nous avons proposé une méthode de chiffrement d'un programme permettant de déchiffrer les instructions une par une et seulement avant leur décodage. Le mécanisme se base sur des chiffrements par flots. Pour permettre leur utilisation, nous avons identifié une structure adaptée au chiffrement (le bloc de base) et étudié les transformations nécessaires à effectuer sur un programme à chiffrer ainsi que quelques optimisations possibles. Nous avons intégré ces transformations dans une suite de compilation LLVM. Le chiffrement est ainsi complètement transparent au programmeur qui n'a qu'à ajouter une option additionnelle au compilateur pour chiffrer un programme.

L'intégration du mécanisme dans un processeur est plus complexe que les méthodes de chiffrement utilisées dans les processeurs sécurisés. Cependant, nous avons vu qu'elle était tout à fait possible et que l'on pouvait même ajouter le support d'exceptions et de changements de contexte sans trop de difficultés. Les résultats de l'évaluation présentés dans le chapitre 6 confirment qu'un tel chiffrement est beaucoup moins lourd qu'une approche basée sur un chiffrement par bloc, mais reste néanmoins efficace.

Chemin de données homomorphe

4

Nous avons vu jusqu'à présent qu'il était possible de chiffrer efficacement les instructions d'un programme, mais qu'en est-il des données ? Ce chapitre présente les travaux sur l'utilisation de chiffrement homomorphe pour protéger les manipulations de données. Bien loin des objets théoriques qu'ils étaient en 2009, aujourd'hui leur utilisation est envisageable dans des systèmes réels, même contraints en ressources.
Nous allons dans un premier temps donner une vue d'ensemble des algorithmes de chiffrement homomorphe. Puis, dans quelle mesure il est possible de les intégrer efficacement dans une architecture de processeur en vue de la protection de donnée.

Sommaire

4.1. Introduction

Avant de développer plus rigoureusement la théorie des algorithmes de chiffrement homomorphes, étudions d'abord les propriétés qu'ils peuvent nous apporter à l'aide d'une analogie simple. Un chiffrement homomorphe permet d'effectuer des calculs «à l'aveugle» sur des données chiffrées. On peut imaginer une boîte opaque, dans laquelle il serait possible d'ajouter des éléments. Un certain nombre sont déjà présents initialement. Une fois tous les éléments nécessaires placés dans la boîte, un résultat en sort. Cette analogie très simple permet déjà d'illustrer le type de garanties que peut fournir un chiffrement homomorphe. Par exemple, si la boîte calcule $z = x + c$, avec c une constante interne à la boîte, il est alors aisé, à partir d'un x fixé de déterminer la valeur de c, il suffit de calculer $z - x$. Dans ce cas, l'utilisation d'un calcul homomorphe n'améliore en rien la confidentialité de la donnée c.

En revanche, si la fonction calculée dans la boîte présente de bonnes propriétés, par exemple qu'il est difficile de retrouver des informations sensibles à partir des entrées-sorties, alors calculer cette fonction en boîte noire présente un intérêt : toutes les valeurs intermédiaires de l'algorithme sont protégées. C'est-à-dire que le calcul ne fuite aucune autre information que les entrées-sorties.

Dans le contexte d'un processeur, les données déjà présentes dans la boîte sont les données en mémoire dont on veut protéger la confidentialité, les données entrantes sont des données en clair, spécifiées par l'utilisateur. Enfin les résultats, sortant de la boîte sont ceux observables sur différentes sorties du système : écriture en mémoire, GPIO, carte réseau... Nous sommes malheureusement confrontés au même problème que ci-dessus : si le calcul est trivial, il n'y a aucun intérêt à effectuer le calcul dans une boîte.

Le chiffrement homomorphe nous permet, à quelques hypothèses près, de mettre en place cette enveloppe de calcul sécurisé. L'avantage par rapport à une méthode classique de chiffrement de la mémoire est que les données manipulées ne révèlent aucune information. Avec un chiffrement de la mémoire, il faut déchiffrer pour effectuer les calculs sur les données en clair, ce qui naturellement induit des fuites par des canaux auxiliaires (consommation, rayonnement EM).

Cependant, intégrer du chiffrement homomorphe dans un processeur pose un certain nombre de difficultés, auxquelles nous apporterons des solutions tout au long de cette partie. Il faut tout d'abord préciser un modèle d'exécution et comment traduire des programmes pour les évaluer de manière homomorphe. L'autre difficulté est les performances qui doivent rester acceptables, les propositions se déclinent sous deux angles :

1. Une méthode de rechiffrement légère (section 4.3), adaptée à une implémentation matérielle où un attaquant n'observe qu'une version bruitée des différents signaux du circuit (comme c'est le cas lors d'une attaque par canaux auxiliaires). Cette opération permet de réduire le bruit dans un message chiffré homomorphe et ainsi d'évaluer des opérations arbitraires sur des données chiffrées.
2. Une architecture matérielle efficace pour l'évaluation, qui sera présentée dans

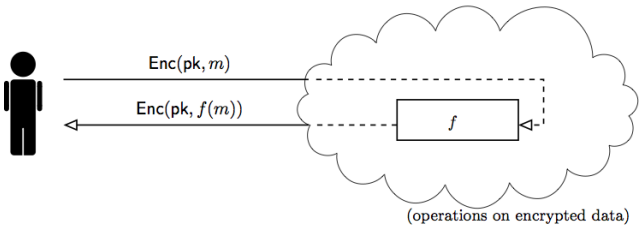


FIGURE 4.1. – Utilisation de chiffrement homomorphe pour le calcul sécurisé.

le chapitre suivant.

4.2. Chiffrement homomorphe

Un chiffrement homomorphe possède la propriété remarquable que les éléments chiffrés sont malléables. C'est-à-dire que l'on peut modifier un chiffré, de manière volontaire et contrôlée, pour obtenir le chiffré d'un autre élément (addition, multiplication, voir une fonction arbitraire). Des propriétés simples d'homomorphisme apparaissent déjà dans des schémas asymétriques couramment utilisés. Par exemple, un chiffré RSA [RSA78] est de la forme $m^e \mod N$, où e et N sont des éléments de la clé publique. On remarque que le produit de deux chiffrés modulo N, vaut ${m_0}^e {m_1}^e = (m_0 m_1)^e \mod N$. Cette valeur représente clairement le chiffré du produit des deux messages. À l'opposé, le schéma de Paillier [Pai99] est homomorphe pour l'addition.

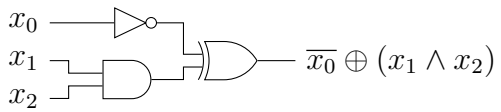
L'existence d'un schéma pouvant évaluer des fonctions arbitraires sur des données chiffrées fut conjecturée en 1978 par [RAD78]. Les applications impressionnantes présentées dans ce papier ont très vite placé le chiffrement homomorphe comme le Graal de la cryptographie. Une progression significative vers cet objectif fut proposée par Boneh et al. [BGN05] en 2005, avec un chiffrement à base de *pairings* capable d'évaluer un nombre arbitraire d'additions, mais une seule multiplication. C'est en 2009 que Graig Gentry [Gen09b, Gen09a] a complètement révolutionné le domaine en construisant un schéma complètement homomorphe.

Aujourd'hui, l'application phare du chiffrement homomorphe est pour sécuriser les calculs effectués sur le *cloud*. Les utilisateurs peuvent ainsi envoyer leurs données chiffrées et laisser le fournisseur *cloud* effectuer les calculs sans qu'il puisse connaître les données manipulées (voir figure 4.1).

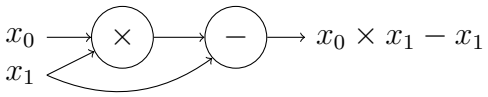
4.2.1. Définitions et terminologie

Nous considérons dans ces travaux principalement les schémas homomorphes asymétriques (bien que des symétriques existent). Un tel schéma est défini par les fonctions suivantes :

— `Init`(λ), qui en fonction d'un paramètre de sécurité λ, génère une paire clé publique, clé privée (pk, sk).



(a) Un circuit logique



(b) Un circuit arithmétique

FIGURE 4.2. – Exemples de circuits

— $\texttt{Encrypt}_{pk}(m)$, la fonction de chiffrement, une procédure non déterministe qui à partir de la clé publique pk produit un chiffré d'un message m donné en entrée. Lorsqu'il n'y a pas d'ambiguïté, nous noterons $\overline{m}$ le chiffré d'un message m. C'est-à-dire, pour une certaine clé pk, $\overline{m} = \texttt{Encrypt}_{pk}(m)$.
— $\texttt{Decrypt}_{sk}(c)$, la fonction de déchiffrement, qui prend en paramètre la clé secrète sk. Cette fonction garantit, avec forte probabilité, que si $c = \texttt{Encrypt}_{pk}(m)$, alors $\texttt{Decrypt}_{sk}(c) = m$.
— $\texttt{Eval}_{pk}(f, c_0, \ldots c_n)$, qui évalue une fonction f sur un ensemble de chiffrés. Cette fonction a seulement besoin de la clé publique, en général une sous-partie de clé publique appelée clé d'évaluation. C'est cette primitive qui confère la propriété d'homomorphisme au schéma.

Une *fonction* est représentée par un circuit, c'est-à-dire un graphe sans cycles (voir figure 4.2), dont les nœuds sont étiquetés avec les opérations à effectuer. Contrairement à un programme au sens informatique, un circuit représente une expression mathématique, un graphe de réductions à effectuer. En conséquence, il n'est pas possible de représenter des branchements, ou toute forme d'exécution conditionnelle. Nous verrons section 4.5 que l'on peut traduire en circuit des structures conditionnelles simples à base de *if-else*, ainsi que des boucles dont le nombre d'itérations est statiquement borné.

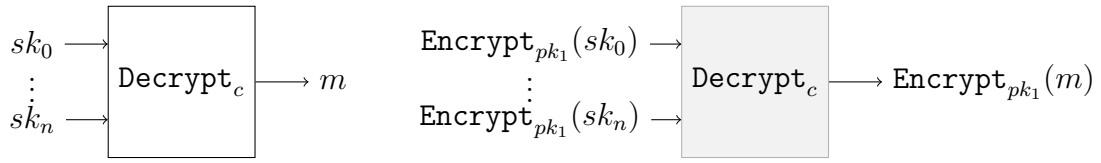
La propriété fondamentale fournie par ces schémas est que l'évaluation sur les données chiffrées préserve la structure des opérations sur les données en clair. Formellement, pour n'importe qu'elle fonction f supportée par le schéma, si

$$c = \texttt{Eval}_{pk}(f, \texttt{Encrypt}_{pk}(m_0), \ldots, \texttt{Encrypt}_{pk}(m_n))$$

alors,

$$\texttt{Decrypt}_{sk}(c) = f(m_0, \ldots, m_n)\,.$$

Lorsqu'un schéma ne peut évaluer qu'une classe restreinte de fonctions, on parle de schéma «Somewhat Homomorphic» (SHE). Nous avons déjà évoqué le schéma RSA, homomorphe pour la multiplication ou le schéma de Paillier, homomorphe pour l'addition. Lorsqu'un schéma peut évaluer des fonctions arbitraires sur des données chiffrées, il est dit complètement homomorphe ou *Fully Homomorphic* en anglais.



(a) Déchiffrement classique, opérant sur les bits de la clé secrète

(b) Déchiffrement effectué de manière homomorphe.

FIGURE 4.3. – Représentation de la fonction de déchiffrement en tant que circuit

4.2.2. Construction d'un schéma complètement homomorphe

Si un certain nombre de schémas *somewhat homomorphic* sont connus, il n'existe pas à ce jour de chiffrements qui soient directement complètement homomorphe. Au cours de sa thèse [Gen09a], Craig Gentry a proposé une méthode pour construire un schéma complètement homomorphe à partir d'un schéma *somewhat homomorphic*. Aujourd'hui encore, il s'agit de la seule manière connue de construire un tel schéma. Le mécanisme central de cette construction est connu sous le nom de *bootstrapping*.

Pour un chiffré donné c, la fonction déchiffrement spécialisée pour ce chiffré $\texttt{Decrypt}_c$ est représentable comme un circuit prenant en entrée la clé secrète sk et qui produit en sortie le message m (voir figure 4.3a). Si le schéma est en capacité d'évaluer de manière homomorphe ce circuit pour n'importe quel chiffré c, suivi d'une ou plusieurs opérations, on dit qu'il est *bootstrappable*.

Lorsque cette condition est vérifiée, le circuit $\texttt{Decrypt}_c$ peut être évalué de manière homomorphe, comme montré sur la figure 4.3b. Il n'est plus nécessaire d'avoir la clé secrète « en clair», mais un chiffré de la clé secrète par une autre clé publique pk_1. On obtient après évaluation, un chiffré selon la clé pk_1 du message m. Tout l'intérêt est que pour un chiffré c, nous sommes revenus après évaluation de $\texttt{Decrypt}_c$ à un chiffré du même message original, mais dont le nombre d'évaluations est connu statiquement (le circuit $\texttt{Decrypt}_c$). En particulier, si la profondeur du circuit évalué sur le chiffré en entrée c est supérieure à celle du circuit $\texttt{Decrypt}_c$, alors le *bootstrapping* a pour effet de réduire le bruit et permettre plus d'opérations homomorphes. En général, une seule et même clé publique est utilisée, $pk_1 = pk$. Il faut alors faire l'hypothèse que connaître $\texttt{Encrypt}_{pk}(sk)$ ne compromet pas la sécurité du schéma (hypothèse de sécurité circulaire).

Clairement, cette opération de *bootstrapping* est extrêmement coûteuse. En effet, la clé secrète et le circuit de déchiffrement sont en pratique assez conséquents. De plus, pour offrir une construction la plus générique possible, nous avons supposé une représentation booléenne des données et circuits , ce qui rend la construction encore moins efficace. Par ailleurs, trouver un schéma *bootstrappable* est loin d'être une tâche aisée. En 2009, une autre contribution majeure de Gentry fut de proposer un schéma *bootstrappable*. Cette construction était néanmoins assez complexe et utilisait des techniques astucieuses pour réduire la complexité du déchiffrement.

Une fois en possession d'un schéma *boostrappable*, il est aisé de construire un schéma

pouvant évaluer des fonctions arbitraires. Il suffit de «rafraîchir» périodiquement un chiffré après une ou plusieurs opérations de base en effectuant un *bootstrapping*. C'est en chaînant ces opérations dîtes augmentées (*bootstrapping*, puis évaluation de circuits réduits), qu'il est possible d'évaluer des fonctions arbitraires de manière homomorphe.

4.2.3. Évolution des schémas homomorphes

On compte aujourd'hui trois évolutions majeures dans les chiffrements homomorphes. La première génération de schémas complètement homomorphes construits par Gentry en 2009, construits sur des problèmes peu communs sur des idéaux de réseaux euclidiens. Un résultat important du domaine fut la première évaluation réelle d'un *boostrapping* en 2011 [GH11], qui nécessitait une clé publique de 2.3 GB et environ 30 minutes sur un serveur de calcul. Dans la même période, des chiffrements homomorphes basés sur les entiers se sont développés, mais présentaient des performances similaires.

La seconde évolution majeure fut l'apparition de schémas basés sur le problème *Learning With Error* (LWE) ou Ring-LWE [BV11a, BV11b]. Leur sécurité est beaucoup plus facile à analyser, grâce à des réductions à des problèmes relativement standards sur des réseaux euclidiens. Ces schémas sont également beaucoup plus simples conceptuellement et plus efficaces. Cette génération compte de nombreux schémas qui restent aujourd'hui très compétitifs pour l'évaluation de type *somewhat homomorphic* quelques exemples sont : LTV [LTV12], YASHE [BLLN13], FV [FV12], BGV [BGV12].

Enfin une troisième génération de schémas a vu le jour, construits spécialement pour avoir un *bootstrapping* très efficace [GSW13]. Ces schémas permettent d'atteindre aujourd'hui l'état de l'art en termes de vitesse de *boostrapping* [CGGI16].

4.3. Méthode de *bootstrapping* matérielle

Le *bootstrapping* est obligatoire pour évaluer des calculs arbitraires. Cependant, cette procédure reste un des points noirs du chiffrement en termes de performances. Nous proposons dans cette partie une procédure fonctionnellement équivalente, mais beaucoup plus efficace.

La procédure proposée est représentée figure 4.4. Pour obtenir la fonctionnalité d'un *bootstrapping*, un déchiffrement est effectué suivi d'un nouveau chiffrement. En déchiffrant, le bruit est complètement retiré du chiffré puis, en le chiffrant à nouveau, un chiffré avec un bruit fortement borné est produit. Pour protéger le flot, un masque aléatoire r est appliqué de manière homomorphe avant le déchiffrement et retiré une fois le message à nouveau chiffré. Parallèlement à ces travaux, une approche très similaire a été proposée dans [RCR+16] pour introduire de l'aléa dans le déchiffrement contre les attaques par canaux auxiliaires. La protection a un très faible surcoût et semble efficace expérimentalement. Cette approche améliore donc clairement la sécurité dans un modèle d'observation globale du circuit. Néanmoins, prouver cette

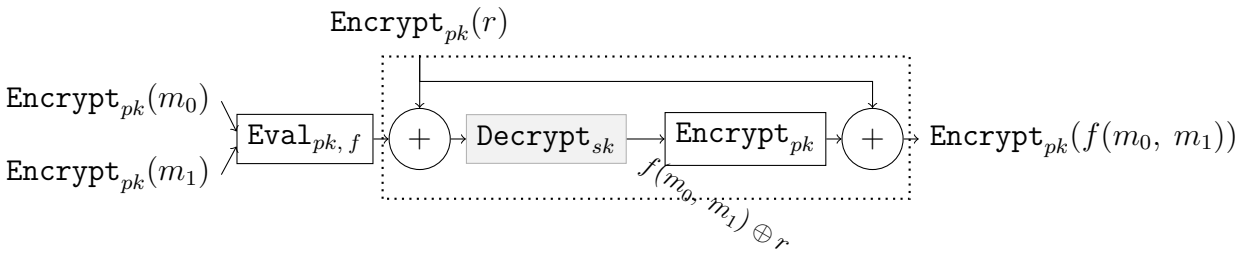


FIGURE 4.4. – Circuit de rechiffrement masqué après une opération

sécurité dans un modèle d'observation plus formel, comme de *d-probing* de Ishai et al. [ISW03] reste un problème ouvert.

Une alternative aurait été de protéger le bloc entier composé de $\texttt{Decrypt}_{sk}$ et $\texttt{Encrypt}_{pk}$. En revanche, l'utilisation du chiffré d'un masque offre plusieurs avantages. Tout d'abord, cela introduit de l'aléa dans le déchiffrement et dans une certaine mesure, une protection [RCR+16]. Enfin, plusieurs options intéressantes sont envisageables pour générer ce masque :

— Chiffrer un aléatoire provenant d'un RNG. Cette opération doit en revanche être effectuée dans une zone protégée.
— Pré-calculer et stocker un certain nombre de masques et ne les utiliser qu'une seule fois.
— Une autre alternative serait de concevoir un générateur homomorphe de ces valeurs.

Analyse de sécurité. Pour prouver la sécurité de ce rechiffrement, nous supposons que les blocs suivants sont résistants aux fuites par canaux auxiliaires :

— la génération du masque r, un attaquant ne doit accéder qu'à $\texttt{Encrypt}_{pk}(r)$.
— La fonction de déchiffrement, qui naturellement ne doit pas révéler d'information sur la clé secrète utilisée.

Sous ces hypothèses, la fonction $\texttt{Encrypt}_{pk}$ se comporte comme une boîte noire. Aussi, un attaquant ne peut dans le meilleur des cas obtenir que des triplés $\texttt{Encrypt}_{pk}(m)$, $\texttt{Encrypt}_{pk}(r)$ et $m \oplus r$ qui correspondent aux valeurs intermédiaires de la figure 4.4. L'obtention de ces trois valeurs ne constitue en rien une menace. Il s'agit d'une forme d'encapsulation de clé (KEM), un protocole couramment utilisé pour établir un tunnel de communication symétrique, et dont la sécurité est prouvable (voir [KL14], théorème 11.12). Ici le chiffrement symétrique utilisé serait le *one-time pad* et la clé secrète associée r.

Protection des blocs critiques. Le fait d'implémenter les blocs critiques de manière matérielle offre une protection naturelle. De plus, introduire de l'aléa dans le déchiffrement permet de contrer des attaques par canaux auxiliaires basiques [RCR+16] expérimentalement. Il serait extrêmement intéressant d'obtenir une protection prou-

vable de ces blocs dans un modèle de fuite classique. Cela constitue indéniablement un axe de recherche en soi. Cependant une telle étude n'était pas un des objectifs premiers de cette thèse. Mentionnons néanmoins que dans [RRVV15], les auteurs proposent une protection à base de masquage pour un chiffrement de type RLWE (donc, applicable également pour un schéma BGV). Ces résultats sont très prometteurs et montrent que le déchiffrement de ces schémas est très adapté à un masquage (seulement 17% de logique supplémentaire dans [RRVV15]).

Une autre approche possible pour la protection des blocs critiques est de recourir à des contre-mesures basées sur la technologie du circuit, visant à réduire et/ou brouiller les diverses émanations du circuit. L'avantage offert par l'approche proposée est que les zones à protéger sont réduites et clairement identifiées.

Avantages. La conséquence immédiate d'un *bootstrapping* léger est de pouvoir réduire considérablement les paramètres du schéma. En effet, le schéma a seulement besoin d'être paramétré pour évaluer une opération simple, chaînée avec le circuit de la figure 4.4, qui lui aussi a une très faible profondeur arithmétique. Les paramètres influencent la taille des données manipulées et affectent donc directement les performances.

4.4. Le schéma Brakerski, Gentry, Vaikutanathan (BGV)

Le schéma de Brakerski, Gentry, Vaikutanathan est actuellement l'un des schémas homomorphes les plus performants. La bibliothèque C++ en libre accès HELib [HS, HS14, HS15], qui en offre une implémentation, témoigne de la maturité de ce schéma. Il a été utilisé avec succès pour l'évaluation homomorphe d'un algorithme AES [GHS12] et reste aujourd'hui un point de comparaison dans la recherche [KL15].

Initialement le schéma YASHE [BLLN13] avait été retenu comme cadre d'expérimentation. Malheureusement, une attaque théorique [ABD16] fut publiée en 2016 et compromet réellement la sécurité de ce schéma, en particulier dans sa version *bootstrappable*. Heureusement les espaces mathématiques (messages, chiffrés) utilisés dans le YASHE sont relativement proches de ceux utilisés dans les chiffrements basés sur RLWE. C'est pourquoi le schéma BGV dans sa version polynomiale offre une alternative très satisfaisante au schéma YASHE. Ajoutons pour terminer qu'une attaque a récemment été découverte sur le BGV [Alb17], et a nécessité de réévaluer les paramètres en conséquence. Ce point sera discuté plus en détail dans la section 4.4.4.

4.4.1. Préliminaires, notations

Nous notons $\mathbb{Z}_q$ l'anneau des entiers modulo q. Pour un élément $x \in \mathbb{Z}$, $[x]_q$ désigne l'unique entier dans l'intervalle $[-q/2, q/2]$ vérifiant $[x]_q = x \mod q$. Les vecteurs sont notés en gras. La norme infinie d'un vecteur $\mathbf{a}$ est la valeur absolue du plus

grand coefficient $\|\mathbf{a}\|_\infty = \max_i |a_i|$. Dans le cas d'un polynôme, la norme est calculée sur le vecteur constitué de ses coefficients.

Une racine m-ième primitive de l'unité, notée ζ_m est un élément vérifiant $\zeta_m^m = 1$ et $\zeta_m^i \neq 1$ pour $i < m$. Dans le corps des complexes $\zeta_m = e^{2i\pi/m}$. Mais elles peuvent également être définies dans un groupe multiplicatif $\mathbb{Z}_q^*$.

Définition 3. Le m-ième polynôme cyclotomique, noté Φ_m est le polynôme ayant pour racines l'ensemble des racines m-ième primitives de l'unité complexes.

$$\Phi_m(X) = \prod_{i \in \mathbb{Z}_m^*} (X - \zeta_m^i)$$

Ces polynômes très particuliers ont un certain nombre de propriétés remarquables :
- Le degré de Φ_m est donné par $\phi(m)$, l'indicatrice d'Euler, définie comme le cardinal de $\mathbb{Z}_m^*$, ou de manière équivalente le nombre d'entiers dans l'intervalle $[0, m-1]$ premiers avec m.
- Les coefficients de Φ_m sont tous entiers et Φ_m est irréductible sur $\mathbb{Z}$.
- Lorsque m est une puissance de 2, $\Phi_m(X) = X^{m/2} + 1$

Définition 4. Nous introduisons les anneaux polynomiaux suivant :

- L'anneau R, des polynômes à coefficients entiers modulo Φ_m, $R = \mathbb{Z}[X]/\Phi_m(X)$.
- Pour un entier q, R_q est défini par $R_q = R/q\mathbb{Z}$.

Le facteur d'expansion de l'anneau R, noté δ_∞ est défini comme :

$$\delta_\infty = \sup\{\frac{\|a(X)b(X)\|_\infty}{\|a(X)\|_\infty \|b(X)\|_\infty} \ : \ a, b \in R\}$$

Ce paramètre caractérise l'expansion maximale en norme infinie après une multiplication. Lorsque m est une puissance de deux et donc que $\Phi_m(X) = X^{m/2} + 1$, on vérifie aisément que $\delta_\infty = \phi(m)$. On a ainsi dans ce cas particulier

$$\|a \cdot b\|_\infty \leq \phi(m) \|a\|_\infty \|b\|_\infty \tag{4.1}$$

Représentations des éléments de R, R_q

La définition 4 suggère naturellement de représenter un élément de R comme un polynôme modulo Φ_m. Cependant, l'anneau R a une structure algébrique très riche venant de la théorie algébrique des nombres, il s'agit de l'anneau des entiers du m-ième corps cyclotomique $\mathbb{Q}(\zeta_m)$. Une conséquence est qu'il existe un plongement, $\sigma^{\text{can}}(\cdot) \ : \ R \to \mathbb{C}^{\phi(m)}$, défini par

$$\sigma^{\text{can}}(a) = (a(\zeta_m^i))_{i \in \mathbb{Z}_m^*} .$$

Un élément de R est ainsi représenté de manière équivalente par son évaluation aux différentes racines primitives de l'unité. Dans cette nouvelle représentation, les opérations sont effectuées point à point, par conséquent, la borne obtenue sur la norme infinie d'un produit de polynômes est bien meilleure que celle de l'équation (4.1) :

$$\|a \cdot b\|_\infty^{\text{can}} \leq \|a\|_\infty^{\text{can}} \|b\|_\infty^{\text{can}}$$

De plus, il existe une constante c_m dépendante de l'anneau, telle que $\|a\|_\infty \leq c_m \|a\|_\infty^{\text{can}}$.

4.4.2. Chiffrement et déchiffrement

Le schéma que nous présentons ici, est décrit dans [BGV12]. Il s'agit d'une version améliorée du schéma initialement proposé par Brakerski [BV11c]. Il existe une version basée sur le problème LWE [Reg09] et une sur Ring-LWE [LPR10]. Nous présentons ici seulement cette dernière, qui est la déclinaison la plus performante.

Pour définir le schéma BGV, nous introduisons les distributions aléatoires suivantes :

— $\mathcal{U}(R_q)$, la distribution uniforme sur R_q.
— $\mathcal{DG}(\sigma^2)$, un polynôme dont les coefficients sont échantillonnés selon une gaussienne discrète centrée en 0 et de variance σ^2. Une définition plus précise de cette distribution sera donnée dans le chapitre suivant 5.4.
— $\mathcal{HW}(h)$, avec $h \leq \phi(m)$ est un vecteur pris uniformément dans $\{-1, 0, 1\}^{\phi(m)}$ ayant exactement h coefficients non-nuls.
— $\mathcal{ZO}(\rho)$, un polynôme dont chacun des coefficients est pris dans $\{-1, 0, 1\}$, avec comme probabilité respectivement $\{\rho/2, 1-\rho, \rho/2\}$.

Une instance particulière du schéma BGV est spécifiée par le 6-uplet de paramètres $(m, q, p, \sigma, h, \rho)$, où :

— L'entier m, spécifie l'index du polynôme cyclotomique utilisé. Concrètement, les polynômes manipulés dans le schéma seront de degrés $\phi(m)$.
— Les entiers q et p sont respectivement le module de l'espace des chiffrés et l'espace des messages.
— Enfin, les paramètres σ, h, ρ servent à caractériser les distributions aléatoires utilisées.

4.4.2.1. Génération de clé (Keygen)

Nous supposons qu'un jeu de paramètres valides a été généré $(m, q, p, \sigma, h, \rho)$. La génération de clé procède de la manière suivante :

1. Échantillonner les polynômes suivants :
   a) $a \hookleftarrow U(R_q)$, un élément uniforme de R_q
   b) $s \hookleftarrow \mathcal{HW}(h)$, qui sera la clé secrète du schéma
   c) $e \hookleftarrow \mathcal{DG}(\sigma^2)$, un polynôme d'erreur
2. Calculer $b = -a \cdot s + pe$

La clé publique est définie comme le couple $(a, b) \in R_q^2$ et la clé secrète est le polynôme s. On voit bien que la clé publique est quasiment un échantillon d'une distribution RLWE de la forme $(a, a \cdot s + e)$.

4.4.2.2. Chiffrement (Encrypt)

1. Échantillonner
   a) $e_0, e_1 \hookleftarrow \mathcal{DG}(\sigma^2)$, deux polynômes d'erreur
   b) $r \hookleftarrow U(R_q)$, un polynôme uniforme

2. Calculer ensuite $c_0 = b \cdot r + pe_0 + m$ et $c_1 = a \cdot r + pe_1$.
3. Retourner $\mathbf{c} = (c_0, c_1)$.

4.4.2.3. Déchiffrement (Decrypt)

Étant donné un chiffré $\mathbf{c} = (c_0, c_1) \in R_q \times R_q$, le déchiffrement consiste à calculer dans un premier temps $z = \langle \mathbf{c}, \mathbf{s} \rangle = c_0 + c_1 \cdot s \mod q$, où $\mathbf{s}$ est le vecteur $(1, s)$. On retourne alors $m = [[z]_q]_p$.

On constate que le déchiffrement est extrêmement simple en termes d'opérations, en particulier la profondeur multiplicative est très faible. Il ne s'agit pas d'un hasard, pour que le *bootstrapping* soit réalisable, il faut que la profondeur multiplicative du circuit de déchiffrement spécialisé pour un chiffré c $\texttt{Decrypt}_c$ soit la plus faible possible (voir section 4.2.2).

Proposition 1. *Étant donné un chiffré* $\mathbf{c} = \texttt{Encrypt}_{pk}(m)$*, alors* $\texttt{Decrypt}_{sk}(\mathbf{c}) = m$.

Démonstration. Cette preuve est l'occasion d'établir une condition nécessaire pour que le déchiffrement réussisse. Calculons d'abord $c_0 + c_1 \cdot s$ dans $\mathbb{Z}_q$:

$$\begin{aligned} c_0 + c_1 s &= (b \cdot r + pe_0) + m + (a \cdot r + pe_1) \cdot s \\ &= ((-a \cdot s + pe) \cdot r + pe_0) + m + (a \cdot r + pe_1) \cdot s \\ &= -a \cdot s \cdot r + pe \cdot r + pe_0 + m + a \cdot r \cdot s + e_1 \cdot s \\ &= p(\underbrace{e \cdot r + e_0 + e_1 \cdot s}_{\tilde{e}}) + m \end{aligned}$$

Le résultat a une forme très particulière $p\tilde{e} + m$. Si la valeur du terme d'erreur $\tilde{e}$ est assez faible, plus précisément quand $\|p\tilde{e} + m\|_\infty < q/2$, alors

$$[p\tilde{e} + m]_q = p\tilde{e} + m$$

Autrement dit, $p\tilde{e}+m$ est son propre représentant modulo q dans l'intervalle $[-q/2, q/2]$. En réduisant modulo p on obtient alors le message :

$$[p\tilde{e} + m]_p = m\,.$$

Si cette contrainte n'est pas respectée, alors le déchiffrement n'est plus garanti. □

Au cours de cette preuve, nous avons pu établir ce que signifie un chiffré valide, nous formalisons cette notion dans la définition suivante :

Définition 5. Un chiffré valide c d'un message $m \in R_p$ est solution de l'équation suivante

$$\langle \mathbf{c}, \mathbf{s} \rangle = pe + m\,, \tag{4.2}$$

avec $\|pe + m\|_\infty < q/2$.

Nous établissons maintenant une borne probabiliste sur l'amplitude du bruit initial dans un chiffré, qui permet de garantir le déchiffrement correct.

TABLE 4.1. – Tableau de valeur de la fonction erfc(·)

n	lg(erfc($n/\sqrt{2}$))	n	lg(erfc($n/\sqrt{2}$))
1	-1.656	7	-38.507
2	-4.458	8	-49.514
3	-8.533	9	-61.942
4	-13.946	10	-75.797
5	-20.734	11	-91.080
6	-28.917	12	-107.795

Proposition 2. *En notant B_{clean} l'amplitude du bruit dans un chiffré initial, on a avec forte probabilité*

$$B_{clean} \leq c_m p \left(\phi(m) \left(\frac{1}{2} + 36\sigma\sqrt{\rho} \right) + \sqrt{\phi(m)} \left(8\sigma + 36\sigma\sqrt{h} \right) \right)$$

Démonstration. Pour établir cette borne, nous reprenons l'approche heuristique utilisée dans [GHS12]. Soit $a \in R$, échantillonné coefficient par coefficient selon une distribution aléatoire de variance σ^2 et centrée en 0. Alors la i-ème composante en représentation canonique, vaut $a(\zeta_m^i) = \sum_k a_k \zeta_m^{ik}$. Il s'agit d'une somme de variables aléatoires indépendantes, leur somme a donc pour variance $\sum \sigma^2 |\zeta_m^{ik}|^2 = \phi(m)\sigma^2$. De plus, comme un grand nombre de variables aléatoires sont sommées ($\phi(m)$ au total), le théorème central limite nous dit que la distribution converge vers une distribution normale. Il est ainsi possible de la borner très fortement la distribution résultante, en particulier

$$Pr(|X| > n\sigma) \leq \text{erfc}(\frac{n}{\sqrt{2}}),$$

où erfc(·) est la fonction d'erreur complémentaire. On observe dans le tableau 4.1 que la probabilité devient très vite négligeable. En appliquant cette méthodologie, les polynômes e et s admettent les bornes suivantes (pour n assez grand) :

$$\begin{cases} \|e\|_\infty^{\text{can}} \leq n\sqrt{\phi(m)}\sigma & \text{pour } e \hookleftarrow \mathcal{DG}(\sigma^2) \\ \|s\|_\infty^{\text{can}} \leq nh & \text{pour } s \hookleftarrow \mathcal{HW}(h) \end{cases}$$

Le produit de deux telles variables de variances σ_1, σ_2 est borné par $n'\sigma_1\sigma_2$ avec une probabilité $1 - 2\,\text{erfc}(\sqrt{n'}/\sqrt{2})$.

Nous pouvons maintenant borner la norme infinie d'un chiffré, c'est-à-dire de l'expression $\|p(e \cdot r + e_0 + e_1 \cdot s) + m\|_\infty^{\text{can}}$. Les différents termes peuvent être bornés par :

$$\begin{cases} \|e \cdot r\|_\infty^{\text{can}} & \leq n'\sigma\phi(m)\sqrt{\rho} \\ \|e_0\|_\infty^{\text{can}} & \leq n\sigma\sqrt{\phi(m)} \\ \|e_1 \cdot s\|_\infty^{\text{can}} & \leq n'\sigma\sqrt{\phi(m)h} \\ \|m\|_\infty^{\text{can}} & \leq \frac{p}{2}\phi(m) \end{cases}$$

En sommant toutes ces bornes et simplifiant on obtient :

$$B_{clean} \leq c_m p \left(\phi(m) \left(\frac{1}{2} + n'\sigma\sqrt{\rho} \right) + \sqrt{\phi(m)} \left(n\sigma + n'\sigma\sqrt{h} \right) \right)$$

En fixant $n = 9$ et $n' = 36$ pour borner avec une probabilité supérieure à $1 - 2^{-55}$, on obtient la formule de la proposition. □

4.4.3. Opérations homomorphes

Nous allons maintenant montrer comment effectuer des opérations homomorphes, c'est-à-dire produire des nouveaux chiffrés valides à partir d'anciens. Pour le reste de la section, nous considérons deux chiffrés $\mathbf{c}_x = \texttt{Encrypt}_{pk}(m_x)$ et $\mathbf{c}_y = \texttt{Encrypt}_{pk}(m_y)$. D'après la proposition 1 ces deux chiffrés vérifient l'équation 4.2. On notera B_x et B_y l'amplitude du bruit dans les chiffré $\mathbf{c}_x$ et $\mathbf{c}_y$.

4.4.3.1. Addition Homomorphe

L'addition homomorphe s'obtient en ajoutant terme à terme les deux chiffrés. C'est-à-dire

$$\mathbf{c}_{add} = (c_{x,0} + c_{y,0},\ c_{x,1} + c_{y,1})\,.$$

En effet, par linéarité du produit scalaire, on a bien

$$\begin{aligned} \langle \mathbf{c}_x + \mathbf{c}_y,\ s \rangle &= pe_0 + m_0 + pe_1 + m_1 \\ &= p(e_0 + e_1) + m_0 + m_1 \end{aligned}$$

L'amplitude du bruit dans ce nouveau chiffré est bornée par $B_{add} = B_x + B_y$. Sous réserve que $B_{add} < q/2$, il s'agit bien d'un chiffré valide de l'addition des deux messages.

4.4.3.2. Multiplication Homomorphe

La multiplication n'apparaît pas de façon aussi spontanée que l'addition. Pourtant, en faisant le produit de deux équations (4.2), on obtient

$$\begin{aligned} \langle \mathbf{c}_x,\ s \rangle \langle \mathbf{c}_y,\ s \rangle &= (pe_0 + m_0)(pe_1 + m_1) \\ &= p^2 e_0 e_1 + pe_0 m_1 + m_0 pe_1 + m_0 m_1 \\ &= p(\underbrace{pe_0 e_1 + e_0 m_1 + m_0 e_1}_{e''}) + m_0 m_1 \\ &= pe'' + m_0 m_1 \end{aligned}$$

Autrement dit, il s'agit d'un chiffré valide d'une multiplication. En développant le membre de gauche de l'équation précédente,

$$\begin{aligned}\langle \mathbf{c}_x,\ s\rangle\langle \mathbf{c}_y,\ s\rangle &= (c_{x,0} + c_{x,1}s)(c_{y,0} + c_{y,1}s) \\ &= c_{x,0}c_{y,0} + (c_{x,0}c_{y,1} + c_{x,1}c_{y,0})s + c_{x,1}c_{y,1}s^2 \qquad (4.3) \\ &= c_{mul,0} + c_{mul,1}s + c_{mul}s^2 \\ &= \langle \mathbf{c}_{mul},\ (1,\ s,\ s^2)\rangle \qquad (4.4)\end{aligned}$$

Ainsi, $\mathbf{c}_{mul} \in R_q^3$ est un chiffré étendu, déchiffrable par une clé étendue $\mathbf{s} = (1,\ s,\ s^2)$. Une borne large sur l'amplitude du bruit dans le chiffré résultant est $\gamma_\infty B_x B_y$.

Il est donc possible de généraliser la définition 5 d'un chiffré valide à un chiffré de taille n. Un chiffré de taille n est valide s'il vérifie $\langle \mathbf{c},\ (1,\ s,\ \ldots,\ s^{n-1})\rangle = pe + m$. La linéarité du produit scalaire étant toujours valide, l'addition homomorphe est toujours possible sur des chiffrés étendus.

Malheureusement, la taille n augmente bien trop rapidement au fil des multiplications. Pour un circuit ayant n_{mul}, multiplications, la taille du chiffré final peut atteindre $2^{n_{mul}} + 1$. Pour éviter cette explosion en taille, le BGV utilise une procédure nommée le *key switching*, qui permet de transformer un chiffré de taille 3 à nouveau vers un chiffré de taille 1. L'idée est de publier des chiffrements de s^2 pour pouvoir retirer de manière homomorphe le terme quadratique de l'équation 4.3.

Un autre problème lié à cette multiplication est l'augmentation du bruit, qui est elle aussi exponentielle. La solution adoptée par le BGV est une technique appelée le *modulus switching*. Elle permet de passer le chiffré d'un module q_i vers un module plus petit q_{i-1}. Une bonne propriété est que le bruit est réduit d'un facteur q_i/q_{i-1}. Donc en prenant $q_i/q_{i-1} > B$, le *modulus switching* permet de revenir à une croissance linéaire du bruit. Le schéma n'a alors plus un seul module q, mais une chaîne décroissante de L modules $(q_{L-1},\ \ldots,\ q_0)$, avec un module associé à chaque niveau de multiplication.

4.4.4. Détermination des paramètres

Déterminer les paramètres d'un schéma homomorphe comme le BGV n'est absolument pas trivial. Il faut satisfaire parallèlement deux contraintes :

— la sécurité, il doit être impossible à un adversaire de distinguer des messages chiffrés ou de retrouver la clé privée à partir de la clé publique.
— La validité, c'est-à-dire qu'un chiffré valide, qui respecte l'ensemble des circuits supportés doit être correctement déchiffrable avec très forte probabilité.

Recherche d'une contrainte de sécurité concrète

Pour évaluer la sécurité, l'approche adoptée dans la littérature est de considérer les meilleures attaques connues et de fixer les paramètres de telle sorte que ces attaques ne soient pas réalisables. Le problème est que ces attaques sont très complexes et se basent sur des algorithmes de réduction de réseaux euclidiens, dont il est difficile de prédire le temps d'exécution par rapport à la qualité des résultats obtenus.

Une heuristique souvent utilisée est celle de Lindner et Peikert [LP11]. Appliquée au BGV dans [GHS12], cette heuristique permet d'établir la contrainte suivante pour obtenir une sécurité de λ-bits :

$$\phi(m) \geq \frac{\lg(q/\sigma)(\lambda + 110)}{7.2} \tag{4.5}$$

Récemment, Albrecht [Alb17] a découvert une attaque sur le schéma BGV, qui exploite le fait que la clé secrète a un très faible nombre de coefficients non nuls. En conséquence, l'estimation de l'équation (4.5) n'est plus totalement valide.

Dans l'optique d'estimer la sécurité plus précisément, nous avons utilisé le module Sage *LWE Estimator* [APS15] développé par Albrecht et qui centralise un nombre important d'attaques sur les problèmes LWE ou RLWE. Ce module inclut en particulier la récente attaque [Alb17] qui concerne le BGV. Notre but est de fournir une nouvelle estimation plus précise que l'équation (4.5). Pour cela, nous avons appliqué l'algorithme 4.1. Pour chaque valeur de q dans un intervalle donné, l'espace des valeurs possibles de n est exploré, est pour chaque valeur le coût des attaques est estimé. Le parcours des valeurs est guidé à l'aide d'un algorithme d'optimisation existant [1].

Algorithme 4.1 Recherche d'estimation de la sécurité

```
for j ← lg(q_min) to lg(q_max) do
    Générer q valide de j bits
    for n ← explore do
        costs ← simulateAttacks(n, q)
        Rejeter si ∃c ∈ costs, tel que c < 2^λ
    end for
    n_min ← valeur minimale de n valide
    return n_min
end for
```

Cet algorithme a nécessité environ 12 heures pour générer une nouvelle courbe d'estimation, représentée figure 4.5. En effet, pour chaque q et n fixé, plusieurs attaques sont simulées ce qui prend un temps non négligeable, de l'ordre de plusieurs secondes. Nous avons observé que la plus récente attaque [Alb17] est toujours celle qui offre le meilleur taux de succès. Il est assez intéressant d'observer que la courbe obtenue figure 4.5 semble linéaire en fonction de $\lg q$. En appliquant une régression linéaire, nous obtenons une nouvelle borne inférieure, plus stricte :

$$\phi(m) \geq 49.348 \lg(q) - 82.645 \tag{4.6}$$

À titre de comparaison, dans les mêmes conditions, l'heuristique de Lindner-Peikert impose :

$$\phi(m) \geq 33.056 \lg(q) - 37.399$$

1. Disponible dans la librairie Python *scipy*, scipy.optimize.

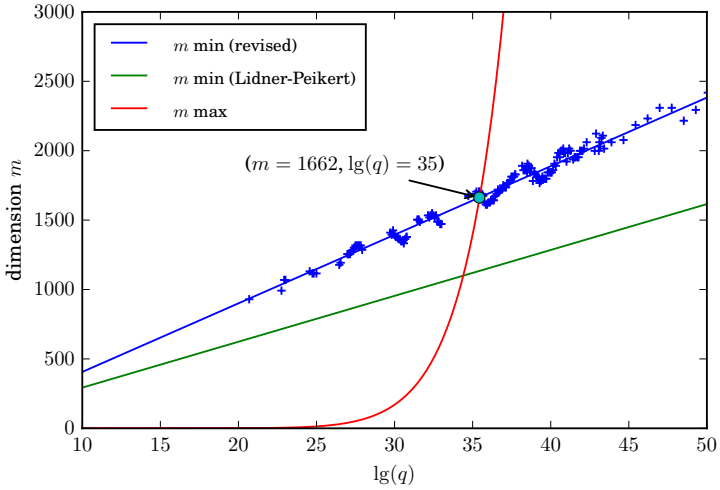


FIGURE 4.5. – Représentation des différentes contraintes sur le choix des paramètres

Contrainte de déchiffrement

Pour renforcer la difficulté des attaques, la solution naturelle est d'augmenter la dimension du problème, c'est pourquoi la sécurité impose une borne minimale sur le degré des polynômes. Cependant, plus la dimension est grande, plus l'amplitude du bruit augmente et donc le déchiffrement risque de devenir incorrect.

Dans la proposition 2, nous avons établi une borne sur le bruit initial d'un chiffré. Dans le but de supporter le schéma d'évaluation figure 4.4, le schéma doit supporter une addition (démasquage), une multiplication (ou addition) puis une nouvelle addition (masquage). Autrement dit, il faut que

$$2(2B_{clean})^2 < \frac{q}{2}\,.$$

En remplaçant B_{clean} par sa borne de la proposition 2, on obtient une contrainte sur $\phi(m)$. La fonction est en effet strictement croissante en fonction de $\phi(m)$. En résolvant numériquement $8B_{clean}^2 - q/2 = 0$, nous obtenons la courbe rouge sur la figure 4.5. Cette courbe donne ainsi la valeur maximale de $\phi(m)$ pour supporter le schéma d'évaluation proposé.

Jeu de paramètres

Pour trouver un jeu de paramètres valide, il ne reste plus qu'à trouver un couple (n, q) valide entre les deux courbes figure 4.5. Il y a également des contraintes supplémentaires à satisfaire, par exemple, les puissances de 2 sont préférables pour n. Ou encore, q doit être un nombre premier et $\phi(m)|q-1$, pour que $\mathbb{Z}_q$ contienne des racines m-ième de l'unité.

Le tableau 4.2 liste des propositions de paramètres ainsi obtenues. Notre cas d'utilisation correspond à la première ligne $\phi(m) = 2048$ et $\lg q$. À titre de comparaison, nous avons également calculé les paramètres pour le schéma BGV dans les mêmes

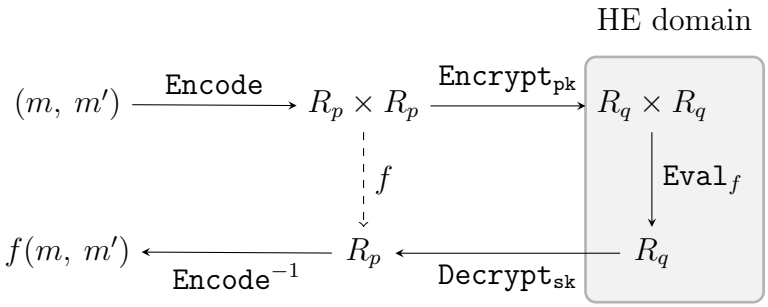


FIGURE 4.6. – Diagramme commutatif de l'encodage

conditions. On constate que le jeu de paramètres minimal offre un gain important en taille de données.

TABLE 4.2. – Différents jeux de paramètres ($k = 128$, $p = 2$ et $\sigma = 1.3$)

nb. mult	min. $\phi(m)$	$\lg q$	taille d'un chiffré (kB)
1	2048	47	24.1
2	2630	55	36.2
5	5100	105	133.9
30	38944	790	7691.4

4.4.5. L'espace des messages, encodages

Un choix essentiel à faire lors de la conception d'une évaluation homomorphe est le choix de la représentation des données. L'espace des messages utilisé dans le BGV (et bien d'autres [LTV12, BLLN13, FV12]), est un anneau de polynômes, R_p. Aussi, comment peut-on de cette arithmétique polynomiale obtenir des opérations exploitables ? Le terme «exploitable» reste à préciser, pour un processeur il faut supporter le plus efficacement possible les opérations communes. Ces opérations sont très générales, la majeure partie est constituée d'opérations logiques et arithmétiques (modulo une puissance de 2).

Nous cherchons à mettre en place le diagramme représenté figure 4.6, où une couche d'encodage est ajoutée avant toute évaluation. Il s'agit de définir le couple d'opérations `Encode`, `Decode`, il est essentiel que ces fonctions présentent également des propriétés d'homomorphisme. Dans le contexte du BGV, le choix de l'entier p (l'espace des messages) conditionne les encodages disponibles et impose éventuellement des contraintes supplémentaires sur les autres paramètres.

4.4.5.1. Encodage direct

On parlera d'encodage direct lorsque l'on place des éléments de $\mathbb{Z}_p$ directement sur des coefficients du polynôme. Cet encodage a l'avantage de pouvoir évaluer très effi-

cacement une addition arithmétique modulo p. En effet, l'addition de deux messages correspond à l'addition coefficient par coefficient. On pourra utiliser efficacement un tel encodage pour évaluer une moyenne de manière homomorphe. En revanche, il est plus difficile d'exploiter la multiplication. Le produit de deux polynômes génère un certain nombre de produits croisés aussi, il n'est pas possible d'obtenir directement les différents produits coefficient par coefficient.

Une façon d'obtenir seulement les produits est d'espacer suffisamment les coefficients dans les polynômes initiaux. Les coefficients du message sont placés aux indices $2^i - 1$, c'est-à-dire que l'on ne peut placer que $\lfloor \lg(\phi(m)+1) - 1 \rfloor$ bits dans un message. En revanche après une multiplication de polynômes a, b ainsi encodés, les produits $a_i b_i$ sont disponibles aux indices $2(2^i - 1)$.

Évidemment, il est impossible d'effectuer plus d'une multiplication, l'encodage espacé n'étant plus respecté. Plusieurs solutions sont envisageables :

— Extraire les coefficients utiles et les replacer sous forme déchiffrée dans le diagramme 4.4. Cette opération est purement structurelle et ne compromet donc pas la sécurité de la solution. Cependant, elle n'est pas complètement satisfaisante, car applicable seulement dans notre cas d'utilisation.

— Implémenter cette opération de manière homomorphe, en utilisant les automorphismes de l'anneau, pour effectuer les permutations.

4.4.5.2. Techniques de *batching*

Une approche bien plus avantageuse pour la représentation des messages est la technique de *batching* introduite par [SV14]. Dans le cas du BGV, si p est correctement choisi, alors le polynôme Φ_m se factorise modulo p en un produit de d polynômes irréductibles tous de même degré l et premiers entre eux. La condition précise est $m | p^l - 1$, avec $\phi(m) = dl$. On a alors

$$\Phi_m(X) = \prod_{i=1}^{d} F_i \mod p$$

En utilisant le théorème des restes chinois, l'anneau des messages est isomorphe au produit de d copies de $\mathbb{F}_{p^l}$, le corps fini à p^l élément. On appellera forme résiduelle la représentation d'un élément de R_p comme un d-uplet de $R_p' = \mathbb{Z}_p[X]/F_1(X) \times \cdots \times \mathbb{Z}_p[X]/F_d(X)$.

Le cas $l = 1$ Dans ce premier cas, Φ_m se factorise en produit de polynômes de degré 1. Ceci nécessite $m \mid p - 1$ et donc $p > m$. L'anneau $\mathbb{Z}_p$ admet alors des racines m-ièmes primitives de l'unité. Par conséquent, la transformation sous forme résiduelle peut se faire très efficacement par une procédure de type NTT.

Le cas $l > 1$ La condition $p > m$ n'est alors plus nécessaire et l'on peut ainsi utiliser des valeurs de p plus faibles, $p = 2$ par exemple. En revanche l'encodage et

le décodage s'effectuent par un théorème des restes chinois plus classique et moins efficace. La décomposition sous forme résiduelle (`Decode`) est calculée par :

$$\begin{aligned} \texttt{CRT} : & R_p \to R_p^{'} \\ & m \mapsto (m \mod F_1, \ldots, m \mod F_d) \end{aligned}$$

Pour l'inversion, on note $\tilde{F_i} = \prod_{j=1}^{d} F_j / F_i$ le produit des polynômes F_j privé de l'indice i. La transformation inverse du théorème des restes chinois peut alors s'écrire :

$$\begin{aligned} \texttt{CRT}^{-1} : & R_p \to R_p^{'} \\ & (m_1, \ldots, m_d) \mapsto \sum_{i=1}^{d} \tilde{F_i} ({\tilde{F_i}}^{-1} m_i \mod F_i) \end{aligned}$$

Opérations parallèles

Un encodage basé sur du *batching* permet donc de placer dans un même message d éléments de $\mathbb{F}_{p^l}$ (où un sous corps de ce dernier), on parlera de *slots* pour les désigner. Les opérations d'additions et multiplications polynomiales homomorphes effectueront ainsi les mêmes opérations sur chacun des *slots* indépendamment. Un tel encodage fait apparaître un parallélisme de type *Single Instruction Multiple Data* (SIMD), où une seule instruction est exécutée sur plusieurs données en parallèle. Ce modèle de programmation très commun et largement utilisé sur les grilles de calcul parallèle (GPU), mais est également présent sur la plupart des processeurs modernes avec les extensions SSE, AVX chez Intel ou Neon chez ARM.

Au-delà des opérations purement arithmétiques, il est également possible d'effectuer des opérations structurelles, via l'utilisation des automorphismes de l'anneau, les fonctions de la forme $\tau_i : X \to X^i$, pour $i \in \mathbb{Z}_m^*$. L'effet d'une partie de ces automorphismes est de permuter les différents slots. Les automorphismes de Frobenius, de la forme $\tau_i : X \to X^{p^i}$, agissent sur chacun des slots indépendamment. À l'opposé, de telles opérations sont très difficiles à effectuer avec un encodage direct.

Les techniques de *batching* présentent néanmoins deux inconvénients :

— tout d'abord les contraintes imposées sur p, rendent impossible l'utilisation d'anneaux où les opérations sont efficaces. Il sera par exemple impossible de prendre $p = 2$ et $m = 2^i$, un cas pourtant intéressant, car la NTT est alors réalisable de manière très efficace avec l'algorithme de Cooley-Tukey [CT65].

— Dans le cas général $l > 1$, l'encodage des messages, autrement dit, l'inversion du théorème des restes chinois est une opération très coûteuse $O(m^2)$. Il est important d'effectuer assez d'opérations avant de déchiffrer pour pouvoir rentabiliser ce coût.

4.5. Compilation pour l'évaluation homomorphe

Nous avons jusqu'à présent vu les caractéristiques principales du schéma homomorphe utilisé. Nous présentons maintenant les problématiques liées à la transformation de programmes pour une exécution homomorphe.

TABLE 4.3. – Construction des opérateurs booléens classiques à partir des opérations de base du schéma.

Opérateur	Notation alternative	Fonction équivalente
$\texttt{Or}(c_0, c_1)$	$c_0 \vee c_1$	$(c_0 \oplus c_1) \oplus (c_0 \wedge c_1)$
$\texttt{Not}(c)$	$\neg c$	$c \oplus 1 \ldots 1$
$\texttt{NAnd}(c_0, c_1)$		$(c_0 \wedge c_1) \oplus 1 \ldots 1$
$\texttt{Iff}(c_0, c_1)$	$c_0 \iff c_1$	$\neg(c_0 \oplus c_1)$
$\texttt{Implies}(c_0, c_1)$	$c_0 \implies c_1$	$\neg(c_0 \wedge \neg c_1)$

4.5.1. Primitives usuelles

Nous rappelons les propriétés essentielles du schéma utilisé :
- Un octet est encodé sur un chiffré (8 bits par chiffré).
- Les opérations suivantes sont supportées sur un octet chiffré
  - $\texttt{Xor}(c_0, c_1)$, noté $c_0 \oplus c_1$ de manière infixe, correspond au «ou exclusif» bit à bit.
  - $\texttt{And}(c_0, c_1)$, noté $c_0 \wedge c_1$ de manière infixe, représente le «et» bit à bit.
  - $\texttt{ShiftL}(i, c)$, $\texttt{ShiftR}(i, c)$, notées $c \ll i$ et $c \gg i$, qui représentent respectivement le décalage à gauche et à droite de la représentation binaire. Il est important de noter que le paramètre i n'est pas dans le domaine chiffré. Pour effectuer un décalage avec un indice chiffré (c'est-à-dire inconnu à l'exécution), il faudra générer un circuit pour cette opération (*barrell shifter* par exemple).

De ces opérations de base, il est possible de construire des opérations arbitraires. Il s'agit en effet de synthétiser (au sens matériel du terme) une fonction donnée, ce problème est résolu depuis bien des années en micro-électronique. En poussant le raisonnement plus loin, il est envisageable de décrire le circuit d'un processeur, sur lequel on pourrait stimuler de manière homomorphe l'exécution de n'importe quel programme (pour un nombre de cycles donné).

Les opérations booléennes usuelles (bit à bit) se déduisent très simplement. Le tableau 4.3 liste la traduction homomorphe des plus communes. Le reste de la partie est dédiée à la transformation de structures conditionnelles et boucles en circuit.

4.5.1.1. Propagation de bits

La propagation de bits, permet de copier la valeur d'un bit (le bit 0 d'un message par convention) sur tous les bits d'un mot de sortie. Tous les autres bits de l'entrée sont supposés nuls. Cet opérateur est très utile pour générer des masques permettant d'implémenter un multiplexeur par exemple. La sémantique de cette opération est la suivante :

$$\texttt{Broadcast}(c = \overline{m}) = \begin{cases} \overline{\texttt{0b11}\ldots 1} & \text{si } m_0 = 1 \\ \overline{0} & \text{si } m_0 = 0 \end{cases}$$

La notation $\overline{m}$ est utilisée pour désigner un chiffré de m, c'est-à-dire $\overline{m} = \texttt{Encrypt}(m)$. Il s'agit ici, formellement de calculer $c_n = c \oplus (c \ll 1) \oplus \cdots \oplus (c \ll n-1)$. De cette expression, nous voyons que le calcul ne nécessite que les opérations `Xor` et $\ll$, toutes deux relativement efficaces dans le schéma homomorphe utilisé.

Pourtant même sur cette opération simple, nous sommes confrontés à plusieurs choix quant à la détermination du circuit. Plaçons nous dans le cas où n est une puissance de 2.

Une approche « diviser pour régner »

Remarquons qu'une approche « diviser pour régner» est possible pour calculer c_n, on a en effet la relation de récurrence suivante :

$$\begin{cases} c_1 = c \\ c_n = c_{n/2} \oplus (c_{n/2} \ll \frac{n}{2}) \end{cases}$$

Exemple 1. Pour $n = 8$, comme dans l'encodage considéré dans cet exemple, cette propagation de bits peut s'effectuer de la manière suivante :

1. $c_2 \leftarrow c \oplus (c \ll 1)$
2. $c_4 \leftarrow c_2 \oplus (c_2 \ll 2)$
3. $c_8 \leftarrow c_4 \oplus (c_4 \ll 4)$

Une approche *map-reduce*

Une façon alternative de décrire ce circuit est par une approche *map-reduce*, où dans un premier temps on calcule $c \ll i$, pour toutes les valeurs de i. Puis on effectue une réduction binaire sur toutes ces valeurs avec l'opérateur $\oplus$.

Pour déterminer la méthode la plus adaptée, il faut analyser les caractéristiques de ces deux méthodes. Les métriques qui nous intéressent sont la profondeur du circuit, et les différents nombres d'opérations. L'approche *map-reduce* minimise la profondeur, mais nécessite plus d'instructions (ou portes logiques). On observe dans le tableau 4.4 qu'asymptotiquement, l'approche diviser pour régner est plus avantageuse. En revanche pour des valeurs de n assez faibles, l'approche *map-reduce* peut devenir avantageuse si assez d'unités d'exécutions homomorphes sont disponibles.

Méthode	Profondeur	Nombre de portes	Nombre de $\ll$
Diviser pour régner	$2 \lg n$	$2 \lg n$	$\lg n$
map-reduce	$1 + \lg n$	$2n - 1$	n

TABLE 4.4. – Comparaison des méthodes pour l'opération `Broadcast`

De manière générale, en supposant une exécution plutôt séquentielle des instructions, l'approche la plus avantageuse sera celle qui minimise le nombre d'instructions (ici, l'approche « diviser pour régner»). À l'opposé, si le support d'exécution a un très fort potentiel de parallélisme, la seconde approche pourrait s'avérer plus efficace. Cependant, au vu de l'empreinte nécessaire pour une seule unité d'exécution (voir chapitre 6 page 93), la première approche est clairement plus adaptée.

4.5.1.2. Multiplexeur

Une opération essentielle, qui n'est pas directement supportée dans le schéma est le multiplexeur. Cette opération est notée `Mux` et possède la sémantique suivante :

$$\texttt{Mux}(c = \overline{s},\, c_x = \overline{x},\, c_y = \overline{y}) = \begin{cases} \overline{x} & \text{si } s = 1 \\ \overline{y} & \text{si } s = 0 \end{cases}$$

Cette opération `Mux` peut être implémentée relativement facilement, on a en effet $\texttt{Mux}(c,\, c_x,\, c_y) = mc_x \oplus \texttt{Not}(m)c_y$, avec $m = \texttt{Broadcast}(c)$. L'avantage de cette expression est que le problème d'évaluer efficacement l'opération `Mux` est réduit à évaluer efficacement l'opération `Broadcast`. L'intérêt principal de cette opération est de pouvoir évaluer des structures conditionnelles, comme nous verrons ci-dessous

4.5.1.3. Suppression des branchements par *if-conversion*

Nous avions évoqué l'opération de *if-conversion* (section 3.3.3.2 page 29) comme une optimisation au chiffrement de code permettant en particulier d'augmenter la taille des blocs de base. Cette transformation est très commune dans le domaine de la compilation. Elle permet de réécrire à l'aide de l'opération `Mux` (parfois aussi appelée `select`) une structure conditionnelle en une séquence d'opérations sans branchements. Une fois transformée, la séquence d'instructions exécutée sera la même, quelle que soit la valeur de la condition. Cette propriété est très appréciée en sécurité pour enlever toute relation entre le temps d'exécution et des données sensibles [CVDBDS09]. De telles dépendances peuvent en effet mener à des attaques par *timing* [Koc96]. La *if-conversion* est également utilisée par les compilateurs pour améliorer les performances [MLC+92]. Les processeurs modernes ont une exécution hautement spéculative [HP11], il s'avère qu'une mauvaise prédiction (*branch-miss)* coûte une dizaine de cycles. Si une portion de code critique a une condition de branchement difficile à prédire, il vaut mieux, dans une certaine mesure, exécuter plus d'instructions (car la *if-conversion* génère plus d'instructions) mais de manière déterministe. Les heuristiques permettant de déterminer si la *if-conversion* doit avoir lieu sont assez complexes et nécessitent des traces d'exécutions pour détecter les zones critiques.

Un exemple de *if-conversion* est représenté figure 4.7. Étant donné une structure conditionnelle binaire *if-else*, il faut procéder de la manière suivante, pour chacune des variables affectées dans au moins l'une des deux branches :

1. associer deux variables x_{if}, x_{else} correspondant à valeur de x après l'exécution de chacune des branches. Si x est affecté dans seulement l'une des deux branches, l'une de ces deux variables vaudra simplement x.
2. Affecter à x la valeur $\texttt{Mux}(cond,\, x_{if},\, x_{else})$.

```
if (cond) {
    x = x + 1;
} else {
    x = x * x;
}
```

(a) Programme original

```
x_if   = x + 1;
x_else = x * x;

x = Mux(cond, x_if, x_else);
```

(b) Après *if-conversion*

FIGURE 4.7. – Exemple de *if-conversion*

4.5.1.4. Transformation des boucles

Les boucles sont des éléments essentiels des programmes, elles apparaissent par exemple sous forme de `for` ou `while` dans les langages impératifs. Malheureusement, elles ne sont pas directement représentables par un circuit. Pour transformer les boucles en circuit, il faut voir une boucle comme une itération bornée d'une fonction sur un certain état (un ensemble de variables). Cet état, même de manière chiffrée est représentable, il s'agit simplement des différentes données chiffrées en mémoires. La fonction itérée sur cet état est également transformable en circuit.

Pour prendre cette approche, il est impératif que le nombre d'itérations de la boucle soit connu et borné. Ensuite pour pouvoir évaluer de telles boucles, nous devons munir la plateforme d'évaluation d'une primitive supplémentaire que nous nommerons `iter`. Cette primitive permet d'appliquer un circuit un nombre de fois fixe sur un état. La transformation d'une boucle en circuit s'effectue ainsi de la manière suivante :

1. Identifier l'état S de la boucle
2. Transformer la boucle en une boucle à itération fixe `While` $\rightarrow$ `iter`, il s'agit ici d'effectuer la transformation représentée dans la figure 4.8.
3. Transformer le corps de la boucle en circuit. Cette procédure est potentiellement récursive si plusieurs boucles sont imbriquées. On remarque figure 4.8, qu'au moins une *if-conversion* invoquée lors de cette étape.

4.6. Évaluation homomorphe de l'algorithme AES

L'algorithme *Avanced Encryption Standard* est un algorithme de chiffrement par bloc standardisé par le NIST en 2001 à travers la publication du FIPS-197 [NIS01]. Son prédécesseur, le *Data Encryption Standard* (DES) proposé en 1977, devenait à cette époque dangereusement vulnérable vis-à-vis des attaques exhaustives, en grande partie à cause de sa faible taille de clé, seulement 56 bits. L'algorithme Triple-DES défini dans la troisième révision du standard FIPS-46 [NIS99] en 1999, permet d'étendre la taille de clé jusqu'à 112 bits. À ce jour, il est encore utilisé dans certaines applications. Cependant, la taille bloc (64 bits) et de clé restent trop faibles par rapport aux recommandations actuelles de 128 bits. Par ailleurs, le chaînage de trois algorithmes DES est très coûteux et peu efficace. C'est dans cette optique qu'une compétition a

```
while (cond) {
    // corps de boucle
  update(s);
}
```

(a) Boucle originale

```
for (int i = 0; i < maxIter; i++) {
    if (cond) {
        // corps de boucle
        update(s);
    }
}
```

(b) Boucle transformée

FIGURE 4.8. – Transformation de boucles à nombre d'itérations connu

été initiée par le NIST en 1997 auprès des chercheurs en cryptographie, dans le but de déterminer le successeur du DES. Après quatre ans de compétition, l'algorithme Rijndael, proposé par Vincent Rijmen et Joan Daemen fut retenu.

Nous allons dans cette partie détailler la transformation d'un algorithme AES, dans sa version 128 bits, pour l'exécuter de manière homomorphe. Mais avant cela, il est légitime de se demander l'intérêt d'exécuter un algorithme de chiffrement, alors même que les données sont déjà chiffrées. La première motivation est qu'il s'agit d'un cas d'évaluation relativement commun dans la littérature des chiffrements homomorphes [GHS12, DHS16, MS13].

De manière plus générale, une application intéressante motivant l'évaluation homomorphe de primitives cryptographiques est pour l'optimisation des transmissions. Il est en effet extrêmement inefficace de transférer directement des chiffrés homomorphes, qui ont un facteur d'expansion important. Un message chiffré prend beaucoup plus de place que le message original. À la place, il a été suggéré par [NLV11] d'utiliser un chiffrement hybride. Supposons que l'on veuille transmettre n messages à une entité tierce pour qu'elle effectue une opération homomorphe dessus. Au lieu d'envoyer

$$c = (\mathtt{HE.Enc}_{pk}(m_1), \ldots, \mathtt{HE.Enc}_{pk}(m_n))$$

un vecteur de chiffrés homomorphes, il est plus avantageux, en vue de minimiser les donnée transmises, d'envoyer

$$c' = (\mathtt{HE.Enc}_{pk}(k), \mathtt{AES}_k(m_1), \ldots, \mathtt{AES}_k(m_n)),$$

contenant seulement le chiffré homomorphe d'une clé d'un AES, suivi d'un vecteur de n messages chiffrés par un algorithme AES [2] avec cette même clé (voir figure 4.9). Tout l'intérêt de cette approche est qu'un chiffré AES possède un très faible facteur

2. La notation $\mathtt{AES}_k$ désigne un algorithme AES étendu pour des tailles de message arbitraires.

(a) Approche traditionnelle (b) Approche hybride

FIGURE 4.9. – Comparaison des méthodes de transmission des données pour un calcul homomorphe.

d'expansion par rapport à un chiffré homomorphe. Supposons que la taille des messages soit fixe, on note T_{he} et T_{aes} respectivement, la taille d'un chiffré homomorphe et la taille d'un chiffré AES. Il faut évidement que $T_{aes} < T_{he}$, sinon la méthode ne présente aucun avantage. Dans le premier cas, la taille du chiffré est $T(c) = nT_{he}$, alors que dans le second cas $T(c') = T_{he} + nT_{aes}$. Ainsi, si le nombre de chiffré est assez important, on aura $T(c')<T(c)$, en résolvant cette inégalité en fonction de n, on arrive à la condition

$$n > \frac{T_{he}}{T_{he} - T_{aes}},$$

qui donne la valeur minimale de n nécessaire pour réduire réellement la taille des données transmises. En revanche, pour pouvoir évaluer une fonction sur le chiffré modifié c', le protocole est plus complexe. Il faut tout d'abord chiffrer les messages AES reçus, c'est-à-dire calculer $x_i = \mathtt{HE.Enc}_{pk}(\mathtt{AES}_k(m_i))$. Ensuite, il faut évaluer de manière homomorphe la fonction $\mathtt{AES}_k^{-1}$ (le chiffré de la clé secrète fait partie du message) sur tous les x_i. On obtient alors le vecteur c, sur lequel il est maintenant possible d'évaluer la fonction souhaitée. Pour rendre cette application possible, il est crucial que l'exécution homomorphe de la primitive cryptographie utilisée soit assez rapide. Mentionnons, pour terminer qu'un des axes de recherche actuels est de trouver des primitives cryptographiques particulièrement adaptées à une exécution homomorphe [CCF+16, ARS+15].

En résumé, l'algorithme AES offre à la fois un cas d'exemple réaliste, assez complexe, relativement standard et bénéficie d'applications intéressantes. Dans un premier temps, nous présenterons brièvement les points essentiels de l'AES, puis nous verrons ensuite comment le transformer en une version homomorphe. Notons pour terminer que ce cas d'exemple ne présente pas un cas favorable pour notre schéma. Certains travaux, comme [GHS12] spécialisent le schéma et l'encodage spécifiquement pour l'AES.

4.6.1. L'algorithme AES

Toutes les opérations de l'AES sont définies dans le corps fini $\mathbb{F}_{2^8}$, avec la représentation spécifique induite par le polynôme irréductible dans $\mathbb{Z}_2$, $P(X) = X^8 + X^4 + X^3 + X + 1$. Ainsi, un élément $b \in \mathbb{F}_{2^8}$ peut être représenté par un polynôme de degré 8, $b = b_7X^7 + \cdots + b_1X + b_0$ et les différentes opérations sont faites modulo $P(X)$.

L'AES 128 bits a pour état interne une matrice de 4×4 octets, notée S et représentée de la manière suivante :

$$S = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

L'algorithme consiste en une succession de plusieurs rounds, eux-mêmes décomposés en étapes élémentaires. La boucle principale est donnée dans l'algorithme 4.2.

Algorithme 4.2 Boucle principale de l'algorithme AES

```
1: AddRoundKey(S, w_0)
2: for i = 1 to 10 do
3:     SubBytes(S)
4:     ShiftRows(S)
5:     MixColumns(S)
6:     AddRoundKey(S, w_i)
7: end for
8: SubBytes(S)
9: ShiftRows(S)                    ▷ pas de MixColumn à la dernière itération
10: AddRoundKey(S, w_10)
```

ShiftRow

L'opération `ShiftRow` est seulement structurelle, il s'agit de permuter de manière cyclique les lignes de la matrice S. La ligne i est décalée de i positions vers la gauche. Ainsi, on obtient

$$\texttt{ShiftRow}(S) = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

MixColumn

L'étape `MixColumn` est plus complexe, il s'agit d'appliquer une transformation linéaire à chaque colonne. La colonne C_i de la matrice S devient

$$C'_i = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} C_i.$$

Comme les opérations `ShiftRow` et `MixColumn` sont exécutées séquentiellement, elles sont en général composées en une seule opération. On remarque que les coefficients de cette opération sont petits et prennent des valeurs très restreintes, ce qui permet une implémentation plus efficace.

SubBytes

L'opération `SubBytes` applique sur chacun des octets de la matrice d'état S une bijection de $\mathbb{F}_8 \to \mathbb{F}_8$. Pour calculer $\texttt{SubBytes}(x)$, on commence par calculer $y = x^{-1}$ (avec pour convention $y = 0$ si $x = 0$), puis une transformation affine est appliquée à y pour obtenir le résultat. Plus précisément,

$$\texttt{SubBytes(x)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} y + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Cette opération semble à première vue très coûteuse, d'une part à cause de l'inversion de x, puis, par l'application de la transformation affine. C'est pourquoi, il est commun dans les implémentations, matérielles ou logicielles, de recourir à une *lookup table*. L'ensemble de valeurs d'entrées de cette fonction est en effet très réduit, 256 éléments seulement, il est avantageux de calculer un tableau de 256 octets, stockant la valeur $\texttt{SubBytes}(x)$ à l'indice x. Le calcul de $\texttt{SubBytes}(x)$ consiste alors en une simple lecture en mémoire, une opération bien plus efficace.

AddRoundKey

Enfin, l'opération `AddRoundKey` effectue un «ou exclusif» avec la clé étendue (calculée par la procédure d'expansion de clé), notée w_k, colonne par colonne. Ainsi, la colonne C_i devient

$$C_i' = C_i \oplus w_{k+i}$$

La procédure d'expansion de clé n'est pas rappelée par souci de concision. Cependant, elle est construite seulement à partir des primitives présentées dans cette section.

4.6.2. Transformation de l'algorithme AES

Pour exécuter de manière homomorphe un algorithme comme un AES, il s'agit de le transformer dans un premier temps en un circuit équivalent, constitué seulement d'opérations élémentaires supportées par le schéma. À partir de ce circuit, il est alors possible générer une suite d'instructions qui évalue ce circuit.

Quel encodage ?

Il est dans un premiers temps essentiel de définir comment l'état de l'AES est représenté. L'évaluation de l'AES proposée dans [GHS12], utilise de manière avancée les techniques de *batching* : l'état S est encodé dans un seul chiffré. L'évaluation de l'AES n'implique la manipulation de presque seulement un seul chiffré.

Avec un encodage direct, il n'est pas possible d'utiliser ces techniques de calcul *Single Instruction Multiple Data* (SIMD). Pour cette raison nous encodons les octets de la clé dans des chiffrés distincts, de même que les différents octets de l'état. Le stockage nécessaire est évidemment beaucoup plus élevé qu'avec une représentation de type *batching*. Cependant toutes les transformations structurelles à l'échelle des octets (permutations) consistent seulement à des manipulations mémoire (changement de pointeurs par exemple).

Multiplication dans $\mathbb{F}_{2^8}$

Pour évaluer la multiplication dans $\mathbb{F}_{2^8}$, nous pouvons utiliser une méthode assez simple n'utilisant que des opérations bits à bits élémentaires. Pour cela, définissons dans un premier temps la fonction `xtime`, qui pour un élément de $\mathbb{F}_{2^8}$, représenté comme un polynôme, effectue l'opération $Xa(X) \mod p(X)$. Si $a(X) = a_7X^7 + \cdots + a_0X^0$, on remarque que seulement deux cas se présentent :

$$\begin{cases} a_7 = 0 & \text{, alors } a(X)X = a_6X^6 + \cdots + a_0X \\ a_7 = 1 & \text{, alors } a(X)X = a_6X^6 + \cdots + a_0X - P(X) \end{cases}$$

Pour cette raison, l'algorithme `xtime` nécessite seulement des opérations élémentaires (voir l'algorithme 4.3) et également l'opération `Mux`.

Algorithme 4.3 Fonction `xtime`, version homomorphe

1: $t \leftarrow a \gg 1$
2: $s \leftarrow \texttt{Mux}(a \gg 7,\ \texttt{0x1b},\ 0)$
3: **return** $t \oplus s$

Pour faire le produit de deux éléments de $\mathbb{F}_{2^8}$ a et b, on considère leur représentation polynomiale. On peut alors l'exprimer comme une accumulation de la fonction `xtime`, comme montré dans l'algorithme 4.4. En écrivant le produit des polynômes, on a en effet :

$$\begin{aligned} a(X)b(X) &= (a_7X^7 + \cdots + a_0)b(X) \\ &= a_0b(X) + (a_7X^6 + \cdots + a_1)Xb(X) \\ &= a_0b(X) + (a_7X^6 + \cdots + a_1)\texttt{xtime}(b) \\ &= a_0b(X) + a_1\texttt{xtime}(b) + \cdots + a_7\texttt{xtime}(\texttt{xtime}(\ldots\texttt{xtime}(b))) \end{aligned}$$

Traduction des opérations

La plupart des opérations de l'AES se traduisent directement de manière homomorphe :

— L'opération `ShiftRow` est structurelle, elle est donc effectuée seulement en manipulant les adresses des chiffrés.
— L'opération `AddRoundKey` ne fait appel qu'à un XOR, qui est nativement supporté et efficace dans notre encodage.
— L'opération `MixColumn` se traduit relativement bien, la seule difficulté est d'émuler la multiplication de $\mathbb{F}_8$ avec notre encodage (voir l'algorithme 4.4).

Algorithme 4.4 Évaluation homomorphe de la multiplication dans $\mathbb{F}_8$

1: $x \leftarrow b$
2: **for** i from 0 downto 7 **do**
3: $\quad res \leftarrow res \oplus \texttt{Mux}(a_i, x)$
4: $\quad x \leftarrow xtime(x)$
5: **end for**
6: **return** x

Évaluation de l'opération `SubBytes`

L'opération qui pose le plus de difficultés à traduire est la fonction `SubBytes`. Dans l'optique d'une exécution homomorphe, une implémentation par table n'est pas la plus adaptée. Il est plus efficace d'évaluer algébriquement cette fonction. Tout d'abord, pour calculer x^{-1}, il suffit de remarquer que $x^{255} = 1$, ainsi $x(x^{254}) = 1$ donc $x^{-1} = x^{254}$. Autrement dit, on peut calculer x^{-1} par une élévation à la puissance 254 de x.

L'algorithme retenu pour effectuer cette élévation à la puissance est représenté sur la figure 4.5. Il s'agit d'une version itérative d'une exponentiation modulaire rapide (*square-and-multiply*). Dans [GHS12], les auteurs utilisent les automorphismes de Frobenius pour évaluer encore plus rapidement cette expression. Malheureusement, dans la représentation choisie, les automorphismes de Frobenius ne correspondent à aucune opération sur les messages en clair.

Algorithme 4.5 Exponentiation modulaire homomorphe

Entrée : x, pow
1: $res \leftarrow \overline{1}$
2: **for** i from 0 downto 7 **do**
3: $\quad res \leftarrow \texttt{Mux}((pow \gg i) \wedge \overline{1},\ res * x,\ res)$
4: $\quad x \leftarrow x * x$
5: **end for**
6: **return** res

Pour évaluer l'inversion, on peut donc spécialiser l'algorithme 4.5 pour l'exposant 254. Le circuit d'évaluation de la fonction $x \to x^{254}$, même avec une exponentiation modulaire rapide reste assez profond, il faut effectuer 8 étapes d'élévation au carré.

Performances

Pour évaluer les performances, nous avons modifié en grande partie un AES existant [3]. Tout l'algorithme de chiffrement a été réécrit en appliquant les transformations décrites précédemment et en remplaçant les opérations de base par les primitives homomorphes. Le programme est compilé pour cible MIPS avec le compilateur LLVM et l'option de compilation -O2.

3. Disponible à l'adresse https://github.com/kokke/tiny-AES-c

Table 4.5. – Temps d'évaluation homomorphe de l'AES

Méthode	Chiffrement	Déchiffrement	Éval. AES	Total
[GHS12], SHE	245.1 s	394.3 s	252 s	14 min
[GHS12], FHE	1049.9 s	1630.5 s	1050 s	62 min
Proposée (estimation)	154.8 μs	154.9 μs	256 s	4.2 min

Le simple fait d'avoir une représentation sous forme de circuit de l'AES introduit un fort ralentissement. Sur le processeur MIPS d'évaluation (présenté dans le chapitre 6), le temps d'évaluation même sur des données en clair augmente de 6364 à 699747 cycles (soit une multiplication par 109).

Nous avons ensuite exécuté ce programme et généré une trace des instructions. Après analyse, la répartition des instructions est listée dans le tableau 4.6. À chaque catégorie d'instructions, on associe le temps estimé en se basant sur les résultats obtenus dans le chapitre 6 (en supposant le jeu de paramètres minimal). Le temps d'une opération mémoire correspond au temps de transfert d'un élément chiffré depuis la mémoire principale vers les mémoires locales, l'analogue des registres. Nous prenons ici une estimation optimiste, calculée à partir de la bande passante théorique de la mémoire DRAM de la carte de test utilisée (33.31 Gbit/s). Mis bout à bout, on obtient donc un temps d'évaluation approximatif de 256 secondes, soit environ 4 minutes pour un AES (voir le tableau 4.5). Pour comparaison, la version *somewhat homomorphic* de [GHS12] nécessite (sur un processeur Intel i5 fonctionnant à 2.6GHz) également 4 minutes et celle avec un schéma complètement homomorphe 18 minutes. Mais cette comparaison ne prend pas en compte les temps de chiffrement et de déchiffrement. Dans [GHS12], ces derniers sont particulièrement importants. Si nous les considérons, les temps d'évaluation homomorphe de l'AES sont alors de 14 minutes pour la version *somewhat homomorphic* et de 62 minutes pour la version complètement homomorphe.

En analysant les accès mémoire, seulement 290 octets différents sont accédés au cours de l'exécution, aussi le stockage nécessaire est environ de 171MB. Cette valeur se compare favorablement aux 3GB utilisés dans [GHS12]. La réduction est en grande partie liée à l'utilisation d'un jeu de paramètres plus réduit (les chiffrés sont naturellement plus petits).

Si les gains en mémoire sont clairement visibles, les gains en temps d'évaluation homomorphe le sont moins. Cependant, il reste de nombreux leviers d'optimisation. Le premier étant l'algorithme lui-même. Nous avons appliqué ici une traduction très générique de l'algorithme AES en circuit. Il est clair qu'une traduction plus optimisée en fonction du schéma homomorphe utilisé peut apporter des gains importants [GHS12]. L'autre point d'amélioration est l'architecture, en réduisant les temps de calcul des opérations de base ou en augmentant la fréquence du circuit. Le circuit sur logique programmable utilisé pour les estimations (voir chapitre 6) est cadencé à 100Mhz, ce qui laisse une marge importante d'augmentation.

TABLE 4.6. – Histogramme des instructions pour un AES sous forme de circuit

instruction	nombre	temps estimé
`Or`	165233	$929.8\mu s$ $(2 * t_{xor} + t_{and})$
`ShiftL`	151830	$309.9\mu s$ (t_{xor})
`And`	110201	$310\mu s$
`Xor`	72502	$309.9\mu s$
opération mémoire	68770	$4.7\mu s$
`ShiftR`	67990	$309.9\mu s$ (t_{xor})
non homomorphe	63221	$\approx 10ns$

4.7. Conclusion

Le chiffrement homomorphe offre une solution très générique pour protéger la manipulation des données : les données restent chiffrées même pendant les calculs. Un processeur, ne connaissant pas à l'avance les opérations à effectuer il est nécessaire d'utiliser un schéma complètement homomorphe, qui supporte l'évaluation d'opérations arbitraires. Ces schémas, bien que réalisables, amènent de forts ralentissements et une consommation considérable en mémoire.

Aussi, nous avons présenté et étudié un compromis : sécuriser matériellement des composants clairement identifiés pour construire une procédure de rechiffrement permettant de ramener le bruit dans un message chiffré à un niveau limité. Ceci permet d'éviter de recourir au coûteux *bootstrapping* tout en fournissant un schéma capable d'évaluer des opérations arbitraire*s.*

Nous avons ensuite dérivé un jeu de paramètres minimal, représentant la limite atteignable par l'approche proposée en se basant sur le schéma BGV. Après avoir étudié les problématiques liées à la transformation des programmes pour l'évaluation homomorphe, nous avons détaillé un exemple complet avec l'algorithme AES. Nous avons ensuite estimé le temps et les ressources utilisées pour effectuer ce calcul de manière homomorphe. Une nette réduction de la mémoire utilisée est observée. En revanche, le temps de calcul obtenu est similaire à ceux de la littérature (4 minutes pour un AES). Ceci reste néanmoins encourageant étant donné que l'algorithme a été traduit de manière générique, nous n'avons pas appliqué d'optimisations spécifiques à l'AES. De plus, l'estimation du temps des opérations est effectuée pour une architecture fonctionnant à une fréquence relativement faible (100Mhz). Il reste donc un large espace d'optimisation, tant au niveau de la construction du circuit que dans son évaluation efficace.

Accélération matérielle pour le chiffrement homomorphe

5

Dans le chapitre précédent, nous avons présenté une première approche pour réduire le coût de l'utilisation d'un chiffrement homomorphe. En sécurisant une zone bien définie, les paramètres du schéma peuvent être fortement réduits et mécaniquement, les performances sont améliorées. Un autre levier d'optimisation est l'accélération matérielle des calculs homomorphes. Ces opérations en plus de s'exécuter plus rapidement sont généralement plus efficaces en terme énergétique. Enfin, implémenter ces accélérateurs permet au processeur de décharger les opérations homomorphe et de continuer à exécuter en parallèle des opérations indépendantes. Après avoir identifié les blocs critiques, nous proposerons et comparerons divers accélérateurs pour ces fonctions.

Sommaire

5.1. Introduction

Construire le processus d'évaluation homomorphe autour d'une fonction de rechiffrement légère a permis de réduire considérablement les paramètres du schéma. L'accélération matérielle de l'évaluation homomorphe permet d'améliorer encore plus les performances. Le schéma de chiffrement BGV est construit sur un chiffrement de type RLWE. Aussi, les accélérateurs décrits dans ce chapitre s'appliquent également dans un contexte bien plus large.

La conception d'un tel accélérateur comprend deux aspects majeurs. En premier lieu, accélérer les opérations de base que sont les calculs modulaires dans $\mathbb{Z}_q$, l'arithmétique polynomiale et la génération de nombres aléatoires. L'autre aspect de la conception est de combiner efficacement ces briques pour construire les différentes fonctions plus haut niveau du schéma (chiffrement, opérations homomorphes, rechiffrement, ...). Nous décrivons tout d'abord l'architecture complète proposée avant de présenter en détail les choix et la conception des différentes briques élémentaires.

5.1.1. Les architectures existantes

On trouve dans la littérature plusieurs travaux sur l'accélération d'algorithmes de chiffrement basés sur les réseaux euclidiens. Un nombre important traite de l'accélération de blocs spécifiques (Multiplication polynomiale, NTT, échantillonnage de gaussiennes discrètes), une proportion plus restreinte présente des architectures complètes. Le tableau 5.1 liste plusieurs architectures matérielles et implémentations orientées pour l'évaluation homomorphe ou RLWE. Ces travaux illustrent clairement la faisabilité et les gains possibles avec l'accélération matérielle. Les implémentations pour RLWE [PG14, RVM+14, RJV+15] sont particulièrement légères. En revanche les architectures pour le chiffrement homomorphe occupent une grande partie de la surface des FPGAs haut de gamme [ÖDSS15, PNPM15, RJV+15]. Cette différence s'explique principalement par la dimension des données manipulées, plus élevée dans le second cas.

Les paramètres déterminés pour notre schéma d'évaluation sont heureusement plus proches de ceux que l'on peut trouver pour un chiffrement asymétrique RLWE et promettent donc une implémentation relativement compacte de l'accélération homomorphe.

5.1.2. Vue d'ensemble de l'architecture proposée

L'étude de paramètres menée dans la section 4.4.4, a permis d'obtenir un jeu de paramètres optimisé pour l'évaluation de circuits de faible profondeur. Dans la configuration obtenue pour une sécurité $\lambda = 128$ bits, un chiffré est une paire de polynômes de degré 2048 avec des coefficients sur 37 bits, soit une taille d'environ 19 KB. Un chiffre conséquent certes, mais il reste dans des ordres de grandeur accessibles pour un stockage sur FPGA. À titre d'illustration, le tableau 5.2 donne l'espace occupé par

TABLE 5.1. – Synthèse des implémentations homomorphes

Références	Caractérisiques	Performances
[PG14]	Accélération HW complète RLWE. $m = 512$, $\lg q = 13$.	Virtex 6@251 MHz 5595 LUT, 4760 FF — Encrypt 13.7K cycles (54μs) — Decrypt 8.8K cycles (35 μs)
[RVM+14]	Accélération HW complète RLWE. $m = 512$, $\lg q = 13$.	Virtex 6@278 MHz 1536 LUT, 953 FF — Encrypt 13.3Kcycles (47 μs) — Decrypt 5.8Kcycles (21 μs)
[RJV+15]	Schéma YASHE, $m = 2^{15}$, $\lg q_i = 30$. Architecture orientée hautes performances, mixte HW/SW, exploite le batching (2048 slots).	Virtex 7@143MHz 360,353 LUT, 339,086 FF — SIMON-64/128 (157.7 s)
[ÖDSS15]	Schéma LTV, mixte HW/SW, accélération des opérations critiques seulement (NTT, *key-switching*, *modulus-switching*). $m = 2^{15}$, $\lg q_i = 32$	Virtex7@250MHz 433,200 LUT, 866,400 FF AES-128 (estimé) : 15min, 0.4s / bloc
[PNPM15]	Schéma YASHE, $m = 2^{14}$. Optimisation pour NTT de grande dimension.	Stratix V@66MHz 141,090 ALM, 391,773 FF.
[GHS12]	Schéma BGV, $m \approx 2^{16}$, $\lg q_i \approx 40$, SW basée sur [HS].	Intel XEON AES-128 : ≈4min, 2s / bloc
[DHS16]	Schéma LTV, $m = 2^{15}$, $\lg q_i = 31$, SW.	AES-128 : 31h, 55s par bloc

FPGA	Mem.	Taille rel.	Nb. max
Cyclone V	394.24 KB	4.8 %	20
Arria V GX	2.2 MB	0.84 %	116

TABLE 5.2. – Coût de stockage d'un chiffré entier, sur différentes cartes FPGA

un tel chiffré, sur un FPGA d'entrée de gamme (Cyclone V) et moyen de gamme (Arria V GX). On constate qu'il est possible de stocker entièrement un, voire plusieurs chiffrés dans les mémoires internes BRAMs d'un FPGA. En plus d'être fortement intégrées avec la logique de calcul du FPGA, elles peuvent fonctionner à fréquence élevée.

L'architecture proposée est construite pour supporter de manière matérielle toutes les primitives homomorphes. Seul l'encodage des données, le *batching* par exemple est effectué au besoin de manière logicielle. L'accélération matérielle complète présente des avantages indéniables à commencer par les performances. De plus, la forte intégration de la fonction de déchiffrement offre une protection naturelle de cette dernière (la clé secrète n'est pas dans une mémoire accessible directement au processeur).

L'inconvénient majeur est le manque de souplesse par rapport à une approche mixte matérielle-logicielle. L'architecture proposée ici est clairement dans un cas favorable : les degrés sont faibles et restreints aux puissances de deux, les modules considérés sont faibles. Dans des cas moins favorables, une approche mixte permet d'utiliser des algorithmes plus complexes et éventuellement d'obtenir de meilleures performances.

La figure 5.1 représente l'architecture proposée pour l'accélérateur de calculs homomorphes. Elle possède de nombreuses similitudes avec une architecture de processeur (ALU, banc de registres, ...). La différence fondamentale est qu'il s'agit ici d'une architecture *dataflow*, c'est-à-dire que sur les fils transitent les différents coefficients des polynômes au cours du temps et pas le polynôme en un seul bloc. La partie flot de données de l'architecture est découpée en trois parties :

1. la lecture des polynômes sources. L'unité d'exécution prend deux opérandes «x » et « y». L'opérande « y » est lue directement depuis le banc de registres. Pour l'opérande « x » plusieurs sources sont possibles :
   — Directement depuis la mémoire principale, c'est-à-dire un polynôme déjà chiffré.
   — Depuis la mémoire principale, mais en chiffrant le polynôme.
   — Depuis le banc de registre.
2. L'exécution de l'instruction par l'ALU homomorphe.
3. Le stockage du résultat, ici encore on peut vouloir le stocker en mémoire principale ou dans un banc de registre.

Les instructions sont placées dans une file et exécutées séquentiellement. L'unité de contrôle est chargée de décoder l'instruction et de définir les signaux qui contrôlent le chemin suivi par les données. On donne sur la figure 5.2 des exemples de décodage

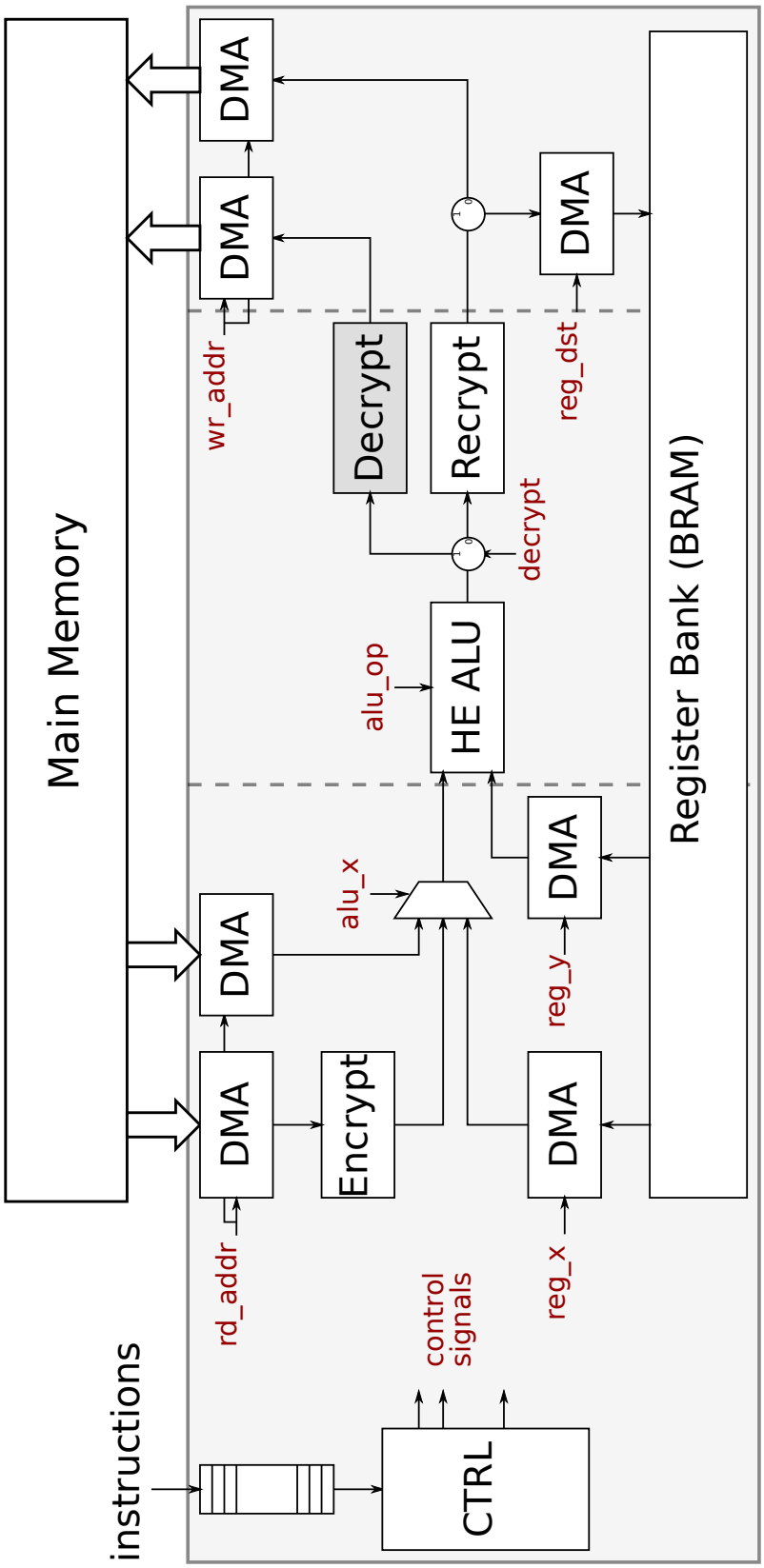


FIGURE 5.1. – Accélérateur de calculs homomorphes

signal	valeur
reg_x	1
reg_y	2
alu_x	`REG`
alu_op	`XOR`
reg_dst	3
$decrypt$	false

(a) `xor r3, r1, r2` , addition registre à registre

signal	valeur
rd_addr	`ADDR`
alu_x	`MEM`
alu_op	`NOP`
reg_dst	1
$decrypt$	false

(b) `load r1, ADDR` chargement depuis la mémoire d'une donnée chiffrée

FIGURE 5.2. – Exemples de décodage des signaux de contrôle pour différentes instructions

d'instructions.

5.2. Réduction modulaire

La réduction modulaire est une opération omniprésente dans les calculs homomorphes. En effet, tous les calculs élémentaires sont effectués dans l'anneau $\mathbb{Z}_q$. Par comparaison avec les autres opérations arithmétiques (additions, multiplications), la division Euclidienne pose de plus grandes difficultés à être implémentée efficacement. L'algorithme est intrinsèquement très séquentiel. D'après l'analyse de paramètres menée dans le chapitre précédent, les modules q considérés sont de taille relativement faible, entre 30 et 64 bits. Il s'agit d'un cas plutôt favorable par rapport aux schéma asymétriques classiques comme le RSA ou les courbes elliptiques, où les réductions modulaires peuvent aller de 256 à 4096 bits.

Nous allons dans cette sous-partie, passer en revue les approches applicables pour effectuer cette réduction modulaire efficacement. Elles se divisent en trois catégories :

— l'utilisation de modules de formes spécifiques, avec lesquels la réduction peut-être effectuée très efficacement. C'est évidemment le cas de la division par une puissance de deux, qui consiste en un simple décalage de la représentation binaire. Mais il existe d'autres formes particulières où la réduction peut être effectuée efficacement comme les nombres de Mersenne, de la forme $q = 2^n - 1$. Une étude étendue du sujet est effectuée dans [Pla05].

— Les méthodes de réduction génériques, qui peuvent réduire avec des modules arbitraires. Il s'agit typiquement des circuits de division qui équipent les processeurs.

— Les méthodes avec pré-calculs, qui sont plus efficaces que les précédentes, mais nécessitent que le module soit connu à l'avance. S'il n'est pas connu à l'avance, il faut que suffisamment de réductions soient effectuées avec le même module pour que ces méthodes deviennent avantageuses.

5.2.1. Notations

Pour $x \in \mathbb{R}$, la partie entière de x, notée $\lfloor x \rfloor$, est définie comme le plus grand entier inférieur ou égal à x. Formellement, $\lfloor x \rfloor$ est l'unique entier vérifiant :

$$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$$

Proposition 3 (Division euclidienne)**.** *Étant donné $x \in \mathbb{Z}$ et $q \in \mathbb{Z}^*$, il existe un unique couple $(k, r) \in \mathbb{Z}^2$ tel que $x = kq + r$, avec $0 \leq r < |q|$. k est appelé le* quotient *et r le* reste*. On notera $\bmod$, l'opération qui retourne le reste, $r = x \bmod q$.*

Cette définition n'est cependant pas aisée à manipuler en particulier lorsqu'il s'agit d'établir des encadrements. Aussi, le quotient est également défini comme la partie entière de la fraction $\frac{x}{q}$.

Proposition 4. *Pour $x \in \mathbb{Z}$ et $q \in \mathbb{Z}^*$, $\lfloor \frac{x}{q} \rfloor$ est le quotient de la division euclidienne de x par q.*

Démonstration. D'après la définition de la partie entière.$\lfloor \frac{x}{q} \rfloor \leq \frac{x}{q} < \lfloor \frac{x}{q} \rfloor + 1$. En multipliant par q et soustrayant $q\lfloor \frac{x}{q} \rfloor$, on obtient

$$0 \leq \frac{x}{q} - q\lfloor \frac{x}{q} \rfloor < q\,.$$

D'après l'unicité de la division euclidienne, $\lfloor \frac{x}{q} \rfloor$ est bien égal au quotient. □

5.2.2. Réduction par approximation du quotient (Barrett)

Nous rappelons dans cette section la méthode de Barrett [Bar87] pour effectuer la réduction modulaire. Il ne s'agit pas de la version originale, mais d'une version généralisée proposée par Dhem [DQ98]. Soit α, $\beta \in \mathbb{Z}$ les paramètres de la réduction, nous donnerons plus loin (proposition 5) des contraintes plus précises sur ces derniers.

L'idée générale est relativement intuitive, on remarque que le reste $x \mod q$ est défini comme $x \mod q = x - q\lfloor x/q \rfloor$, la méthode de Barrett consiste à estimer au mieux le quotient $\lfloor x/q \rfloor$. À partir de cette approximation, notée k, il suffit alors de calculer $r' = x - qk$. Comme k n'est qu'une approximation du quotient, il sera éventuellement nécessaire d'ajouter un certain nombre de multiples de q pour ramener r' dans l'intervalle $[0, q)$.

Pour obtenir l'approximation de Barrett, on commence par remarquer que l'on peut écrire

$$\lfloor \frac{x}{q} \rfloor = \lfloor \frac{\frac{x}{2^{n+\beta}} \frac{2^{n+\alpha}}{q}}{2^{\alpha-\beta}} \rfloor$$

l'approximation de $\lfloor x/q \rfloor$ proposée par Barrett est

$$k = \lfloor \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}} \rfloor = \lfloor \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \mu}{2^{\alpha-\beta}} \rfloor \tag{5.1}$$

avec $\mu = \lfloor 2^{n+\alpha}/q \rfloor$. L'avantage de cette décomposition, est que μ est une constante, indépendante de x, qui peut donc être pré-calculée si q est connu par avance. De plus, le calcul de k ne fait apparaître que des multiplications ou des divisions par des puissances de deux, des opérations qui peuvent être implémentées efficacement. La proposition suivante nous assure que k ainsi calculé, est une approximation inférieure très proche de $\lfloor x/q \rfloor$.

Proposition 5. *Si $\alpha \geq n+1$ et $\beta \leq -2$, et $0 < x < 2^{2n}$, alors k tel que définit dans l'équation (5.1) vérifie*

$$\lfloor \frac{x}{q} \rfloor \geq k > \lfloor \frac{x}{q} \rfloor - 2$$

Démonstration. La preuve consiste à borner k en utilisant principalement les propriétés de la partie entière. Les calculs étant assez longs, la preuve est donnée en annexe A. □

Remarque 1. La méthode proposée initialement par Barrett correspond au choix de paramètres $\alpha = n$ et $\beta = -1$. Elle ne rentre donc pas dans le cadre de la proposition précédente. En effet, elle approxime plus largement le quotient. Plus précisément, on a dans ce cas $\lfloor \frac{x}{q} \rfloor \geq k > \lfloor \frac{x}{q} \rfloor - 3$. En conséquence, une étape supplémentaire de soustraction par q est nécessaire dans l'algorithme original.

Remarque 2. La proposition 5 n'impose pas que $x < q^2$, mais une borne plus large $x < 2^{2n}$. Autrement dit, on peut réduire des valeurs légèrement supérieures à la sortie d'une multiplication de deux éléments de $\mathbb{Z}_q$.

Nous avons maintenant tous les éléments pour comprendre l'algorithme 5.1, qui effectue une réduction modulaire complète. Cet algorithme est encore plus lisible sous forme de circuit, comme montré sur la figure 5.3. Pour réaliser ce circuit, il faut donc deux multiplicateurs : un premier de dimension $(n-\beta) \times (\alpha+1)$ puis un second de $(n+1) \times n$. Il faut également deux soustracteurs et un multiplexeur. Dans le but de minimiser la taille de la première multiplication, il faut minimiser α et maximiser β, en appliquant les contraintes de la proposition 5, cela se traduit par $\alpha = n+1$ et $b = -2$. Le coût est alors

$$\begin{aligned} c_{barrett} =& c_{mul}((n+2) \times (n+2)) + c_{mul}((n+1) \times n) \\ &+ c_{sub}(2n) + c_{sub}(n+1) + c_{mux}(n)\,. \end{aligned}$$

Les paramètres α et β permettent de trouver un compromis entre la taille des multiplications effectuées et la précision de l'estimation du quotient. La proposition 5 donne les contraintes pour avoir à faire au plus une soustraction par q. En tolérant plus d'étapes de corrections, la dimension des multiplications peut être réduite.

Exemple 2. Calculons avec la méthode de Barrett $31 \mod 5$. Pour représenter $q = 5$, 3 bit sont nécessaires ($n = 3$), on peut ainsi réduire n'importe quelle valeur entre 0 et $63 = 2^6 - 1$. Pour commencer, calculons $\mu = \lfloor 2^7/5 \rfloor = 25$. Ensuite, on calcule :

$$\lfloor \frac{31}{2^1} \rfloor \mu = 15 \times 25 = 375$$

Algorithme 5.1 Algorithme de Barrett généralisé

Entrée : $q \in \mathbb{Z}^*$, n tq. $q < 2^n$, $0 \leq x < 2^{2n}$, $\alpha \geq n+1$ and $\beta \leq -2$

1: $\mu \leftarrow \lfloor 2^{n+\alpha}/q \rfloor$ ▷ Pré-calcul
2: $t \leftarrow x \gg (n+\beta)$
3: $k \leftarrow (t\mu) \gg (\alpha - \beta)$
4: $r' \leftarrow x - kq$
5: **if** $r' > q$ **then**
6: $\quad r' \leftarrow r' - q$
7: **end if**
8: **return** r'

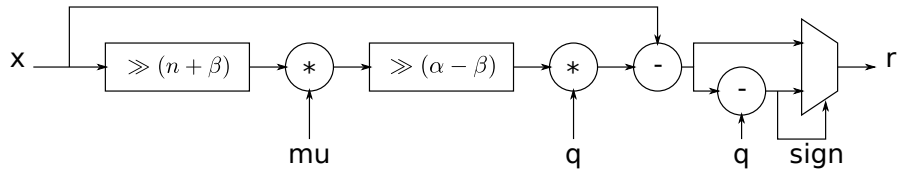


FIGURE 5.3. – Circuit calculant la réduction de Barrett

puis,

$$31 - \lfloor \frac{\lfloor \frac{31}{2^2} \rfloor \mu}{2^6} \rfloor 5 = 31 - 5 \times 5 = 6$$

il reste maintenant les deux dernières étapes de réductions. Ici, il faut seulement soustraire $1 \times q = 5$ pour retomber dans l'intervalle $[0, 5)$, on obtient alors $31 \equiv 1 \mod 5$, comme attendu.

Dans ces conditions, le circuit de réduction peut-être implémenté sous forme d'un pipeline complet. C'est-à-dire, capable d'effectuer en régime établi une réduction par cycle. Un des avantages d'une telle implémentation est que le temps de réduction est constant, les étapes de réduction constituent en effet souvent une vulnérabilité fortement exploitable par des attaques de *timing*.

5.2.3. Réduction de Montgomery

La réduction de Montgomery [Mon85] est une autre méthode bien connue pour la multiplication modulaire (plus que celle de Barrett en réalité). Supposons que l'on sache effectuer efficacement la multiplication modulaire dans $\mathbb{Z}_R$, avec $R > q$ et $q \wedge R = 1$, c'est par exemple le cas quand $R = 2^k$.

Comme q et R sont premiers entre eux, l'application $\varphi : x \to xR$ est un automorphisme de $\mathbb{Z}_q$. Notons $\texttt{red}(z)$, l'opération *réduction de Montgomery*, définie pour $0 \leq z < qR$ par $\texttt{red}(z) = zR^{-1} \mod q$. On remarque que cette opération permet d'évaluer la fonction φ :

$$\begin{cases} \varphi(x) = \texttt{red}(x(R^2 \mod q)) \\ \varphi^{-1}(x) = \texttt{red}(x) \end{cases}$$

Cet automorphisme est homomorphe pour l'addition, en effet

$$\varphi(x) + \varphi(y) = xR + yR = (x+y)R = \varphi(x+y)\,.$$

Cependant, φ n'est pas un homomorphisme multiplicatif. Pour effectuer une multiplication modulaire, l'idée est de calculer d'abord $\varphi(x)$ et $\varphi(y)$, puis de faire le produit de ces deux valeurs. En appliquant ensuite la réduction de Montgomery au résultat, on obtient $\varphi(xy)$. En effet,

$$\texttt{red}(\varphi(x)\varphi(y)) = \varphi(x)\varphi(y)R^{-1} = (xR)(yR)R^{-1} = xyR = \varphi(xy)\,.$$

Une fois toutes les opérations effectuées en représentation de Montgomery, le résultat final est obtenu en calculant φ^{-1}. Tout l'intérêt de la méthode de Montgomery vient du fait que l'opération `red` peut être effectuée très rapidement [Mon85], l'algorithme 5.2 donne une implémentation possible de cette réduction.

Algorithme 5.2 Calcul efficace de la réduction de Montgomery $\texttt{red}(z)$

Entrée : $q \in \mathbb{Z}^*$, $z < qR$

1: $M \leftarrow -q^{-1} \mod R$ ▷ Pré-calcul
2: $k \leftarrow zM \mod R$
3: $r' \leftarrow \lfloor (z + kq)/R \rfloor$
4: **if** $r' > q$ **then**
5: $\quad r' \leftarrow r' - q$
6: **end if**
7: **return** r'

Proposition 6. *L'algorithme 5.2 est correct.*

Démonstration. Pour prouver la validité de cet algorithme, on remarque tout d'abord que $z + kq = 0 \mod R$. Ainsi, la division par R (ligne 3 de l'algorithme) fait sens. En effet,

$$\begin{aligned} z + kq &\equiv z + q(z(-q^{-1})) \mod R \\ &\equiv z - z \mod R \\ &\equiv 0 \mod R \end{aligned}$$

Par conséquent, il existe $u \in \mathbb{Z}$ tel que $z + qk = uR$. En prenant cette équation modulo q, on obtient

$$z \equiv uR \mod q \iff u \equiv zR^{-1} \mod q \tag{5.2}$$

Autrement dit, à un multiple de q près, r' vaut bien $zR^{-1} \mod q$. Or, on a

$$0 \leq z + qk < qR + qR$$

et donc

$$0 \leq \frac{z + qk}{R} < 2q\,.$$

Ceci termine la preuve, car l'équation (5.2) nous dit que $r' = (zR^{-1} \mod q) + vq$ et l'équation ci-dessus que $0 \leq r' < 2q$. Donc $v \in \{0, 1\}$. Ainsi, l'étape de correction des lignes 4 à 6 de l'algorithme permet donc d'obtenir la bonne valeur $zR^{-1} \mod q$. □

Exemple 3. Calculons, avec la méthode de Montgomery $17 \times 13 \mod 19$. Nous allons naturellement prendre $R = 2^5 = 32$, la première puissance de deux supérieure à 19. On pré-calcule alors les constantes $R^2 \mod 19 = 17$ et $-q^{-1} \mod R = 5$. On commence par mettre 17 et 13 sous forme de Montgomery :

— $\varphi(17) = \texttt{red}(17 \times R^2) = 12$
— $\varphi(13) = \texttt{red}(13 \times R^2) = 17$

Ensuite, on calcule $\texttt{red}(\varphi(17) \times \varphi(13)) = \texttt{red}(12 \times 17) = \texttt{red}(204)$, le déroulement de l'algorithme `red` donne :

1. $k \leftarrow 204 \times (-q^{-1}) \mod 32 = 28$
2. $r \leftarrow (204 + 19 \times 28)/32 = 736/32 = 23$
3. on retourne $4 = 23 - 19$ (car $23 > 19$)

Pour terminer, il faut remettre le résultat sous forme normale $\varphi^{-1}(4) = \texttt{red}(4) = 12$. On a ainsi bien retrouvé $17 \times 13 \equiv 12 \mod 19$.

Pour un q donné, nous posons $n = \lceil \lg(q+1) \rceil$, la taille en bits de q, et $R = 2^n$, en supposant bien sûr que q est premier avec R. L'algorithme 5.2 nécessite alors deux multiplications $n \times n$ (ligne 2 et ligne 3), une soustraction $2n \times 2n$ et une soustraction $(n+1) \times n$. Aussi le coût total est

$$c_{mont} = 2c_{mul}(n) + c_{sub}(2n) + c_{sub}(n+1) + c_{mux}(n)\,.$$

5.2.4. Comparaison

Par rapport à la méthode de Barrett, la méthode de Montgomery utilise des multiplications de tailles légèrement inférieures. Par conséquent, cette méthode est plus compacte en termes de surface. Cependant, il n'est pas dit que cette différence soit visible sur un FPGA, car les multiplications de grands nombres sont découpées en plus petites multiplications (de taille 18×18 par exemple pour les FPGAs que nous avons utilisés).

En contrepartie, il est nécessaire de convertir les entrées sous forme de Montgomery ce qui ajoute de la complexité de conception ainsi qu'une latence supplémentaire aux calculs.

5.3. Multiplication polynomiale

Une des opérations particulièrement critiques en termes de performances est la multiplication polynomiale. Les degrés considérés sont plutôt élevés, supérieurs à 2048, aussi la méthode de multiplication «à la main» (*shift-add)*, que nous avions utilisée pour effectuer la multiplication dans $\mathbb{F}_{2^8}$ (Algorithme 4.4) ne passe clairement pas à l'échelle. Si cette méthode requiert très peu de ressources, son temps d'exécution est quadratique en fonction du degré $O(m^2)$.

Ceci motive l'exploration de stratégies plus efficaces pour la multiplication. Le sujet est évidemment extrêmement riche et sujet à de nombreux travaux de recherche. Parmi les méthodes plus efficaces asymptotiquement, on pourra mentionner celle de Karatsuba [KO62], qui a une complexité en $O(m^{\ln 3/\ln 2})$, ou encore celle de Toom-Cook [Coo66].

Pour les degrés élevés, l'approche offrant la meilleure complexité asymptotique est la *Number Theoric Transform* (NTT) [Pol71], qui est l'analogue de la transformée de Fourier rapide (FTT) sur un corps fini. La NTT est un algorithme permettant d'évaluer efficacement un polynôme aux puissances ζ_m^i, pour $i \in [0, m-1]$, avec ζ_m une racine m-ième primitive de l'unité, à condition qu'il en existe dans $\mathbb{Z}_q$. Il est important de noter que cette représentation n'est pas la même que la représentation sous forme d'évaluation évoquée dans la partie précédente 4.4.1, qui était l'évaluation sur ζ_m^i pour $i \in \mathbb{Z}_m^*$. Aussi, les opérations point à point ne sont pas équivalentes, mais nous verrons dans la section suivante qu'il est possible de les rendre équivalentes.

La NTT est d'autant plus avantageuse pour l'évaluation homomorphe, qu'un grand nombre d'opérations peuvent être effectuées en représentation d'évaluation. Aussi, par rapport à une multiplication brute de polynômes, le coût de la NTT est amorti par plusieurs opérations intermédiaires.

5.3.1. Multiplication dans R_q basée sur la transformée de Fourier

Définition 6. La convolution discrète de deux séquences $(a_i)_{0\leq i<n}$ et $(b_i)_{0\leq i<n}$ est la séquence de longueur $2n-2$, $(\tilde{z}_i)_{0\leq i\leq 2n-2}$ définie par :

$$\tilde{z}_k = \sum_{i+j=k} a_i b_j$$

La convolution circulaire de (a_i) et (b_i) est définie par $z_i = \tilde{z}_i + \tilde{z}_{i+n}$. De manière analogue, la convolution anti-circulaire (*negacyclic*) est définie par $z_i = z_i - z_{i+n}$. Ces convolutions correspondent respectivement à une multiplication polynomiale dans $R[X]/X^n - 1$ et $R[X]/X^n + 1$. On comprend donc que cette dernière est utilisable pour calculer un produit d'éléments homomorphes. Une propriété fondamentale de la NTT est qu'elle permet d'évaluer efficacement une convolution circulaire ou anti-circulaire.

Théorème 1 (Théorème de convolution)**.** *Pour deux séquences a et b de longueur n, on a*

$$a * b = \mathtt{NTT}^{-1}(\mathtt{NTT}(a) \odot \mathtt{NTT}(b))$$

Démonstration. Nous allons prouver l'égalité coefficient par coefficient. On note $\texttt{NTT}_k$ le k-ième coefficient de la NTT et ζ_n une racine n-ième primitive de l'unité. Le k-ième coefficient du produit des deux NTT vaut :

$$\begin{aligned}
\texttt{NTT}_k(a)\texttt{NTT}_k(b) &= (\sum_{i=0}^{n} a_i\zeta_n^{ik})(\sum_{i=0}^{n} b_i\zeta_n^{ik}) \\
&= \sum_{i=0}^{2n-2}(\sum_{i0+i1=i} a_{i0}b_{i1})\zeta_n^{ik} \\
&= \sum_{i=0}^{n-1}(\sum_{i0+i1=i} a_{i0}b_{i1})\zeta_n^{ik} + \sum_{i=n}^{2n-2}(\sum_{i0+i1=i} a_{i0}b_{i1})\zeta_n^{ik} \\
&= \sum_{i=0}^{n-1}\tilde{z}_i\zeta_n^{ik} + \sum_{i=0}^{n-1}\tilde{z}_{i+n}\zeta_n\zeta_n^{ik} \\
&= \sum_{i=0}^{n-1}(\tilde{z}_i + \tilde{z}_{i+n})\zeta_n^{ik} \\
&= \texttt{NTT}_k(a * b)
\end{aligned}$$

Ainsi, en prenant la NTT inverse on obtient bien le résultat souhaité. □

Pour obtenir une convolution anti-cyclique, une méthode astucieuse est d'évaluer les entrées a et b à ζ_{2n}, puis le résultat ζ_{2n}^{-1} à la fin. En reprenant le calcul de la preuve précédente, on obtient :

$$\begin{aligned}
\texttt{NTT}_k(a(\zeta_{2n}))\texttt{NTT}_k(b(\zeta_{2n})) &= \sum_{i=0}^{n-1}(\sum_{i0+i1=i} a_{i0}b_{i1})\zeta_{2n}^{i}\zeta_n^{ik} + \sum_{i=n}^{2n-2}(\sum_{i0+i1=i} a_{i0}b_{i1})\zeta_{2n}^{i}\zeta_n^{ik} \\
&= \sum_{i=0}^{n-1}\tilde{z}_i\zeta_{2n}^{i}\zeta_n^{ik} + \zeta_{2n}^{n}\sum_{i=0}^{n-1}\tilde{z}_{i+n}\zeta_{2n}^{i}\zeta_n^{ik} \\
&= \sum_{i=0}^{n-1}(\tilde{z}_i - \tilde{z}_{i+n})\zeta_{2n}^{i}\zeta_n^{ik} \\
&= \texttt{NTT}_k((a * b)(\zeta_{2n}^{i}))
\end{aligned}$$

Pour passer de la deuxième à la troisième ligne, on remarque que $\zeta_{2n}^{n} = \zeta_2 = -1$.

On peut donc, avec un faible surcoût encapsuler une NTT pour obtenir une vraie représentation d'évaluation. Qui plus est, l'opération $\texttt{Eval}_{\zeta_{2m}} : X \to \zeta_{2m}X$ est une opération efficace. Concrètement, il s'agit du produit point à point avec le polynôme dont les coefficients sont $(\zeta_{2m}^{0}, \zeta_{2m}^{1} \ldots, \zeta_{2m}^{m-1})$.

La figure 5.4 montre un circuit permettant de calculer un produit d'éléments dans R_q en utilisant cette approche. Sur cet exemple une seule opération point à point est effectuée (un produit). Mais il est possible et souhaitable d'effectuer plusieurs opérations sur cette représentation d'évaluation.

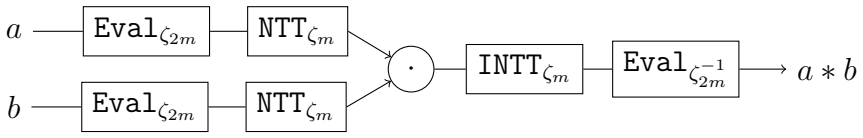


FIGURE 5.4. – Évaluation d'un multiplication dans $\mathbb{Z}_q[X]/X^m + 1$ basée sur la NTT

Pour cette raison, dans notre implémentation, tout élément chiffré appartenant à l'anneau R_q est représenté et stocké sous forme d'évaluation. Autrement dit, les opérations effectuées lors du chiffrement, déchiffrement et évaluation homomorphes sont toutes effectuées point à point. Cela permet d'amortir les coûts des transformations entre la représentation polynomiale et d'évaluation. Elles sont effectuées seulement au moment du chiffrement et du chiffrement.

Remarque 3. Si l'on utilise la méthode de Montgomery pour la réduction modulaire, la transformation des éléments sous forme de Montgomery peut être incluse dans l'opération `Eval` et n'a donc aucun surcoût. Le i-ème coefficient de `Eval` devient alors $a_i \to \texttt{red}(a_i, R^2\zeta_{2m}^i)$, où la partie droite peut être entièrement pré-calculée.

5.3.2. Calcul efficace de la NTT

Nous abordons à présent l'implémentation efficace de la NTT. Lorsque la dimension m est composée ($m = m_0 m_1$), l'algorithme de Cooley-Tukey [CT65] permet une approche « diviser pour régner » au problème. Un cas particulièrement favorable est quand m est une puissance de deux, où la transformation peut s'effectuer de manière très efficace et structurée. L'algorithme 5.3, adapté de [Cor09] (partie 30.3) effectue une NTT en place de rayon deux.

Algorithme 5.3 NTT itérative, en place, de rayon 2

Entrée : X, un vecteur de taille n, où n est une puissance de deux. ζ_n, une racine n-ième de l'unité dans $\mathbb{Z}_q$.

1: $A \leftarrow \texttt{reverse\_copy}(X)$
2: **for** $s = 1$ to k **do**
3: $\quad m \leftarrow 2^s$
4: $\quad w \leftarrow 1$
5: $\quad$ **for** $j = 0$ to $m/2$ **do**
6: $\quad\quad$ **for** $k = 0$ to $n - 1$ by m **do**
7: $\quad\quad\quad a,\ b \leftarrow A[j + k],\ A[j + k + m/2]$
8: $\quad\quad\quad A[j + k] \leftarrow a + bw$
9: $\quad\quad\quad A[j + k + m/2] \leftarrow a - bw$
10: $\quad\quad$ **end for**
11: $\quad\quad w \leftarrow w * \zeta_m$
12: $\quad$ **end for**
13: **end for**

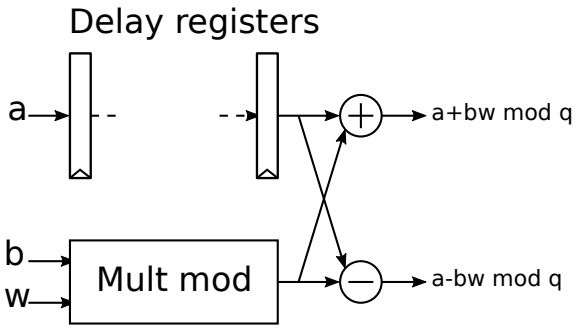


FIGURE 5.5. – Opération *butterfly* complètement pipelinée.

L'opération de base de cet algorithme est appelée le *butterfly*, effectué des lignes 7 à 9. Il est assez immédiat de réaliser un opérateur *butterfly* matériel très efficace (voir figure 5.5). Ce circuit est capable de consommer deux entrées et produire deux sorties par cycles d'horloge. Pour exploiter complètement ce potentiel de calcul, il faut pouvoir gérer deux lectures et écritures en mémoire par cycle d'horloge.

5.3.2.1. Architecture

L'accélérateur matériel pour la NTT est représenté figure 5.6. Il possède une mémoire interne, constituée de deux mémoires RAM mises en parallèle, toutes deux de dimension $m/2$. Cette séparation permet de supporter deux lectures et écritures par cycle d'horloge. Le chemin de données est un pipeline composé de 4 parties distinctes :

Fetch. Lit la commande à exécuter dans le registre d'instruction. Deux sources différentes sont possibles pour l'instruction :

1. depuis une mémoire d'instruction (*Instruction ROM*). Un registre d'adresse (PC) est associé à cette mémoire et incrémenté linéairement. Cette mémoire contient les instructions de chargement et déchargement des coefficients, qui sont exécutées en début et fin de NTT.
2. Une seconde source est le *Pattern Generator*, qui génère les différents indices (s, k, j) de la boucle 5.3.

Read. À partir du registre d'instruction, cette étape lit les différentes opérandes depuis les bancs mémoires ainsi que la mémoire *Twiddle ROM* qui contient les valeurs pré-calculées w^i.

Execute. Effectue l'opération *butterfly* (figure 5.5) puis permute le résultat (ce composant sera présenté dans la section 5.3.2.2). Lors de la dernière phase de la NTT, le résultat du *butterfly* est écrit directement en sortie. L'étage *execute* est évidement un pipeline de plusieurs cycles.

Writeback. Est chargé de l'écriture dans les mémoires. La donnée écrite peut être soit le résultat venant de l'étage *execute* ou alors directement depuis l'extérieur lors du chargement des coefficients.

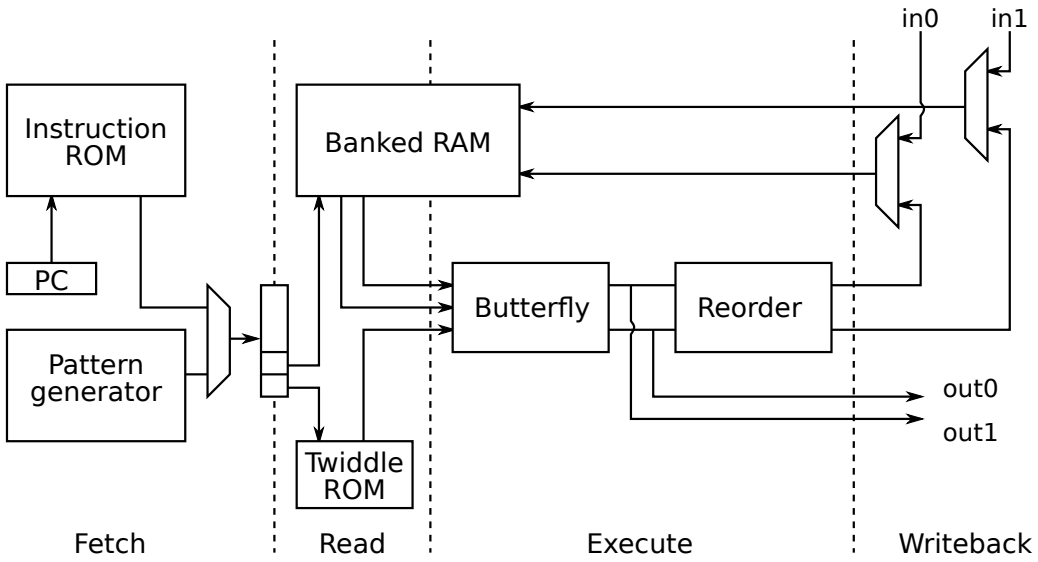


FIGURE 5.6. – Architecture pour la NTT

Banc mémoire	$j,\ k$	$j,\ k+m$
0	$A[j+k]$	$A[j+k+m]$
1	$A[j+k+m/2]$	$A[j+k+3m/2]$

(a) À l'étape s

Banc mémoire	$j,\ k$	$j+m/2,\ k$
0	$A[j+k]$	$A[j+k+m/2]$
1	$A[j+k+m]$	$A[j+k+3m/2]$

(b) À l'étape $s+1$

FIGURE 5.7. – Accès mémoires effectués à deux étapes consécutives d'une NTT

5.3.2.2. Organisation des accès mémoires

La structuration de la mémoire en plusieurs bancs impose l'invariant suivant : à chaque passe de l'algorithme 5.3 (la boucle principale sur s) les valeurs a et b doivent être localisées dans des bancs différents. L'analyse des accès mémoire figure 5.7 montre que cet invariant n'est pas satisfait pour l'algorithme 5.3. Par exemple $A[j+k]$, $A[j+k+m/2]$, sont dans le banc 0 et 1 à l'étape s, mais doivent tous deux être lus depuis le banc 0 dans l'étape $s+1$.

Pour résoudre ce conflit, il est suffisant de permuter la sortie de deux *butterfly* successifs pour maintenir cet invariant. On échange donc les valeurs $A[j+k+m/2]$ et $A[j+k+m]$ au cours de deux itérations successives. Cette opération s'effectue naturellement à l'aide de registres à décalage.

5.3.3. Performances, limites et améliorations possibles

Cette architecture a l'avantage d'être relativement légère et plutôt efficace. Ce composant nécessite seulement 3 mémoires de taille $m/2$. Pour ce qui est du temps de calcul, il est divisé en trois phases :

— le chargement des coefficients : m cycles (l'opération `reverse_copy`)
— Les étapes intermédiaires de calcul (excepté la dernière), il y en a en tout $\lg m - 1$, chacune d'elles nécessitant $m/2$ cycles.
— La dernière étape de l'algorithme, qui après le calcul émet le résultat sur les ports de sorties. Cette étape peut normalement être effectuée en $m/2$ cycles. Cependant, ceci présuppose que l'interface de sortie accepte deux données par cycles. Cette étape prend en réalité m cycles dans l'architecture complète (figure 5.1), qui par conception ne supporte qu'un coefficient par cycle sur les canaux de communication.

Le temps total de calcul attendu est approximativement de

$$t_{NTT}(m) = m \times (\frac{\lg m - 1}{2} + 2)\,. \tag{5.3}$$

Le temps sera en pratique légèrement supérieur, car une latence de quelques cycles est nécessaire pour l'initialisation des pipelines.

Cependant, une telle architecture n'est pas adaptée aux grandes dimensions. En effet, la mémoire interne doit être rapide, de type SRAM par exemple et n'est pas adaptée au stockage de gros volumes de données. En revanche, pour effectuer des NTT de plus grandes dimensions, le calcul peut-être décomposé en NTT de plus petite dimension.

Les principales améliorations possibles de cette architecture sont :

— Réduire le temps de chargement, l'opération *reverse copy* empêche d'exploiter complètement toute la bande passante des mémoires (2 données par cycles). Une réduction allant jusqu'à $m/2$ cycles est envisageable en réordonnant les accès.
— Augmenter le rayon de la NTT, qui a pour effet de réduire le temps des calculs intermédiaires. En contrepartie, cela augmentera la logique nécessaire, car plusieurs opérations *butterfly* seront nécessaires. Cette stratégie est celle qui aura le plus d'impact sur les performances.

5.4. Échantillonnage de gaussiennes discrètes

L'échantillonnage de gaussiennes discrètes est une partie importante de la procédure de chiffrement, deux échantillons (des polynômes) différents sont nécessaires, nous les avions appelés e_0 et e_1 dans la section 4.4.2. Dans un cas d'utilisation normal, le chiffrement est utilisé peu fréquemment par rapport aux évaluations homomorphes. À l'opposé, avec notre approche, le chiffrement est au cœur du processus d'évaluation. C'est pourquoi il est essentiel de concevoir un circuit, avec la plus faible empreinte

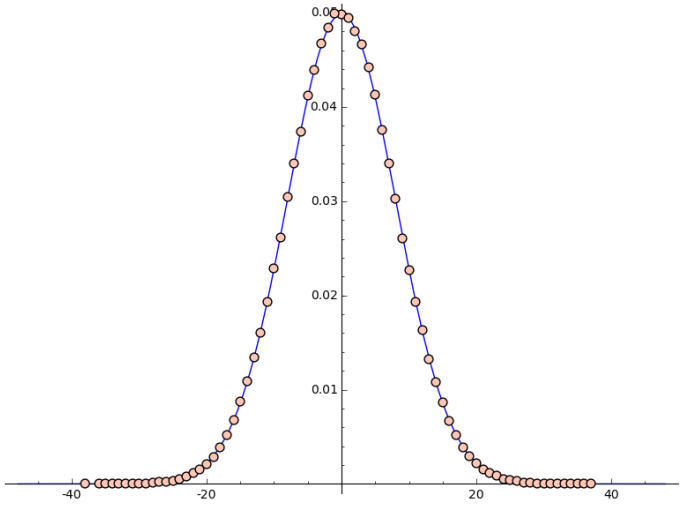


FIGURE 5.8. – Gaussienne discrète sur un intervalle fini

matérielle possible capable de fournir des échantillons à un rythme suffisant pour ne pas ralentir le reste du circuit.

Une gaussienne discrète est une distribution normale continue restreinte et normalisée sur un ensemble dénombrable (voir figure 5.8). Pour une moyenne μ et une variance σ^2, la densité de probabilité d'une loi normale est définie par

$$\rho_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} .$$

La distribution sur un ensemble dénombrable Λ est alors caractérisée par la densité de probabilité suivante pour $x \in \Lambda$:

$$\rho_{\mu,\sigma,\Lambda}(x) = \frac{\rho_{\mu,\sigma^2}(x)}{\sum_{k\in\Lambda} \rho_{\mu,\sigma^2}(k)} .$$

Au même titre qu'une loi normale, cette distribution est fortement bornée autour de sa moyenne. En particulier, en adaptant la borne établie dans le lemme 4.4 de [Lyu11] (version longue) on a pour $n > 1$:

$$\Pr_{x\leftarrow\mathcal{DG}(\sigma^2)}(|x| > n\sigma) < e^{(1-n^2)/2}$$

Par exemple, la probabilité d'avoir un échantillon plus grand que 12σ est inférieure à 2^{-103}. Cette propriété est fondamentale, elle nous permet d'échantillonner une approximation la distribution $\mathcal{DG}(\sigma^2)$.

En effet, échantillonner directement selon une gaussienne discrète pose plusieurs problèmes pratiques :

— les entiers manipulés sont finis, alors que la distribution $\mathcal{DG}(\sigma^2)$ a l'ensemble des entiers relatifs comme support.

— La précision des calculs est bornée (on ne peut pas approximer parfaitement ρ).

Soit D'_σ la distribution gaussienne discrète restreinte sur l'ensemble $B = [-n\sigma, n\sigma]$, c'est-à-dire définie par la densité de probabilité $\rho_{\sigma,B}(k)$. Enfin, on introduit D''_σ , une version approximée de la distribution D'_σ sur $\mathbb{Q}$, avec une précision ϵ. La distribution D''_σ répond aux deux problèmes énoncés ci-dessus, elle est définie sur un ensemble fini et approximée avec une précision fixée. La propriété suivante, établie une borne permettant de choisir les paramètres n et ϵ.

Proposition 7. *Soit $e(n) = e^{(1-n^2)/2}$. La distance statistique entre D''_σ et D_σ est bornée par*

$$\Delta(D''_\sigma, D_\sigma) < n\sigma\epsilon + e(n)$$

Démonstration. On commence par borner $\Delta(D_\sigma, D'_\sigma)$, puis $\Delta(D'_\sigma, D''_\sigma)$. On terminera alors la preuve par une inégalité triangulaire $\Delta(D''_\sigma, D_\sigma) \leq \Delta(D_\sigma, D'_\sigma) + \Delta(D'_\sigma, D''_\sigma)$. D'après la définition de la distance statistique :

$$\begin{aligned}\Delta(D_\sigma, D'_\sigma) &= \frac{1}{2}\sum_{k\in\mathbb{Z}} |\rho_{\sigma,B}(k) - \rho_{\sigma,\mathbb{Z}}(k)| \\ &= \frac{1}{2}\sum_{k\in B} |\rho_{\sigma,B}(k) - \rho_{\sigma,\mathbb{Z}}(k)| + \frac{1}{2}\sum_{k\in\mathbb{Z}\setminus B} \rho_{\sigma,\mathbb{Z}}(k)\end{aligned}$$

Pour borner le terme de droite ci-dessus, on a

$$\begin{aligned}\sum_{k\in\mathbb{Z}\setminus B} \rho_{\sigma,\mathbb{Z}}(k) &= \Pr(|X| > n\sigma) \\ &< e^{(1-n^2)/2} = e(n)\end{aligned}$$

Exprimons maintenant $\rho_{\sigma,B}(k)$ en fonction de $\rho_{\sigma,\mathbb{Z}}(k)$.

$$\begin{aligned}\rho_{\sigma,B}(k) &= \frac{\rho_\sigma(k)}{\rho_\sigma(B)} \\ &= \frac{\rho_\sigma(k)}{\rho_\sigma(\mathbb{Z}) - \rho_\sigma(\mathbb{Z}\setminus B)} \\ &= \frac{\rho_\sigma(k)}{\rho_\sigma(\mathbb{Z})}\left(\frac{1}{1 - \frac{\rho_\sigma(\mathbb{Z}\setminus B)}{\rho_\sigma(\mathbb{Z})}}\right) \\ &= \rho_{\sigma,\mathbb{Z}}(k)\left(\frac{1}{1 - \rho_{\sigma,\mathbb{Z}}(\mathbb{Z}\setminus B)}\right) \\ &= \rho_{\sigma,\mathbb{Z}}(k)\left(\frac{\rho_{\sigma,\mathbb{Z}}(\mathbb{Z}\setminus B)}{1 - \rho_{\sigma,\mathbb{Z}}(\mathbb{Z}\setminus B)} + 1\right)\end{aligned}$$

En reprenant la première équation,

$$\begin{aligned}
\Delta(D_\sigma,\, D'_\sigma) &< \frac{1}{2}\sum_{k\in B} |\rho_{\sigma,\,B}(k) - \rho_{\sigma,\,\mathbb{Z}}(k)| + \frac{e^{(1-n^2)/2}}{2} \\
&= \frac{1}{2}\sum_{k\in B} |\rho_{\sigma,\,\mathbb{Z}}(k)\frac{\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B)}{1-\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B)}| + \frac{e(n)}{2} \\
&= \frac{1}{2}\rho_{\sigma,\,\mathbb{Z}}(B)\frac{\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B)}{1-\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B)} + \frac{e(n)}{2} \\
&= \frac{1}{2}(1-\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B))\frac{\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B)}{1-\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B)} + \frac{e(n)}{2} \\
&= \frac{1}{2}\rho_{\sigma,\,\mathbb{Z}}(\mathbb{Z}\setminus B) + \frac{e(n)}{2} \\
&< e(n)
\end{aligned}$$

Enfin, si l'on a une version approximée D''_σ de la distribution D'_σ sur $\mathbb{Q}$, avec une précision ϵ, alors

$$\Delta(D''_\sigma,\, D'_\sigma) = \frac{1}{2}\sum_{k\in B} |\rho''_\sigma(k) - \rho'_\sigma(k)| < n\sigma\epsilon$$

Au final en combinant les deux inégalités, on obtient

$$\Delta(D''_\sigma,\, D_\sigma) < n\sigma\epsilon + e(n)$$

□

5.4.1. Choix de paramètres

La proposition 7 nous prouve que D''_σ est une approximation très proche de D_σ au sens de la distance statistique Δ. Maintenant, si l'on utilise D''_σ à la place de D_σ, quel est l'impact sur la sécurité du schéma ? Il est montré dans [MW17, lemme 3.1], que si le schéma a une sécurité de λ bits et $\Delta(D''_\sigma,\, D_\sigma) < 2^{-\lambda}$, alors le schéma utilisant D''_σ a une sécurité de $\lambda - 1$ bits. Dans ce même article, les auteurs suggèrent l'utilisation d'une autre distance, avec laquelle seulement $d(D''_\sigma,\, D_\sigma) < \lambda/2$ est nécessaire pour maintenir la sécurité de λ bits. La distance proposée pour d est la distance *max-log* $\Delta_{ML}(D',\, D) = \max_{x\in S} |\ln \rho'(x) - \ln \rho(x)|$. Cette contrainte moins forte a pour effet direct de réduire la précision nécessaire à l'échantillonnage, mais nécessite des calculs avec erreur relative bornée.

Revenons à notre cas, pour garantir une distance statistique inférieure à 2^{-128} et maintenir ainsi le niveau de sécurité du schéma quand $\sigma = 3.1$, on pourra prendre par exemple $\epsilon = 134$ et $n = 14$.

5.4.2. Méthodes d'échantillonnage

Plusieurs méthodes d'échantillonnage de gaussiennes discrètes sont étudiées et comparées dans [DG14]. Nous sommes intéressés par trois critères : la surface, le temps

d'échantillonnage et l'utilisation en quantité d'aléa pour générer un échantillon.

5.4.2.1. Échantillonnage par rejet

L'approche la plus simple est l'échantillonnage par rejet, cette méthode étant applicable pour n'importe quelle distribution. Tout d'abord on tire aléatoirement un élément dans l'intervalle B. Soit k l'élément obtenu, on calcule alors $p = \rho(k)$ et l'on accepte la valeur k avec une probabilité p, sinon k est rejeté et l'échantillonnage est réitéré. Cette méthode requiert de calculer à chaque échantillonnage une valeur $\rho(k)$. Pour éviter ce calcul, il est envisageable de pré-calculer toutes les valeurs et les stocker dans une table.

5.4.2.2. Méthode d'inversion

Une autre méthode couramment utilisée pour échantillonner des distributions aléatoires est la méthode d'inversion. Une distribution uniforme sur l'intervalle [0, 1] est directement transformée vers la distribution souhaitée.

On note F la densité de probabilité cumulée. L'algorithme 5.4 donne la procédure d'échantillonnage. Nous allons maintenant vérifier la validité de cet algorithme. Soit K la variable aléatoire obtenue en sortie de l'échantillonneur et K' une gaussienne discrète, on voit que :

$$\begin{aligned}\Pr(K = k) &= \Pr_{x \leftarrow \mathcal{U}(0,1)}(F(k-1) < x \leq F(k)) \\ &= F(k) - F(k-1) \\ &= \Pr(K' \leq k) - \Pr(K' \leq k-1) \\ &= \Pr(k-1 < K' \leq k) \\ &= \Pr(K' = k)\end{aligned}$$

Algorithme 5.4 Échantillonnage par inversion

1: $x \leftarrow \mathcal{U}(0,\ 1)$
2: Rechercher k tel que $F(k-1) < x \leq F(k)$
3: Retourner k

Le terme «rechercher k» laisse volontairement le flou sur la stratégie à adopter. La fonction F est par construction strictement croissante. Il est ainsi possible de rechercher linéairement, ou bien par une recherche dichotomique. Pour implémenter encore plus efficacement l'algorithme 5.4, les valeurs $F(k)$ peuvent même être pré-calculées et stockées dans une table. La taille du stockage sera alors proportionnelle au nombre de valeurs de la distribution finale, par exemple 12σ. On comprend que cette méthode n'est pas adaptée à l'échantillonnage de distributions avec une variance élevée.

Une réduction du stockage est possible lorsque la distribution est symétrique en 0 (ce qui est le cas des distributions considérées). En effet, on peut n'échantillonner que

la partie positive de la distribution et utiliser un bit aléatoire pour décider du signe de la sortie.

5.4.3. Échantillonnage pour la procédure de chiffrement

Il est essentiel pour ne pas ralentir le chiffrement de fournir les échantillons à un rythme assez rapide. Cependant, les échantillonneurs peuvent présenter un temps d'échantillonnage variable et potentiellement important.

Nous avions initialement cherché à concevoir un échantillonneur produisant un échantillon par cycle d'horloge. L'approche utilisée avait été de pipeliner complément un échantillonnage par inversion. La recherche linéaire dans la table CDT pouvant être déroulée statiquement, on peut ainsi créer un pipeline où l'étage i effectue la comparaison avec le i-ième élément de la table. Le circuit résultant est complètement déterministe et produit un échantillon par cycle d'horloge. Cependant, cette approche s'est avérée beaucoup trop gourmande en ressources [1].

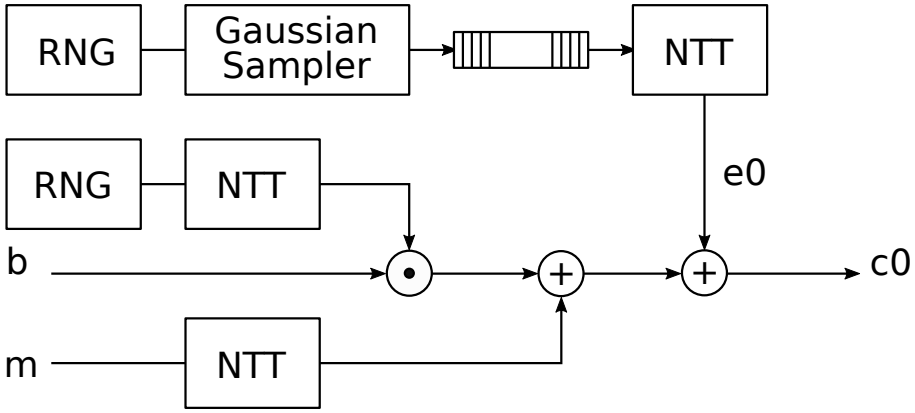


FIGURE 5.9. – Flot de données pour le chiffrement, ici seule la partie qui calcule $c_0 = b \cdot s + pe_0 + m$ est représentée

Cet objectif d'un échantillon par cycle n'est en réalité pas nécessaire lorsque l'on considère le chiffrement dans son ensemble. En supposant une réalisation en flot de données comme représentée sur la figure 5.9, c'est la procédure de NTT qui limite le débit [2]. Le temps pris par ce bloc est donné dans notre architecture par t_{NTT} (équation 5.3, environ 14K cycles pour $m = 2048$). Pour cette raison, les échantillons peuvent être calculés en avance et placés dans une file (figure 5.9) pour être consommés plus tard. Ceci permet d'étaler la procédure d'échantillonnage pendant le calcul de la NTT. Idéalement, il faut que l'échantillonnage prenne (en moyenne) un temps inférieur à la NTT.

1. Rien que pour $\epsilon = 64$ (un peu faible pour la sécurité), le circuit est 2.5× plus important (1881 ALMs) que celui finalement utilisé qui offre une précision largement supérieure ($\epsilon = 134$).
2. Cette observation est confirmée dans l'implémentation présentée dans le chapitre suivant.

Ces observations nous ont conduit à utiliser une méthode d'inversion avec une recherche linéaire effectuée par une machine à états finis. Le circuit résultant a une empreinte matérielle très faible. L'échantillonnage a un temps variable, car la recherche s'arrête dès que le premier indice valide est trouvé. La file utilisée entre l'échantillonneur et la NTT est de taille supérieure à m, de sorte qu'un polynôme entier puisse être échantillonné pendant le calcul de NTT.

Temps d'exécution

Nous nous intéressons maintenant au temps pour obtenir un échantillon (nous noterons T la variable aléatoire associée). Une fois les paramètres fixés, nous pouvons calculer exactement le temps moyen pris par l'échantillonnage à partir de la CDT. En effet, la probabilité d'obtenir un temps $t+1$ [3] est directement la probabilité que l'indice de l'échantillon soit t dans la CDT. On obtient ainsi pour les paramètres $m = 2048$, $\sigma = 3.1$, $\epsilon = 134$ et $n = 14$:

$$\begin{cases} E[T] = 4.45 \\ Var[T] = 3.47 \end{cases}$$

Le temps d'échantillonnage d'un polynôme complet est la somme de m variables T. Encore une fois, le théorème central limite nous suggère que la distribution résultante est proche d'une distribution normale de paramètres $\mu = mE[T]$ et $\sigma^2 = mVar[T]$. Dans ce cas, $\mu = 9117.4$ et $\sigma = 157.17$. Autrement dit, le temps d'exécution moyen est approximativement de 9117.4 cycles. De plus, la marge avec le temps de calcul de la NTT est suffisamment large par rapport à l'écart type. Ainsi, la probabilité que l'échantillonnage prenne plus de temps que la NTT est négligeable. En effet, on a

$$\begin{aligned} t_{NTT} - \mu &= 13312 - 9117.4 \\ &= 4194.6 \\ &\approx 26\sigma\,. \end{aligned}$$

5.5. Conclusion

Le schéma BGV et plus généralement la cryptographie basée sur les réseaux euclidiens peuvent bénéficier de gains importants en temps ou consommation par l'accélération matérielle. Dans ce chapitre, nous avons présenté en détail l'architecture d'un coprocesseur de calculs homomorphes. Ce composant a une structure très similaire à un processeur traditionnel. Il stocke dans des mémoires locales un petit nombre d'éléments chiffrés, l'équivalent des registres pour un processeur. Ce cœur ne peut évaluer que les opérations de base du chiffrement homomorphe. Après chaque opération homomorphe, la fonction de rechiffrement est appliquée au résultat, de sorte à maintenir un niveau de bruit limité dans les messages chiffrés. Cette architecture est

3. Un cycle supplémentaire est nécessaire pour déterminer le signe du résultat

clairement rendue possible par l'utilisation d'un jeu de paramètres relativement petit, qui permet aux données intermédiaires de résider entièrement dans des mémoires locales.

L'architecture est composée de différentes briques matérielles assemblées selon un modèle *dataflow*. Nous avons présenté les choix et la conception des briques essentielles. Au plus bas niveau d'abstraction, la multiplication modulaire est utilisée à de nombreux endroits et peut être effectuée efficacement avec des méthodes utilisant des pré-calculs. La procédure de NTT est un élément clé de l'architecture, elle permet de transformer les polynômes dans une représentation d'évaluation où les opérations d'addition et de multiplication sont effectuées coefficient à coefficient. Enfin, l'échantillonnage de gaussiennes discrètes, est une étape essentielle de la procédure de chiffrement, nous avons ici utilisé un échantillonnage étalé sur le temps de calcul d'une NTT entière pour réduire la taille du circuit.

Validation et Évaluation

6

Sommaire

Ce dernier chapitre est consacré à la mise en place, la validation et l'évaluation des solutions proposées sur des composants logiques programmables. Une première section donne une vue d'ensemble de ces composants dans le but de mieux interpréter les résultats. Nous présentons ensuite le processeur utilisé et auquel sont ajoutées les différentes extensions de sécurité. Enfin, nous discuterons de l'intégration des différentes solutions et l'effet sur le circuit. Dans un premier temps la solution de chiffrement de code, puis, dans un second temps l'ajout du coprocesseur de calculs homomorphes.

6.1. Composant logique programmable (FPGA)

Les *Field Programmable Gate Arrays* (FPGAs) sont des composants sur lesquels il est possible de programmer un circuit. Ils se placent en termes de flexibilité à mi-chemin entre les circuits complètement dédiés (ASIC), conçus pour des fonctionnalités très spécifiques et les processeurs beaucoup plus génériques. D'un côté, par rapport à un ASIC, un FPGA est reconfigurable et offre plus de flexibilité dans les fonctionnalités. Selon l'étude menée en 2007 dans [KR07], un circuit sur FPGA verrait ses performances diminuer d'un facteur 3 à 4, pour une consommation dynamique multipliée par 12 par rapport à un équivalent ASIC. D'autre part, en comparaison aux

processeurs les FPGAs permettent de s'adapter de manière plus flexible aux calculs et d'obtenir ainsi des performances et une consommation meilleures.

Les FPGAs sont des outils essentiels pour le prototypage et la validation des circuits. Grâce à leurs performances très correctes, ils permettent d'avoir tôt dans le cycle de développement un circuit fonctionnel permettant d'obtenir des résultats qui seraient beaucoup trop longs à obtenir en simulation. Pour les faibles quantités de circuits, les FPGAs offrent même une alternative avantageuse aux ASICs. En effet, le processus de développement pour cible FPGA est beaucoup plus simple, les coûts et les temps sont largement réduits. Enfin, ils servent également comme accélérateurs très flexibles pour les calculs. Les fournisseurs de calcul *cloud* (EC2 chez Amazon, Azure chez Windows) proposent aujourd'hui des instances équipées de FPGAs. Les paradigmes de programmation évoluent et permettent avec des langages comme OpenCL et les outils de synthèse haut niveau de déporter des calculs sur FPGA sans avoir à décrire directement un circuit.

6.1.1. Architecture générale

De manière générale, les FPGAs sont constitués d'une matrice hétérogène de blocs logiques configurables (voir figure 6.1). Les principaux éléments sont des cellules d'entrées-sorties, des blocs de logique (combinatoire et registres), de la mémoire sur puce ou encore des multiplieurs. Tous ces blocs sont reliés par un réseau d'interconnexion, lui aussi reconfigurable. Pour programmer le FPGA, tous ces blocs sont configurés par une mémoire. Les technologies les plus courantes sont les SRAMs (utilisées dans les FPGAs Xilinx et Intel/Altera), FLASH et également les anti-fusibles (cette dernière technologie ne permet de programmer le circuit qu'une seule fois). La configuration complète du circuit, qui est écrite dans ces mémoires pour programmer le circuit, est appelée un *bitstream*.

6.1.2. La Famille de FPGAs Arria V

Nous détaillons ici plus précisément l'architecture de la famille des FPGAs Arria V de l'entreprise Altera (rachetée par Intel en fin 2015). Il s'agit des FPGAs utilisés pour l'évaluation et la validation des travaux de cette thèse. Aussi, l'objectif de cette partie est de donner les éléments permettant de mieux comprendre les résultats. Cette partie se base en grande sur la documentation officielle [Int16] à laquelle le lecteur pourra se référer pour plus de détails.

L'élément de logique configurable de base des Arria V [1] est appelé *Adaptative Logic Module* (ALM), représenté figure 6.2. Chacun de ces blocs est constitué de deux tables de correspondance (LUT). La sortie de chaque table peut-être optionnellement chaînée avec un additionneur. Enfin, chaque ALM possède 4 registres en sortie. On comprend sur la figure 6.2 que les ALMs peuvent être configurés de nombreuses manières différentes par les LUT et les différents multiplexeurs.

1. Plus précisément, la notion d'ALM est présente dans la plupart des FPGAs Altera/Intel. En

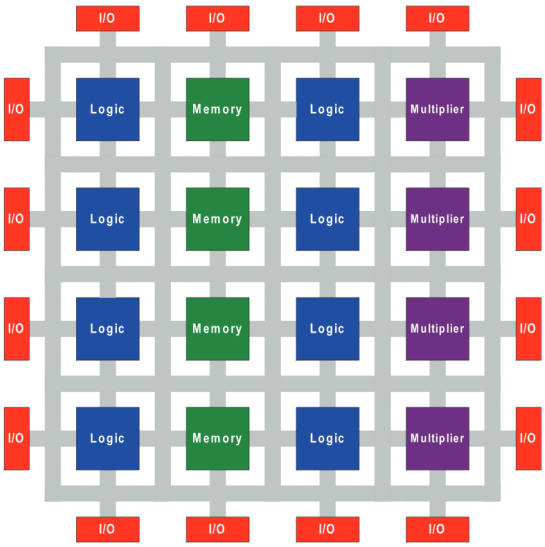


FIGURE 6.1. – Structure générale d'un FPGA, tiré de [KTR08]

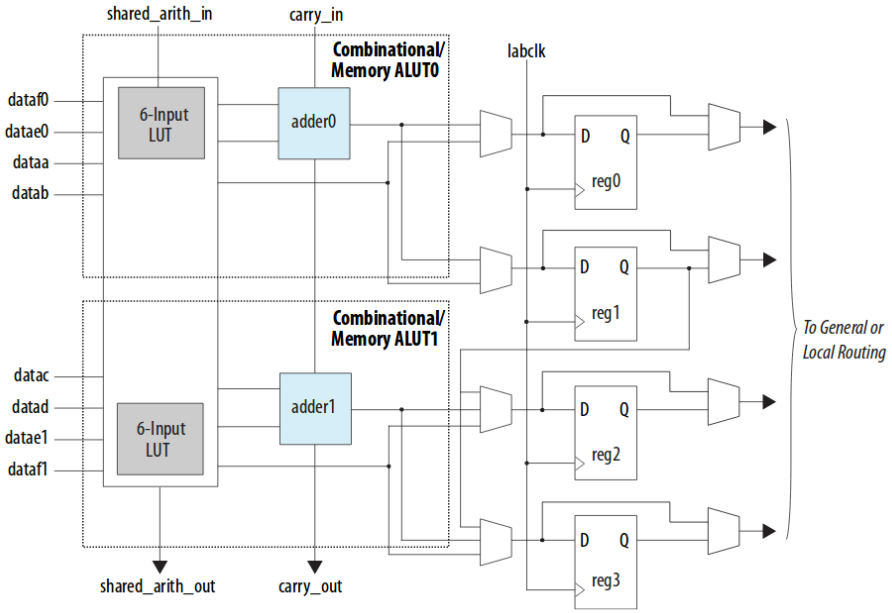


FIGURE 6.2. – Vue haut niveau d'un ALM, tiré de [Int16]

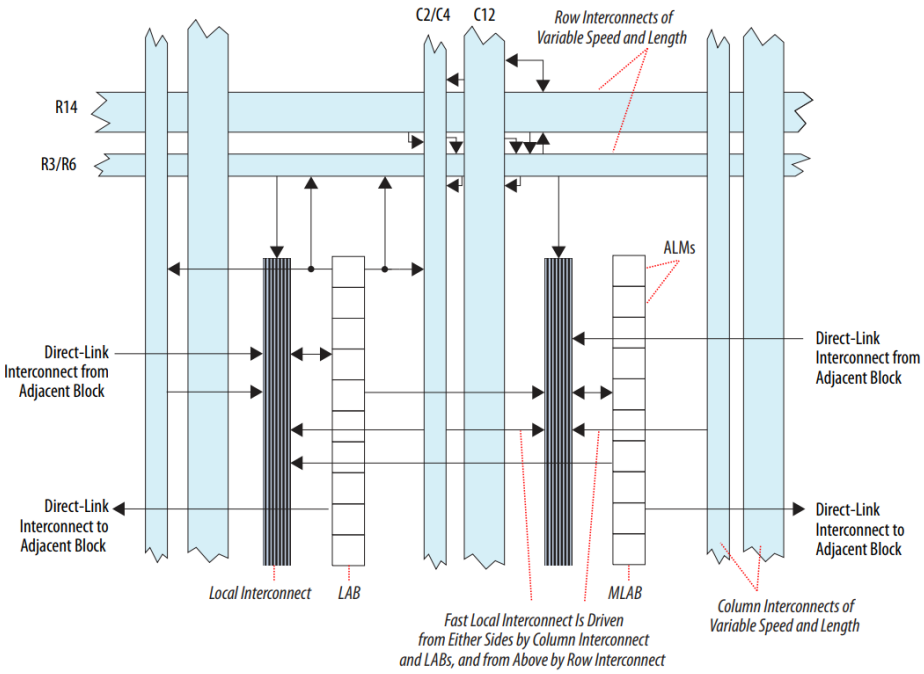


FIGURE 6.3. – Organisation des ALMs en LABs, tiré de [Int16]

Dans la matrice du FPGA, les ALMs sont organisés en colonnes par groupes de 10, appelés des *Logic Array Blocs* (LABs). Chaque LAB est connecté aux LABs directement adjacents (situés à droite et gauche) par des réseaux de communication locaux extrêmement rapides. Un second type d'interconnexion, plus lent permet d'effectuer des routages arbitraires entre LABs.

Plusieurs types de blocs mémoires sont disponibles. Une première forme est un LAB en mode mémoire (MLAB), seul un sous ensemble des LABs propose cette capacité. La mémoire est alors implémentée sous forme de LUT et chaque MLAB peut stocker jusqu'à 640 bits. Le FPGA possède également des blocs de mémoire dédiés permettant de stocker des données plus volumineuses. Chaque bloc (M10K) peut stocker un total de 10Kbits et la largeur est adaptable de 1 bit$\times$10K, jusqu'à 40 bits$\times$256. Les deux types de mémoire (MLAB et M10K) supportent simultanément un port en lecture/écriture et un second en lecture (*simple dual port RAM*). Les M10K supportent deux ports en lecture/écritures (*true dual port RAM*).

Enfin, le FPGA dispose également de blocs de multiplication configurables. Ces blocs sont assez complexes et très flexibles dans les configurations qu'il peuvent prendre. La figure 6.4 représente schématiquement un tel bloc. Les éléments principaux sont les deux circuits de multiplication de rayon 18×18, les bancs de registres en entrée et sortie, ainsi que des additionneurs, présent à la fois avant et après multiplication. Ainsi un bloc peut être configuré pour effectuer entre autres (la liste est loin d'être exhaustive) :

revanche les fonctionnalités de ces blocs varient selon les familles.

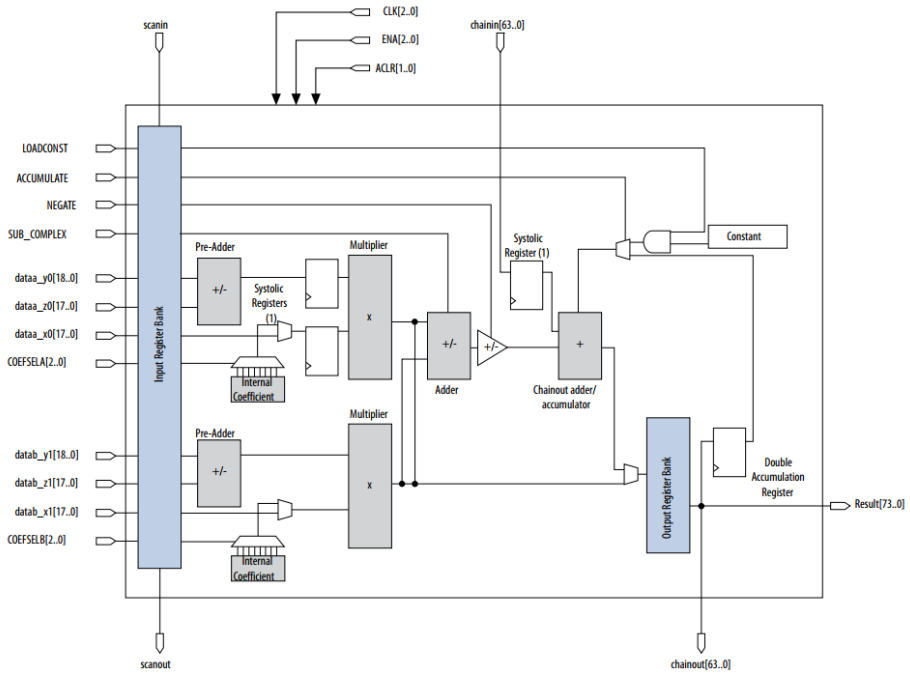


FIGURE 6.4. – Structure d'un bloc DSP pour la multiplication, tiré de [Int16]

— 2 multiplications indépendantes de rayon 18×18,
— 1 multiplication de rayon 27×27,
— 1 multiplication de rayon 18×18 sommée avec une valeur de 36 bits.

Carte d'évaluation utilisée

La carte d'évaluation utilisée est un kit de démarrage Arria V GX, équipé d'un FGPA Arria V GX 5AGXFB3H4F35C4N dont les caractéristiques principales sont résumées dans le tableau 6.1. Le FPGA est connecté à de nombreux autres périphériques : mémoire SDRAM, FLASH, port PCIe, LEDs, GPIO, ...

TABLE 6.1. – Caractéristiques principales du FPGA utilisé

ALM	MLAB	Bloc M10K	Nombre de DSPs
136 880	3 357	1 726	1045

6.2. Processeur d'évaluation

Les différentes solutions proposées sont évaluées sur un processeur qui supporte un sous-ensemble du jeu d'instructions MIPS 32 [Tec]. Le modèle RTL, optimisé pour une synthèse sur logique programmable est entièrement décrit en langage VHDL. Il est basé sur le pipeline RISC classique à 5 étages [HP11, annexe C] :

Fetch. Contient l'adresse d'instruction du programme (PC) et effectue la lecture dans la mémoire d'instructions.

Decode. Dans cet étage, l'instruction lue depuis la mémoire est décodée. C'est à ce moment que les registres nécessaires à l'instruction sont lus dans le banc de registres et que l'exécution est configurée. La décision d'effectuer un branchement est également prise à ce moment.

Execute. Contient l'unité arithmétique et logique et effectue les opérations purement arithmétiques.

Memory. Cet étage est responsable d'effectuer les opérations avec la mémoire de données.

Writeback. Cette dernière étape écrit le résultat du calcul dans le banc de registre.

Le processeur développé est relativement basique, les instructions sont exécutées une par une et sans réordonnancement (*single-issue, in-order*). Le processeur comporte une unité de multiplication et division, mais pas d'unité de calcul flottant. Enfin, le processeur dispose d'une mémoire d'instructions (en lecture seule) et de données (en lecture / écriture) dédiées. Ces deux mémoires sont locales au FPGA (BRAM) de taille 32KB et ont un délai d'accès d'un seul cycle d'horloge.

6.3. Mécanisme de chiffrement des instructions

Nous présentons dans cette section les résultats d'évaluation de la solution de chiffrement des instructions.

6.3.1. Validation et programmes d'évaluations

La validation du chiffrement de code est basée sur la suite de vérification du processeur. Il s'agit d'un ensemble de programmes C et assembleur annotés. Une routine additionnelle `track_output` permet au logiciel de tracer à l'exécution la valeur d'un registre ou d'une variable. Les annotations sous forme de commentaires permettent de spécifier la trace d'exécution attendue. Dans l'exemple de la figure 6.5, les commentaires de la forme `track <valeur>` listent les valeurs attendues pour les appels à `track_output`, c'est-à-dire le tableau trié.

Lorsque le processeur est simulé, la trace d'exécution est stockée dans un fichier, puis comparée avec la trace attendue. Si le processeur est exécuté sur cible réelle (FPGA), la trace est transmise sur une unité de communication (UART) et peut être réceptionnée sur un PC pour être vérifiée. Il est également possible de compiler les tests nativement (sur l'architecture hôte), où la fonction `track_output` est simplement remplacée par une écriture dans un fichier.

6.3.1.1. Vérification de code chiffré

Le chiffrement de code n'interfère pas avec le mécanisme de vérification du processeur et donc la suite de tests peut aisément être adaptée. En effet, la fonction

```
// On spécifie ci-dessous la trace attendue pour ce test
// track FFFFFFF7
// track FFFFFFFD
// track 00000001
// track 00000003
void test_quicksort() {
  static int32_t example_array[4] = {-3, -9, 1, 3};
  quicksort(example_array, 0, 3);
  for (size_t i = 0; i < 4; i++) {
    // Trace de l'élément i
    track_output(example_array[i]);
  }
}
```

FIGURE 6.5. – Un exemple de test pour un algorithme de tri rapide

`track_output` fait concrètement une écriture mémoire à une adresse spéciale. Les tests écrits en langage C doivent seulement être recompilés avec le chiffrement activé. En revanche, les tests écrits en assembleur n'ayant pas bénéficié des pré-traitements (décrits dans la section 3.3), ils ne peuvent pas être chiffrés correctement. Ils ne sont donc pas inclus dans les tests du chiffrement.

6.3.1.2. Programmes d'évaluation

Pour évaluer les performances, nous avons sélectionné plusieurs programmes d'évaluation :

— Chiffrement avec un AES en mode ECB.
— Calcul d'une fonction de hachage SHA1, tiré de la suite de *benchmark* MiBench.
— Tri d'un tableau avec un algorithme de tri rapide *(quicksort)*.
— Chiffrement d'un message sur la courbe elliptique secp160r1.

Ces programmes servent à l'évaluation de performances, mais ils vérifient en même temps la validité des résultats de la même manière que dans l'exemple figure 6.5.

6.3.2. Empreinte matérielle

La section 3.4 a présenté les modifications nécessaires pour supporter le chiffrement. Nous étudions dans un premier temps l'empreinte matérielle de la solution dans différentes configurations.

6.3.2.1. Matériel et paramètres d'évaluation

Tous les résultats de synthèse de cette partie sont donnés pour un FPGA Intel/Altera Arria V que nous avons présenté dans la section 6.1.2. L'outil de synthèse et

placement routage est Quartus dont les paramètres d'optimisations sont ceux d'origine (mode *balanced*). Le circuit est synthétisé pour fonctionner à la fréquence de 80 MHz. Les fréquences maximales sont obtenues par analyse statique dans les conditions de fonctionnement à 1.1 V et 85°C.

6.3.2.2. Synthèse du chiffrement par flots Trivium

Le tableau 6.2 présente les résultats de synthèse du circuit de Trivium pour différents niveaux de parallélisme. Ce dernier inclut également la logique pour le chargement de clé, d' IV et la gestion de l'initialisation. On observe clairement une augmentation de la surface (nombre d'ALM) quand le débit de Trivium augmente. Le nombre de registres quant à lui diminue, ce qui est cohérent, car le registre d'état du chiffrement par flot reste le même (288 bits) et en augmentant le parallélisme, la logique d'initialisation est réduite (en particulier le compteur de nombre de cycles).

TABLE 6.2. – Utilisation du circuit de chiffrement seul

Circuit	ALM	Registres	Débit [2]	Latence	f_{max} (MHz)
Trivium×1	177 (0.12%)	302	1	1152	408
Trivium×32	237 (0.17%)	297	32	36	367
Trivium×64	380 (0.27%)	296	64	18	301
Trivium×128	731 (0.5%)	295	128	9	227
Tiny AES	3403 (2.4%)	4305	128	21	189

À titre comparatif, nous avons également synthétisé dans les mêmes conditions un AES 128 bits complètement pipeliné. Le projet est *open source* et disponible sur OpenCores (https://opencores.org/project,tiny_aes). Ceci est l'occasion d'illustrer le coût matériel d'un chiffrement par blocs capable de produire des chiffrés au même débit que le processeur. On observe que le circuit d'AES occupe 3 fois plus de surface que le processeur de base (première ligne du tableau 6.3). Cependant, la comparaison n'est pas complètement équitable dans le sens où seuls 32 bits par cycle d'horloge, c'est-à-dire la largeur d'une instruction, sont nécessaires au déchiffrement pour l'architecture MIPS 32 bits utilisée.

Les performances du chiffrement de code proposé sont directement liées à la latence d'initialisation du chiffrement par flots. Cette dernière doit idéalement être la plus faible possible. La manière la plus directe de réduire cette latence est d'augmenter le nombre de bits par itération. Par exemple, la version 128 bits nécessite seulement 9 cycles d'initialisation, deux fois moins que la version 64 bits. Mais cette approche augmente inévitablement la surface du circuit. Aussi, à quel point est-il rentable de dérouler ainsi Trivium ? Pour répondre à cette question, nous traçons sur la courbe 6.6 le débit d'initialisation par ALM (mesuré en Mbits/ALM) en fonction du nombre de bits par itération p (allant de 1 jusqu'à 192). Cette métrique est calculée par

$$d_{init}(p) = F_{max}(p) \times \frac{1152}{p} \times \frac{1}{n_{ALM}(p)}\,.$$

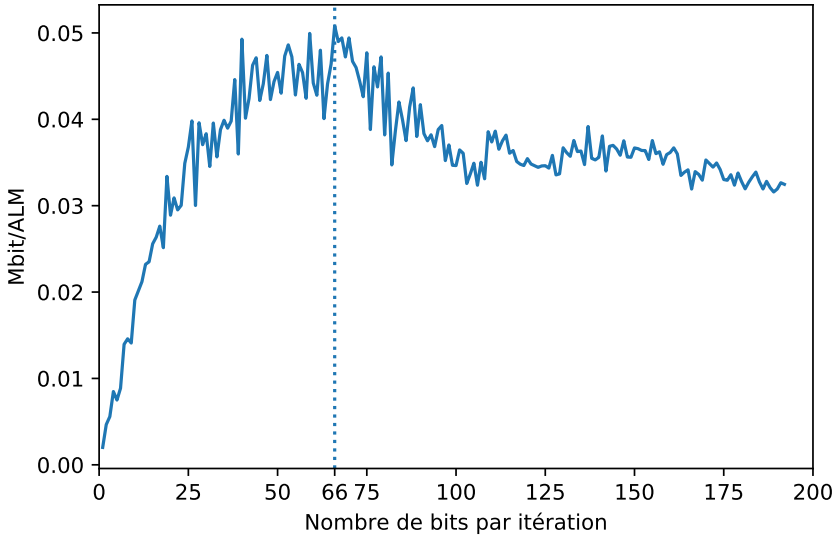


FIGURE 6.6. – Débit à l'initialisation par ALM en fonction du nombre de bits par itération

Le terme $1152/p$ est une approximation inférieure du nombre de cycles écoulés avant d'obtenir au moins un bit de la suite chiffrante. Cette valeur est multipliée par la fréquence maximale du circuit, puis le résultat est ramené au nombre d'ALMs utilisés. Le meilleur compromis, qui correspond au maximum de la courbe, est obtenu pour un nombre de bits par itération $p = 66$. Sans surprise, il s'agit du niveau maximal de parallélisme (au sens du circuit) supporté par Trivium (voir section 3.2.1). À l'aide de cette courbe, on remarque qu'il reste avantageux d'opter pour un niveau de parallélisme plutôt élevé pour réduire la latence. Même la variante pour $p = 128$ maintient un compromis acceptable entre les performances et le nombre d'ALMs nécessaires.

6.3.2.3. Empreinte matérielle du chiffrement dans le processeur

Nous analysons maintenant l'effet du mécanisme de chiffrement de code présenté dans le chapitre 3 sur le circuit du processeur. Pour cela, ce dernier est synthétisé en intégrant différentes versions du chiffrement. En particulier, nous faisons varier conjointement :

— la méthode de choix des vecteurs d'initialisation : complètement lus depuis la mémoire d'instructions dans une première variante (notée IMEM) et calculés comme l'adresse de la première instruction de chaque bloc de base dans la seconde variante (notée PC).
— Le nombre de bits par cycle d'horloge généré par Trivium : fixé à 64 ou 128. Les latences liées aux branchements sont respectivement de 18 cycles et 9 cycles.

Le tableau 6.3 donne les résultats de synthèse obtenus. La première ligne du tableau

correspond au processeur de référence. Lorsque le chiffrement est ajouté, on observe naturellement une augmentation de la surface (nombre d'ALMs). L'augmentation relativement au processeur de base est au minimum de 25%. Il faut cependant noter que le processeur de référence est de taille relativement faible. Le surcoût serait nettement moins élevé pour un processeur plus important.

TABLE 6.3. – Utilisation en ressources du mécanisme de déchiffrement dans le processeur

Chiffrement	Débit	ALM	Registres	f_{max} (MHz)
Absent	-	1136	987	116.74
IMEM	64	1412 (+25%)	1439 (+45%)	114.47 (-1.9%)
IMEM	128	1710 (+50%)	1501 (+52%)	114.03 (-2.3%)
PC	64	1419 (+25%)	1350 (+36%)	115.19 (-1.3%)
PC	128	1712 (+50%)	1418 (+43%)	116.96 (+0.1%)

Les versions basées sur un chiffrement à plus haut débit (128 bits par cycle) sont plus performantes, le temps d'initialisation du chiffrement par flot étant divisé par deux. En contrepartie la logique augmente significativement avec environ 50% d'ALMs supplémentaires. Le surcoût est principalement combinatoire, le nombre de registres n'augmente pas de 25 % (nous avions même observé un nombre de registres plus faible pour Trivium 128 bits par rapport à la version 64 bits dans le tableau 6.2). Des registres additionnels sont quand même nécessaires pour découper le flot de 128 bits en blocs de 32 bits. La fonction `genBits` du chiffrement par flots fonctionne dans ce cas seulement un cycle sur 4.

En revanche, le mécanisme de chiffrement n'a que très peu d'effet sur la fréquence maximale du circuit. On observe de très légères variations, inférieures à 3%. Lorsque l'IV est calculé depuis l'adresse, la fréquence maximale est même plus élevée. Une explication possible est que les variations observées sont probablement liées aux algorithmes de placement routage, basé sur des algorithmes hautement heuristiques. Nous avons pris garde de recréer complètement un projet Quartus pour chaque synthèse pour éviter tout phénomène d'amélioration lié à une compilation incrémentale.

6.3.3. Performances

Après avoir étudié les effets du chiffrement de code sur le circuit, nous nous penchons sur les conséquences sur le logiciel. En particulier, quel est l'effet sur la taille des programmes et leur temps d'exécution ? Pour répondre à ces questions, la méthodologie suivante est employée. Les programmes d'évaluation sont compilés avec, puis sans le chiffrement. Ensuite, les deux versions sont comparées en taille et temps d'exécution (mesuré en nombre de cycles d'horloges). Les performances du chiffrement de code sont hautement liées à la compilation. C'est pourquoi la comparaison est effectuée pour plusieurs profils d'optimisation, « -Oz » qui active les optimisations visant

à réduire la taille du programme (l'équivalent de « -Os » pour le compilateur GCC). Puis, le profil « -O3» qui optimise de manière agressive dans le but de minimiser le temps d'exécution.

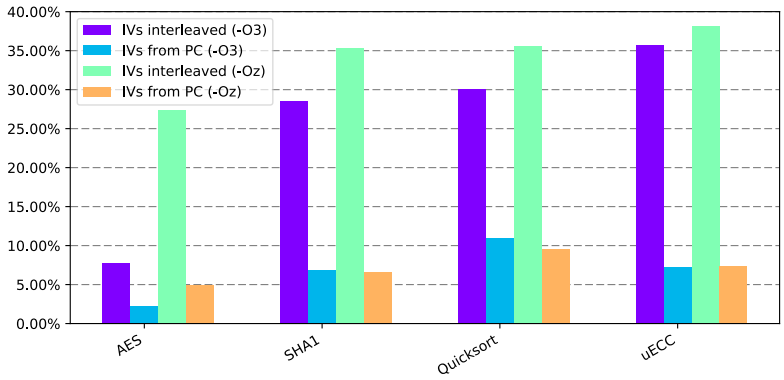
TABLE 6.4. – Effet du chiffrement de code sur la taille et le temps d'exécution

	IVs depuis l'adresse				IVs stockés dans le code			
	LLVM -Oz		LLVM -O3		LLVM -Oz		LLVM -O3	
	taille	temps	taille	temps	taille	temps	taille	temps
AES	+5 %	×2.39	+2.3 %	×1.29	27.4 %	×2.93	7.8 %	×1.41
SHA1	+6.6 %	×2.10	+6.8 %	×1.56	35.3 %	×2.52	28.6 %	×1.76
Quicksort	+9.5 %	×2.26	+11 %	×1.59	35.6 %	×2.75	30.1 %	×1.82
uECC	+7.4 %	×2.16	+7.2 %	×1.5	38.1 %	×2.61	35.7 %	×1.70
Moyenne	+7.12 %	×2.23	+6.8 %	×1.48	34.1 %	×2.70	25.5 %	×1.67

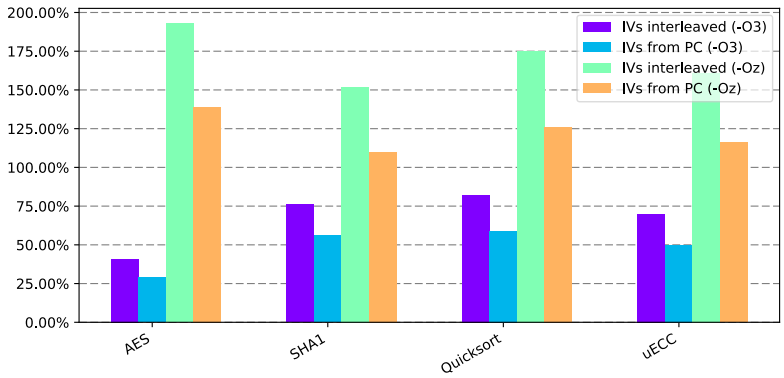
Le tableau 6.4 présente les résultats obtenus avec un Trivium à débit de 128 bits par cycle (soit 9 cycles d'initialisation à chaque branchement). Ces résultats sont aussi représentés graphiquement sur la figure 6.7. Comme attendu, le profil d'optimisation affecte fortement les résultats, indépendamment de la politique de choix des vecteurs d'initialisation. Pour le profil « -Oz », on observe une dégradation significative des résultats que cela soit en taille de programme ou en temps d'exécution. Un code optimisé pour minimiser la taille induit inévitablement une réutilisation plus forte de parties du code. Ce qui signifie plus de blocs de base et plus de branchements, d'où un effet plus marqué du chiffrement de code. À l'opposé, les optimisations pour les performances cherchent à agrandir les blocs de base et donc améliorent les performances du chiffrement.

La méthode de choix des vecteurs d'initialisation influe également sur les résultats. L'utilisation d'IVs calculés à partir de l'adresse présente un cas bien plus favorable. Naturellement, l'effet sur la taille du programme chiffré est bien plus réduit qu'avec des IVs stockés dans le code. L'augmentation en taille reste néanmoins perceptible. Elle est due intégralement à l'ajout des branchements nécessaire à l'exécution correcte du programme chiffré (voir section 3.3.2). Qui plus est, sur l'architecture MIPS, la plupart des instructions de branchement possèdent un *delay slot*. Si le compilateur n'arrive pas à remplir cet espace, ce branchement additionnel coûte deux instructions en espace (le branchement et un NOP). L'effet de la méthode de choix des IVs est moins visible sur les temps d'exécutions. Même si, ici encore, la variante avec les IVs calculés à partir de l'adresse obtient de meilleurs résultats.

En conclusion, le profil le plus adapté au chiffrement de code est la combinaison du mode de compilation « -O3 » avec des IVs calculés seulement à partir de l'adresse.



(a) Effet sur la taille de programme



(b) Effet sur le temps d'exécution

FIGURE 6.7. – Représentation graphique des résultats

6.4. Coprocesseur de calculs homomorphes

Nous abordons maintenant le développement et l'évaluation du coprocesseur de calculs homomorphes.

6.4.1. Stratégie de développement

Le développement d'accélérateurs matériels pour le chiffrement homomorphe n'est pas une tâche aisée. Une des difficultés du point de vue de la conception est la gestion des configurations. Un nombre important de paramètres et constantes interviennent. Pour définir le schéma BGV, un 6-uplets de valeurs sont nécessaire (m, q, p, σ, h, ρ), à cela s'ajoutent diverses constantes pour l'accélération de blocs spécifiques, comme :

— les *twiddle factors* de la NTT,
— les constantes pour la réduction modulaire,
— la table CDT du générateur de gaussienne discrète.

Le langage de description matériel VHDL utilisé pour décrire le processeur n'offre clairement pas assez de généricité pour adresser ce problème de manière satisfaisante. Certes, le langage possède le mécanisme des *generics*, qui permet de spécifier des paramètres à la compilation aux composants. Cependant les usages sont assez limités, par exemple, il n'est pas possible de passer des tableaux dont la taille des éléments varie (ce qui est le cas pour toutes les tables utilisées). Une autre fonctionnalité manquante est le passage par paramètre de sous composants, ou de sous routines [3]. De plus, les procédures qui calculent les différents paramètres sont assez complexes, elles font appel à de l'arithmétique avancée qu'il est difficile d'exprimer directement en VHDL.

Une solution est donc de générer les différents accélérateurs à partir d'un langage de programmation plus traditionnel (C++, Python ou autre). Une approche intéressante en ce sens est la librairie Chisel [BVR+12] développée par l'université de Berkeley qui permet de construire des circuits dans le langage de programmation Scala. Ce langage offre de larges possibilités d'introspection et d'abstraction, ce qui permet de décrire des circuits avec une syntaxe très expressive. Un succès notable de Chisel est son utilisation pour décrire l'ensemble des processeurs de référence de l'architecture RISC-V ainsi qu'un nombre important d'IPs permettant de construire un système sur puce complet [AAB+16]. Pour des besoins d'interfacage avec des blocs écrits en VHDL, nous avons utilisé SpinalHDL [4]. Cette librairie Scala est très similaire à Chisel mais supporte en plus la génération et l'intégration avec des composants écrits en VHDL.

Illustration avec la fonction de déchiffrement du schéma homomorphe

Pour illustrer concrètement l'aspect d'un composant matériel décrit en Scala, nous prenons la fonction de déchiffrement du schéma BGV comme cas d'exemple. Par souci de concision, nous ne montrons ici que la première partie qui calcule $\texttt{INTT}(c_0 + c_1 \odot$

3. La révision 2008 du langage VHDL apporte des solutions à ces problèmes. Malheureusement, très peu de fonctionnalités sont supportées à ce jour par les outils de synthèse.
4. Disponible à l'addresse https://github.com/SpinalHDL/SpinalHDL

```
1  class Decrypt(cfg: BgvConfig) extends Component {
2    val io = new Bundle {
3      val start = in Bool()
4
5      /** Coefficients du message chiffré en entrée */
6      val ct0 = slave Stream(Bits(cfg.w bit))
7      val ct1 = slave Stream(Bits(cfg.w bit))
8      val ct2 = slave Stream(Bits(cfg.w bit))
9
10     /** Coefficients de la clé secrète */
11     val sk = slave Stream(Bits(cfg.w bit))
12     val sk2 = slave Stream(Bits(cfg.w bit))
13
14     val result = master Stream(Bits(cfg.w bit))
15     val done = out Bool()
16   }
17
18   // Spécialisation des opérateurs arithmétiques
19   def mulMod(x : Stream[Bits], y : Stream[Bits]) =
20     BarrettReducer(cfg.q, x, y)
21   def addMod(x : Stream[Bits], y : Stream[Bits]) = AddMod(cfg.q, x, y)
22
23   val x = addMod(io.ct0,
24                  addMod(mulMod(io.ct1, io.sk),
25                         mulMod(io.ct2, io.sk2))).queue(8)
26   val (done, result) = InverseNTT(cfg.getNttConfig(), io.start, x)
27   io.result << result
28   io.done << done
29 }
```

FIGURE 6.8. – Description de la première partie du déchiffrement BGV

$s + c_2 \odot s^2$). Le code Scala, basé sur la librairie SpinalHDL pour cette fonction est représenté sur figure 6.8.

Un composant particulier est créé comme une classe qui dérive d'une classe principale `Component`. L'attribut spécial `io` sert à définir les entrées-sorties. Ces dernières sont ici pour la plupart des flots de données (le type `Stream`), qui correspondent aux coefficients des polynômes transmis successivement (un par cycle d'horloge) dans le temps. La configuration du composant est entièrement représentée par la classe opaque `BgvConfig`. À partir de cette classe, toutes les constantes et tables sont calculées.

Sans chercher à comprendre tous les détails du code de la figure 6.8, on peut remarquer avec quelle simplicité l'expression $c_0 + c_1 \odot s + c_2 \odot s^2$ est calculée des lignes 23 à 25. La NTT inverse est également instanciée de manière très synthétique à la ligne

25. Pourtant, des circuits complexes sont impliqués : la réduction de Barrett, NTT inverse. Aussi, l'approche de Chisel et SpinalHDL permet d'abstraire complètement cette complexité et d'avoir une intégration très expressive des composants.

6.4.2. Evaluation

Les paramètres exacts utilisés pour le schéma BGV sont spécifiés dans le tableau 6.5. La synthèse est ici effectuée sur Arria V GX pour une fréquence cible de 100MHz. Le tableau 6.9 présente les résultats obtenus en termes de surface et performances.

m	q	p	σ	h	ρ
2048	195382210561	2	3.1	64	0.5

TABLE 6.5. – Paramètres utilisés pour le BGV

	ALM	Reg.	Mem.	DSP	Fmax (Mhz)
Mult. modulaire	260 (0.1%)	436	-	11 (1%)	143.35
Ech. gaussienne	762 (0.5%)	569	-	-	193.27
NTT	509 (0.37%)	860	170 Kb	11 (1%)	132.17
Chiffrement	4 624 (3.3%)	6789	1.2 Mb (7%)	77 (7%)	135.75
Déchiffrement	1 573 (1.5%)	2618	403 Kb (2%)	55 (5%)	133.73
ALU	12 434 (9%)	16384	2.9 Mb (16%)	253 (24%)	128.35

(a) Synthèse sur Arria V

	Délai (cycles)	Délai @100Mhz
NTT	15415	154.1 μs
Chiffrement	15486	154.8 μs
Déchiffrement	15496	154.9 μs
ALU XOR	30994	309.9 μs
ALU AND	31002	310 μs

(b) Performances

FIGURE 6.9. – Résultats du coprocesseur de chiffrement homomorphe

6.4.2.1. Empreinte matérielle

L'ALU qui représente le coprocesseur entier est indéniablement un composant de taille importante. Si l'on compare au processeur avec le chiffrement de code (tableau 6.3 page 102) ce circuit est plus de 7 fois plus important en nombre ALMs. De

plus, la répartition des ressources utilisées sur le FPGA est très inégale. Le coprocesseur nécessite en particulier une quantité importante de blocs de multiplication (24%). Comparativement, moins de logique est nécessaire (9% du total d'ALM).

On remarque que le bloc de chiffrement seul est déjà très important. Néanmoins ce résultat était prévisible, un total de 4 blocs NTT sont utilisés. À cela s'ajoute les générateurs de gaussienne discrète (2 sont nécessaires) et un circuit de Trivium pour générer un élément uniforme de R_q. On comprend mieux que l'ALU homomorphe, qui inclut 2 blocs de chiffrement et un de déchiffrement soit aussi volumineuse.

6.4.2.2. Performances

Quant aux performances, le composant qui introduit les latences [5] majeures dans le flot de données est la NTT. Le temps de traitement incompressible de la NTT est de 154μs. Autrement dit, pendant cet intervalle de temps, aucune donnée n'est acceptée en entrée. Ce composant mis à part, les autres opérateurs sont complètement pipelinés et produisent une donnée par cycle en régime établi. On comprend mieux que les opérations de chiffrement et déchiffrement ont un délai de traitement comparable à la NTT. Les latences supplémentaires introduites sont dues aux opérateurs arithmétiques (addition, multiplication modulaire, ...) et aux files utilisées pour relier les opérateurs et s'élèvent une fois additionnées à une dizaine de cycles. Par exemple, un bloc d'addition modulaire a une latence de 2 cycles. La multiplication modulaire (basée sur la méthode de Barrett) a une latence de 6 cycles.

Les temps obtenus pour les calculs homomorphes effectués par l'ALU sont de 310 μs pour un « AND » et 309.9 μs pour un « XOR». Ici encore, les temps obtenus sont très proches, l'évaluation du « AND» a un circuit légèrement plus profond d'où une latence supérieure de quelques cycles.

L'approche que nous avons suivie pour la conception de ce coprocesseur est de minimiser les temps de latence. Pour cette raison, tous les opérateurs sont dupliqués et fonctionnent en parallèle. La conséquence est évidemment une surface importante. Cependant, le circuit résultant reste synthétisable sur des FPGAs de gamme moyenne comme celui utilisé. Pour réduire la surface, les différents blocs peuvent être mis en communs, mais au prix d'une latence plus élevée.

6.5. Conclusion

Ce chapitre a présenté la mise en place et l'évaluation des solutions proposées pour le chiffrement d'un programme et de ses données. Une première conclusion de cette évaluation est que la méthode de chiffrement de code offre une solution très légère au chiffrement, dont les performances sont tout à fait acceptables. De plus, nous avons pu observer l'effet important des options de compilation sur les performances.

5. La latence (ou délai) d'un bloc est définie ici comme le temps écoulé entre l'envoi du premier coefficient en entrée et la réception du premier coefficient en sortie.

En particulier les profils d'optimisation orientés performances sont bien plus adaptés au chiffrement. Cette observation suggère une étude plus poussée des méthodes de placement et d'agrandissement des blocs de base conjointement au chiffrement pour optimiser les performances. Un autre axe d'amélioration est l'utilisation d'une architecture plus avancée. Par exemple, un chiffrement fonctionnant de manière asynchrone à fréquence plus élevée pour l'initialisation, ou encore utiliser un prédicteur de branchement dédié pour anticiper les initialisations.

Nous avons également évalué le coprocesseur de calculs homomorphes. Un premier résultat intéressant est que l'accélérateur entièrement parallèle est synthétisable sur un FPGA de gamme moyenne. Cette approche permet d'avoir une idée des performances limites et des points limitants. En ce sens, l'opération de NTT est le principal contributeur aux latences mesurées. Clairement, au vu des surfaces impliquées, il est difficile d'envisager une utilisation embarquée ou sur système contraint en ressources. Néanmoins, les résultats invitent à réfléchir sur la conception d'une architecture plus compacte, mais également étendre ce type de coprocesseurs à d'autres champs d'applications.

7 Conclusion générale

Assurer la confidentialité des programmes sur les processeurs représente un enjeu majeur en termes de sécurité. L'état de l'art fournit aujourd'hui un panel de solutions tout à fait viables pour parvenir à cet objectif. Les solutions basées sur du chiffrement sont particulièrement intéressantes pour leurs performances et la sécurité prouvable qu'elles apportent. Néanmoins, ces solutions sont pour la plupart conçues pour être placées aux interfaces avec la mémoire. Aujourd'hui, les besoins évoluent et il est de plus en plus nécessaire de maintenir la sécurité non seulement aux interfaces mémoire, mais également au plus profond de l'architecture.

C'est pourquoi nous avons cherché tout au long de cette thèse des méthodes permettant de maintenir le chiffrement au plus proche des calculs. D'une part, pour pouvoir cibler les processeurs contraints où il n'existe presque aucun élément entre la mémoire et le processeur. Mais également dans le souci de limiter les fuites d'information à une zone la plus réduite possible.

Dans un premier temps, nous avons proposé un mécanisme de chiffrement dédié aux instructions. Cette méthode exploite efficacement la nature séquentielle des accès mémoire en chiffrant à l'aide d'un algorithme de chiffrement par flots les instructions. Nous avons montré dans ce contexte, comment baser le chiffrement sur le flot de contrôle du programme en chiffrant à l'échelle des blocs de bases étendus que nous avons définis. Ceci permet de fournir un chiffrement complètement invisible au programme, tous les traitements étant effectués au moment de la compilation.

Dans une seconde partie de la thèse, nous avons exploré dans quelle mesure les algorithmes de chiffrement homomorphe peuvent être utilisés pour protéger les données d'un programme. Cette approche permet de maintenir les données chiffrées non seulement en mémoire, mais également durant les calculs. Aussi, cela permet par construction d'empêcher toute fuite d'information intermédiaire. Cependant, l'utilisation de chiffrement complètement homomorphe apparaît difficile compte tenu de la dimension des calculs impliqués. Nous avons ainsi proposé un compromis, à savoir protéger une zone bien définie qui effectue un rechiffrement efficace. Dans cette zone, le bruit présent dans un message chiffré est retiré.

Nous avons ensuite montré comment construire l'évaluation de programmes autour de ce rechiffrement. Dans un premier temps en spécifiant un encodage des données, puis en présentant la construction d'opérations primitives. Nous avons abouti à la conception d'un coprocesseur permettant d'effectuer des opérations homomorphes sur des données élémentaires de 8 bits, avec une structure similaire à celle d'un processeur.

Un aspect important du travail fut le développement de ces blocs et la mise en place de l'architecture sur logique programmable. De la synthèse et des temps obtenus, nous tirons plusieurs conclusions :

— l'approche de rechiffrement permet de réduire les paramètres et en conséquence d'obtenir un circuit compact. Nous avons pu ici paralléliser entièrement tous les blocs de calculs. L'architecture résultante a une surface acceptable.
— La réduction des paramètres permet de réduire considérablement l'utilisation en mémoire nécessaire aux calculs homomorphes.
— La procédure de rechiffrement est un circuit important et dont le temps d'exécution est relativement élevé. Par conséquent, le temps des opérations de base l'est aussi. À l'opposé, un schéma *somewhat homomorphic* évaluera les opérations plus rapidement.

Ainsi, les gains en temps ne sont pas forcément visibles, mais masqués par l'approche générique utilisée pour encoder les données et traduire les calculs. De plus, le coprocesseur de calcul fonctionne à relativement basse fréquence.

Perspectives

Les résultats de la méthode de chiffrement d'un code basée sur un chiffrement par flots sont encourageants. Il reste évidemment un large champ d'exploration quant à l'amélioration des performances. Un des axes de poursuite de ces travaux est de spécialiser plus fortement la compilation pour le chiffrement, en particulier par des algorithmes de placement de blocs de base et d'allongement de blocs de base. Il y a évidemment un compromis à trouver entre l'augmentation en taille du programme et les gains en performances.

L'autre axe d'amélioration possible consiste en une architecture plus avancée du circuit de chiffrement intégré au processeur. La partie coûteuse en temps étant effectuée lors des branchements, on peut envisager la conception et l'utilisation d'un prédicteur de branchement dédié pour le chiffrement dans le but d'anticiper le chiffrement en le commençant à l'avance.

Se pose également la question de l'intégrité du programme, que nous avons supposée vérifiée. Nous avons observé que ce type de chiffrement permettait de forcer une certaine intégrité des chemins de code et de prévenir le déplacement de code. Mais il semble intéressant d'explorer si une vérification également efficace de l'intégrité des instructions peut être mise en place, parallèlement ou conjointement au chiffrement.

Un grand nombre d'optimisations sont également possibles pour le coprocesseur de calculs homomorphe. Ces améliorations peuvent avoir lieu au niveau matériel. Au vu des résultats, le bloc à optimiser en priorité est la NTT, qui se trouve sur le chemin critique du rechiffrement. L'échantillonnage de gaussiennes discrètes en particulier gagnerait à être plus compact. Mais des améliorations peuvent être faites également au niveau de la génération des circuits et de l'encodage des données pour le calcul.

Enfin, dans la solution proposée dans cette thèse, l'amélioration notable de l'efficacité des calculs homomorphes est apportée une grande partie grâce à une nouvelle procédure de rechiffrement dont la sécurité est facile à établir dans les modèles d'atta-

quant les plus simples. Une prolongation naturelle de ces travaux consiste à pousser davantage l'analyse de la sécurité, en particulier vis-à-vis des attaques par canaux auxiliaires, afin d'étendre le modèle de sécurité et -par là- les cas d'utilisation de ce type de processeur sécurisé.

Preuve de la proposition 5

A

Nous donnons ici une preuve de la proposition 5 qui assure la validité de la réduction de Barrett. Nous rappelons que le quotient est approximé (voir section 5.2.2) par

$$k = \lfloor \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}} \rfloor$$

Essayons d'identifier les contraintes sur les paramètres valides. D'après la définition de la partie entière, on a

$$\frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}} \geq \lfloor \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}} \rfloor > \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}} - 1$$

La partie gauche est clairement inférieure à $\lfloor x/q \rfloor$, car

$$\lfloor \frac{x}{q} \rfloor = \frac{\frac{x}{2^{n+\beta}} \frac{2^{n+\alpha}}{q}}{2^{\alpha-\beta}} \geq \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}}$$

pour ce qui est du membre droit

$$\begin{aligned}
\frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \lfloor \frac{2^{n+\alpha}}{q} \rfloor}{2^{\alpha-\beta}} - 1 &\geq \frac{\left(\frac{x}{2^{n+\beta}} - 1\right)\left(\frac{2^{n+\alpha}}{q} - 1\right)}{2^{\alpha-\beta}} - 1 \\
&= \frac{x}{q} - \frac{x}{2^{n+\alpha}} - \frac{2^{n+\beta}}{q} + \frac{1}{2^{\alpha-\beta}} - 1 \\
&\geq \lfloor \frac{x}{q} \rfloor - \frac{x}{2^{n+\alpha}} - \frac{2^{n+\beta}}{q} + \frac{1}{2^{\alpha-\beta}} - 1 \\
&\geq \lfloor \frac{x}{q} \rfloor - 2^{n-\alpha} - 2^{\beta+1} + 2^{\beta-\alpha} - 1 \\
&\geq \lfloor \frac{x}{q} \rfloor - e(n,\, \alpha,\, \beta) - 1
\end{aligned}$$

il s'agit donc de borner le terme d'erreur $e(n,\, \alpha,\, \beta) = 2^{n-\alpha} + 2^{\beta+1} - 2^{\beta-\alpha}$ si l'on veut garantir que l'estimation est assez proche du quotient réel. Ainsi, si $n - \alpha \leq -1$ et $\beta + 1 \leq -1$, c'est-à-dire si $\alpha \geq n + 1$ et $\beta \leq -2$, alors $e(n,\, \alpha,\, \beta) \leq 1$. On vérifie aisément que au plus 1 étape de réduction est nécessaire, car dans ce cas

$$\lfloor \frac{x}{q} \rfloor \geq \lfloor \frac{\lfloor \frac{x}{2^{n+\beta}} \rfloor \mu}{2^{\alpha-\beta}} \rfloor > \lfloor \frac{x}{q} \rfloor - 2$$

et donc

$$0 \leq x - q\lfloor\frac{\lfloor\frac{x}{2^{n+\beta}}\rfloor\mu}{2^{\alpha-\beta}}\rfloor < 2q$$

le reste approximé est donc bien entre 0 et $2q$. Ainsi, une étape de correction (soustraction de q) est éventuellement nécessaire.

Bibliographie

[AAB+16] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo et Andrew Waterman : The rocket chip generator. Rapport technique UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[AAF+16] Ehsan Aerabi, A Elhadi Amirouche, Houda Ferradi, Rémi Géraud, David Naccache et Jean Vuillemin : The conjoined microprocessor. *In Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*, pages 67–70. IEEE, 2016.

[ABD16] Martin R. Albrecht, Shi Bai et Léo Ducas : A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. *In* Matthew Robshaw et Jonathan Katz, éditeurs : *CRYPTO 2016, Part I*, volume 9814 de *LNCS*, pages 153–178. Springer, Heidelberg, août 2016.

[Alb17] Martin R. Albrecht : On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. *In* Jean-Sébastien Coron et Jesper Buus Nielsen, éditeurs : *EUROCRYPT 2017, Part II*, volume 10211 de *LNCS*, pages 103–129. Springer, Heidelberg, mai 2017.

[APS15] Martin R. Albrecht, Rachel Player et Sam Scott : On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. http://eprint.iacr.org/2015/046.

[ARP07] Jude Angelo Ambrose, Roshan G Ragel et Sri Parameswaran : Rijid : random code injection to mask power analysis based side channel attacks. *In Proceedings of the 44th annual Design Automation Conference*, pages 489–492. ACM, 2007.

[ARS+15] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen et Michael Zohner : Ciphers for mpc and fhe. *In Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer, 2015.

[BAP+03] Elena Gabriela Barrantes, David H Ackley, Trek S Palmer, Darko Stefanovic et Dino Dai Zovi : Randomized instruction set emulation to disrupt binary code injection attacks. *In Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289. ACM, 2003.

[Bar87] Paul Barrett : Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. *In* Andrew M. Odlyzko, éditeur : *CRYPTO'86*, volume 263 de *LNCS*, pages 311–323. Springer, Heidelberg, août 1987.

[BBT+15] Florent Bruguier, Pascal Benoit, Lionel Torres, Lyonel Barthe, Morgan Bourree et Victor Lomne : Cost-effective design strategies for securing embedded processors. 2015.

[Ber05] Daniel J Bernstein : Cache-timing attacks on aes, 2005.

[Bes79] R.M. Best : Microprocessor for executing enciphered programs, septembre 18 1979. US Patent 4,168,396.

[Bes80] Robert M Best : Preventing software piracy with crypto-microprocessors. *In Proceedings of IEEE Spring COMPCON*, volume 80, pages 466–469, 1980.

[Bes81] R.M. Best : Crypto microprocessor for executing enciphered programs, juillet 14 1981. US Patent 4,278,837.

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan et Ke Yang : On the (im) possibility of obfuscating programs. *In Annual International Cryptology Conference*, pages 1–18. Springer, 2001.

[BGN05] Dan Boneh, Eu-Jin Goh et Kobbi Nissim : Evaluating 2-dnf formulas on ciphertexts. *Theory of cryptography*, pages 325–341, 2005.

[BGV12] Zvika Brakerski, Craig Gentry et Vinod Vaikuntanathan : (leveled) fully homomorphic encryption without bootstrapping. *In Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[BHMT16] Joppe W. Bos, Charles Hubain, Wil Michiels et Philippe Teuwen : Differential computation analysis : Hiding your white-box designs is not enough. *In* Benedikt Gierlichs et Axel Y. Poschmann, éditeurs : *CHES 2016*, volume 9813 de *LNCS*, pages 215–236. Springer, Heidelberg, août 2016.

[Bil08] Matthew Robshaw Olivier Billet : New stream cipher designs. 2008.

[BLLN13] Joppe W. Bos, Kristin Lauter, Jake Loftus et Michael Naehrig : Improved security for a ring-based fully homomorphic encryption scheme. *In* Martijn Stam, éditeur : *14th IMA International Conference on Cryptography and Coding*, volume 8308 de *LNCS*, pages 45–64. Springer, Heidelberg, décembre 2013.

[Boi06] Adam Boileau : Hit by a bus : Physical access attacks with firewire, 2006.

[BV11a] Zvika Brakerski et Vinod Vaikuntanathan : Efficient fully homomorphic encryption from (standard) LWE. *In* Rafail Ostrovsky, éditeur : *52nd FOCS*, pages 97–106. IEEE Computer Society Press, octobre 2011.

[BV11b] Zvika Brakerski et Vinod Vaikuntanathan : Fully homomorphic encryption from ring-LWE and security for key dependent messages. *In* Phillip Rogaway, éditeur : *CRYPTO 2011*, volume 6841 de *LNCS*, pages 505–524. Springer, Heidelberg, août 2011.

[BV11c] Zvika Brakerski et Vinod Vaikuntanathan : Fully homomorphic encryption from ring-LWE and security for key dependent messages. *In Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.

[BVR+12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek et Krste Asanović : Chisel : constructing hardware in a scala embedded language. *In Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[CCF+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier et Renaud Sirdey : Stream ciphers : A practical solution for efficient homomorphic-ciphertext compression. *In International Conference on Fast Software Encryption*, pages 313–333. Springer, 2016.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva et Malika Izabachène : Faster fully homomorphic encryption : Bootstrapping in less than 0.1 seconds. *In* Jung Hee Cheon et Tsuyoshi Takagi, éditeurs : *ASIACRYPT 2016, Part I*, volume 10031 de *LNCS*, pages 3–33. Springer, Heidelberg, décembre 2016.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao et Pankaj Rohatgi : Towards sound approaches to counteract power-analysis attacks. *In* Michael J. Wiener, éditeur : *CRYPTO'99*, volume 1666 de *LNCS*, pages 398–412. Springer, Heidelberg, août 1999.

[Coo66] Stephen A. Cook : *On the minimum computation time of functions.* Thèse de doctorat, 1966.

[Cor09] Thomas H Cormen : *Introduction to algorithms.* MIT press, 2009.

[CRSP09] Siddhartha Chhabra, Brian Rogers, Yan Solihin et Milos Prvulovic : Making secure processors os-and performance-friendly. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(4):16, 2009.

[CT65] James W Cooley et John W Tukey : An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[CVDBDS09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere et Bjorn De Sutter : Practical mitigations for timing-based side-channel attacks on modern x86 processors. *In Security and Privacy, 2009 30th IEEE Symposium on*, pages 45–60. IEEE, 2009.

[DCDKC+16] Ruan De Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen De Bosschere, Bart Preneel, Bjorn De Sutter et Ingrid Verbauwhede : Sofia : software and control flow integrity architecture. *In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1172–1177. IEEE, 2016.

[DCP08] Christophe De Cannière et Bart Preneel : *Trivium*, pages 244–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[DG14] Nagarjun C Dwarakanath et Steven D Galbraith : Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, 2014.

[DGP14] Jean-Luc Danger, Sylvain Guilley et Florian Praden : Hardware-enforced protection against software reverse-engineering based on an instruction set encoding. *In Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 5 :1–5 :11, New York, NY, USA, 2014. ACM.

[DHS16] Yarkın Doröz, Yin Hu et Berk Sunar : Homomorphic aes evaluation using the modified ltv scheme. *Designs, Codes and Cryptography*, 80(2):333–358, 2016.

[DQ98] J-F Dhem et J-J Quisquater : Recent results on modular multiplications for smart cards. *In International Conference on Smart Card Research and Advanced Applications*, pages 336–352. Springer, 1998.

[Duc07] Guillaume Duc : *Support matériel, logiciel et cryptographique pour une éxécution sécurisée de processus*. Thèse de doctorat, 2007. Thèse de doctorat dirigée par Stern, Jacques Informatique Télécom Bretagne 2007.

[FV12] Junfeng Fan et Frederik Vercauteren : Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/2012/144.

[FV14] Pierre-Alain Fouque et Thomas Vannet : Improving key recovery to 784 and 799 rounds of Trivium using optimized cube attacks. *In* Shiho Moriai, éditeur : *FSE 2013*, volume 8424 de *LNCS*, pages 502–517. Springer, Heidelberg, mars 2014.

[Gen09a] Craig Gentry : *A fully homomorphic encryption scheme*. Thèse de doctorat, Stanford University, 2009.

[Gen09b] Craig GENTRY : Fully homomorphic encryption using ideal lattices. *In STOC*, volume 9, pages 169–178, 2009.

[GGH+13] Sanjam GARG, Craig GENTRY, Shai HALEVI, Mariana RAYKOVA, Amit SAHAI et Brent WATERS : Candidate indistinguishability obfuscation and functional encryption for all circuits. *In 54th FOCS*, pages 40–49. IEEE Computer Society Press, octobre 2013.

[GH11] Craig GENTRY et Shai HALEVI : Implementing Gentry's fully-homomorphic encryption scheme. *In* Kenneth G. PATERSON, éditeur : *EUROCRYPT 2011*, volume 6632 de *LNCS*, pages 129–148. Springer, Heidelberg, mai 2011.

[GHS12] Craig GENTRY, Shai HALEVI et Nigel P SMART : Homomorphic evaluation of the AES circuit. *In Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[GJM+16] Hannes GROSS, Manuel JELINEK, Stefan MANGARD, Thomas UNTERLUGGAUER et Mario WERNER : Concealing secrets in embedded processors designs. *In Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*, pages 89–104, 2016.

[GO96] Oded GOLDREICH et Rafail OSTROVSKY : Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3): 431–473, 1996.

[GSW13] Craig GENTRY, Amit SAHAI et Brent WATERS : Homomorphic encryption from learning with errors : Conceptually-simpler, asymptotically-faster, attribute-based. *In* Ran CANETTI et Juan A. GARAY, éditeurs : *CRYPTO 2013, Part I*, volume 8042 de *LNCS*, pages 75–92. Springer, Heidelberg, août 2013.

[HMC+93] Wen Mei W. HWU, Scott A. MAHLKE, William Y. CHEN, Pohua P. CHANG, Nancy J. WARTER, Roger A. BRINGMANN, Roland G. OUELLETTE, Richard E. HANK, Tokuzo KIYOHARA, Grant E. HAAB, John G. HOLM et Daniel M. LAVERY : The superblock : An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, May 1993.

[HP11] John L. HENNESSY et David A. PATTERSON : *Computer Architecture, Fifth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th édition, 2011.

[HS] Shai HALEVI et Victor SHOUP : HELib. https://github.com/shaih/HElib.

[HS14] Shai HALEVI et Victor SHOUP : Algorithms in HELib. *In International Cryptology Conference*, pages 554–571. Springer, 2014.

[HS15] Shai HALEVI et Victor SHOUP : Bootstrapping for HElib. *In* Elisabeth OSWALD et Marc FISCHLIN, éditeurs : *EUROCRYPT 2015, Part I*, volume 9056 de *LNCS*, pages 641–670. Springer, Heidelberg, avril 2015.

[HS17] T. Hiscock et O. Savry : Méthode d'exécution confidentielle d'un programme opérant sur des données chiffrées par un chiffrement homomorphe, 2017. EP Patent App. EP20,170,157,049.

[HSG17] T. Hiscock, O. Savry et L. Goubin : Lightweight software encryption for embedded processors. *In 2017 Euromicro Conference on Digital System Design (DSD)*, pages 213–220, Aug 2017.

[HSH+09] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum et Edward W Felten : Lest we remember : cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu et Murat Kantarcioglu : Access pattern disclosure on searchable encryption : Ramification, attack and mitigation. *In NDSS*, volume 20, page 12, 2012.

[Int16] Intel : Arria V device handbook, volume 1 : Device interfaces and integration, 2016.

[ISW03] Yuval Ishai, Amit Sahai et David Wagner : Private circuits : Securing hardware against probing attacks. *In Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[KL14] Jonathan Katz et Yehuda Lindell : *Introduction to modern cryptography*. CRC Press, 2014.

[KL15] Miran Kim et Kristin Lauter : Private genome analysis through homomorphic encryption. *BMC medical informatics and decision making*, 15(Suppl 5):S3, 2015.

[KMN12] Simon Knellwolf, Willi Meier et María Naya-Plasencia : Conditional differential cryptanalysis of Trivium and KATAN. *In* Ali Miri et Serge Vaudenay, éditeurs : *SAC 2011*, volume 7118 de *LNCS*, pages 200–212. Springer, Heidelberg, août 2012.

[KO62] A Karabutsa et Yu Ofman : Multiplication of many-digital numbers by automatic computers. *DOKLADY AKADEMII NAUK SSSR*, 145(2):293, 1962.

[Koc96] Paul C Kocher : Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *In Advances in Cryptology – CRYPTO 96*, pages 104–113. Springer, 1996.

[KR07] Ian Kuon et Jonathan Rose : Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[KTR08] Ian Kuon, Russell Tessier et Jonathan Rose : Fpga architecture : Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[Kuh98] Markus G Kuhn : Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP. *IEEE Transactions on Computers*, (10):1153–1157, 1998.

[LA04] Chris Lattner et Vikram Adve : LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[LMTH03] David Lie, John Mitchell, Chandramohan A Thekkath et Mark Horowitz : Specifying and verifying hardware for tamper-resistant software. *In Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 166–177. IEEE, 2003.

[LP11] Richard Lindner et Chris Peikert : Better key sizes (and attacks) for LWE-based encryption. *In* Aggelos Kiayias, éditeur : *CT-RSA 2011*, volume 6558 de *LNCS*, pages 319–339. Springer, Heidelberg, février 2011.

[LPR10] Vadim Lyubashevsky, Chris Peikert et Oded Regev : On ideal lattices and learning with errors over rings. *In* Henri Gilbert, éditeur : *EUROCRYPT 2010*, volume 6110 de *LNCS*, pages 1–23. Springer, Heidelberg, mai 2010.

[LTM+00] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell et Mark Horowitz : Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[LTV12] Adriana López-Alt, Eran Tromer et Vinod Vaikuntanathan : On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *In* Howard J. Karloff et Toniann Pitassi, éditeurs : *44th ACM STOC*, pages 1219–1234. ACM Press, mai 2012.

[Lyu11] Vadim Lyubashevsky : Lattice signatures without trapdoors. Cryptology ePrint Archive, Report 2011/537, 2011. http://eprint.iacr.org/2011/537.

[Mar97] Kuhn Markus : The trustno 1 cryptoprocessor concept. *CS555 Report, Purdue Univ*, 1997.

[Mer88] Ralph C. Merkle : A digital signature based on a conventional encryption function. *In* Carl Pomerance, éditeur : *CRYPTO'87*, volume 293 de *LNCS*, pages 369–378. Springer, Heidelberg, août 1988.

[MLC+92] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank et Roger A Bringmann : Effective compiler support for predicated execution using the hyperblock. *In ACM SIGMICRO Newsletter*, volume 23, pages 45–54. IEEE Computer Society Press, 1992.

[MMS01a] David May, Henk L Muller et Nigel P Smart : Non-deterministic processors. *In Information Security and Privacy*, pages 115–129. Springer, 2001.

[MMS01b] David May, Henk L Muller et Nigel P Smart : Random register renaming to foil DPA. *In Cryptographic Hardware and Embedded Systems - CHES 2001*, pages 28–38. Springer, 2001.

[Mon85] Peter L Montgomery : Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MS13] Silvia Mella et Ruggero Susella : On the homomorphic computation of symmetric cryptographic primitives. *In IMA International Conference on Cryptography and Coding*, pages 28–44. Springer, 2013.

[MW17] Daniele Micciancio et Michael Walter : *Gaussian Sampling over the Integers : Efficient, Generic, Constant-Time*, pages 455–485. Springer International Publishing, 2017.

[NIS99] FIPS PUB 46-3, Data Encryption Standard (DES), 1999. U.S.Department of Commerce/National Institute of Standards and Technology.

[NIS01] FIPS PUB 197, Advanced Encryption Standard (AES), 2001. U.S.Department of Commerce/National Institute of Standards and Technology.

[NLV11] Michael Naehrig, Kristin Lauter et Vinod Vaikuntanathan : Can homomorphic encryption be practical ? *In Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[ÖDSS15] Erdinç Öztürk, Yarkın Doröz, Berk Sunar et Erkay Savaş : Accelerating somewhat homomorphic evaluation using FPGAs. Cryptology ePrint Archive, Report 2015/294, 2015. http://eprint.iacr.org/2015/294.

[Pai99] Pascal Paillier : Public-key cryptosystems based on composite degree residuosity classes. *In Advances in cryptology–EUROCRYPT 1999*, pages 223–238. Springer, 1999.

[PG14] Thomas Pöppelmann et Tim Güneysu : Towards practical lattice-based public-key encryption on reconfigurable hardware. *In* Tanja Lange, Kristin Lauter et Petr Lisonek, éditeurs : *SAC 2013*, volume 8282 de *LNCS*, pages 68–85. Springer, Heidelberg, août 2014.

[PGM+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz et Stefan Mangard : Drama : Exploiting dram addressing for cross-cpu attacks. *In USENIX Security Symposium*, pages 565–581, 2016.

[Pla05] Thomas Plantard : *Arithmétique modulaire pour la cryptographie*. Theses, Université Montpellier II - Sciences et Techniques du Languedoc, décembre 2005.

[PLPI13] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou et Sotiris Ioannidis : ASIST : architectural support for instruction set randomization. *In Proceedings of the 2013 ACM SIGSAC*

conference on Computer & communications security, pages 981–992. ACM, 2013.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam et Adrian Macias : Accelerating homomorphic evaluation on reconfigurable hardware. *In International Workshop on Cryptographic Hardware and Embedded Systems*, pages 143–163. Springer, 2015.

[Pol71] John M Pollard : The fast fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[PPSH17] F. Pebay-Peyroula, O. Savry et T. HISCOCK : Encryption method for an instructions stream and execution of an instructions stream thus encrypted, 2017. US Patent App. 15/412,252.

[PR13] Emmanuel Prouff et Matthieu Rivain : Masking against side-channel attacks : A formal security proof. *In* Thomas Johansson et Phong Q. Nguyen, éditeurs : *EUROCRYPT 2013*, volume 7881 de *LNCS*, pages 142–159. Springer, Heidelberg, mai 2013.

[QW14] Frank Quedenfeld et Christopher Wolf : Advanced algebraic attack on trivium. Cryptology ePrint Archive, Report 2014/893, 2014. http://eprint.iacr.org/2014/893.

[RAD78] Ronald L Rivest, Len Adleman et Michael L Dertouzos : On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[RCPS07] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic et Yan Solihin : Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. *In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196. IEEE Computer Society, 2007.

[RCR+16] Oscar Reparaz, Ruan Clercq, Sujoy Sinha Roy, Frederik Vercauteren et Ingrid Verbauwhede : Additively homomorphic ring-LWE masking. *Post-Quantum Cryptography*, janvier 2016.

[Reg09] Oded Regev : On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[RJV+15] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil S. Dimitrov et Ingrid Verbauwhede : Modular hardware architecture for somewhat homomorphic function evaluation. *In* Tim Güneysu et Helena Handschuh, éditeurs : *CHES 2015*, volume 9293 de *LNCS*, pages 164–184. Springer, Heidelberg, septembre 2015.

[Rog07] Marcin Rogawski : Hardware evaluation of eSTREAM candidates. 2007.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren et Ingrid Verbauwhede : A masked ring-LWE implementation. *In* Tim Güneysu et Helena Handschuh, éditeurs : *CHES 2015*, volume 9293 de *LNCS*, pages 683–702. Springer, Heidelberg, septembre 2015.

[RSA78] Ronald L Rivest, Adi Shamir et Leonard Adleman : A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen et Ingrid Verbauwhede : Compact ring-LWE cryptoprocessor. *In* Lejla Batina et Matthew Robshaw, éditeurs : *CHES 2014*, volume 8731 de *LNCS*, pages 371–391. Springer, Heidelberg, septembre 2014.

[Sem] Dallas Semiconductor : DS5002FP secure microprocessor chip.

[SOD05] G Edward Suh, Charles W O'Donnell et Srinivas Devadas : AEGIS : A single-chip secure processor. *Information Security Technical Report*, 10(2):63–73, 2005.

[SOSD05] G Edward Suh, Charles W O'Donnell, Ishan Sachdev et Srinivas Devadas : Design and implementation of the AEGIS single-chip secure processor using physical random functions. *In ACM SIGARCH Computer Architecture News*, volume 33, pages 25–36. IEEE Computer Society, 2005.

[SV14] N.P.a Smart et F.b Vercauteren : Fully homomorphic SIMD operations. *Designs, Codes, and Cryptography*, 71(1):57–81, 2014.

[Tec] Imagination Technologies : MIPS32 instruction set reference. https://imgtec.com/?do-download=4291.

[ZZP04] Xiaotong Zhuang, Tao Zhang et Santosh Pande : Hide : an infrastructure for efficiently protecting information leakage on the address bus. *In ACM SIGPLAN Notices*, volume 39, pages 72–84. ACM, 2004.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Microcontrôleur à flux chiffré d'instructions et de données
Mots-clés : Sécurité matérielle; Conception de processeurs; Chiffrement de code; Chiffrement par flots; Chiffrement homomorphe

Résumé. Un nombre important et en constante augmentation de systèmes numériques nous entoure. Tablettes, smartphones et objets connectés ne sont que quelques exemples apparents de ces technologies omniprésentes, dont la majeure partie est enfouie, invisible à l'utilisateur. Les microprocesseurs, au cœur de ces systèmes, sont soumis à de fortes contraintes en ressources, sûreté de fonctionnement et se doivent, plus que jamais, de proposer une sécurité renforcée. La tâche est d'autant plus complexe qu'un tel système, par sa proximité avec l'utilisateur, offre une large surface d'attaque.

Cette thèse, se concentre sur une propriété essentielle attendue pour un tel système, la confidentialité, le maintien du secret du programme et des données qu'il manipule. Une première contribution de ces travaux est une méthode de chiffrement d'un code, basée sur le graphe de flot de contrôle, rendant possible l'utilisation d'algorithmes de chiffrement par flots, légers et efficaces. Protéger les accès mémoires aux données d'un programme s'avère plus complexe. Dans cette optique, nous proposons l'utilisation d'un chiffrement homomorphe pour chiffrer les données stockées en mémoire et les maintenir sous forme chiffrée lors de l'exécution des instructions. Enfin, nous présenterons l'intégration de ces propositions dans une architecture de processeur et les résultats d'évaluation sur logique programmable (FPGA) avec plusieurs programmes d'exemples.

Title: Design and implementation of a microprocessor working with encrypted instructions and data
Keywords: Hardware security; Processor Design; Software Encryption; Stream cipher; Homomorphic encryption

Abstract. Embedded processors are today ubiquitous, dozen of them compose and orchestrate every technology surrounding us, from tablets to smartphones and a large amount of invisible ones. At the core of these systems, processors collect data, process it and interact with the outside world. As such, they are excepted to meet very strict safety and security requirements. From a security perspective, the task is even more difficult, since the user has a physical access to the device, allowing a wide range of specifically tailored attacks.

Confidentiality, both in terms of software code and data is one of the fundamental properties expected for such systems. The first contribution of this work is a software encryption method based on the control flow graph of the program. It allows the use of stream ciphers to provide lightweight and efficient encryption, suitable for constrained processors. The second contribution is a data encryption mechanism based on homomorphic encryption. With this scheme, sensible data remain encrypted not only in memory, but also during computations. Then, the integration and evaluation of these solutions on Field Programmable Gate Array (FPGA) with some example programs will be discussed.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France