Extensions cryptographiques pour processeurs embarqués

Fabrice Lozachmeur

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ BRETAGNE SUD

ÉCOLE DOCTORALE N° 644
Mathématiques et Sciences et Technologies de l'Information et de la Communication en Bretagne Océane
Spécialité : *Informatique et Architectures numériques*

Par
Fabrice LOZACHMEUR

Extensions cryptographiques pour processeurs embarqués

Thèse présentée et soutenue à Lorient, le 23 janvier 2024
Unité de recherche : Lab-STICC UMR 6285, Thales LAS France SAS
Thèse N° : 687

Rapporteurs avant soutenance :

Régis LEVEUGLE — Professeur des Universités, INP - Université Grenoble Alpes
Sébastien PILLEMENT — Professeur des Universités, Polytech - Nantes Université

Composition du Jury :

Président :	Sébastien PILLEMENT	Professeur des Universités, Polytech - Nantes Université
Examinateurs :	Sonia BELAÏD	Ingénieure en cryptographie, CryptoExperts
	Régis LEVEUGLE	Professeur des Universités, INP - Université Grenoble Alpes
Dir. de thèse :	Arnaud TISSERAND	Directeur de Recherche, CNRS

Invité(s) :

Nicolas TRANCHANT — Expert Cybersécurité, Thales LAS France

Remerciement

Je présente en premier lieu mes sincères remerciements à mon directeur de thèse, Arnaud Tisserand, qui m'a encadré tout au long de cette thèse et qui m'a fait partager ses observations avisées et ses précieux conseils. Je le remercie notamment pour la confiance qu'il m'a témoignée durant ces 3 années de thèse. Sa capacité d'analyse, son expérience et sa connaissance approfondie du domaine ont été d'une aide précieuse et ont contribué à l'aboutissement de cette thèse.

Je remercie à présent mes encadrants de Thalès. Mes premiers remercients sont à l'égard d'Arnaud Phelipot qui encadra mes travaux en 2021. Je tiens également à remercier Nicolas Tranchant, mon encadrant de 2022 jusqu'à la fin de ma thèse, pour ses remarques constructives et son travail de relecture. Je leur exprime toute ma gratitude pour tout ce qu'ils ont apporté à cette thèse.

Je tiens à remercier mon employeur, l'entreprise Thalès LAS France, de m'avoir donné l'opportunité de réaliser cette thèse. Je remercie également le laboratoire commun LATERAL (entre Thalès et le laboratoire Lab-STICC).

J'aimerais aussi adresser des remerciements à toute l'équipe d'ARCAD. Je remercie plus particulièrement tous les thésards qui ont partagé mon bureau.

Pour terminer, j'adresse des remerciements à ma famille pour m'avoir accompagné et soutenu tout au long de la thèse. Je tiens à remercier particulièrement mon frère, Gaëtan Lozachmeur, pour son soutien et ses précieux conseils.

Table des matières

Acronyme

Terme	Description	Pages
SPN	Substitution-permutation network	21
ECB	Electronic codebook block	22
CBC	Cipher block chaining	22
IV	Initialization vector	22
OFB	Output feedback	22
CTR	Counter mode	22
NIST	National institute of standards and technology	23
AES	Advanced Encryption Standard	23
SCA	Side-channel analysis	29
CMOS	Complementary metal-oxide-semiconductor	33
SPA	Simple power analysis	34
DPA	Differential power analysis	35
CPA	Correlation power analysis	38
MIA	Mutual information analysis	39
HO-SCA	Higher-order side-channel analysis	40
BM	Boolean masking	43
MPC	Multi party computation	43
DAG	Directed acyclic graph	44
TI	Threshold implementation	51
NI	Non-interference	52
SNI	Strong non-interference	52
PINI	Probe isolating non-interference	54
O-PINI	Output probe isolating non-interference	55
CMS	Consolidating masking schemes	56
DOM	Domain oriented masking	56
UMA	Unified masking approach	56
GLM	Generic low-latency masking	56
HPC	Hardware private circuits	56
BS	Bit slicing	60
ISE	Instruction set extension	69, 72
ISA	Instruction set architecture	69
CISC	Complex instruction set computer	69
RISC	Reduced instruction set computer	69
RF	Register file	75
PK	Proxy kernel	82

Terme	**Description**	**Pages**
HTIF	Host target interface	82
ISS	Instruction set simulator	82
CSR	Control and status registers	85
TVLA	Test vector leakage assessment	89
SNR	Signal-to-noise ratio	89
PRNG	Psendorandom number generator	97
TRNG	True random number generator	97
FSM	Finite state machine	97
MSB	Most significant bit	105

Notation

Terme	Description	Pages
Enc	Fonction de chiffrement	20
Dec	Fonction de déchiffrement	20
p	Message en clair	20
c	Message chiffré	20
k	Clé de chiffrement	20
$\mathrm{GF}(2^8)$	Corps fini de caractéristique 2, à 256 éléments, défini par le polynôme de réduction $P[X] = X^8 + X^4 + X^3 + X + 1$	24
R	Nombre de rondes	24
$\mathrm{GF}(2)$	L'unique corps fini à deux éléments	25
N	Nombre de traces	30
r	Une valeur aléatoire	41
n	Nombre de parts d'une variable masquée	42
$\hat{\mathbf{x}}$	Vecteur représentant un masquage de la variable x	42
$\hat{\mathbf{x}}[i]$	$i^{ème}$ part du masquage $\hat{\mathbf{x}}$	42
Masque	Procédure de masquage	42
$\mathsf{Démasque}$	Procédure de démasquage	42
d	Ordre de masquage	42
F	Les termes en police sans empattement désignent une implémentation, une fonction ou un graphe	43
$\hat{\mathsf{F}}$	Une implémentation masquée	43
$\mathbb{K}$	Un corps fini de caractéristique 2	44
$\mathcal{P}$	Les termes en police calligraphique désignent un ensemble	45
$[\cdot]_{\mathsf{reg}}$	L'opérande est stocké dans un registre	57
`rs1`	Indice du premier registre source dans le fichier de registres	71
`rs2`	Indice du deuxième registre source dans le fichier de registres	71
`rd`	Indice du registre destination dans le fichier de registres	71
`imm`	Valeur immédiate	71

Terme	**Description**	**Pages**
`reg`[`x`]	Valeur du registre d'indice `x` dans le fichier de registres.	75
d_{H}	Ordre de masquage en matériel	118
d_{S}	Ordre de masquage en logiciel	118
$\oplus$	Porte booléenne XOR	
$\otimes$	Porte booléenne AND	
$\neg$	Porte booléenne NOT	
$\mathcal{O}(\cdot)$	Notation grand O de Landau	
$\mathsf{rnd}(\cdot)$	Arrondi de l'opérande	
n.a.	Valeur non disponible	

Introduction

La *cryptographie* est, en particulier, la science de la protection des informations sensibles. La cryptographie permet notamment d'assurer la confidentialité en utilisant des cryptosystèmes. La cryptographie se divise en deux grandes catégories, la *cryptographie asymétrique* et la *cryptographie symétrique*. Un des cryptosystèmes symétriques les plus utilisés actuellement est le chiffrement AES.

Les implémentations de cryptosystèmes, même robustes mathématiquement, sont susceptibles de divulguer des informations via l'*observation de canaux auxiliaires* du circuit sur lequel s'exécute le cryptosystème (par exemple le temps de calcul, la consommation d'énergie, le rayonnement électromagnétique). Les attaques par *analyse des canaux auxiliaires* [KJJ99], abrégées en SCA pour «*side-channel analysis*» en anglais, exploitent de potentielles corrélations entre les *valeurs physiques mesurées* et les *opérations et opérandes* traités dans le circuit pour récupérer des données sensibles.

Le *masquage* est une contre-mesure couramment utilisée pour protéger les systèmes logiciels et/ou matériels contre certaines attaques SCA [Cha+99 ; GP99]. Le *masquage booléen* d'ordre d représente chaque variable sensible intervenant lors du calcul en $d + 1$ parts. La sécurité d'un schéma de masquage peut être prouvée dans certains *modèles théoriques d'attaquant*. Un schéma de masquage est par exemple sécurisé dans le *modèle de sondage* [ISW03] si un attaquant peut sonder d variables dans l'implémentation sans obtenir des informations sensibles. [Cha+99 ; PR13] ont montré que le nombre de traces nécessaires pour attaquer une implémentation masquée et sécurisée dans le modèle de sondage augmente avec l'ordre de masquage. Cependant, les implémentations purement logicielles du masquage imposent un surcoût important en termes de tailles de code et temps de calcul [GR17]. De plus, les implémentations masquées en logiciel n'atteignent pas toujours le niveau de sécurité attendu en raison de fuites liées à la micro-architecture des processeurs [Gao+19] ou d'un niveau de bruit trop faible. [BS21] ont ainsi réalisé une attaque sur des implémentations purement logicielles de AES masquées jusqu'à l'ordre 5.

Différentes techniques ont été proposées dans la littérature pour implémenter le chiffrement AES en logiciel. Une des techniques les plus efficaces est appelée le «*bit slicing*» [Bih97], abrégé en BS, et permet de produire des implémentations de chiffrement à haut débit grâce au parallélisme de données dans un même mot machine. Les implémentions BS peuvent être masquées en répartissant les parts dans différents registres du processeur. Une autre solution, appelée «*share slicing*», consiste à placer toutes les parts d'un bit masqué dans un même mot mémoire.

Des *extensions de jeux d'instructions*, abrégées en ISE pour «*instruction set extension*» en anglais, ont été proposées pour implémenter le masquage de façon efficace et sécurisée. Une ISE dédiée au masquage permet d'accélérer les protections en maintenant de la flexibilité, d'augmenter la sécurité en prenant en compte la micro-architecture des processeurs et de rendre l'utilisation du masquage plus simple. Contrairement aux implémentations logicielles du mas-

quage, les ISE dédiées au masquage de l'état de l'art sont cependant limitées aux petits ordres de masquage.

Nos contributions Dans cette thèse, nous avons développé différentes ISE dédiées au masquage pour le processeur RISC-V CV32E40P afin de protéger certains cryptosystèmes contre des attaques SCA en utilisant un masquage d'ordre élevé :

1. Nous avons dans un premier temps mis en place un environnement expérimental permettant de compiler un code source, simuler son exécution sur une architecture RISC-V et de synthétiser le cœur CV32E40P. Plusieurs solutions matérielles et logicielles de l'état de l'art ont ensuite été réimplantées sur le cœur CV32E40P pour analyser leurs performances. Un simulateur a été étendu pour générer des traces simulées de consommation. Les traces simulées sont utilisées pour évaluer la résistance d'une implémentation contre des attaques SCA.
2. Nous proposons une ISE dédiée au masquage, que nous nommons ISE1, permettant d'implémenter de façon sécurisée et efficace des implémentations «*share slicing*» de différents cryptosystèmes. L'ISE1 a été implémentée pour des ordres de masquage élevés, allant jusqu'à l'ordre 31. L'ordre de masquage de l'ISE1 est fixé au moment de la synthèse. Une grande flexibilité est ainsi possible au moment de la conception pour divers compromis coût/performance. L'ISE1 a été implantée sur le processeur RISC-V CV32E40P pour réaliser des évaluations de performance et de surface.
3. Nous proposons une nouvelle ISE dédiée au masquage supportant des ordres de masquage $d \in \{1, 3, 7, 15\}$, que nous nommons ISE2, qui étend l'ISE1 pour réduire le nombre de blocs indépendants chiffrés en parallèle. Nous présentons de nouvelles implémentations «*share slicing*» de AES permettant de chiffrer un bloc à la fois. L'utilisation de tous les modes de chiffrement devient alors possible. L'ordre de masquage de l'ISE est fixé au moment de la synthèse. L'ISE2 a été implantée sur le processeur RISC-V CV32E40P pour réaliser des évaluations de performance et de surface.
4. Une implémentation masquée utilisant les ISE précédentes est sécurisée contre un attaquant pouvant sonder d bits dans l'exécution complète. Pour augmenter la sécurité apportée par le masquage, nous proposons une nouvelle ISE dédiée au masquage, que nous nommons ISE3, qui modifie l'ISE1 pour obtenir une sécurité dans le *modèle de sondage par régions*, où chaque région comprend une instruction de l'ISE3. L'ISE3 masquée à l'ordre d permet alors à l'attaquant de sonder $d/2$ bits par instruction sans compromettre la sécurité du schéma de masquage.
5. Nous proposons une solution de masquage matériel/logiciel pour masquer à des ordres élevés avec une grande flexibilité au moment de la synthèse et au moment de la compilation logicielle pour divers compromis coût/performance. Un premier niveau de masquage est implémenté en matériel en utilisant l'ISE1 dont l'ordre de masquage est fixé au moment de la synthèse. Un masquage logiciel utilise ensuite les instructions masquées de l'ISE1 pour atteindre divers ordres supérieurs.

Notre solution de masquage matériel/logiciel a été l'objet de notre publication [LT23] dans la conférence MWSCAS.

Organisation de la thèse Le manuscrit de thèse se divise en deux parties consacrées à l'état de l'art et la description de nos contributions. Le chapitre 1 aborde quelques principes

de base de cryptographie symétrique. Le chapitre 2 introduit les attaques SCA. Le chapitre 3 décrit la contre-mesure de masquage. Les différentes techniques d'implémentations logicielles du chiffrement AES sont présentées dans le chapitre 4. Le chapitre 5 établit un état de l'art des ISE dédiées au masquage. Nos contributions sont détaillées dans les chapitres 6, 7, 8, 9 et 10. Le chapitre 6 présente notre environnement expérimental ainsi que la réimplantation sur le cœur CV32E40P de certaines solutions de l'état de l'art. Le chapitre 7 décrit l'ISE1 permettant d'implémenter de façon sécurisée et efficace des implémentations «*share slicing*» de cryptosystèmes. Le chapitre 8 présente l'ISE2 qui étend l'ISE1 pour réduire le nombre de blocs indépendants chiffrés en parallèle. Le chapitre 9 décrit l'ISE3 qui modifie l'ISE1 pour obtenir une sécurité dans le modèle de sondage par régions. Le chapitre 10 présente notre solution flexible de masquage matériel/logiciel.

Première partie

État de l'art

Chapitre 1

Rappels de cryptographie

Dans ce chapitre, nous abordons quelques principes de base de cryptographie symétrique. Pour une description plus complète de la cryptographie, le lecteur peut se rapporter au livre [PP10].

Une introduction à la cryptographie est réalisée dans la section 1.1. La section 1.2 introduit le chiffrement symétrique AES.

1.1 Introduction à la cryptographie

1.1.1 Le rôle de la cryptographie dans la sécurité de l'information

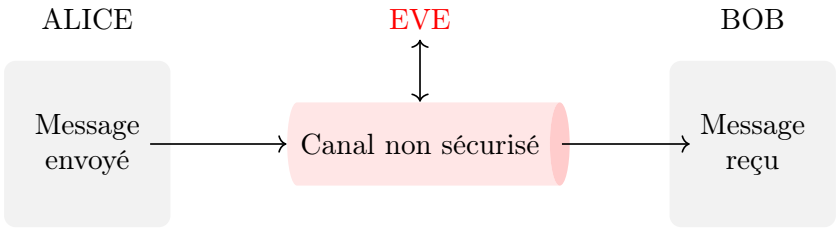


FIGURE 1.1 – Schéma d'une communication entre Alice et Bob.

La *cryptographie* est, en particulier, la science de la protection des informations sensibles. L'objectif de la cryptographie moderne est de fournir certaines fonctionnalités de sécurité pour échanger des informations sur un canal de communication non sécurisé. Comme illustré sur la figure 1.1, on suppose que deux interlocuteurs, communément appelés Alice et Bob, souhaitent échanger de façon sécurisée des données sensibles à travers un canal de communication. Mais ils ne peuvent pas directement s'échanger les données sensibles, car un attaquant espionne le canal de communication. Dans la communauté cryptographique, l'attaquant qui ne fait qu'observer s'appelle Eve. La cryptographie moderne utilise les mathématiques et la théorie de l'information pour créer des protocoles qui garantissent certaines propriétés de sécurité telles que :

— la *confidentialité* : un message est inintelligible pour quiconque sauf pour Alice et Bob,
— l'*intégrité* : Bob peut vérifier que le message n'a pas été modifié lors de son transfert,
— l'*authenticité* : Bob peut vérifier l'identité de l'expéditeur d'un message,
— la *non-répudiation* : Alice ne peut nier avoir écrit et envoyé un message.

1.1.2 Assurer la confidentialité

Pour assurer la confidentialité des données échangées et empêcher Eve de récupérer de l'information, Alice et Bob utilisent un *système de chiffrement*, aussi appelé *cryptosystème*, comme par exemple AES qu'on va étudier à la section 1.2. Alice va chiffrer le message à transmettre de telle façon que Bob puisse le déchiffrer, mais pas Eve (voir figure 1.2). Tout d'abord, ils se mettent d'accord sur une fonction de chiffrement Enc et une fonction de déchiffrement Dec. Puis, Alice va chiffrer le message en *clair* à transmettre, noté p pour «*plaintext*» en anglais, en un *chiffré* c en utilisant une *clé de chiffrement* k_A. Ensuite, Alice transmet le chiffré c à Bob à travers le canal de communication. Lorsque Bob reçoit le chiffré c, il utilise une *clé de déchiffrement* k_B pour le déchiffrer et retrouver le clair p. Alice et Bob réalisent le chiffrement et le déchiffrement dans leur *zone de confiance* dans laquelle tout ce qui est manipulé n'est pas visible par Eve. Eve peut obtenir le chiffré en espionnant le canal de communication. Mais si le système de chiffrement est robuste, alors il est calculatoirement impossible de déterminer le clair à partir du chiffré sans la clé de déchiffrement. Selon un des principes de Kerckhoffs [Ker83], la sécurité d'un système de chiffrement ne doit pas reposer sur la dissimulation de son fonctionnement, mais uniquement sur les valeurs des clés et une complexité calculatoire suffisamment grande pour éviter qu'une attaque puisse réussir. En effet, si un certain nombre de personnes souhaitent communiquer en toute sécurité, elles ne peuvent espérer cacher le système qu'elles utilisent.

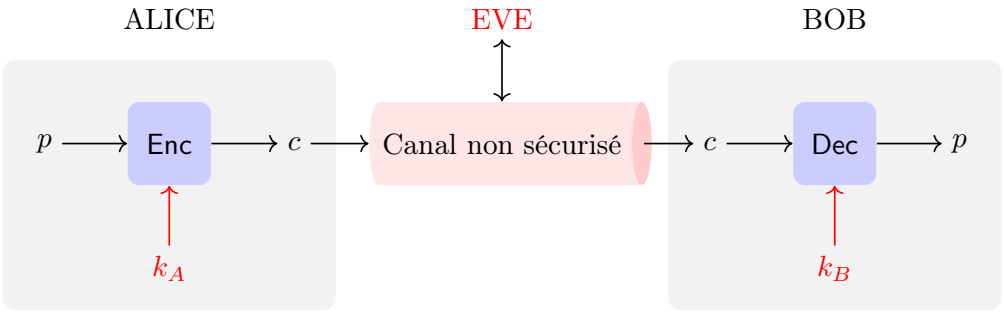


FIGURE 1.2 – Système de chiffrement.

Cryptographie symétrique et asymétrique Il existe deux grandes catégories de cryptographie, la *cryptographie asymétrique* (ou à clé publique) et la *cryptographie symétrique* (ou à clé secrète). Les cryptosystèmes symétriques utilisent la même clé secrète k pour le chiffrement et le déchiffrement. Ce type de cryptosystèmes est généralement assez performant, mais nécessite que Alice et Bob se soient préalablement échangés de façon sécurisée la clé secrète. Le concept de cryptographie asymétrique a été proposé par [DH76] dans le but de résoudre le problème de distribution des clés posé par la cryptographie symétrique. Dans les cryptosystèmes asymétriques, les clés de chiffrement et de déchiffrement sont distinctes et l'une des clés ne peut pas se déduire de l'autre. On peut donc rendre l'une des deux publique tandis que l'autre reste privée. Tout le monde peut chiffrer en utilisant la clé publique, mais seul le propriétaire de la clé privée peut déchiffrer. En pratique, on utilise d'abord un cryptosystème asymétrique pour échanger la clé secrète et ensuite un cryptosystème symétrique pour l'échange des données. Dans cette thèse, on ne s'intéressera qu'à la cryptographie symétrique dont fait partie le cryptosystème AES.

Attaques calculatoires La cryptographie symétrique est la cible d'attaques dont l'objectif est de deviner la clé secrète utilisée ou des informations sur le clair. Les *attaques calculatoires* essayent de déterminer des données sensibles en utilisant uniquement la connaissance de l'algorithme, ainsi que certaines paires de clairs-chiffrés. Un cryptosystème symétrique est alors sécurisé s'il est *calculatoirement impossible* de déterminer des données sensibles, même en utilisant des implémentations informatiques extrêmement performantes. Un exemple d'attaque est la *recherche exhaustive de clé*, appelée également *attaque force brute*, qui permet de récupérer la clé secrète en utilisant une seule paire de clair-chiffré. Le clair est chiffré avec toutes les clés possibles jusqu'à obtenir le chiffré attendu. Pour rendre la recherche exhaustive de clé informatiquement impossible, un chiffrement doit utiliser un espace de clé suffisamment grand. Par exemple, pour attaquer une clé de 128 bits, une recherche exhaustive nécessite d'effectuer 2^{128} chiffrements qui est considérée comme calculatoirement infaisable à ce jour. D'autres attaques plus évoluées ont été proposées comme par exemple la *cryptanalyse différentielle* [BS91] et la *cryptanalyse linéaire* [Mat93].

Propriétés pour concevoir un cryptosystème symétrique sécurisé [Sha49] a défini deux propriétés mathématiques importantes pour concevoir des cryptosystèmes symétriques sécurisés, il s'agit de la *confusion* et de la *diffusion*. La confusion reflète la complexité de la relation entre le clair et le chiffré. Dans un cryptosystème raisonnable, la confusion doit être forte pour qu'on ne puisse pas facilement retrouver le clair ou la clé secrète à partir du chiffré. Certaines fonctions non-linéaires sont suffisamment complexes pour garantir un peu de confusion. La diffusion reflète la capacité du cryptosystème à mélanger les bits en cours de chiffrement. Pour qu'un cryptosystème soit robuste, une petite modification sur le clair doit engendrer une modification importante sur le chiffré. La diffusion peut être apportée par certaines fonctions linéaires.

1.1.3 Chiffrement par bloc

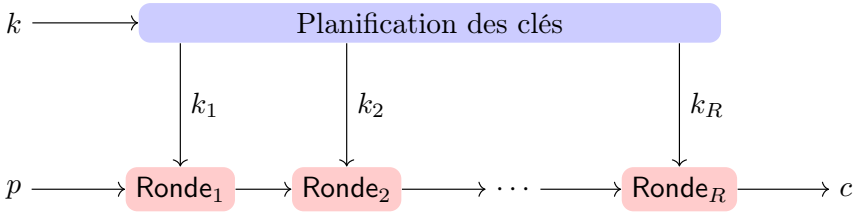


FIGURE 1.3 – Chiffrement par bloc itératif.

Un *chiffrement par bloc* est un cryptosystème permettant de chiffrer des données de taille fixe (comme par exemple 128 bits) appelées *blocs*. Un *chiffrement par bloc itératif* est un type de chiffrement par bloc dans lequel le chiffré d'un bloc est obtenu en itérant une transformation qui apporte de la confusion et de la diffusion à chaque itération. Les itérations sont aussi appelées des *rondes* ou *tours*. Chaque ronde Ronde_i prend en entrée la sortie de la ronde précédente et une *clé de ronde* k_i. Les clés de ronde sont dérivées de la clé principale k par un algorithme de *planification des clés*. Une représentation d'une telle construction est donnée dans la figure 1.3. Le *schéma de permutation substitution*, abrégé en SPN pour «*substitution-permutation network*» en anglais, est une architecture de chiffrement par bloc itératif couramment utilisée.

Une ronde se compose de trois étapes successives. Premièrement, l'étape de *mélange des clés* consiste à additionner la clé de ronde et l'entrée de la fonction de ronde. Deuxièmement, l'étape de *substitution* apporte de la confusion en appliquant une fonction non linéaire. Troisièmement, l'étape de *permutation* apporte de la diffusion en mélangeant les bits.

Mode de chiffrement Pour chiffrer des textes en clair de taille arbitraire, le texte en clair est divisé en blocs puis chaque bloc est chiffré selon un *mode de chiffrement*. La communauté cryptographique utilise différents modes de chiffrement pour appliquer de manière sécurisée un chiffrement par bloc sur plusieurs blocs consécutifs. Certains de ces modes de chiffrement sont décrits dans la figure 1.4. Le *mode ECB*, abrégé de «*electronic codebook block*» en anglais, consiste à chiffrer chaque bloc indépendamment des autres. ECB est généralement considéré comme non sécurisé, car des blocs en clair identiques produisent des blocs chiffrés identiques. Cela peut être exploité par un attaquant pour obtenir des informations sur les blocs en clair. Le *mode CBC*, abrégé de «*cipher block chaining*» en anglais, rend le bloc chiffré en cours de chiffrement dépendant des blocs en clair précédemment chiffrés. Le bloc chiffré est obtenu en chiffrant le résultat du XOR entre le bloc en clair et le bloc chiffré précédent. Initialement, le XOR est effectué avec un vecteur d'initialisation, abrégé en IV. Le *mode OFB*, abrégé de «*output feedback*» en anglais, transforme un chiffrement par bloc en un chiffrement par flux. Pour générer le bloc chiffré, le XOR est réalisé entre le bloc en clair et la sortie du chiffrement par flux. Étant donné que le bloc en clair n'est utilisé que pour le XOR final, les opérations du chiffrement par bloc peuvent être effectuées à l'avance. Le *mode compteur*, abrégé en CTR, transforme un chiffrement par bloc en un chiffrement par flux en chiffrant les valeurs successives d'un compteur. Le compteur peut être n'importe quelle fonction produisant une séquence ne se répétant pas sur une longue période. Le mode CTR a des caractéristiques similaires à OFB, mais permet également le chiffrement des blocs en clair en parallèle. Le tableau 1.1 résume les caractéristiques des différents modes de chiffrement.

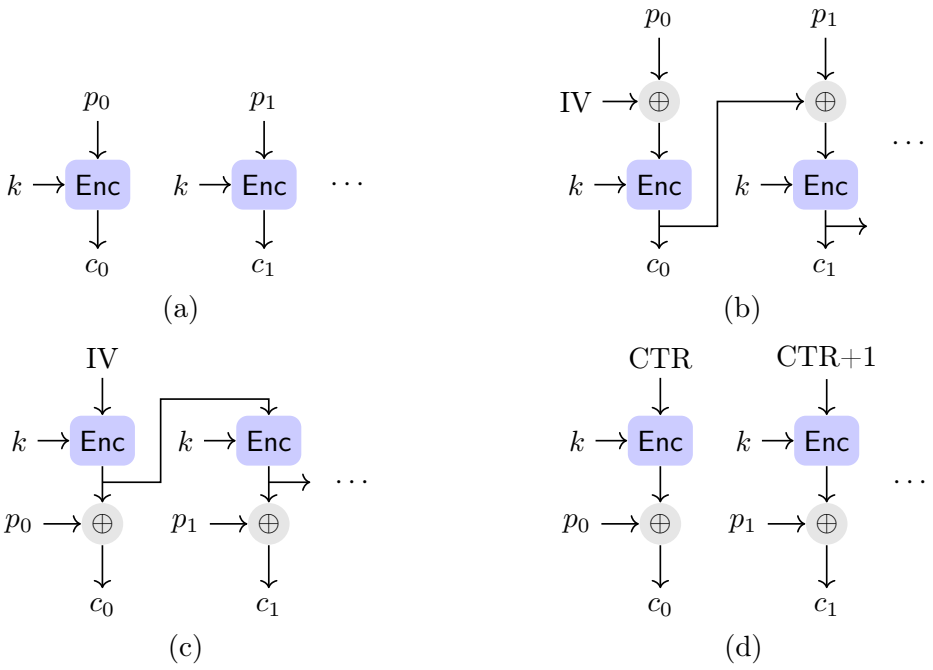


FIGURE 1.4 – (a) Mode ECB ; (b) Mode CBC ; (c) Mode OFB ; (d) Mode CTR.

Mode	Sécurisé	Précalcul	Parallèle
ECB	✗	✗	✓
CBC	✓	✗	✗
OFB	✓	✓	✗
CTR	✓	✓	✓

TABLE 1.1 – Caractéristiques des différents modes de chiffrement.

1.2 AES

1.2.1 Histoire de AES

Dans les années 90, on utilisait un algorithme de chiffrement symétrique, appelé DES, dont on voyait apparaître des limites de sécurité vis-à-vis de certaines attaques mathématiques. L'organisme de normalisation américain, le NIST, a alors organisé en 1997 un concours ouvert internationalement pour sélectionner un nouvel algorithme de chiffrement symétrique qui sera appelé *Advanced Encryption Standard* [NIS01], abrégé en AES. Le but était de le standardiser pour le diffuser à l'échelle mondiale. L'algorithme de chiffrement Rijndael [DR02], créé par des chercheurs européens, remporta le concours en 2000 et devint le chiffrement AES [1]. AES est un chiffrement par bloc itératif avec des blocs de 128 bits. Les tailles de clé possibles sont 128, 192 et 256 bits et permettent d'atteindre différents niveaux de sécurité. L'algorithme AES est très flexible et peut être implémenté très efficacement en logiciel et en matériel. Le chiffrement AES peut être implémenté en logiciel de façon efficace, à la fois sur de petits processeurs 8 bits et sur des processeurs 32 bits plus performants. Les implémentations matérielles dédiées à AES peuvent être adaptées en taille et en latence pour répondre à des contraintes spécifiques.

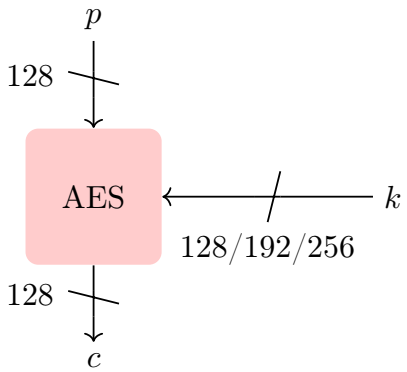


FIGURE 1.5 – Le chiffrement AES.

1. À `https://competitions.cr.yp.to/aes.html`, il y a toutes les explications sur le déroulement du concours.

1.2.2 Fonction de chiffrement et de déchiffrement

Toutes les opérations de l'algorithme AES transforment un bloc de données appelé *état*. L'état de AES, illustré à la figure 1.6, est une matrice de taille 4×4 dont les éléments sont des octets, représentant des valeurs dans le corps fini GF(2^8) défini par le polynôme de réduction $P[X] = X^8 + X^4 + X^3 + X + 1$. Au tout début, l'état est le bloc en clair à chiffrer, découpé en octets et rangé par colonne dans la matrice. Pendant le chiffrement, AES va itérer des transformations de l'état avec des fonctions qu'on va définir dans la suite. À la toute fin, l'état contient les octets du bloc chiffré rangés par colonne dans la matrice. Le déchiffrement est similaire au chiffrement, l'état est initialisé avec le texte chiffré et les transformations inverses sont appliquées dans l'ordre inverse.

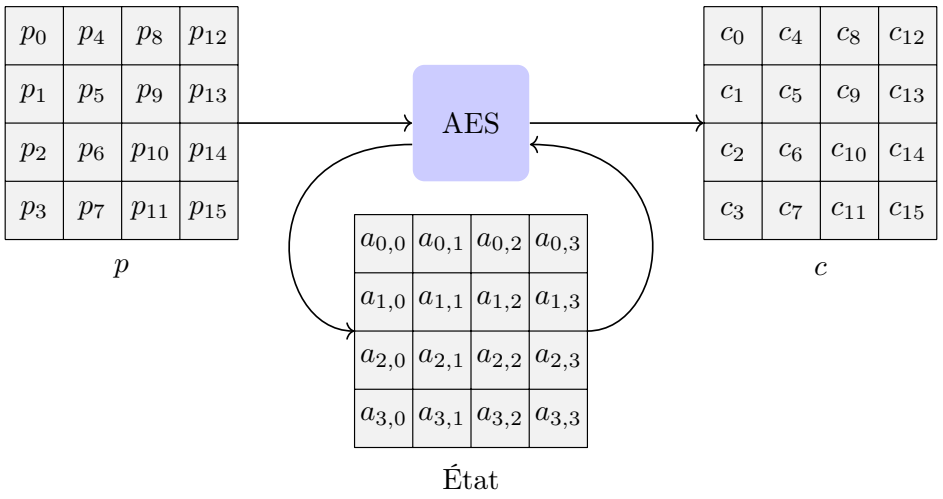


FIGURE 1.6 – L'état de AES.

Fonction de chiffrement de AES La fonction de chiffrement de AES est illustré par la figure 1.7. AES est un chiffrement par bloc itératif de type SPN. Le nombre de rondes R dépend de la taille des clés : 10 rondes pour une clé de 128 bits, 12 rondes pour une clé de 192 bits et 14 rondes pour une clé de 256 bits. Dans une ronde, il y a 4 sous-fonctions. Il y a tout d'abord SubBytes qui est non linéaire et apporte de la confusion. Puis ShiftRows et MixColumns qui sont linéaires et apportent de la diffusion. Et enfin AddRoundKey qui mélange l'état courant avec la clé de ronde de la ronde en cours. Toutes les clés de rondes sont dérivées de la clé de chiffrement par un algorithme de planification de clé. Pour des raisons de performance et de sécurité, on applique AddRoundKey avant la toute première ronde, et dans la dernière ronde, MixColumns est omise.

Fonction de déchiffrement de AES La fonction de déchiffrement de AES peut être effectuée en utilisant l'inverse des quatre sous-fonctions SubBytes, ShiftRows, MixColumns, et AddRoundKey, appliquées dans l'ordre inverse de la fonction de chiffrement. Les clés de ronde sont également prises dans l'ordre inverse. Étant donné que les opérations SubBytes et ShiftRows commutent, et que MixColumns est omise de la ronde finale, il existe une description du déchiffrement de AES qui imite la séquence des opérations pendant le chiffrement.

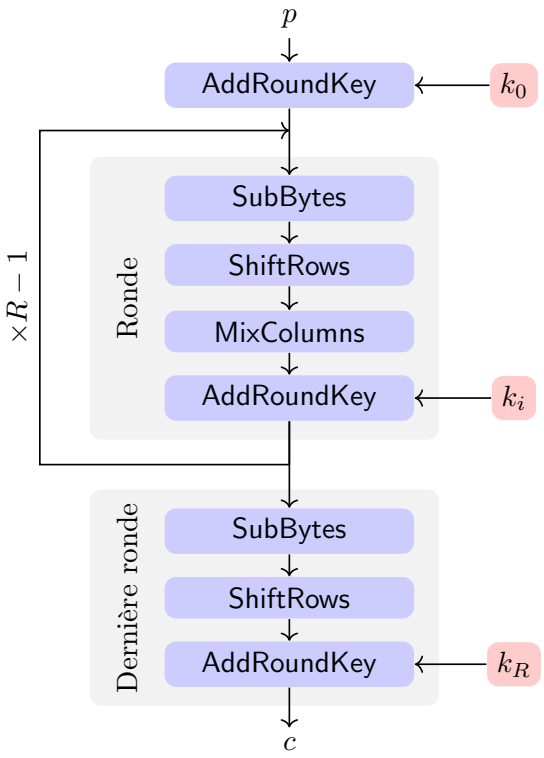


FIGURE 1.7 – Fonction de chiffrement de AES.

1.2.3 Sous-fonctions de AES

SubBytes La première sous-fonction de la ronde est SubBytes (voir figure 1.8). SubBytes applique une table de substitution, appelée généralement SBOX, sur chaque octet de l'état. La fonction SBOX est une inversion dans GF(2^8) suivie d'une application affine sur GF(2). SBOX est non-linéaire et apporte de la confusion. La fonction SBOX peut être implémentée en logiciel sous la forme d'une table précalculée contenant 256 valeurs, ou calculée au moment de l'exécution en utilisant certaines instructions de base du processeur. Pour les implémentations matérielles, de nombreuses conceptions optimisées exploitant les structures mathématiques de la fonction SBOX ont été proposées [Can05 ; BP12].

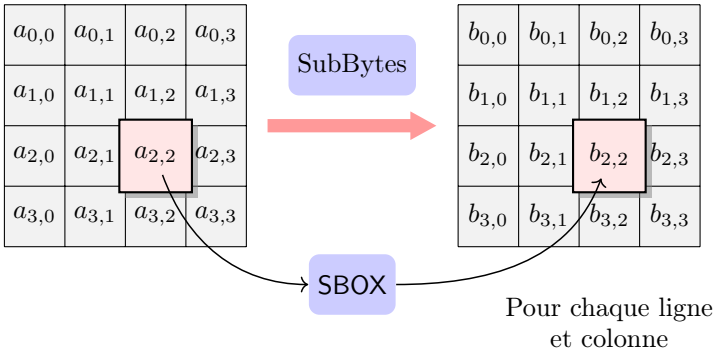


FIGURE 1.8 – Sous-fonction SubBytes.

ShiftRows La deuxième sous-fonction de la ronde est ShiftRows (voir figure 1.9). ShiftRows décale cycliquement les lignes de l'état. La première ligne n'est pas décalée, la deuxième est décalée de 1 vers la gauche, la troisième de 2 et la quatrième de 3. Elle apporte de la diffusion. L'implémentation en logiciel de ShiftRows consiste en un réadressage des octets de l'état et peut être effectuée en combinaison de la SBOX. En matériel, ShiftRows peut être implémentée comme un simple câblage.

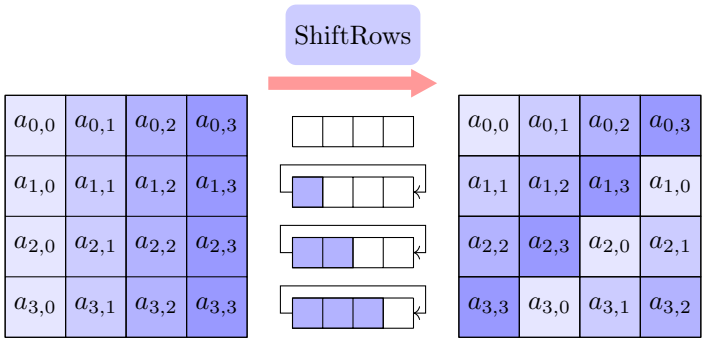


FIGURE 1.9 – Sous-fonction ShiftRows.

MixColumns La troisième sous-fonction est MixColumns (voir figure 1.10). MixColumns multiplie chaque colonne de l'état par la matrice M définie par :

$$\mathrm{M} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

Les bits dans une colonne sont mélangés par du calcul pour former la nouvelle colonne. C'est une fonction linéaire sur GF(2^8) qui apporte de la diffusion. MixColumns peut être implémentée en logiciel en calculant des multiplications par $X \in \mathrm{GF}(2^8)$, réalisées avec un certain nombre d'instructions de décalage et de XOR, ou par une table pré-calculée. Les implémentations matérielles de MixColumns ont tendance à être très efficaces car elles sont constituées principalement d'un nombre relativement faible de portes XOR.

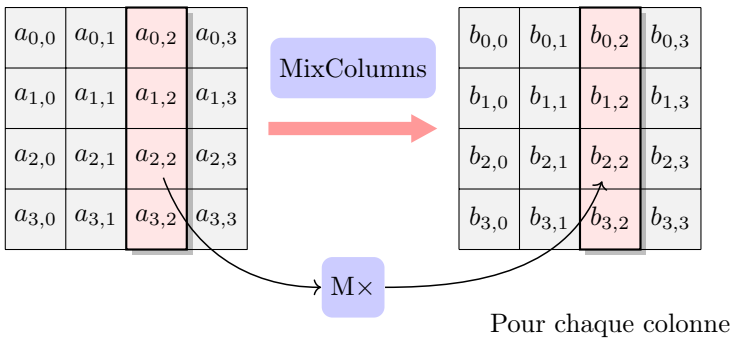
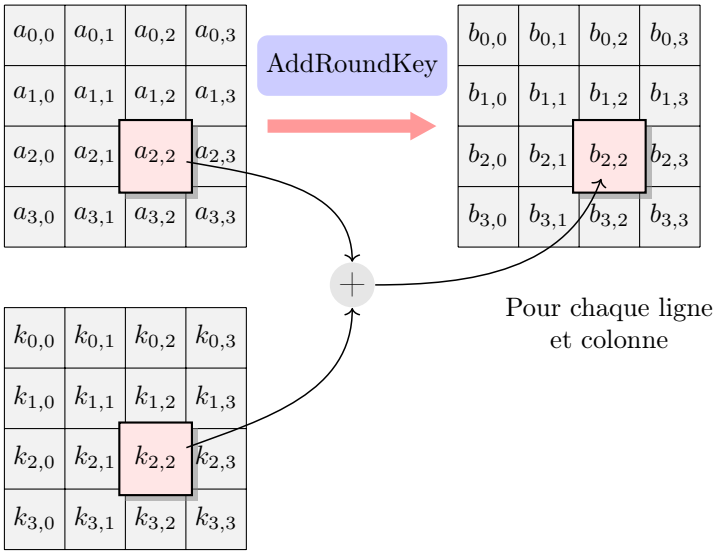


FIGURE 1.10 – Sous-fonction MixColumns.

AddRoundKey La dernière sous-fonction est AddRoundKey qui est illustrée par la figure 1.11. AddRoundKey réalise l'addition dans GF(2^8) de chaque octet de l'état avec l'octet correspondant de la clé de ronde de la ronde courante. Elle consiste simplement en XOR bit à bit entre l'état et la clé de ronde.



k: clé de ronde

FIGURE 1.11 – Sous-fonction AddRoundKey.

Chapitre 2

Attaques par analyse de canaux auxiliaires

Les cryptosystèmes modernes sont mathématiquement sécurisés contre les attaques calculatoires. Cependant, l'implémentation d'un cryptosystème sur un dispositif physique est susceptible de laisser fuir de l'information. Les *attaques physiques* exploitent l'implémentation physique des cryptosystèmes afin d'extraire des données sensibles. Depuis leur introduction, cette famille d'attaques suscite un grand intérêt à la fois dans la communauté académique et dans la communauté industrielle. En particulier, ce type d'attaques a un impact très important sur la sécurité des systèmes embarqués. Les attaques physiques sont divisées en deux catégories principales : les attaques par *analyse de canaux auxiliaires* [Koc96], abrégées en SCA pour «*side-channel analysis*» en anglais, et les *attaques par faute* [BS97]. Les attaques SCA observent et analysent les fuites physiques pour déterminer des données sensibles. Les *attaques par faute* perturbent le calcul cryptographique en générant des fautes dans le circuit afin de déduire des informations sensibles ou passer outre des protections. Les perturbations peuvent être des variations de tension ou d'horloge, des impulsions électromagnétiques ou encore des impulsions lasers. Dans la suite de cette thèse, nous nous intéresserons uniquement aux attaques SCA.

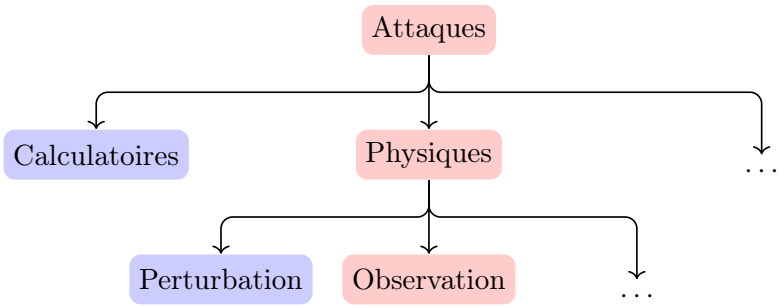


FIGURE 2.1 – Classification des attaques.

La section 2.1 introduit les attaques SCA. La section 2.2 décrit les attaques par observation du temps d'exécution. La section 2.3 présente les attaques par observation de la consommation d'énergie, ainsi que les principales contre-mesures.

2.1 Introduction aux attaques SCA

L'implémentation d'un cryptosystème sur un dispositif physique est susceptible de laisser fuir de l'information à travers des mesures physiques appelées *canaux auxiliaires*. Ces mesures physiques peuvent dépendre des opérations effectuées et des données manipulées. Une attaque SCA exploite alors certaines corrélations entre des valeurs physiques mesurées et des valeurs manipulées pour retrouver des variables sensibles. La SCA est une attaque utilisée depuis longtemps, comme par exemple pour ouvrir les coffres-forts en écoutant le son émis par les mouvements internes de la serrure. L'application de la SCA aux implémentations électroniques est plus récente. La première référence d'une attaque SCA ciblant une implémentation électronique date de 1943, lorsqu'un chercheur des laboratoires Bell remarqua que les émissions électromagnétiques d'un dispositif cryptographique dépendait des valeurs manipulées. Il essaya alors de récupérer, plus ou moins directement, le message en clair en exploitant les émissions électromagnétiques. Cette découverte a conduit le gouvernement américain à lancer dans les années 50 le programme TEMPEST [Fri72] pour exploiter les émissions électromagnétiques de différents appareils électroniques. Plusieurs attaques SCA ont ensuite été proposées, mais il faudra attendre 1996 pour qu'une attaque SCA contre une implémentation cryptographique soit publiée [Koc96]. [Koc96] ont montré comment les implémentations classiques de RSA et d'autres cryptosystèmes à clé publique pouvaient être efficacement attaquées en exploitant les variations du temps de calcul. Depuis, les attaques SCA se sont avérées très efficaces dans la pratique pour attaquer une large gamme de cryptosystèmes, notamment DES, AES, RSA, ECC ainsi que les chiffrements post-quantiques. La SCA est devenue aujourd'hui une branche à part entière de la recherche en cryptographie et en sécurité des systèmes embarqués.

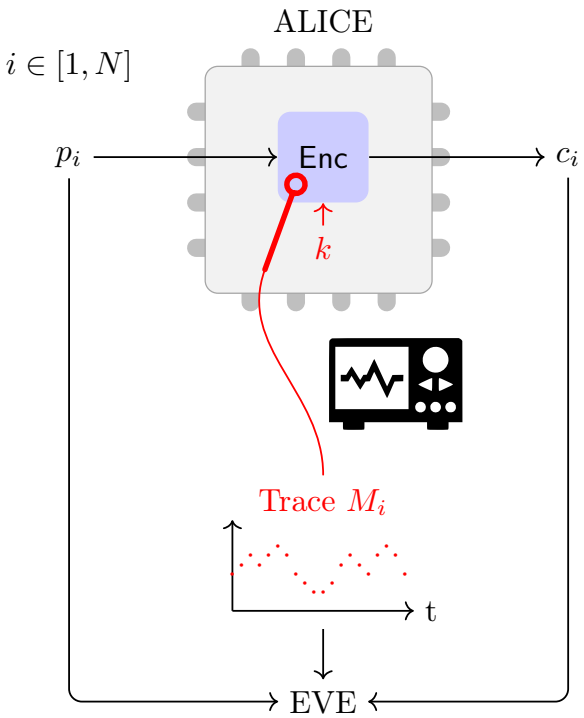


FIGURE 2.2 – Attaque par observation. N désigne le nombre de traces.

Description de l'attaque Pendant une attaque SCA, l'attaquant peut chiffrer plusieurs messages en clair avec une même clé secrète k. Pendant le chiffrement de chaque clair, l'attaquant mesure un ou plusieurs canaux auxiliaires, pour obtenir une *trace de mesures physiques* (voir figure 2.2). L'attaquant a alors accès aux messages en clair et/ou aux messages chiffrés associés, ainsi qu'aux traces de mesures physiques correspondantes. L'attaquant analyse ensuite certaines *corrélations* entre les traces de mesures physiques et les valeurs manipulées. Une analyse directe des traces de mesures physiques est parfois suffisante pour déterminer des variables sensibles. Lorsque le bruit est trop important pour mener une analyse directe, des *méthodes statistiques* [KJJ99] sont utilisées ou des techniques plus sophistiquées à base d'*intelligence artificielle* [MDP19]. Les attaques SCA nécessitent le plus souvent un *accès physique* au circuit électronique exécutant le cryptosystème. Cela se produit en pratique pour les implémentations cryptographiques embarquées. Par exemple, lorsqu'une carte bancaire intervient dans l'exécution d'un paiement, le client insère la carte dans le terminal du commerçant et saisit son code PIN. Le terminal dispose alors d'un accès physique à la carte et est en mesure d'interroger la carte pour différents calculs cryptographiques.

Les différents canaux auxiliaires Il existe différents canaux auxiliaires permettant d'attaquer un cryptosystème. Un des premiers canaux auxiliaires utilisé dans la littérature est le *temps d'exécution* du cryptosystème [Koc96]. [KJJ99] ont ensuite exploité l'évolution de la *consommation d'énergie* d'un circuit électronique pendant l'exécution d'un cryptosystème. La consommation d'énergie est mesurée en utilisant une sonde de courant qui agit un peu comme une résistance en série. Par la suite, [GMO01 ; QS01] ont montré que le *rayonnement électromagnétique* pouvait être exploité pour attaquer un cryptosystème. L'attaquant capte les émissions électromagnétiques en plaçant une sonde électromagnétique au dessus du circuit électronique. La principale différence avec la consommation d'énergie est la *localité de la mesure*. Alors que la consommation du circuit électronique est produite par l'ensemble de ses composants, une sonde électromagnétique peut être placée sur un composant spécifique et fournir des informations uniquement par l'élément situé en dessous de la sonde. Plus récemment, d'autres canaux auxiliaires ont été exploités, tels que les *émissions de photons* [Krä+13], l'*acoustique* [GST14] ou la *température* [HS13].

2.2 Attaques par observation du temps d'exécution

[Koc96] a réalisé en 1996 une attaque SCA exploitant les différences de temps d'exécution de certains calculs cryptographiques. Afin d'illustrer le fonctionnement de cette attaque, rappelons l'exemple classique de vérification d'un code PIN. On considère l'algorithme 1 qui vérifie le code PIN chiffre par chiffre. Si un chiffre est incorrect, l'algorithme s'arrête. En mesurant le temps mis par l'algorithme pour répondre à un code PIN donné, l'attaquant peut déduire le code PIN correct. En effet, l'attaquant peut essayer toutes les valeurs différentes pour le premier chiffre du code PIN. Une valeur erronée arrête l'algorithme alors que la valeur correcte engendre la comparaison du chiffre suivant. En comparant le temps d'exécution nécessaire pour répondre à chaque requête, l'attaquant peut déduire la valeur du premier chiffre du code PIN. L'attaquant peut ensuite attaquer le chiffre suivant de la même manière. En itérant cette technique, l'attaquant obtient le code PIN complet avec une complexité linéaire au lieu d'exponentielle.

Algorithme 1 Vérification de code PIN

Entrée: $S[.]$ un vecteur de taille k contenant le code PIN saisi ; $C[.]$ un vecteur de taille k contenant le code PIN correct.
Sortie: PIN valide ou invalide

```
pour i = 1 à k faire
    si S[i] ≠ C[i] alors
        retourne PIN invalide
    fin si
fin pour
retourne PIN valide
```

Un autre exemple classique d'analyse du temps d'exécution est l'attaque de [Koc96] contre RSA basée sur l'exponentiation modulaire décrite dans l'algorithme 2. L'exponentiation modulaire réalise une boucle qui requiert plus ou moins d'opérations en fonction de l'exposant secret manipulé. En effet, si le bit de l'exposant manipulé vaut 0, seule l'élévation au carré sera effectuée, alors que si le bit vaut 1, une multiplication supplémentaire est réalisée. Ainsi, le temps global du calcul de RSA dépend du nombre de bits à 1 de l'exposant, ce qui permet de récupérer l'exposant secret.

Algorithme 2 Exponentiation modulaire utilisant la méthode «*Square and Multiply*»

Entrée: $a, m \in \mathbb{N}$,$e = (e_{k-1}, \ldots, e_0)$
Sortie: $x = a^e \mod m$

```
x ← 1
pour i = k − 1 à 0 faire
    x ← x^2 mod m
    si e[i] = 1 alors
        x ← x × a mod m
    fin si
fin pour
retourne x
```

Depuis les travaux de [Koc96], de nombreuses autres attaques exploitant le temps d'exécution ont été développées, telles que [HH98 ; Dhe+98]. Les attaques par analyse du temps d'exécution sont également rendues possibles par certains mécanismes d'optimisation des processeurs modernes tels que le *cache* [Tsu+03 ; Ber05 ; BM06] ou l'*unité de prédiction de branchement* [AKS07]. Pour protéger un cryptosystème de l'analyse du temps d'exécution, une contre-mesure consiste à implémenter des cryptosystèmes s'exécutant en *temps constant* [OST05 ; Kön08], c'est-à-dire tel que le temps d'exécution ne varie pas, ou tout au moins que sa variation ne dépende pas de valeurs sensibles.

2.3 Attaques par observation de la consommation

Dans cette section, nous décrivons les attaques SCA exploitant la consommation d'énergie des circuits numériques.

2.3.1 Origine de la fuite

Consommation d'énergie dans les circuits numériques La plupart des circuits intégrés actuels sont basés sur la technologie CMOS. La consommation et le rayonnement électromagnétique émis par un circuit CMOS dépendent des valeurs manipulées et des calculs effectués [MDS99]. Par exemple, la figure 2.3 montre la consommation d'un microprocesseur 8 bits pendant le transfert d'un octet de données de la mémoire vers un registre. On constate que la consommation est directement liée au nombre de bits qui commutent (appelés transitions). Outre les transferts mémoires, les unités de calcul laissent fuir également de l'information. Par exemple, l'inverseur CMOS consomme davantage lorsque sa sortie commute.

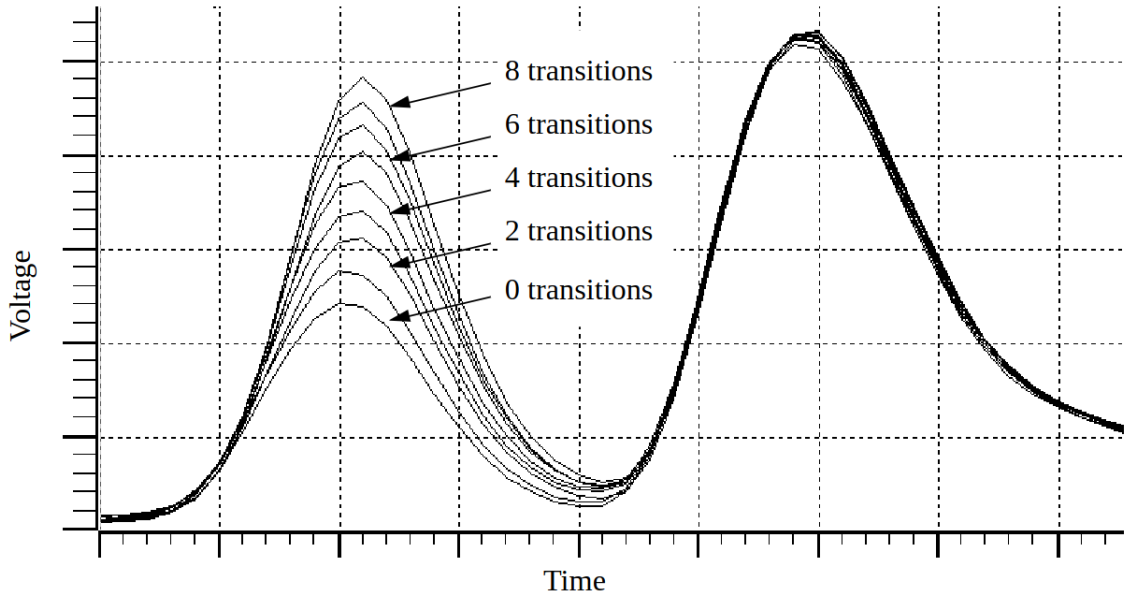


FIGURE 2.3 – Consommation d'un microprocesseur 8 bits pendant le transfert d'un octet de données de la mémoire vers un registre. Extrait de [MDS99].

Glitch Les circuits ne sont pas idéaux et possèdent des *délais de propagation*. Les délais de propagation peuvent causer des modifications transitoires du signal de sortie qui n'ont pas de sens fonctionnel [MOP07], qu'on appelle des *glitchs*. Illustrons le concept de glitch avec l'exemple de la figure 2.4. Au temps T_0, les deux entrées sont stabilisées à 0, donc C et OUT sont à 1. Au temps T_1, on suppose que les deux entrées passent simultanément à 1. À cause du délai de propagation dans l'inverseur et la porte NAND, C et OUT changent respectivement de valeur à T_2 et T_3. Quand tous les délais de propagation sont passés, la sortie va se stabiliser sur la valeur correcte. Mais entre T_3 et T_4 s'est produit une modification temporaire et incorrecte du signal de sortie, qu'on appelle un glitch. Les glitchs consomment de l'énergie et la consommation liée aux glitchs dépend des valeurs des opérandes et de leurs commutations. De nouveau, l'énergie consommée par les glitchs peut révéler de l'information sur la clé secrète [MPG05].

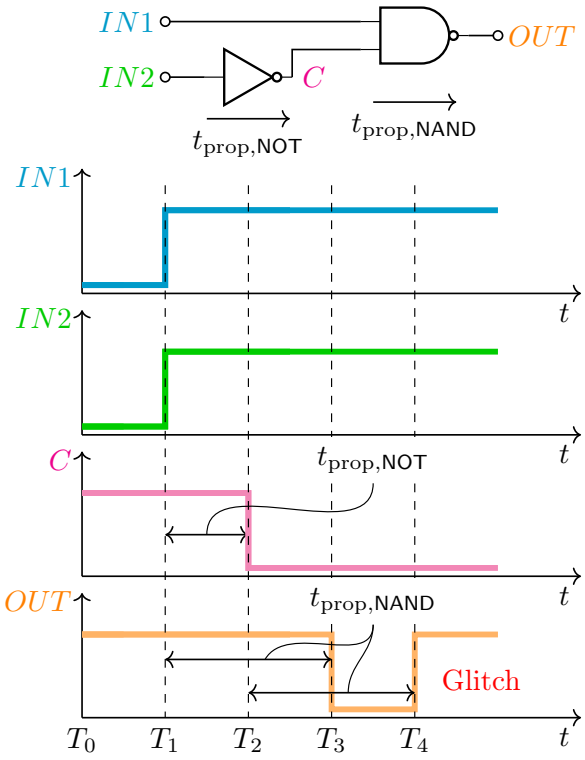


FIGURE 2.4 – Illustration simplifiée d'un glitch (en pratique les fronts montant et descendant ne sont pas verticaux).

2.3.2 L'analyse simple de la consommation

L'*analyse simple de la consommation*, abrégée en SPA pour «*simple power analysis*» en anglais, vise à récupérer des informations sensibles en observant seulement quelques traces de consommation [KJJ99]. Deux stratégies principales existent pour récupérer des informations sensibles par analyse simple de la consommation. La première stratégie tente d'identifier les dépendances entre les instructions effectuées et les traces, tandis que la seconde tente directement d'exploiter les dépendances entre les données manipulées et les traces.

Dépendance aux instructions Lorsque les calculs dépendent des valeurs des bits de la clé secrète, une trace de consommation peut a priori révéler directement la valeur de la clé. Un exemple typique est l'implémentation RSA basée sur l'exponentiation modulaire de l'algorithme 2 [KJJ99]. L'exponentiation modulaire réalise une boucle et effectue à chaque itération une élévation au carré et une multiplication si le bit d'exposant est 1, ou seulement une élévation au carré si bit d'exposant est 0. Si la consommation de la multiplication est différente de la consommation de l'élévation au carré, alors il est possible de distinguer chaque opération sur la trace de consommation. Par exemple, la figure 2.5 représente une trace de consommation de RSA sur laquelle on peut distinguer les consommations liées à la multiplication et à l'élévation au carré. Un attaquant est alors capable de retrouver chacun des bits de l'exposant secret en déterminant si la multiplication a été effectuée ou non. Une bonne manière de se protéger contre ce type d'attaque est de produire des implémentations à *code constant* [Cla+11]. Il faut en effet veiller à ce que la suite des instructions exécutées soit la plus régulière possible, ou au moins qu'elle ne dépende pas de valeurs sensibles (directement ou indirectement).

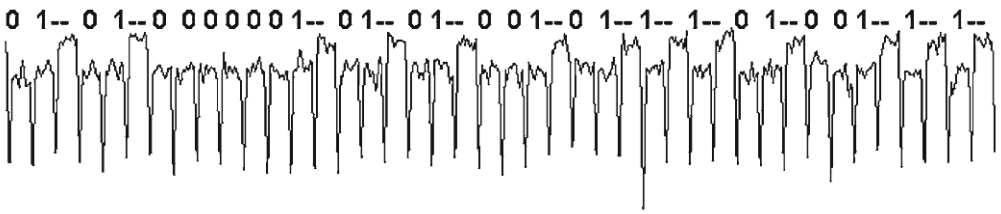


FIGURE 2.5 – Trace de consommation de RSA extraite de [Koc+11]. Une SPA sur l'exponentiation modulaire permet de déterminer la clé.

Dépendance des données La consommation d'énergie d'un dispositif exécutant un cryptosystème dépend des valeurs intermédiaires manipulées. Pour récupérer des valeurs sensibles, l'attaquant réalise dans un premier temps un modèle en mesurant la consommation pour chaque valeur de la variable intermédiaire. En comparant la trace mesurée avec le modèle, l'attaquant peut déduire la valeur sensible.

Limites de la SPA Une implémentation à code constant permet de protéger efficacement contre la SPA exploitant les dépendances liées aux instructions. De plus, les dépendances liées aux données peuvent être difficiles à exploiter en pratique car les différences de consommation entre les différentes valeurs manipulées sont faibles et l'information utile est noyée dans du bruit lié à la mesure et à l'activité du reste du circuit. Pour atténuer le bruit et pour renforcer les différences de consommation aussi petites soient elles, des *méthodes statistiques* [KJJ99] ou des techniques plus sophistiquées à base d'*intelligence artificielle* [MDP19] sont utilisées.

2.3.3 Analyses statistiques de la consommation

L'*analyse différentielle de la consommation* [KJJ99], abrégée en DPA pour «*differential power analysis*» en anglais, utilise des méthodes statistiques pour déterminer des données sensibles à partir de plusieurs traces de consommation mesurées lors du chiffrement de différents messages en clair avec la même clé secrète. L'attaque DPA consiste dans un premier temps à identifier une *variable sensible* $X_{k'}$ manipulée dans l'implémentation du chiffrement et qui dépend d'une partie k' de la clé secrète k, appelée sous-clé, ainsi que d'une donnée connue de l'attaquant (telle que le message en clair ou le message chiffré). Une fois la variable $X_{k'}$ ciblée, l'attaquant chiffre N clairs $p_1, p_2, ..., p_N$ avec la même clé secrète k pour obtenir les chiffrés c_i correspondants. Pour estimer la valeur de k', l'attaque DPA se décompose en deux phases illustrées par la figure 2.6. D'un côté, on va *mesurer la consommation*, de l'autre, on va *prédire la consommation*. Puis on fait une *comparaison statistique* des deux pour trouver la valeur de k' pour laquelle la mesure et la prédiction sont les plus corrélées.

Description de l'attaque La première phase est l'acquisition d'une trace de consommation pendant le chiffrement de chaque clair p_i pour obtenir les différentes lignes de la matrice de consommation mesurée. Les traces de consommation mesurées ne sont pas toujours parfaitement alignées en raison de légers décalages de temps lors de leurs acquisitions. L'attaque DPA étant basée sur une analyse statistique entre les différentes traces, ces décalages peuvent augmenter significativement le nombre de traces nécessaire pour réussir l'attaque. Plusieurs méthodes d'*alignement des traces* ont été proposées dans la littérature (voir [MOP07] pour un

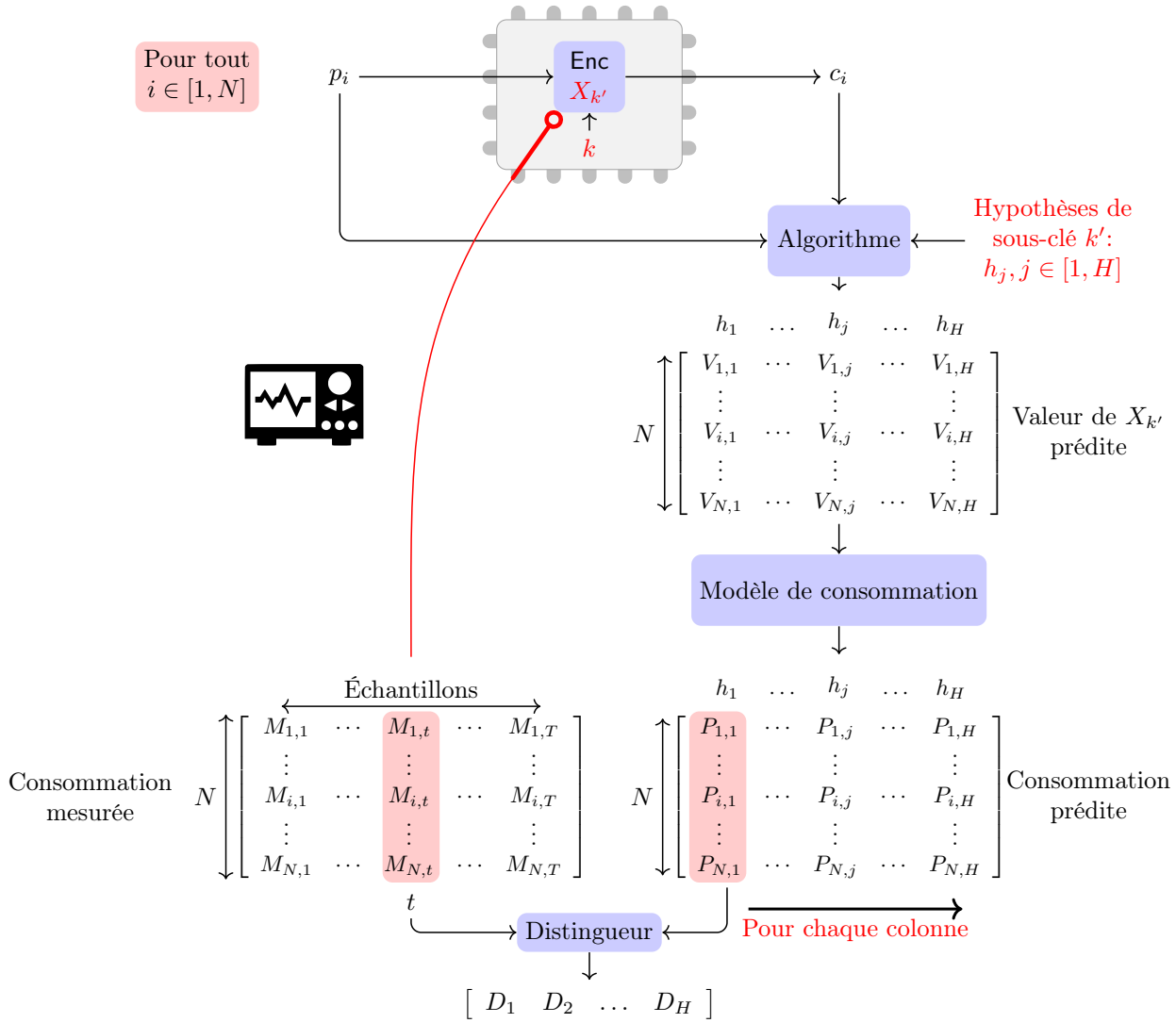


FIGURE 2.6 – Analyse différentielle de la consommation.

aperçu des différentes méthodes). Dans une seconde phase, l'attaquant réalise des *hypothèses* sur la valeur de la sous clé k' et prédit la consommation pour chaque hypothèse de sous-clé. Pour *prédire la consommation*, la valeur de $X_{k'}$ est calculée pour chaque hypothèse de sous-clé et pour chaque clair. Puis, la consommation du chiffrement est prédite en utilisant un *modèle de consommation théorique*. On obtient alors la matrice des consommations prédites dans laquelle chaque colonne correspond à la prédiction de la consommation des N chiffrements pour une hypothèse de sous-clé. Pour déterminer la valeur de la sous clé k', l'attaquant détermine un instant t dans les traces mesurées durant lequel la variable $X_{k'}$ est manipulée dans l'implémentation. Puis une *comparaison statistique* est réalisée entre la consommation mesurée à l'instant t et la consommation prédite. La colonne de la matrice des consommations mesurées correspondant à l'instant t est comparée avec chacune des colonnes de la matrice des consommations prédites. L'hypothèse de sous-clé pour laquelle la corrélation est la plus forte est probablement la sous-clé correcte. En pratique, l'instant t durant lequel la variable $X_{k'}$ est manipulée ne peut pas être déterminé précisément. La comparaison statistique est alors appliquée en chacun des points d'un intervalle de temps. La taille de cet intervalle dépend de la possibilité d'encadrer plus

ou moins précisément l'instant t. Le nombre de courbes nécessaires pour réussir une attaque dépend de plusieurs paramètres tels que le niveau de bruit des mesures, la précision du modèle de consommation ou l'efficacité de la méthode statistique utilisée.

Modèles de fuite Nous mentionnons maintenant deux modèles classiques de consommation utilisés dans les attaques DPA. La consommation liée à une variable peut être modélisée par le *poids de Hamming* de la variable, c'est à dire le nombre de bits à 1. Ce modèle exploite le fait que la consommation d'un bit à 0 peut être différente de la consommation d'un bit à 1. Une autre possibilité pour modéliser la consommation d'une variable est de calculer la *distance de Hamming* par rapport à sa valeur précédente, c'est à dire le nombre de bits ayant commuté. Ce modèle exploite le fait que la consommation est plus élevée lorsque les bits commutent. On a illustré ces deux modèles dans la figure 2.7 avec un registre 2 bits. On constate que les deux modèles donnent des traces prédites un peu différentes et ne modélisent pas tout à fait la même chose. Ces deux modèles de consommation sont les plus couramment adoptés car ils sont simples à utiliser et ils correspondent souvent assez bien aux consommations effectivement mesurées.

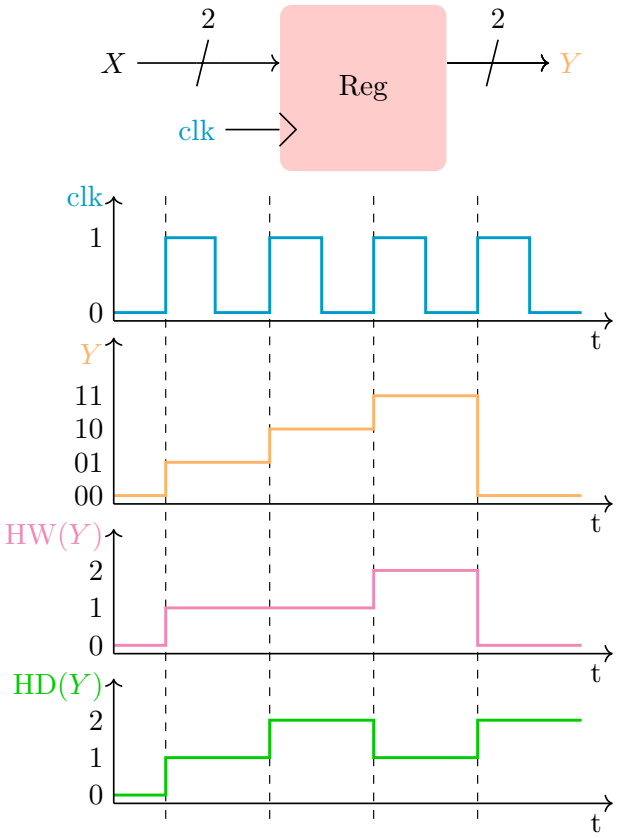


FIGURE 2.7 – Modèles de consommation.

Stratégie d'attaque Déterminer directement la clé secrète complète par une attaque DPA est impossible lorsque la clé secrète est de grande taille. Par exemple, pour déterminer la clé de 128 bits de AES, 2^{128} hypothèses de clé doivent être modélisées, ce qui est calculatoirement infaisable. La méthode *diviser pour régner* cible des variables dépendant de petits morceaux

de la clé secrète puis les attaque séparément pour déterminer la clé complète. Dès qu'on a un morceau de clé connu, on peut l'utiliser pour déterminer un autre morceau de clé et ainsi de suite. La figure 2.8 illustre la méthode diviser pour régner par une attaque DPA de AES ciblant la sortie de chaque SBOX de la première ronde. L'attaque DPA peut alors révéler chaque octet de la clé secrète avec seulement 2^8 hypothèses de sous-clé. Le nombre d'hypothèses de sous-clé à tester est donc réduit de 2^{128} à $16 \times 2^8 = 2^{12}$, qui est facilement réalisable avec les processeurs actuels et les dispositifs de mesure existants.

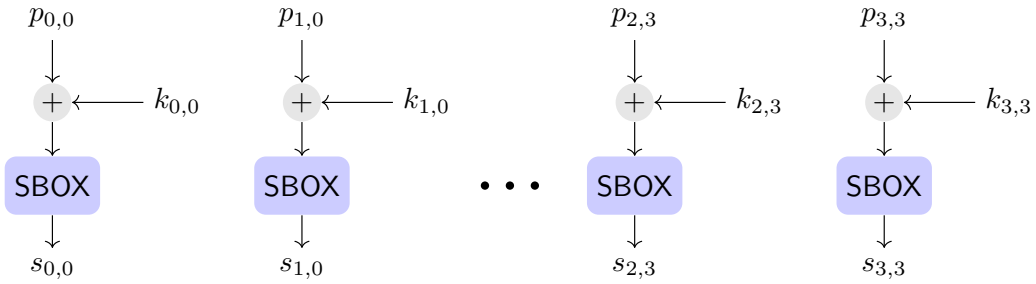


FIGURE 2.8 – AddRoundKey et SubBytes de la première ronde de AES.

Différentes méthodes statistiques Les deux méthodes statistiques les plus utilisées en pratique pour effectuer une attaque DPA sont la différence des moyennes et le coefficient de corrélation de Pearson. La différence des moyennes fut initialement introduite par [KJJ99] et cible un bit d'une valeur intermédiaire de l'implémentation. Pour chaque hypothèse de sous-clé, les traces sont partitionnées en deux ensembles en fonction de la valeur du bit ciblé, puis la différence des moyennes de chaque ensemble est calculée. Il en résulte une courbe de différence des moyennes associée à chaque hypothèse de sous-clé. Lorsque l'hypothèse de sous-clé est incorrecte, la différence des moyennes est proche de 0 car le choix qui a guidé l'affectation de chaque trace à un ensemble est décorrélé de la mesure. Lorsque l'hypothèse de sous-clé est correcte, la différence des moyennes a une valeur élevée à chaque instant où le bit ciblé a été manipulé. L'hypothèse de sous-clé ayant la plus grande différence des moyennes correspond avec une certaine probabilité à la sous-clé correcte. Un des inconvénients de cette méthode statistique est la présence de «pics fantômes» (voir [BCO04] pour une description de ce phénomène). En effet, des hypothèses de sous-clé incorrectes peuvent générer des pics dans la courbe de différence des moyennes. De plus, la différence des moyennes est très sensible à la présence de bruit dans les traces de consommation. Ces contraintes peuvent augmenter de manière significative le nombre de traces nécessaire pour réussir l'attaque. Pour une meilleure tolérance au bruit et aux «pics fantômes», [BCO04] ont proposé l'*analyse de la consommation par corrélation*, abrégée en CPA pour «*correlation power analysis*» en anglais, qui utilise le coefficient de corrélation de Pearson comme méthode statistique. Le coefficient de corrélation de Pearson de deux séries statistiques est le quotient de la covariance par le produit des écarts-types de chaque série. Il renseigne sur le degré de dépendance linéaire entre les séries statistiques. Il est compris entre -1 et 1 et plus le coefficient est proche de -1 ou 1, plus la corrélation linéaire est forte. La CPA semble être plus tolérante au bruit et aux «pics fantômes» que la différence des moyennes et fournit généralement de meilleurs résultats expérimentaux en révélant des corrélations avec un nombre de traces inférieur. Ceci est probablement dû au fait que sa formule tient compte à la fois de la

moyenne et de la variance des mesures. Il existe d'autres méthodes statistiques plus évoluées. On peut notamment citer l'*analyse de l'information mutuelle* [GBT07], abrégée en MIA pour «*mutual information analysis*» en anglais. À la différence des méthodes statistiques précédentes, elle prend en compte certaines relations non-linéaires entre les traces de consommation et les valeurs manipulées.

2.3.4 Contre-mesures

On vient d'étudier les principes de la SPA et de la DPA. On va voir maintenant comment on se protège de ces attaques. Il existe différents types de protection.

Désynchronisation en temps Dans les attaques DPA, les traces doivent être alignées pour qu'une variable sensible soit manipulée à un même instant dans chaque trace. Une contre-mesure consiste à *désynchroniser les traces* afin qu'une variable sensible soit manipulée à des moments différents. Une première méthode est l'*insertion aléatoire d'opérations factices* [TB07 ; CK09]. Le temps d'exécution global doit rester constant pour que l'attaquant n'obtienne pas d'information sur le nombre d'opérations factices et les opérations factices ne doivent pas être détectables par SPA. Augmenter le nombre d'opérations factices insérées améliore la sécurité apportée par la contre-mesure, mais en contrepartie dégrade les performances. La deuxième méthode de désynchronisation des traces est le *mélange aléatoire des opérations* [RPD09 ; Vey+12]. Cette contre-mesure consiste à réordonner aléatoirement l'ordre d'exécution des opérations lorsque c'est possible. Par exemple, les SBOX de AES peuvent être calculées dans un ordre quelconque. Cependant, le nombre d'opérations qui peut être réordonné est généralement assez limité et dépend de l'algorithme. La désynchronisation des traces n'est généralement pas suffisante pour se prémunir complètement des attaques DPA. Associée avec d'autres contre-mesures, elle améliore la protection en augmentant le nombre de traces nécessaires pour réussir une attaque DPA.

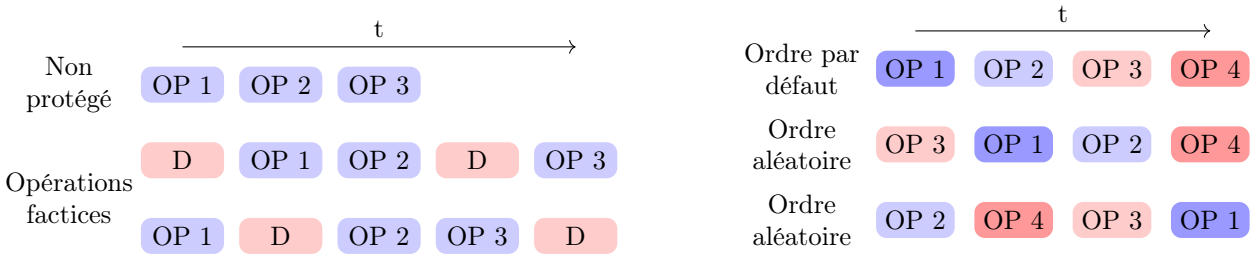


FIGURE 2.9 – Insertion aléatoire d'opérations factices à gauche et mélange aléatoire des opérations à droite.

Consommation équilibrée Une autre contre-mesure consiste à rendre la consommation de l'implémentation constante, supprimant ainsi la corrélation entre les mesures et les variables manipulées [Sok+04 ; PM05 ; Buc+06]. Par exemple, le double rail rend le poids de Hamming constant en manipulant toujours un bit et son complémentaire. Cependant, en pratique, la consommation n'est jamais parfaitement constante en raison de petits déséquilibres de charge entre les deux circuits complémentaires, de variations de processus, de routage, etc. De plus, cette protection impose un surcoût important en terme de surface, puisqu'elle multiplie la surface par 3 ou 4 environ.

FIGURE 2.10 – Porte logique implémentée en double rail.

Randomisation des données La contre-mesure la plus étudiée et la plus utilisée en pratique est le *masquage* [Cha+99 ; GP99]. Le masquage consiste à rendre aléatoires à chaque exécution les variables sensibles manipulées dans une implémentation. Le masquage d'ordre d représente chaque variable sensible par n parts (avec $n > d$) telles que chaque sous-ensemble de d parts soit indépendant de la variable sensible. Un attaquant doit alors connaître plus de d parts pour déterminer une variable sensible. Lorsque le masquage est bien mis en œuvre, la réalisation d'une attaque DPA est d'autant plus compliquée que l'ordre de masquage est élevé. La contre-mesure de masquage est décrite dans le chapitre 3.

2.3.5 Attaques SCA d'ordre supérieur

L'attaque *SCA d'ordre supérieur*, abrégée en HO-SCA pour «*higher-order side-channel analysis*» en anglais, est une attaque SCA qui exploite les consommations résultant de la manipulation dans l'implémentation de plusieurs variables. Les *attaques multivariées* d'ordre d combinent d points de la trace pour déterminer des variables sensibles. Le masquage d'ordre d est ainsi vulnérable aux attaques multivariées d'ordre $d + 1$ qui combinent les consommations liées à $d + 1$ parts. Plusieurs *fonctions de combinaison* ont été proposées dans la littérature [PRB09]. [Cha+99] propose d'effectuer le produit des parts alors que [Mes00b] combine les parts en utilisant la différence absolue. La complexité pour réaliser une attaque multivariée augmente avec l'ordre de masquage. Les *attaques univariées* exploitent la consommation en un seul point de la trace résultant de la manipulation de plusieurs variables en parallèle. C'est notamment ce qui se passe lorsque le masquage est appliqué en matériel, puisque toutes les parts sont manipulées en parallèle. Une attaque univariée d'ordre d utilise le moment statistique d'ordre d pour attaquer le cryptosystème. Le masquage d'ordre d en matériel est vulnérable aux attaques univariées d'ordre $d + 1$ qui impliquent conjointement $d + 1$ parts.

Chapitre 3

Masquage

Le *masquage* est une des contre-mesures les plus largement utilisées pour protéger les implémentations cryptographiques contre la SCA [MOP07]. Le masquage est apparu dans la littérature peu après la publication de l'attaque DPA et fut publié indépendamment dans [Cha+99] et [GP99]. Un *schéma de masquage* mélange chaque variable sensible intervenant lors du calcul avec une ou plusieurs valeurs aléatoires. La sécurité d'un schéma de masquage est prouvable dans des *modèles théoriques d'attaquant*. De nombreux schémas de masquage ont été proposés dans la littérature pour réduire les coûts induits ou augmenter le niveau de protection. Un schéma de masquage peut s'avérer compliqué à mettre en œuvre pour des applications complexes telles que le chiffrement AES. Pour faciliter la mise en œuvre d'un schéma de masquage, une méthode consiste à composer de façon sécurisée des implémentations masquées.

Dans ce chapitre, nous décrivons en détail la contre-mesure de masquage. Dans la section 3.1, nous introduisons la représentation masquée. Dans la section 3.2, nous décrivons ce qu'est un schéma de masquage. Dans la section 3.3, nous analysons la sécurité apportée par un schéma de masquage dans différents modèles d'attaquant. Dans la section 3.4, nous décrivons différentes méthodes pour composer des implémentations masquées de façon sécurisée. Dans la section 3.5, nous décrivons quelques implémentations masquées de la multiplication.

3.1 Représentation masquée

Le *masquage* est une contre-mesure contre l'attaque SCA. Les premières propositions de masquage ont été réalisées indépendamment par [Cha+99] et [GP99]. Le principe du masquage est d'appliquer un *schéma de partage de secret* [Sha79] pour *randomiser* les variables intermédiaires traitées par un algorithme cryptographique et casser les *dépendances statistiques* entre les données sensibles et la fuite mesurée.

Masquage de premier ordre Le *masquage de premier ordre* représente une variable sensible x en deux *parts* telles que chaque part soit *statistiquement indépendante* de x. Par exemple, le *masquage booléen* représente une variable sensible x par les parts $x \oplus r$ et r, où r est une valeur aléatoire uniformément tirée appelée masque. Un attaquant doit donc cibler les deux parts ensembles pour obtenir des informations sur x. Par conséquent, les attaques SCA classiques exploitant la fuite liée à une seule variable intermédiaire ne sont plus possibles. Cependant, le masquage de premier ordre est vulnérable aux *attaques de second ordre* qui exploitent en même temps la fuite liée à $x \oplus r$ et la fuite liée à r (voir par exemple [Mes00b ; Osw+05 ; PRB09]). Pour

atteindre une sécurité contre les attaques d'ordre supérieur, la méthode consiste à augmenter le nombre de parts.

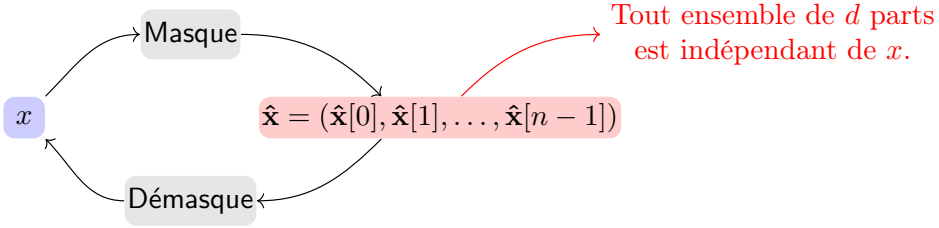


FIGURE 3.1 – Procédure d'encodage et de décodage.

Masquage d'ordre supérieur Pour protéger contre les *attaques d'ordre supérieur*, une variable sensible x est représentée par un vecteur $\hat{\mathbf{x}} \in \mathbb{K}^n$ de n parts [Cha+99], appelé un masquage de x. On utilise des minuscules en gras avec chapeau $\hat{\mathbf{x}}$ pour désigner un masquage d'une variable x, et $\hat{\mathbf{x}}[i]$ pour désigner la $i^{ème}$ part du masquage $\hat{\mathbf{x}}$. Une *procédure de masquage* $\mathsf{Masque} : \mathbb{K} \to \mathbb{K}^n$ est un *algorithme probabiliste* qui permet de passer en représentation masquée. À partir d'un masquage de x, on peut toujours retrouver x en utilisant une *procédure de démasquage*. Une procédure de démasquage $\mathsf{Démasque} : \mathbb{K}^n \to \mathbb{K}$ est un *algorithme déterministe* vérifiant pour tout $x \in \mathbb{K}$, $\mathsf{Démasque}(\mathsf{Masque}(x)) = x$ avec une probabilité égale à 1. Le masquage est dit *d'ordre* d si tout sous-ensemble de d parts est statistiquement indépendant de x. Lorsque $d > 1$, le masquage est dit *d'ordre supérieur*. Dans la meilleure situation souvent étudiée dans la littérature, l'ordre de masquage est égal à $d = n - 1$, mais l'ordre de masquage peut aussi être inférieur à $n - 1$. Le masquage d'ordre d rend les attaques plus difficiles, car l'attaquant doit au moins connaître plus de d parts pour récupérer la valeur d'une variable sensible. Néanmoins, un masquage d'ordre d est toujours théoriquement vulnérable aux attaques d'ordre $d+1$. Mais en pratique, le bruit implique que la difficulté à réaliser une attaque d'ordre supérieur augmente avec son ordre, ce qui fait de l'ordre de masquage un bon paramètre de sécurité contre les attaques SCA [Cha+99].

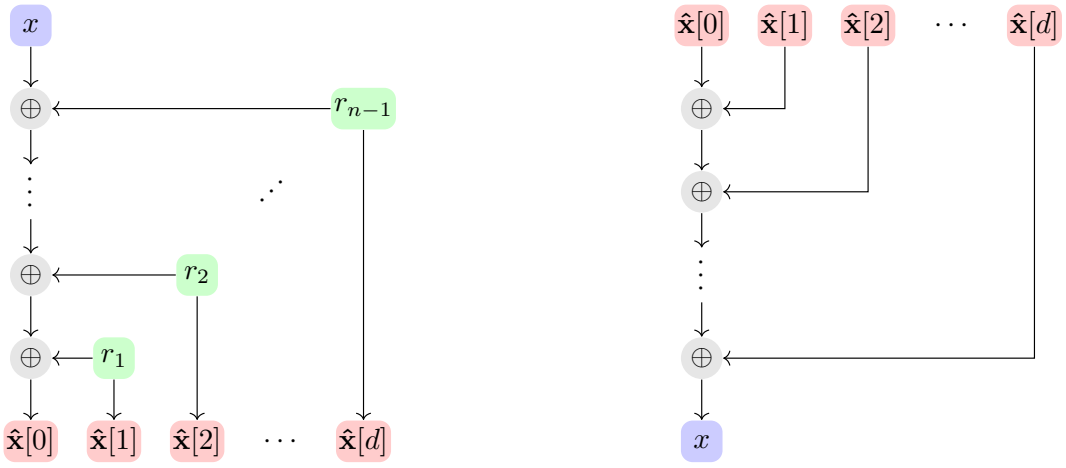


FIGURE 3.2 – Procédure de masquage (à gauche) et de démasquage (à droite) du masquage booléen.

Masquage booléen d'ordre supérieur Dans ce manuscrit, nous nous intéressons uniquement au *masquage booléen* d'ordre d [Cha+99 ; ISW03], abrégé en BM pour «*boolean masking*» en anglais, dans lequel une variable x est représentée par $d+1$ parts satisfaisant la relation $\hat{\mathbf{x}}[0] \oplus \ldots \oplus \hat{\mathbf{x}}[d] = x$. En pratique, les parts $\hat{\mathbf{x}}[1], \ldots, \hat{\mathbf{x}}[d]$ (appelées les masques) sont des valeurs aléatoires et la part $\hat{\mathbf{x}}[0]$ est le XOR de x avec toutes les autres parts (voir figure 3.2 gauche), rendant ainsi tout sous-ensemble de d parts statistiquement indépendant de x. Pour démasquer $\hat{\mathbf{x}}$, il suffit de calculer le XOR de toutes les parts (voir figure 3.2 droite).

3.2 Schéma de masquage

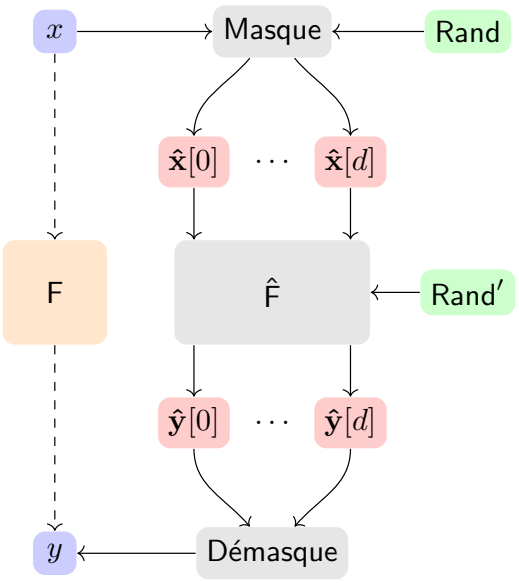


FIGURE 3.3 – Schéma de masquage.

Un *schéma de masquage* est un algorithme permettant de réaliser des calculs en manipulant uniquement des variables masquées tout en préservant certaines propriétés de sécurité. Cette notion est étroitement liée au *calcul multi-parties* (MPC) [BGW88] qui permet à un ensemble de parties de calculer conjointement une fonction tout en gardant leurs entrées respectives secrètes. L'application d'un schéma de masquage à une implémentation à protéger F (qui peut être un circuit matériel ou un programme informatique) se divise en trois phases illustrées par la figure 3.3.

1. Premièrement, une *phase d'encodage* masque les variables d'entrées x à l'ordre de masquage choisi pour donner $\hat{\mathbf{x}}$.
2. Deuxièmement, une *implémentation masquée* de F, notée $\hat{\mathsf{F}}$, est appliquée aux entrées masquées. $\hat{\mathsf{F}}$ a la même fonctionnalité que F dans le domaine masqué et permet de calculer en utilisant uniquement les variables masquées. De façon plus formelle, $\hat{\mathsf{F}}$ est une implémentation vérifiant pour tout x :
$$\Pr(\mathsf{Démasque}(\hat{\mathsf{F}}(\mathsf{Masque}(x))) = \mathsf{F}(x)) = 1$$
3. Troisièmement, une *phase de décodage* permet de récupérer le résultat du calcul à partir de sa représentation masquée. Ce résultat doit toujours être égal au résultat qu'on aurait obtenu en utilisant l'implémentation non masquée F.

Par exemple, lorsqu'on applique un schéma de masquage à AES, les calculs ne sont pas directement effectués sur la clé de chiffrement k et le message en clair p, mais k et p sont préalablement masqués en $\hat{\mathbf{k}}$ et $\hat{\mathbf{p}}$. Ensuite, une implémentation masquée de AES est appliquée à $\hat{\mathbf{k}}$ et $\hat{\mathbf{p}}$ pour obtenir un message chiffré masqué $\hat{\mathbf{c}}$. Pour terminer, le message chiffré masqué $\hat{\mathbf{c}}$ est démasqué pour obtenir le message chiffré attendu.

Évaluation de la sécurité L'évaluation de la sécurité d'un schéma de masquage a été largement étudiée dans la littérature et se divise en deux méthodologies complémentaires. La première méthode consiste à implémenter le schéma de masquage sur un dispositif physique et à réaliser des *évaluations pratiques*. L'évaluation pratique permet une analyse précise des fuites de l'implémentation réelle. La mise en œuvre pratique de ce type d'évaluation est cependant complexe, car elle nécessite du matériel et une forte expertise des attaques. Et surtout la non réussite d'une attaque ne signifie pas que la solution est sûre ! La deuxième méthode consiste à se placer dans un *modèle théorique d'attaquant* et de prouver la sécurité d'un schéma de masquage dans ce modèle. Afin de modéliser les capacités d'un attaquant à extraire des informations, plusieurs modèles ont été introduits visant à atteindre différents compromis entre simplicité et précision. Dans la section suivante, nous introduisons différents modèles d'attaquant utilisés dans la littérature.

3.3 Modèle théorique d'attaquant

3.3.1 Modélisation des implémentations à protéger

Les implémentations à protéger (circuits matériels ou programmes informatiques) sont préalablement modélisées par un *graphe orienté acyclique* sur un corps $\mathbb{K}$ [ISW03 ; CS21], abrégé en DAG pour «*directed acyclic graph*» en anglais, dont les arêtes sont des variables intermédiaires et les sommets sont soit des *fonctions*, des *variables d'entrée*, des *variables de sortie* ou des *variables aléatoires*. Par exemple, tout circuit combinatoire peut être modélisé par un DAG sur GF(2) ayant pour sommet les portes booléennes AND, XOR et NOT. Un attaquant est capable de sonder uniquement les arêtes du DAG. Le modèle d'attaquant dépend donc de la précision de la modélisation.

3.3.2 Modèle de sondage

Le *modèle de sondage* a initialement été introduit dans [ISW03] pour caractériser un adversaire qui serait théoriquement capable de positionner un nombre limité d'aiguilles métalliques de mesure sur un circuit intégré mettant en œuvre un algorithme cryptographique. Le modèle de sondage donne à un attaquant la possibilité de sonder jusqu'à t arêtes dans le DAG d'une implémentation masquée comme le montre la figure 3.4. Les arêtes sondées par l'attaquant sont appelées des *sondes*. Il est supposé que l'attaquant ne peut pas sonder la procédure de masquage et de démasquage. Une implémentation masquée $\hat{\mathsf{F}}$ est sécurisée dans le modèle de sondage avec t sondes si tout ensemble de t sondes dans $\hat{\mathsf{F}}$ est indépendant des variables sensibles.

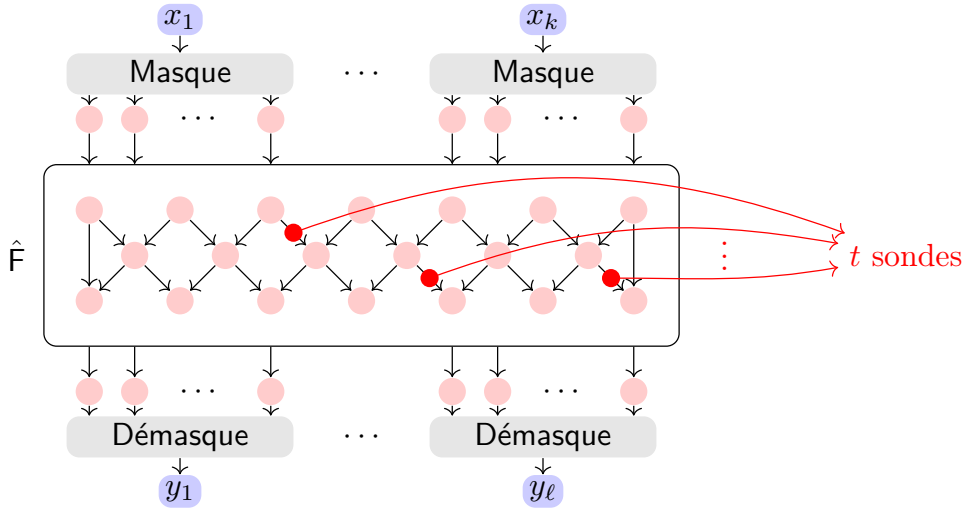


FIGURE 3.4 – Modèle de sondage.

Pertinence du modèle Ce modèle est pratique pour prouver la sécurité d'une implémentation masquée. Cependant, ce modèle n'est pas très réaliste car en pratique un attaquant obtient des fuites bruitées sur le calcul complet et n'a aucune raison d'être limité à t sondes. Pour modéliser plus fidèlement la fuite, le *modèle de fuite bruitée* a été décrit dans [Cha+99] puis formalisé dans [PR13]. Dans ce modèle, l'adversaire a accès à une *fonction bruitée* de chaque variable intermédiaire de l'implémentation masquée. Bien que réaliste, ce modèle n'est pas pratique pour les preuves de sécurité. Les implémentations masquées ont donc continué à être vérifiées dans le modèle de sondage en se reposant sur la *réduction de sécurité* qui a été formellement établie dans [DDF14]. [DDF14] a en effet montré que la sécurité d'une implémentation masquée dans le modèle de fuite bruitée pouvait être obtenue par la sécurité dans le modèle de sondage.

Preuve de sécurité Pour prouver en pratique la sécurité dans le modèle de sondage, on utilise le concept de *simulation d'une sonde* [ISW03] qui permet de décrire les dépendances entre les sondes et les parts d'entrée d'une implémentation masquée. Un ensemble de sondes $\mathcal{P}$ peut être simulé par un ensemble de parts d'entrée $\mathcal{S}$ si et seulement s'il existe un simulateur Sim tel que les distributions de $\mathcal{P}$ et $\mathsf{Sim}(\mathcal{S})$ soient identiques. Une implémentation masquée est alors sécurisée dans le modèle de sondage à t sondes si tout ensemble de t sondes peut être simulé avec au plus d parts de chaque masquage d'entrée, où d est l'ordre de masquage. En effet, les t sondes peuvent alors être simulées indépendamment des variables sensibles, car le masquage est d'ordre d.

Implémentation masquée d'une application linéaire Une application linéaire L peut être masquée en appliquant L indépendamment sur chaque part du masquage booléen de l'entrée x, noté $\hat{\mathbf{x}}$ (voir figure 3.5). Il s'agit d'une implémentation masquée de L car en démasquant le résultat, on retrouve bien $\mathsf{L}(x)$:

$$\mathsf{L}(\hat{\mathbf{x}}[0]) \oplus \cdots \oplus \mathsf{L}(\hat{\mathbf{x}}[d]) = \mathsf{L}(\hat{\mathbf{x}}[0] \oplus \cdots \oplus \hat{\mathbf{x}}[d]) = \mathsf{L}(x)$$

La fonction linéaire ainsi masquée est sécurisée dans le modèle de sondage avec d sondes. En effet, quelle que soit la variable intermédiaire sondée, elle dépend uniquement d'une seule part

de $\hat{\mathbf{x}}$. Ainsi, tout ensemble de d sondes est simulable avec d parts de $\hat{\mathbf{x}}$, qui sont indépendantes de x. Les sondes ne révèlent donc aucune information sur x.

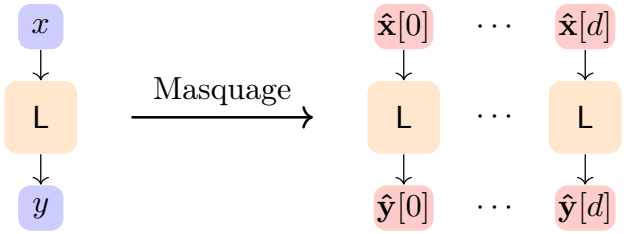


FIGURE 3.5 – Masquage d'une application linéaire.

Implémentation masquée d'une application non linéaire Les fonctions non linéaires sont plus compliquées à masquer que les fonctions linéaires, car elles mélangent plusieurs parts lors des calculs, ce qui peut conduire à une recombinaison des parts. Par exemple, la figure 3.6 de gauche représente une implémentation masquée à l'ordre 1 de la porte AND. En effet, en démasquant le résultat, on retrouve $x \otimes y$:

$$\begin{aligned}\hat{\mathbf{z}}[0] \oplus \hat{\mathbf{z}}[1] =& (\hat{\mathbf{x}}[0] \otimes \hat{\mathbf{y}}[0]) \oplus (\hat{\mathbf{x}}[1] \otimes \hat{\mathbf{y}}[0]) \oplus (\hat{\mathbf{x}}[0] \otimes \hat{\mathbf{y}}[1]) \oplus (\hat{\mathbf{x}}[1] \otimes \hat{\mathbf{y}}[1]) \\ =& (\hat{\mathbf{x}}[0] \oplus \hat{\mathbf{x}}[1]) \otimes (\hat{\mathbf{y}}[0] \oplus \hat{\mathbf{y}}[1]) \\ =& x \otimes y\end{aligned}$$

Mais cette implémentation masquée n'est pas sécurisée dans le modèle de sondage même en supposant que les masquages de x et de y sont indépendants. En effet, $\hat{\mathbf{z}}[0]$ est égal à $x \otimes \hat{\mathbf{y}}[0]$ qui n'est plus indépendant de x. On a fait l'erreur de recombiner $\hat{\mathbf{x}}[0]$ et $\hat{\mathbf{x}}[1]$ dans $\hat{\mathbf{z}}[0]$ ce qui a démasqué x. Pour éviter de recombiner les parts, des *valeurs aléatoires* doivent être ajoutées à certains endroits bien choisis comme dans la figure 3.6 de droite. Pour sécuriser la porte AND masquée, une *valeur aléatoire* r est ajoutée avant de recombiner les deux produits. En démasquant $\hat{\mathbf{z}}$, la valeur r disparaît (car $r \oplus r = 0$ dans GF(2)) et on retrouve bien le produit $x \otimes y$. Grâce à la valeur aléatoire r, chaque variable intermédiaire est indépendante de x et de y et la porte AND masquée est sécurisée dans le modèle de sondage.

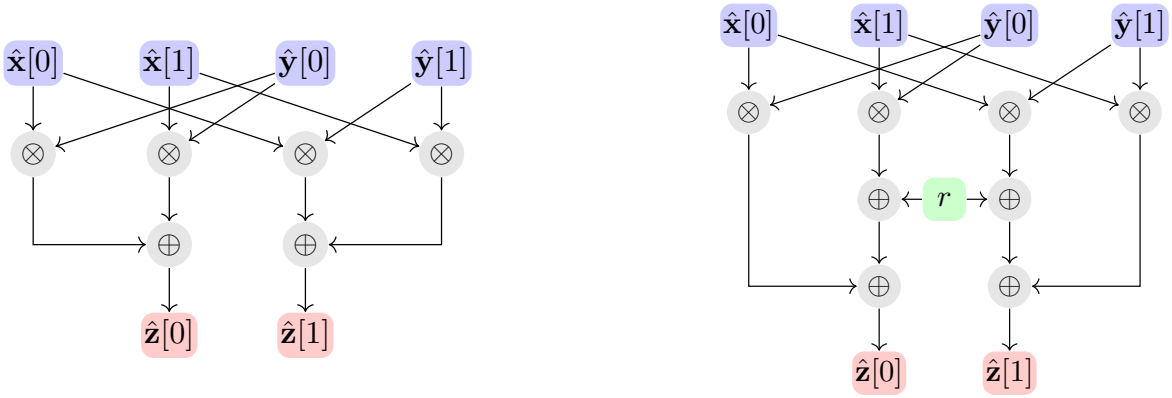


FIGURE 3.6 – Porte AND masquée, non sécurisée à gauche et sécurisée à droite dans le modèle de sondage.

3.3.3 Modèle de sondage robuste

La sécurité d'un schéma de masquage repose en grande partie sur l'hypothèse dite de *fuite indépendante* [Cha+99 ; PR13]. Autrement dit, on suppose généralement que les fuites de différentes valeurs intermédiaires se produisant dans des calculs distincts sont indépendantes les unes des autres. Cependant, cette hypothèse n'est pas toujours vérifiée en pratique dans les implémentations matérielles en raison de *défauts physiques*. Des défauts physiques tels que les *transitions* de mémoire [MPG05 ; MPO05 ; MS06], les *glitchs* [Cor+12 ; Bal+14], ou les *couplages* [De +17] peuvent engendrer une recombinaison des parts et ainsi dégrader la sécurité des implémentations théoriquement sécurisées dans le modèle de sondage. Le *modèle de sondage robuste*, introduit dans [Fau+18], étend le modèle de sondage pour capturer les fuites d'information provenant de certains défauts physiques. Ce modèle vise à modéliser avec précision des défauts physiques tout en étant suffisamment simple pour permettre une vérification efficace des implémentations masquées. Pour chaque défaut physique considéré, des *sondes étendues* ont été introduites pour augmenter le nombre de valeurs observées dans l'implémentation. Une implémentation masquée est sécurisée dans le modèle de sondage robuste à un défaut physique donné si elle est sécurisée dans le modèle de sondage où chaque sonde est étendue pour le défaut physique considéré.

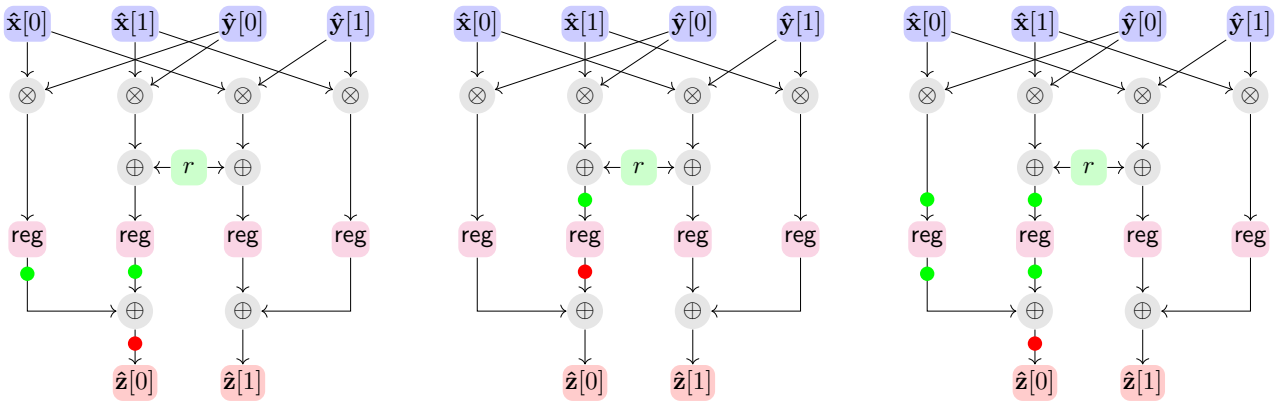


FIGURE 3.7 – Exemples de sondes étendues pour les glitchs à gauche, les transitions au centre, les glitchs et transitions à droite. En vert sont réprésentées les sondes de l'attaquant et en rouge les sondes étendues pour le défaut physique considéré.

Sonde étendue pour les glitchs Les glitchs sont des transitions de signaux involontaires et indésirables causées par les délais de propagation dans les circuits (voir sous-section 2.3.1). En raison des glitchs, la consommation liée à un fil dépend non seulement de la valeur du fil, mais aussi des données contribuant à sa valeur combinatoire. Afin de modéliser les fuites liées aux glitchs, chaque sonde est étendue en un ensemble de sondes placées sur les sorties des registres et les entrées primaires qui entrent dans le calcul combinatoire de la valeur sondée (voir figure 3.7 gauche).

Sonde étendue pour les transitions Les transitions sont des recombinaisons de mémoire causées par l'invocation consécutive d'un circuit. Afin de modéliser les fuites liées aux transitions, chaque sonde placée sur la sortie d'un élément mémoire est étendue en deux sondes, une sur l'entrée et une sur la sortie de l'élément mémoire, doublant ainsi le nombre de sondes (voir figure 3.7 centre).

Sonde étendue pour les glitchs et les transitions Les transitions et les glitchs peuvent également être modélisés simultanément. Les sondes sont d'abord étendues pour les glitchs, puis chaque sonde résultante est étendue pour les transitions (voir figure 3.7 droite). Cela correspond à la propriété physique selon laquelle la propagation d'un glitch peut dépendre à la fois de la nouvelle valeur et de la valeur précédente.

3.3.4 Modèle de sondage par régions

Pour augmenter la sécurité, le *modèle de sondage par régions* a été proposé par [ISW03] et formalisé par [ADF16]. Le modèle de sondage par régions est illustré par la figure 3.8. Dans ce modèle, une implémentation masquée $\hat{\mathsf{F}}$ est partitionnée en régions $(\hat{\mathsf{F}}_1, \hat{\mathsf{F}}_2, \ldots, \hat{\mathsf{F}}_m)$ où chaque $\hat{\mathsf{F}}_i$ est un sous-graphe de $\hat{\mathsf{F}}$ tel que les portes des sous-graphes $\hat{\mathsf{F}}_i$ forment une partition des portes de $\hat{\mathsf{F}}$. Une implémentation masquée est alors sécurisée dans le modèle de sondage par régions avec t sondes par région si l'attaquant peut sonder t variables dans chaque région sans compromettre des variables sensibles.

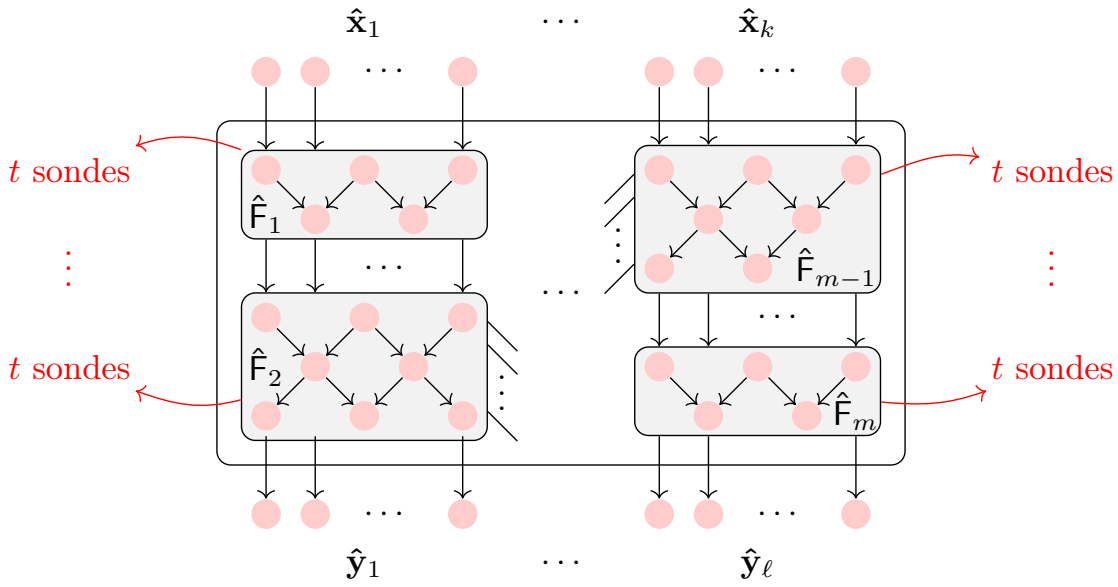


FIGURE 3.8 – Modèle de sondage par régions.

Pertinence du modèle La réduction de sécurité de [DDF14] permet de montrer la sécurité d'une implémentation masquée dans le modèle de fuite bruitée à partir de la sécurité dans le modèle de sondage. Elle fait intervenir un modèle d'attaquant intermédiaire, appelé *modèle de sondage aléatoire*, dans lequel la valeur de chaque fil est transmise à l'attaquant avec une probabilité donnée p, appelée *taux de fuite*. Les implémentations masquées et sécurisées dans le modèle de sondage sont sécurisées dans le modèle de sondage aléatoire, mais le taux de fuite toléré n'est pas constant par rapport à l'ordre de masquage, ce qui n'est pas satisfaisant d'un point de vue pratique. Le *modèle de sondage par régions* permet d'améliorer le taux de fuite toléré, et ainsi d'augmenter la sécurité dans le modèle de fuite bruitée.

3.4 Composition d'implémentations masquées

La sécurité d'une implémentation masquée dans le modèle de sondage peut être vérifiée en utilisant des *outils formels* (par exemple [Bar+19b ; KSM20]). Si de tels outils sont très utiles pour évaluer la sécurité d'implémentations concrètes et développer des implémentations sécurisées, ils sont limités par leur complexité exponentielle par rapport à la *taille de l'implémentation* et *l'ordre de masquage*. Par exemple, ces outils ne permettent pas de prouver la sécurité dans le modèle de sondage d'une implémentation masquée de AES. Pour masquer une application complexe telle que AES, une méthode illustrée à la figure 3.9 consiste à diviser l'application en sous-fonctions $\mathsf{G}_1, \cdots, \mathsf{G}_m$ suffisamment petites pour être facilement masquées en $\hat{\mathsf{G}}_1, \cdots, \hat{\mathsf{G}}_m$. En composant les différentes sous-fonctions masquées, on obtient alors une implémentation masquée de l'application complexe. Cependant, la *composition* d'implémentation masquées sécurisées dans le modèle de sondage n'est pas nécessairement sécurisée dans le modèle de sondage [Cor+14]. Différentes méthodes ont alors été proposées pour composer de façon sécurisée des implémentations masquées.

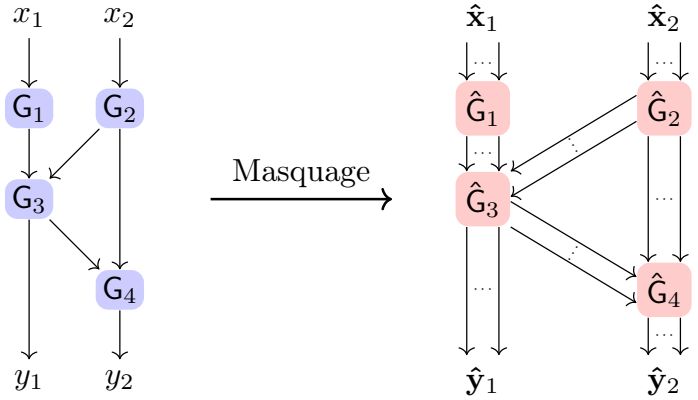


FIGURE 3.9 – Composition d'implémentations masquées.

3.4.1 Schéma de masquage ISW

[ISW03] ont proposé un schéma de masquage, que nous nommons ISW, pour masquer des circuits booléens composés de portes AND et NOT. Leur approche consiste à remplacer chaque porte par un circuit masqué de la porte basé sur un masquage booléen d'ordre d. Chaque porte NOT est remplacée par un circuit masqué qui consiste simplement à appliquer une porte NOT sur l'une des parts. L'exactitude s'ensuit simplement puisque $\mathsf{NOT}(x) = \mathsf{NOT}(\hat{\mathbf{x}}[0]) \oplus \hat{\mathbf{x}}[1] \oplus \cdots \oplus \hat{\mathbf{x}}[d]$ pour tout masquage booléen $\hat{\mathbf{x}}$ de x. Chaque porte AND est remplacée par le circuit masqué décrit dans l'algorithme 3 qui permet de masquer la porte AND à des ordres supérieurs. La porte AND masquée à l'ordre d a une complexité en $\mathcal{O}(d^2)$ et utilise $d(d+1)/2$ aléas. Chaque fil du circuit est également remplacé par $n = d+1$ fils transportant un masquage du fil. Tout circuit composé de M portes est alors transformé en un circuit masqué de taille $\mathcal{O}(Md^2)$. Le circuit masqué à l'ordre d résultant du schéma de masquage est en outre sécurisé dans le modèle de sondage à $d/2$ sondes. Cette solution nécessite ainsi de doubler le nombre de parts par rapport au nombre de sondes.

Algorithme 3 Porte AND masquée de [ISW03]

Entrée: Masquage $\hat{\mathbf{x}} \in \mathrm{GF}(2)^{d+1}$ de x et masquage $\hat{\mathbf{y}} \in \mathrm{GF}(2)^{d+1}$ de y
Sortie: Masquage $\hat{\mathbf{z}} \in \mathrm{GF}(2)^{d+1}$ de z tel que $\bigoplus_i \hat{\mathbf{z}}[i] = x \otimes y$

pour $i = 0$ à d **faire**
  pour $j = i + 1$ à d **faire**
    $r_{i,j} \xleftarrow{\$} \mathbb{K}_2$
    $u_{i,j} \leftarrow r_{i,j}$
    $u_{j,i} \leftarrow r_{i,j} \oplus (\hat{\mathbf{x}}[i] \otimes \hat{\mathbf{y}}[j]) \oplus (\hat{\mathbf{x}}[j] \otimes \hat{\mathbf{y}}[i])$
  fin pour
fin pour
pour $i = 0$ à d **faire**
  $\hat{\mathbf{z}}[i] \leftarrow (\hat{\mathbf{x}}[i] \otimes \hat{\mathbf{y}}[i]) \oplus \bigoplus_{j=0, j\neq i}^{d} u_{i,j}$
fin pour

3.4.2 Schéma de masquage RP

[RP10] ont proposé un schéma de masquage d'ordre d, que nous nommons RP, atteignant la sécurité contre un attaquant pouvant sonder d sondes. Ils montrent notamment que la porte AND masquée de ISW atteint cette sécurité de sondage minimale en supposant que les deux masquages d'entrée sont mutuellement indépendants. [RP10] ont également observé que la preuve de sécurité de [ISW03] était valable pour tout corps fini et pas seulement GF(2). Le schéma de masquage ISW a ainsi été adapté à AES en travaillant sur $\mathrm{GF}(2^8)$ au lieu de GF(2). Cependant, la composition directe d'implémentations masquées n'est pas nécessairement sécurisée lorsque les masquages d'entrée de certaines multiplications ne sont plus mutuellement indépendants. Pour maintenir l'indépendance des masquages d'entrée des multiplications, [RP10] ont suggéré de *rafraîchir* le masquage d'une des entrées en ajoutant des valeurs aléatoires. L'algorithme 4 présente une méthode de rafraîchissement proposée dans [RP10]. Elle consiste simplement à ajouter aux $n - 1$ premières parts un nouveau masque aléatoire et à accumuler la somme des masques dans la dernière part.

Algorithme 4 Méthode de rafraîchissement de [RP10]

Entrée: Masquage $\hat{\mathbf{x}} \in \mathbb{K}^{d+1}$ de x
Sortie: Masquage $\hat{\mathbf{y}} \in \mathbb{K}^{d+1}$ de y tel que $\bigoplus_i \hat{\mathbf{y}}[i] = x$

$\hat{\mathbf{y}}[d] \leftarrow \hat{\mathbf{x}}[d]$
pour $i = 0$ à $d - 1$ **faire**
  $r_i \xleftarrow{\$} \mathbb{K}$
  $\hat{\mathbf{y}}[d] \leftarrow \hat{\mathbf{y}}[d] \oplus r_i$
  $\hat{\mathbf{y}}[i] \leftarrow \hat{\mathbf{x}}[i] \oplus r_i$
fin pour

[Cor+14] ont cependant montré que l'utilisation de la méthode de rafraîchissement de l'algorithme 4 ne permettait pas d'atteindre la sécurité de sondage minimale. Pour obtenir la sécurité de sondage minimale, une nouvelle méthode de rafraîchissement a été proposée dans [DDF14 ;

Bar+16] consistant à appliquer une multiplication masquée ISW entre le masquage d'entrée et le tuple $(1, 0, \ldots, 0)$. Notez que cette méthode de rafraîchissement ne nécessite pas l'évaluation complète de la multiplication masquée ISW, mais peut être calculée comme décrit dans l'algorithme 5. Rafraîchir le masquage d'entrée d'une multiplication masquée implique une perte en performance, mais c'est mieux que de doubler le nombre de parts.

Algorithme 5 Méthode de rafraîchissement basée sur la multiplication masquée ISW [Bar+16]

Entrée: Masquage $\hat{\mathbf{x}} \in \mathbb{K}^{d+1}$ de x
Sortie: Masquage $\hat{\mathbf{y}} \in \mathbb{K}^{d+1}$ de y tel que $\bigoplus_i \hat{\mathbf{y}}[i] = x$

pour $i = 0$ à d **faire**
  $\hat{\mathbf{y}}[i] \leftarrow \hat{\mathbf{x}}[i]$
fin pour
pour $i = 0$ à d **faire**
  pour $j = i + 1$ à d **faire**
    $r_{i,j} \xleftarrow{\$} \mathbb{K}$
    $\hat{\mathbf{y}}[i] \leftarrow \hat{\mathbf{y}}[i] \oplus r_{i,j}$
    $\hat{\mathbf{y}}[j] \leftarrow \hat{\mathbf{y}}[j] \oplus r_{i,j}$
  fin pour
fin pour

3.4.3 Implémentations de seuil

Les *implémentations de seuil*, abrégées en TI pour «*threshold implementation*» en anglais, ont été introduites dans [NRR06 ; NRS11] pour protéger les implémentations matérielles contre un attaquant pouvant sonder une variable. L'idée centrale de TI est le concept d'*incomplétude*. Un circuit combinatoire est incomplet si chaque sous-circuit contribuant au calcul d'une part de sortie n'utilise pas toutes les parts d'entrée. TI a ensuite été étendue par [Bil+14b] pour se protéger d'un attaquant ayant plusieurs sondes. [Bil+14b] ont introduit la notion d'*incomplétude d'ordre d*. Un circuit combinatoire est incomplet d'ordre d si tout ensemble de d sous-circuits contribuant au calcul d'une part de sortie n'utilise pas toutes les parts d'entrée. La sécurité dans le modèle de sondage découle immédiatement de l'incomplétude. TI est également sécurisée en présence de glitchs [NRR06 ; NRS11]. En effet, l'incomplétude assure que chaque sous-circuit combinatoire calculant une part de sortie n'utilise pas toutes les parts d'entrée et ne fournit donc aucune information sur l'entrée non masquée même en présence de glitchs. TI a été appliquée à de nombreuses implémentations matérielles masquées [Mor+11 ; Bil+14a ; De +15 ; UHA17]. Cependant, le niveau de sécurité apporté par TI n'est pas optimal, car une implémentation masquée à l'ordre d est seulement sécurisée par sondage de $d/2$ sondes. Comme pour le schéma ISW, le nombre de parts est doublé par rapport au nombre de sondes.

Exemple Nous illustrons ce concept en développant une implémentation de seuil masquée à l'ordre 2 de la porte AND réalisant $c = a \otimes b$. On cherche à définir trois circuits f_0, f_1 et f_2 qui calculent respectivement $\hat{\mathbf{c}}[0]$, $\hat{\mathbf{c}}[1]$ et $\hat{\mathbf{c}}[2]$ de sorte que l'incomplétude soit vérifiée. La méthode consiste à exprimer la forme normale algébrique du circuit à masquer en fonction des parts

d'entrée et de distribuer en conséquence chaque terme dans les parts de sortie. Une solution est donnée par :

$$\begin{aligned}\hat{\mathbf{c}}[0] =&(\hat{\mathbf{a}}[1] \otimes \hat{\mathbf{b}}[1]) \oplus (\hat{\mathbf{a}}[1] \otimes \hat{\mathbf{b}}[2]) \oplus (\hat{\mathbf{a}}[2] \otimes \hat{\mathbf{b}}[1])\\ \hat{\mathbf{c}}[1] =&(\hat{\mathbf{a}}[0] \otimes \hat{\mathbf{b}}[2]) \oplus (\hat{\mathbf{a}}[2] \otimes \hat{\mathbf{b}}[0]) \oplus (\hat{\mathbf{a}}[2] \otimes \hat{\mathbf{b}}[2])\\ \hat{\mathbf{c}}[2] =&(\hat{\mathbf{a}}[0] \otimes \hat{\mathbf{b}}[0]) \oplus (\hat{\mathbf{a}}[0] \otimes \hat{\mathbf{b}}[1]) \oplus (\hat{\mathbf{a}}[1] \otimes \hat{\mathbf{b}}[0])\end{aligned}$$

On note que $\hat{\mathbf{c}}[i]$ ne dépend ni de $\hat{\mathbf{a}}[i]$ ni de $\hat{\mathbf{b}}[i]$ garantissant ainsi l'incomplétude. Cette porte AND masquée est donc sécurisée par sondage d'une sonde. Cette méthode peut être appliquée à d'autres fonctions avec différents nombres de parts d'entrée ou de sortie.

Composition Les circuits combinatoires incomplets peuvent être composés en intercalant des registres de synchronisation pour arrêter la propagation des glitchs. Cependant, une condition préalable importante est que chaque circuit combinatoire incomplet a un masquage uniforme en entrée. La composition proposée par [Bil+14b] s'est ainsi avérée vulnérable dans [Rep15] par manque d'uniformité des masquages d'entrée. Pour assurer l'uniformité des masquages d'entrée, [Rep15] ont proposé d'ajouter des aléas supplémentaires avant les registres de synchronisation (par exemple en rafraîchissant les masquages, voir 3.4.2).

3.4.4 Propriété de composition NI et SNI

Dans une première tentative de proposer une propriété de composition générique, [Bar+15] ont introduit la notion de *non-interférence*, abrégée en NI, pour assurer la sécurité de la composition d'implémentations masquées en utilisant le concept de *simulation d'une sonde* (voir sous-section 3.3.2). Cette propriété assure que les sondes internes et les sondes de sortie sont simulables par un sous-ensemble limité des parts d'entrée de l'implémentation masquée, comme illustré sur la figure 3.10.

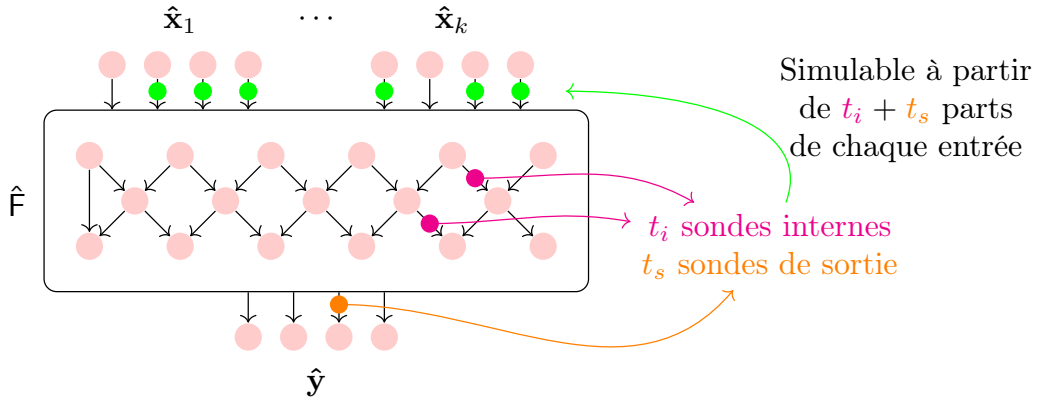


FIGURE 3.10 – 3-NI illustré sur une implémentation masquée à l'ordre 3.

La non-interférence s'est cependant avérée insuffisante pour assurer la sécurité de la composition d'implémentations masquées [Bar+15]. La *non-interférence forte* [Bar+16], abrégée en SNI pour «*strong non-interference*» en anglais, a alors été proposée pour limiter encore davantage le nombre de parts nécessaires pour simuler les sondes, comme le montre la figure 3.11. Les propriétés de composition NI/SNI étaient initialement limitées aux implémentations masquées

à une seule sortie, puis ont été étendues dans [CS20] pour couvrir également les implémentations masquées à sorties multiples. La définition suivante décrit la propriété NI/SNI de façon plus formelle.

Definition 1 (t-non-interférence (forte) [Bar+16])**.** *Une implémentation masquée avec k entrées et une sortie est t-NI (resp., t-SNI), si pour chaque ensemble $\mathcal{I}$ de t_i sondes internes et pour chaque ensemble $\mathcal{O}$ de t_s sondes de sortie telles que $t_i + t_s \leq t$, l'ensemble de sondes $\mathcal{I} \cup \mathcal{O}$ peut être simulé avec $t_i + t_s$ (resp., t_i) parts de chaque entrée.*

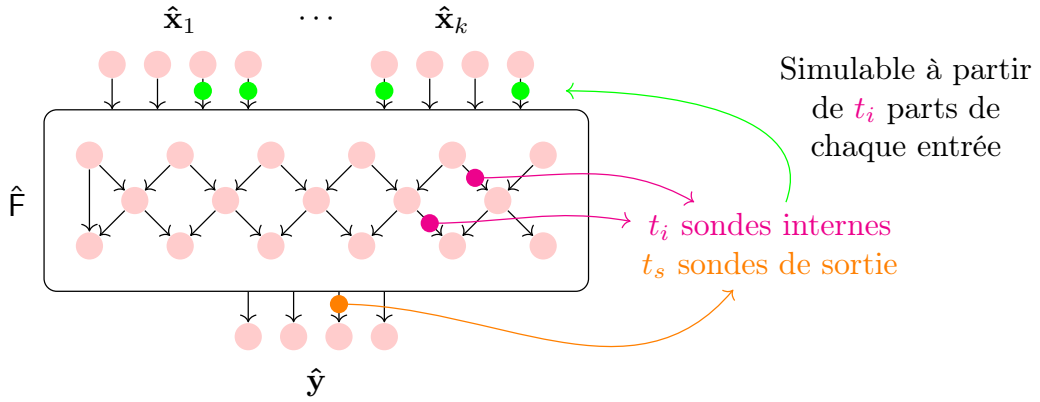


FIGURE 3.11 – 3-SNI illustré sur une implémentation masquée à l'ordre 3.

[Bar+16] ont montré que t-NI implique la sécurité dans le modèle de sondage à t sondes. En effet, l'ensemble des sondes pouvant être simulé par au plus t parts de chaque masquage d'entrée, elles sont indépendantes de toutes variables sensibles tant que t est inférieur ou égal à l'ordre de masquage. De plus, t-SNI implique t-NI et donc la sécurité dans le modèle de sondage à t sondes.

Composition La différence entre SNI et NI réside uniquement dans le nombre de parts d'entrée pouvant être utilisé pour la simulation. En rendant ce nombre indépendant du nombre de sondes de sortie, SNI assure une séparation entre les parts de sortie et d'entrée, ce qui permet une composition générique. En effet, toute composition d'implémentations masquées t-SNI (dans laquelle deux masquages d'entrée d'une implémentation masquée ne sont jamais identiques) est également t-SNI [Bar+16], et donc sécurisée dans le modèle de sondage avec t sondes. Utiliser uniquement des implémentations masquées SNI dans la composition peut induire des coûts importants. Pour réduire les coûts, [Bar+16] ont présenté une méthode de composition sécurisée utilisant à la fois des implémentations masquées NI et SNI. Ils ont montré qu'une composition d'implémentations masquées est d-NI si toutes les implémentations masquées sont d-NI ou d-SNI, selon la règle de composition suivante : chaque masquage est utilisé au plus une fois comme entrée d'une implémentation masquée autre qu'un rafraîchissement d-SNI. [Bar+16] ont développé un compilateur pour masquer une implémentation et rafraîchir les masquages nécessaires pour assurer une composition sécurisée. Ce compilateur génère une implémentation masquée en langage C. Ce compilateur a ensuite été amélioré par [BGR18] pour rafraîchir le nombre minimum de variables masquées dans une implémentation masquée.

Exemples d'implémentations masquées (S)NI La fonction linéaire masquée à l'ordre d présentée à la section 3.3.2 est d-NI car toutes les sondes sur $\mathsf{L}(\hat{\mathbf{x}}[i])$ sont simulables par la part d'entrée $\hat{\mathbf{x}}[i]$. Cependant, cette implémentation masquée n'est pas d-SNI, car pour simuler d parts de sortie différentes, un simulateur doit avoir accès aux d parts d'entrée correspondantes. On peut rendre d-SNI l'implémentation masquée d'une fonction linéaire en appliquant au masquage de sortie un rafraîchissement d-SNI. La multiplication masquée ISW à l'ordre d de l'algorithme 3 a été démontrée d-SNI par [Bar+16]. Ils ont également montré que la méthode de rafraîchissement basée sur la multiplication masquée ISW de l'algorithme 5 est d-SNI alors que la méthode de rafraîchissement de l'algorithme 4 est seulement d-NI. La vulnérabilité de composition du schéma de [RP10] démontrée par [Cor+14] provient du fait que la méthode de rafraîchissement de l'algorithme 4 n'est pas d-SNI.

3.4.5 Propriété de composition PINI

NI et SNI offrent une méthode simple de composer en toute sécurité des implémentations masquées. Cependant, cette méthode nécessite l'utilisation de nombreux rafraîchissements de masquage coûteux. Les rafraîchissements de masquage sont nécessaires car la fonction linéaire masquée n'est pas SNI. Cela suggère naturellement la recherche d'une propriété assurant la sécurité de la composition d'implémentations masquées, tout en tirant parti de l'efficacité du masquage des fonctions linéaires.

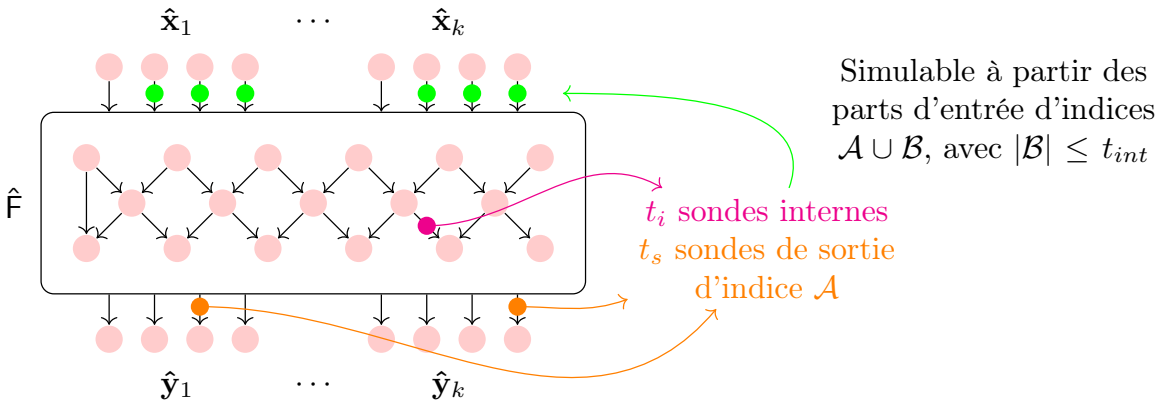


FIGURE 3.12 – 3-PINI illustré sur une implémentation masquée à l'ordre 3.

Non-interférence isolant la sonde [CS20] ont proposé une propriété de composition appelée *non-interférence isolant la sonde*, abrégée en PINI pour «*probe isolating non-interference*» en anglais. Pour qu'une implémentation masquée soit PINI, chaque sonde de sortie doit être simulable avec les parts d'entrée de même indice que la sonde de sortie, tandis que chaque sonde interne doit être simulable avec les parts d'entré d'un indice unique (mais arbitraire). La figure 3.12 illustre la propriété PINI avec une implémentation masquée à l'ordre 3. La définition suivante décrit la propriété PINI de façon plus formelle.

Definition 2 (t-non-interférence isolant la sonde (PINI) [CS20])**.** *Une implémentation masquée est t-PINI, si pour chaque ensemble $\mathcal{I}$ de t_i sondes internes, pour chaque ensemble $\mathcal{A}$ de t_o indices de parts, tels que $t_i + t_o \leq t$, il existe un ensemble $\mathcal{B}$ d'au plus t_i indices de parts tels que l'ensemble de sondes $\mathcal{I} \cup \hat{y}[\mathcal{A}]$ peut être simulé avec les parts d'entrée d'indices $\mathcal{A} \cup \mathcal{B}$.*

[CS20] ont montré que toute implémentation masquée d-PINI est sécurisée dans le modèle de sondage avec d sondes. De plus, le masquage d'une application linéaire est d-PINI. La propriété PINI permet également une composition directe car la composition d'implémentations masquées d-PINI est d-PINI. [Cas+20] ont montré que la composition directe d'implémentations masquées d-PINI restait sécurisée dans le modèle de sondage robuste aux glitchs. Pour faire face aux transitions de données entre différents cycles d'horloge, [CS21] ont défini la notion plus forte de *non-interférence d'isolation de sonde de sortie*, abrégée en O-PINI pour «*output probe isolating non-interference*» en anglais. O-PINI permet de composer de façon sécurisée dans le modèle de sondage robuste aux transitions et glitchs. Une grande variété d'implémentations masquées PINI ont été récemment introduites [CS20 ; Cas+20 ; KSM21 ; CS21 ; KM22]. Ils diffèrent par le nombre d'aléas nécessaires, leur latence et leur robustesse contre les glitchs et les transitions.

3.5 Multiplication masquée

Dans cette section, nous établissons un état de l'art des différentes multiplications masquées proposées dans la littérature. Nous nous intéresserons d'abord aux multiplications masquées en logiciel, puis à celles masquées en matériel.

3.5.1 Multiplication masquée en logiciel

Multiplication masquée (S)NI [Bar+16] ont montré que la multiplication masquée ISW de l'algorithme 3 ainsi que la méthode de rafraîchissement de l'algorithme 5 sont d-SNI. [Bel+16] ont ensuite proposé une multiplication masquée d-NI avec un coût en aléas réduit à $d^2/4 + d$, et qui peut être facilement instanciée sur n'importe quel corps fini de caractéristique 2. Bien que la propriété de composition vérifiée soit plus faible que celle obtenue par la multiplication masquée ISW, elle peut être utilisée en pratique pour réduire la complexité des implémentations masquées. Par exemple, on peut remplacer la moitié des multiplications dans l'implémentation masquée de la SBOX de AES (voir figure 4.6) par celles de [Bel+16] tout en maintenant la sécurité globale d-SNI. Dans [Bel+17], les mêmes auteurs proposent deux nouvelles multiplications masquées d-SNI applicables sur tout corps fini autre que GF(2), l'une avec un nombre linéaire de multiplications et l'autre avec un coût linéaire d'aléas. [Bar+17] ont introduit des multiplications masquées et méthodes de rafraîchissement NI et SNI efficaces en logiciel qui ont ensuite été améliorées dans [Gré+18 ; Gou+18 ; Bar+19a]. [Bat+16] ont proposé une multiplication masquée et une méthode de rafraîchissement SNI renforçant la sécurité contre les attaques horizontales et ayant une complexité en nombres aléatoires en $\mathcal{O}(d \log d)$.

Multiplication masquée PINI [CS20] ont montré que la multiplication masquée «double SNI» proposée dans [GR17], qui consiste à appliquer un rafraîchissement SNI à un des masquages d'entrée de chaque multiplication masquée SNI, est d-PINI. Cette méthode est cependant assez conservative et induit des coûts importants en temps d'exécution et en nombre

d'aléas. Pour réduire les coûts, [CS20] ont introduit la multiplication masquée à l'ordre d décrite dans l'algorithme 6, nommée PINI_1, qui est d-PINI et nécessite $d(d+1)/2$ aléas. PINI_1 est inspirée de la multiplication masquée ISW, en remplaçant le calcul de $\hat{\mathbf{x}}[i] \otimes \hat{\mathbf{y}}[j] \oplus r_{i,j}$ par $\hat{\mathbf{x}}[i] \otimes (\hat{\mathbf{y}}[j] \oplus r_{i,j}) \oplus (\hat{\mathbf{x}}[i] \oplus 1) \otimes r_{i,j}$. Aucune des valeurs intermédiaires de ce calcul ne nécessite la connaissance à la fois de $\hat{\mathbf{x}}[i]$ et de $\hat{\mathbf{y}}[j]$ pour être simulée.

Algorithme 6 Multiplication masquée PINI_1 de [CS20].

Entrée: Masquage $\hat{\mathbf{x}} \in \mathbb{K}^{d+1}$ de x et masquage $\hat{\mathbf{y}} \in \mathbb{K}^{d+1}$ de y
Sortie: Masquage $\hat{\mathbf{z}} \in \mathbb{K}^{d+1}$ de z tel que $\bigoplus_i \hat{\mathbf{z}}[i] = xy$

pour $i = 0$ à d **faire**
  pour $j = i + 1$ à d **faire**
    $r_{i,j} \xleftarrow{\$} \mathbb{K}$
    $u_{i,j} \leftarrow \hat{\mathbf{x}}[i] \otimes (\hat{\mathbf{y}}[j] \oplus r_{i,j}) \oplus (\hat{\mathbf{x}}[i] \oplus 1) \otimes r_{i,j}$
    $u_{j,i} \leftarrow \hat{\mathbf{x}}[j] \otimes (\hat{\mathbf{y}}[i] \oplus r_{i,j}) \oplus (\hat{\mathbf{x}}[j] \oplus 1) \otimes r_{i,j}$
  fin pour
fin pour
pour $i = 0$ à d **faire**
  $\hat{\mathbf{z}}[i] \leftarrow \hat{\mathbf{x}}[i] \otimes \hat{\mathbf{y}}[i] \oplus \bigoplus_{j=0, j\neq i}^{d} u_{i,j}$
fin pour

3.5.2 Multiplication masquée en matériel

Multiplication masquée (S)NI Plusieurs multiplications masquées en matériel ont été proposées sans preuve de sécurité à un ordre arbitraire. Nous pouvons notamment citer le *schéma de masquage consolidé* (CMS) [Rep+15 ; Cnu+16], le *masquage orienté domaine* (DOM) [GMK16 ; GMK17], l'*approche de masquage unifiée* (UMA) [GM17 ; GM18] et le *masquage générique à faible latence* (GLM) [GIB18]. [Moo+19] a montré que CMS, CMS et GLM avaient des failles de sécurité (locales ou de composabilité) dans le modèle de sondage robuste. Seule la multiplication masquée DOM proposée par [GMK16] et décrite dans l'algorithme 7 s'est avérée sécurisée dans le modèle de sondage robuste aux glitchs. [Fau+18] ont en effet montré que la multiplication DOM masquée à l'ordre d est d-NI et qu'elle peut être rendue d-SNI en stockant les parts de sortie dans des registres. Pour réduire la latence de la multiplication masquée d-SNI DOM à un cycle d'horloge, [MPZ21] ont adapté la solution basée sur le *schéma de masquage consolidé* proposée dans [Rep+15] en ajoutant des sommes pré-calculées de bits aléatoires.

Multiplication masquée PINI [Cas+20] ont proposé plusieurs multiplications masquées en matériel. Ils ont tout d'abord montré que la multiplication «double SNI» utilisant la multiplication DOM était PINI en présence de glitchs. «Double SNI» a une latence de 4 cycles d'horloge et nécessite $d(d+1)$ aléas. Plusieurs multiplications masquées PINI, nommées HPC pour «*hardware private circuits*» en anglais, ont également été proposées dans [Cas+20] pour améliorer les performances. HPC1 est une multiplication DOM dont on a ajouté à une des entrées un masquage de 0 généré par un rafraîchissement SNI. HPC1 nécessite le même nombre d'aléas que «double SNI», mais a une latence divisée par deux. [Cas+20] ont proposé la multiplication

Algorithme 7 Multiplication masquée DOM de [GMK16]. Le crochet $[\cdot]_{\mathsf{reg}}$ indiquent que l'argument est stocké dans un registre.

Entrée: Masquage $\hat{\mathbf{x}} \in \mathbb{K}^{d+1}$ de x et masquage $\hat{\mathbf{y}} \in \mathbb{K}^{d+1}$ de y
Sortie: Masquage $\hat{\mathbf{z}} \in \mathbb{K}^{d+1}$ de z tel que $\bigoplus_i \hat{\mathbf{z}}[i] = x \otimes y$

pour $i = 0$ à d **faire**
  pour $j = i + 1$ à d **faire**
    $r_{i,j} \xleftarrow{\$} \mathbb{K}$
    $u_{i,j} \leftarrow [\hat{\mathbf{x}}[i] \otimes \hat{\mathbf{y}}[j] \oplus r_{i,j}]_{\mathsf{reg}}$
    $u_{j,i} \leftarrow [\hat{\mathbf{x}}[j] \otimes \hat{\mathbf{y}}[i] \oplus r_{i,j}]_{\mathsf{reg}}$
  fin pour
fin pour
pour $i = 0$ à d **faire**
  $\hat{\mathbf{z}}[i] \leftarrow \hat{\mathbf{x}}[i] \otimes \hat{\mathbf{y}}[i] \oplus \bigoplus_{j=0, j\neq i}^{d} u_{i,j}$
fin pour

masquée HPC2, adaptée de PINI_1, qui réduit le nombre d'aléas à $d(d+1)/2$ avec une latence de deux cycles d'horloge. [KSM21] ont présenté GHPC qui permet la construction d'implémentations masquées PINI réalisant n'importe quelle fonction booléenne mais limitée à la sécurité de premier ordre dans le modèle de sondage robuste aux glitchs. GHPC ne nécessite qu'un aléa par bit de sortie et 2 cycles d'horloge, mais induit une surface relativement importante. Dans [KSM21], les auteurs ont en outre introduit GHPCLL, une variante à faible latence de GHPC, qui ne nécessite qu'un seul étage de registre pour calculer toute fonction booléenne, mais nécessite $2n$ nouveaux bits aléatoires par bit de sortie pour une fonction booléenne à n entrées. Pour réduire la latence à un cycle d'horloge, [KM22] propose la multiplication masquée PINI nommée HPC3. En contrepartie, le nombre d'aléas nécessaire est doublé par rapport à HPC2. Pour sécuriser dans le modèle de sondage robuste aux glitchs et transitions, [CS21] réalise une multiplication masquée nommée O-PINI1 qui est O-PINI. Leur construction est directement basée sur HPC2, ajoutant seulement un rafraîchissement supplémentaire en sortie. Cependant, cette multiplication masquée est soit résistante aux glitchs, soit résistante aux transitions, mais

Nom	Réf.	Fonction	Ordre	Composition	Latence	Nb. bits d'aléas
DOM	[GMK16]	AND	$d \geq 1$	SNI	2	$d(d+1)/2$
CML_{LL}	[MPZ21]	AND	$d \geq 1$	SNI	1	$2(d+1)^2$
Double SNI	[Cas+20]	AND	$d \geq 1$	PINI	4	$d(d+1)$
HPC1	[Cas+20]	AND	$d \geq 1$	PINI	2	$d(d+1)$
HPC2	[Cas+20]	AND	$d \geq 1$	PINI	2	$d(d+1)/2$
GHPC	[KSM21]	$\mathsf{F} : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$	1	PINI	2	m
GHPC_{LL}	[KSM21]	$\mathsf{F} : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$	1	PINI	1	$2^n \times m$
HPC3	[KM22]	AND	$d \geq 1$	PINI	1	$d(d+1)$
O-PINI2	[CS21]	AND	$d \geq 1$	O-PINI	3	$d(d+1)/2 + d$
HPC3^+	[KM22]	AND	$d \geq 1$	O-PINI	2	$d(d+1) + d$

TABLE 3.1 – Résumé des différentes multiplications masquées en matériel.

pas les deux en même temps. [CS21] introduit O-PINI2 dans le but d'obtenir une composabilité directe en présence de transitions et de glitchs en même temps. Par rapport à O-PINI1, des registres supplémentaires sont introduits après le rafraîchissement engendrant une latence globale de 3 cycles d'horloge. [KM22] ont ensuite adapté HPC3 pour sécuriser contre les transitions et les glitchs en seulement 2 cycles d'horloge. Le tableau 3.1 résume les différentes méthodes de l'état de l'art pour masquer la multiplication en matériel.

Chapitre 4

Techniques d'implémentations

Dans ce chapitre, nous décrivons les différentes techniques d'implémentations logicielles du chiffrement AES. Ces implémentations sont basées sur du calcul direct, sur des tables précalculées, ou sur une technique appelée «*bit slicing*». Dans la littérature, différentes implémentations logicielles masquées de AES ont également été proposées. Elles utilisent les différentes techniques décrites au chapitre 3, ainsi que d'autres méthodes telles que les tables masquées, le «*bit slicing*» masqué ou le «*share slicing*» qui seront présentées dans la suite.

Dans la section 4.1, nous décrivons les implémentations logicielles du chiffrement AES ne comportant pas de protection contre l'attaque SCA. Dans la section 4.2, nous introduisons différentes techniques de masquage du chiffrement AES de l'état de l'art.

4.1 Implémentations logicielles de AES

4.1.1 Calcul direct

Certaines implémentations de AES favorisent le calcul direct, réduisant ainsi l'empreinte mémoire au prix d'une latence accrue. Dans [DR02] est proposée une implémentation, que nous nommons SW-BW, consistant à placer chaque octet de l'état et de la clé de ronde dans des registres différents et à calculer chaque fonction de ronde de manière directe. Sur des plateformes 32 bits, [Ber+02] ont proposé une implémentation, que nous nommons SW-RW, qui améliore la latence d'exécution en plaçant chaque ligne de l'état et de la clé de ronde dans des registres différents. De cette façon, ShiftRows peut être implémentée à l'aide d'instructions de rotation.

Implémentation de la SBOX Il existe différentes possibilités pour implémenter en logiciel la fonction SBOX. Une possibilité est de calculer la fonction SBOX au moment de l'exécution en utilisant les instructions de base du processeur. La SBOX de AES réalise le calcul de l'inverse dans GF(2^8) suivi d'une application linéaire sur GF(2). En utilisant le petit théorème de Fermat, on obtient $x^{-1} = x^{254}$ et on peut réduire le calcul de x^{254} en une séquence de multiplications et de carrés décrite à la figure 4.1. Ce type d'implémentation induit cependant une latence d'exécution élevée car les jeux d'instructions de base sont rarement efficaces pour calculer sur des corps finis. Alternativement, la fonction SBOX peut être implémentée sous la forme d'une table contenant les 256 valeurs possibles. Cela nécessite le calcul préalable et le stockage d'une table de 256 octets, mais réduit considérablement la latence d'exécution.

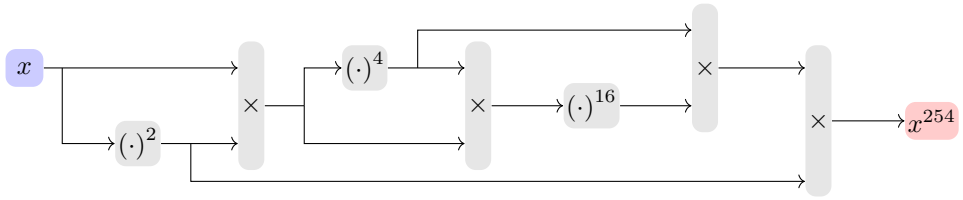


FIGURE 4.1 – Inversion dans GF(2^8).

4.1.2 Tables pré-calculées

Les implémentations de AES utilisant des tables pré-calculées favorisent le pré-calcul, réduisant la latence mais augmentant l'empreinte mémoire. Le principal exemple de cette technique est la méthode dite des T-tables [DR02], que nous nommons SW-TTABLE. Cette technique place chaque colonne de l'état et de la clé de ronde dans des registres différents. Puis, MixColumn $\circ$ SubBytes est pré-calculée à l'aide de 4 tables. Chaque ronde devient une séquence de recherches de table et de XOR pour combiner leurs résultats. Cela nécessite le pré-calcul et le stockage de 1 ko par table. De plus, dans les systèmes avec des caches de données, les implémentations basées sur des tables sont sensibles aux attaques temporelles [Tsu+03 ; Ber05 ; BM06].

4.1.3 Implémentation «bit slicing»

Le «*bit slicing*», abrégé en BS, est une technique d'implémentation introduite par [Bih97], pour favoriser le parallélisme dans les calculs booléens. Sans BS, chaque registre de 32 ou 64 bits ne contient que 1 bit d'information, ce qui n'est pas optimal. En regroupant plusieurs bits de données indépendantes dans un même registre, on obtient des calculs booléens sur 32 ou 64 bits en même temps à chaque cycle.

	Bloc 0	Bloc 1	$\cdots$	Bloc 31
$\mathtt{M}_0$	b_0^0	b_0^1	$\cdots$	b_0^{31}
$\mathtt{M}_1$	b_1^0	b_1^1	$\cdots$	b_1^{31}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathtt{M}_{127}$	b_{127}^0	b_{127}^1	$\cdots$	b_{127}^{31}

FIGURE 4.2 – Représentation «*bit slicing*» de 32 blocs de 128 bits dans 128 mots machines ($\mathtt{M}_i, 0 \leq i \leq 127$) d'une architecture 32 bits. b_i^j désigne le i-ième bit du j-ième bloc.

Implémentation BS de cryptosystèmes Dans une implémentation BS d'un cryptosystème, les blocs en clair sont d'abord convertis en représentation BS en réalisant une transposition. Sur une architecture avec des registres de k bits, k blocs en clair de l bits sont transposés dans l mots machines de k bits, où le mot machine j contient le j-ième bit de chaque bloc en clair. Par exemple, une implémentation BS de AES sur un processeur 32 bits transpose 32 blocs

en clair de 128 bits dans 128 mots machines de 32 bits. Cette représentation BS est illustrée sur la figure 4.2. Transposer les blocs en clair en représentation BS peut être une opération assez coûteuse. Une méthode simple est de transposer les blocs bit par bit comme illustrée dans le code de la figure 4.3. Cet algorithme de transposition peut être amélioré en transposant les blocs en clair de façon récursive [Knu98] (voir annexe A). Le cryptosystème est ensuite exprimé sous la forme d'un circuit F composé de portes booléennes élémentaires (par exemple AND, XOR, OR, NOT). Le circuit F chiffre alors en parallèle les k blocs en clair, et retourne les k blocs chiffrés en représentation BS. On obtient finalement les k blocs chiffrés en réalisant la transposition inverse de la conversion en représentation BS. Le BS produit ainsi des implémentations de chiffrement à haut débit en chiffrant plusieurs blocs en parallèle. Dans la suite, nous désignons par SW-BS une implémentation BS de AES.

```
void transposition(uint32_t data[32],uint32_t transposed_data[32])
    for (int i = 0; i < 32; i++) {
            transposed_data[i] = 0;
    }

    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
            transposed_data[j] |= ((data[i] >> j) & 1) << i;
        }
    }
}
```

FIGURE 4.3 – Code C de la transposition simple de 32 blocs de 32 bits en représentation BS.

Avantages et inconvénients Le BS permet de produire des implémentations de chiffrement à haut débit grâce au parallélisme des données dans un même mot machine. De plus, les implémentations BS de cryptosystèmes s'exécutent en temps constant. En effet, les conditions en BS sont réalisés en exécutant chaque branchement possible, puis en conservant le résultat correct. Cette méthode peut devenir coûteuse pour les grandes conditions, car elle nécessite le calcul de toutes les branches possibles. Ce n'est cependant pas un problème car les cryptosystèmes symétriques ont peu de conditions à protéger. Les implémentations BS empêchent également les attaques ciblant les caches [Ber05] car les tables pré-calculées sont remplacées par des circuits s'exécutant en temps constant. Cependant, le BS possède quelques limitations. Tout d'abord, de nombreux blocs en clair doivent être chiffrés en parallèle pour obtenir les meilleures performances. Par exemple, dans une architecture 32 bits, 32 blocs en clair doivent être chiffrés en parallèle pour être efficaces. Ces différents blocs en clair peuvent être chiffrés en utilisant le mode de chiffrement CTR qui permet le chiffrement parallèle de blocs consécutifs. Cependant, BS empêche l'utilisation des modes de chiffrement CBC et OFB, ainsi que tout autre mode dans lequel le chiffrement d'un bloc en clair utilise le chiffré précédent. Pour utiliser le BS avec ces modes de chiffrement, une solution est de chiffrer des blocs indépendants en parallèle. Cette méthode induit néanmoins un coût lié à la gestion des différents blocs. Une autre limitation est que le BS utilise beaucoup de variables, ce qui engendre une pression sur les registres généraux du processeur. De nombreux déplacements de données entre les registres et la mémoire sont nécessaires, réduisant les performances. Par exemple, 32 blocs en clair d'AES sont représentés en BS sous la forme de 128 registres de 32 bits, ce qui est nettement supérieur au nombre de registres généraux dans la plupart des processeurs.

Usuba Usuba [MD19] est un langage pour décrire des implémentations BS. Un compilateur permet de synthétiser un code Usuba en une implémentation logicielle BS en langage C. De nombreuses optimisations sont réalisées par le compilateur. Le compilateur utilise par exemple un algorithme d'ordonnancement des instructions visant à minimiser la pression sur les registres du processeur. Il a été utilisé avec succès pour déployer des primitives cryptographiques sur des architectures Intel, PowerPC, Arm et Sparc.

```
table SBOX (a:v8) returns (b:v8) {
    99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,
    ...
    140,161,137,13,191,230,66,104,65,153,45,15,176,84,187,22}

node SubBytes (a:b8[16]) returns (b:b8[16]) let
    forall i in [0,15] {
        b[i] = SBOX(a[i])
    } tel

node ShiftRows (a:b8[16]) returns (b:b1[128]) let
    b = a[ 0,  5, 10, 15,
           4,  9, 14,  3,
           8, 13,  2,  7,
          12,  1,  6, 11] tel

node times2 (a:b1[8]) returns (b:b1[8]) let
    b=(a<<1)^(0,0,0,a[0],a[0],0,a[0],a[0]); tel

node times3 (a:b1[8]) returns (b:b1[8]) let
    b=times2(a)^a; tel

node MixColumn_single (a:b8[4]) returns (b:b8[4]) let
    b[0]=times2(a[0])^times3(a[1])^a[2]^a[3];
    b[1]=a[0]^times2(a[1])^times3(a[2])^a[3];
    b[2]=a[0]^a[1]^times2(a[2])^times3(a[3]);
    b[3]=times3(a[0])^a[1]^a[2]^times2(a[3]); tel

node MixColumn (a:b32[4]) returns (b:b32[4]) let
    forall i in [0,3] {
        b[i] = MixColumn_single(a[i])
    } tel

node AddRoundKey(a:b128,key:b128) returns (b:b128) let
    b = a ^ key tel

node AES128 (plain:b128,key:b128[11]) returns (cipher:b1[128])
vars tmp : b128[10] let
    tmp[0]=AddRoundKey(plain, key[0]);
    forall i in [1,9] {
    tmp[i]=AddRoundKey(MixColumn(ShiftRows(SubBytes(tmp[i-1]))),key[i]);
    }
    cipher=AddRoundKey(ShiftRows(SubBytes(tmp[9])),key[10]); tel
```

FIGURE 4.4 – Code Usuba d'une implémentation BS de AES, extrait du code source de Usuba [1].

Le code Usuba d'une implémentation BS de AES est montré à la figure 4.4. Des types spécifiques permettent de manipuler des données en représentation BS. Par exemple, le type `b128` représente un bloc de 128 bits en représentation BS. Pour structurer les programmes,

Usuba utilise des nodes, équivalant aux fonctions dans d'autres langages. Un node spécifie un ensemble de valeurs d'entrée, de valeurs de sortie ainsi qu'un système d'équations reliant ces variables. Pour implémenter SubBytes, la table SBOX est appliquée à chaque octet. La table SBOX est définie par une table de vérité. Un circuit optimisé de la SBOX est intégré au compilateur, permettant de réaliser la SBOX en 113 opérations booléennes. ShiftRows est calculée en mélangeant les registres, puisque chaque registre contient un bit de chaque élément de l'état. Une implémentation spécifique de MixColumns est réalisée pour calculer sur des variables en représentation BS.

4.1.4 Implémentation m-slicing

Le *m-slicing* est une technique d'implémentation, introduite dans [KS09], qui généralise le BS. Comme le BS, le *m-slicing* permet de chiffrer plusieurs blocs en parallèle. Dans une implémentation *m-slicing*, un bloc en clair est divisé en morceaux de m bits et chaque morceau est placé dans un mot machine différent. Le cas $m = 1$ correspond au BS. Si chaque mot machine contient k bits, k/m blocs en clair indépendants peuvent être chiffrés en parallèle au lieu de k pour une implémentation BS. Cette représentation nécessite donc moins de blocs en clair pour remplir les registres et ainsi maximiser le débit. Par exemple, sur une architecture 32 bits, seulement 2 blocs en clair indépendants sont nécessaires pour le 16-*slicing* contre 32 blocs en clair pour le BS. Il en résulte une pression sur les registres généraux réduite. Dans la suite, une implémentation *m-slicing* de AES est nommée SW-mS, avec m un entier positif.

	ligne 3							·····	ligne 0						
	colonne 0			···	colonne 3			·····	colonne 0			···	colonne 3		
	bloc 0	···	bloc 7	···	bloc 0	···	bloc 7	·····	bloc 0	···	bloc 7	···	bloc 0	···	bloc 7
$\mathtt{M}_0$	b^0_{24}	···	b^7_{24}	···	b^0_{120}	···	b^7_{120}	·····	b^0_0	···	b^7_0	···	b^0_{96}	···	b^7_{96}
$\mathtt{M}_1$	b^0_{25}	···	b^7_{25}	···	b^0_{121}	···	b^7_{121}	·····	b^0_1	···	b^7_2	···	b^0_{97}	···	b^7_{97}
$\mathtt{M}_2$	b^0_{26}	···	b^7_{26}	···	b^0_{122}	···	b^7_{122}	·····	b^0_2	···	b^7_2	···	b^0_{98}	···	b^7_{98}
$\mathtt{M}_3$	b^0_{27}	···	b^7_{27}	···	b^0_{123}	···	b^7_{123}	·····	b^0_3	···	b^7_3	···	b^0_{99}	···	b^7_{99}
$\mathtt{M}_4$	b^0_{28}	···	b^7_{28}	···	b^0_{124}	···	b^7_{124}	·····	b^0_4	···	b^7_4	···	b^0_{100}	···	b^7_{100}
$\mathtt{M}_5$	b^0_{29}	···	b^7_{29}	···	b^0_{125}	···	b^7_{125}	·····	b^0_5	···	b^7_5	···	b^0_{101}	···	b^7_{101}
$\mathtt{M}_6$	b^0_{30}	···	b^7_{30}	···	b^0_{126}	···	b^7_{126}	·····	b^0_6	···	b^7_6	···	b^0_{102}	···	b^7_{102}
$\mathtt{M}_7$	b^0_{31}	···	b^7_{31}	···	b^0_{127}	···	b^7_{127}	·····	b^0_7	···	b^7_7	···	b^0_{103}	···	b^7_{103}

FIGURE 4.5 – Représentation 16-*slicing* de 8 blocs dans 8 mots machines ($\mathtt{M}_i, 0 \leq i \leq 127$) d'une architecture 128 bits. b^j_i désigne le i-ième bit du j-ième bloc.

Implémentation *m-slicing* [KS09] ont réalisé une implémentation 16-*slicing* de AES sur une architecture 128 bits en représentant 8 blocs de 128 bits dans 8 mots mémoires contenant

1. `https://github.com/usubalang/usuba`

chacun 16 bits de chaque bloc, comme illustré dans la figure 4.5. [SS16] ont proposé une implémentation 16-*slicing* de AES sur un processeur ARM Cortex-M3, qui a ensuite été porté sur une architecture RISC-V 32 bits dans [Sto19]. Les performances ont encore été améliorées dans [AP20] en utilisant une méthode appelée «*fix slicing*». [AP20] proposent également une implémentation 4-*slicing* de AES appelée «*barrel-shiftrows*», permettant de chiffrer 8 blocs en parallèle sur une architecture 32 bits.

4.1.5 Comparaison des différentes solutions

Le tableau 4.1 présente les performances de différentes implémentations logicielles du chiffrement AES sur des processeurs 32 bits. On constate que l'implémentation SW-TTABLE utilisant des tables est la plus performante, mais est vulnérable aux attaques ciblant les caches. L'implémentation SW-RW est beaucoup moins efficace que SW-TTABLE. Les implémentations *m-slicing* permettent d'obtenir des performances proches de l'implémentation SW-TTABLE. L'implémentation SW-16S de [SS16] permet de chiffrer 2 blocs en parallèle en atteignant 1990 cycles par bloc sur un processeur ARM Cortex-M3. [Sto19] ont ensuite portée cette implémentation sur une architecture RISC-V 32 bits, permettant d'atteindre 1990 cycles par bloc. L'implémentation SW-16S de [AP20] utilisant la méthode «*fix slicing*» améliore encore les performances en atteignant 1468 cycles par bloc. Une comparaison plus précise est réalisée au chapitre 6 en réimplémentant ces différentes implémentations sur un même cœur.

Implém.	Source	Cœur	Temps	
			Cycles	Cmp
SW-TTABLE	[Gao+21]	SCARV	1016	base
SW-RW	[Gao+21]	SCARV	2427	×2,4
SW-16S	[SS16]	Cortex-M3	1617	×1,6
SW-16S	[Sto19]	E31	1990	×2,0
SW-16S	[AP20]	E31	1468	×1,4
SW-4S	[AP20]	E31	1263	×1,2

TABLE 4.1 – Performance de différentes implémentations logicielles du chiffrement AES. Le temps d'exécution est donné en nombre de cycles par bloc.

4.2 Implémentations masquées de AES

4.2.1 Schéma de masquage RP

[RP10] ont proposé une implémentation masquée de AES que nous nommons SW-M-RP. Pour masquer l'inversion dans GF(2^8), ils utilisent la décomposition de la figure 4.1. Chaque multiplication dans GF(2^8) est remplacée par la multiplication masquée ISW. Les exponentiations à des puissances de deux étant linéaires dans GF(2^8), elles sont masquées de façon directe. Cependant, on risque de recombiner les parts en composant ainsi les implémentations masquées, car l'indépendance des masquages d'entrée des multiplications n'est plus satisfaite. En effet, lorsqu'on masque la multiplication d'une variable x et d'une variable $y = \mathsf{L}(x)$, où L est une application linéaire, les entrées de la multiplication masquée ISW sont dépendantes

car pour tout $0 \leq i \leq d$, $\hat{\mathbf{y}}[i] = \mathsf{L}(\hat{\mathbf{x}}[i])$. Pour contourner ce problème, [RP10] ont suggéré d'ajouter des valeurs aléatoires en rafraichissant le masquage à certains endroits bien choisis, comme illustré à la figure 4.6. [RP10] ont d'abord proposé d'utiliser le code de rafraîchissement de l'algorithme 4, qui s'est avéré non sécurisé dans [Cor+14]. Le code de rafraîchissement de l'algorithme 5 fut finalement utilisé pour maintenir la sécurité de la composition.

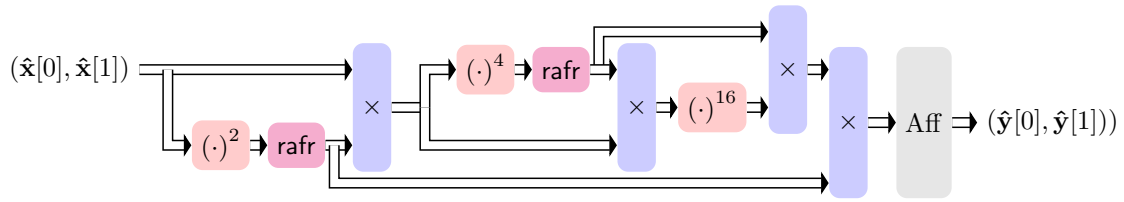


FIGURE 4.6 – Exponentiation modulaire masquée à l'ordre 1 et sécurisée dans le modèle de sondage. rafr désigne un code de rafraîchissement.

4.2.2 Schéma de masquage CPRR

[Cor+14] ont proposé une implémentation masquée de AES que nous nommons SW-M-CPRR. Le schéma de masquage SW-M-CPRR a été initialement proposé pour calculer en toute sécurité des multiplications de la forme $\mathsf{h} : x \mapsto x \otimes \mathsf{L}(x)$ où L est linéaire, sans avoir besoin de rafraîchir le masquage de x. L'algorithme 8 présente l'évaluation masquée de h proposée dans [Cor+14]. Cette évaluation masquée peut être plus efficace que la solution proposée par [RP10] en remplaçant les multiplications de corps coûteuses par de simples recherches de table. En effet, tabuler la multiplication de corps n'est généralement pas réalisable sur des systèmes contraints (la table contient 2^{2n} éléments) alors que tabuler la table de h est facilement réalisable (la table contient 2^n éléments). Le seul inconvénient de cette évaluation masquée est qu'elle double le nombre d'aléas par rapport à la multiplication masquée ISW.

Algorithme 8 Évaluation sécurisée de $\mathsf{h} : x \mapsto x \otimes \mathsf{L}(x)$ [Cor+14]

Entrée: Masquage $\hat{\mathbf{x}} \in \mathbb{K}^{d+1}$ de x
Sortie: Masquage $\hat{\mathbf{z}} \in \mathbb{K}^{d+1}$ de z tel que $\bigoplus_i \hat{\mathbf{z}}[i] = \mathsf{h}(x)$

pour $i = 0$ à d **faire**
  pour $j = i + 1$ à d **faire**
    $r_{i,j} \xleftarrow{\$} \mathbb{K}$
    $r'_{i,j} \xleftarrow{\$} \mathbb{K}$
    $r_{j,i} \leftarrow r_{i,j} \oplus \mathsf{h}(\hat{\mathbf{x}}[i] \oplus r'_{i,j}) \oplus \mathsf{h}(\hat{\mathbf{x}}[j] \oplus r'_{i,j}) \oplus \mathsf{h}((\hat{\mathbf{x}}[i] \oplus r'_{i,j}) \oplus \hat{\mathbf{x}}[j]) \oplus \mathsf{h}(r'_{i,j})$
  fin pour
fin pour
pour $i = 0$ à d **faire**
  $\hat{\mathbf{z}}[i] \leftarrow \mathsf{h}(\hat{\mathbf{x}}[i]) \oplus \bigoplus_{j=0, j\neq i}^{d} r_{i,j}$
fin pour

4.2.3 Tables masquées

[Cha+99 ; Cor14 ; CRZ18] ont proposé des implémentations masquées de AES, que nous nommons SW-M-HTABLE, consistant à pré-calculer une table masquée.

Table masquée à l'ordre 1 Pour évaluer une SBOX dans le domaine masqué, [Cha+99 ; Mes00a] ont proposé de masquer directement la table. Plus précisément, pour appliquer une table S à une variable x, on précalcule d'abord une table masquée T :

$$\mathsf{T}(x) = \mathsf{S}(x \oplus m_{in}) \oplus m_{out}$$

pour tout x, avec m_{in} le masque d'entrée et m_{out} le masque de sortie. En appliquant T à une variable masquée avec le masque m_{in}, on obtient le résultat de la table S masqué avec le masque m_{out}. Le table T doit être recalculée pour toute nouvelle valeur des masques d'entrée et de sortie. Le calcul de la table T entraîne une perte de performance. De plus, la table T doit être stockée en RAM, rendant cette méthode vulnérable aux attaques ciblant les caches [Tsu+03 ; Ber05 ; BM06].

Table masquée à un ordre supérieur Le masquage des tables proposées dans [Cha+99 ; Mes00a] au premier ordre a été étendu aux ordres supérieurs dans [Cor14]. Leur solution consiste à décaler progressivement la table T en rafraîchissant le masquage entre chaque décalage. [CRZ18] ont amélioré [Cor14] en augmentant progressivement le nombre de parts de sortie dans la table T et en adaptant la méthode dite des parts communes introduite dans [Cor+16]. La méthode des tables masquées engendre cependant un surcoût important en termes de temps d'exécution, de mémoire et de complexité en aléas.

4.2.4 Implémentation «bit slicing» masquée

Pour masquer une application BS ou *m-slicing*, une solution est de répartir les parts dans différents registres du processeur et remplacer les instructions par des implémentations masquées en utilisant les méthodes de masquage présentées au chapitre 3. Dans la suite, nous nommons cette solution SW-M-BS. Une telle approche a été appliquée dans [Gro+15] pour concevoir des chiffrements par blocs en utilisant le schéma ISW. Une implémentation BS masquée de AES a également été proposée dans [Bal+15] et utilisée comme étude de cas pour des attaques SCA sur un processeur ARM Cortex-A8 fonctionnant à 1 GHz. Plus récemment, [GR17] ont décrit des implémentations BS masquées de chiffrements par blocs classiques tels que AES ou PRESENT. Les parts d'un bit étant réparties sur de nombreux mots mémoires, la pression sur les registres généraux du processeur augmente avec l'ordre de masquage [JS17].

Tornado Tornado [Bel+20] synthétise une implémentation BS masquée à partir d'une description Usuba d'une primitive cryptographique en utilisant le schéma ISW. Une macro permet d'instancier à la compilation l'ordre de masquage souhaité. Tornado utilise Usuba pour générer automatiquement un code BS qui est ensuite analysé par une extension de tightPROVE [BGR18] pour rafraîchir le masquage lorsque c'est nécessaire.

4.2.5 Implémentation «share slicing»

Une autre technique pour masquer une implémentation BS ou *m-slicing*, appelée «*share slicing*», consiste à placer toutes les parts d'un bit masqué dans un même mot mémoire. Par exemple, la figure 4.7 illustre la représentation «*share slicing*» d'ordre 1 sur une architecture 32 bits. Chaque mot mémoire contient un bit de 16 blocs indépendants, ainsi que le masque de chaque bit. Dans la suite, une implémentation «*share slicing*» de AES est nommé SW-M-SS.

	Bloc 0		$\cdots$	Bloc 15	
	part 0	part 1	$\cdots$	part 0	part 1
$\mathtt{M}_0$	$\hat{\mathbf{b}}_0^0[0]$	$\hat{\mathbf{b}}_0^0[1]$	$\cdots$	$\hat{\mathbf{b}}_0^{15}[0]$	$\hat{\mathbf{b}}_0^{15}[1]$
$\mathtt{M}_1$	$\hat{\mathbf{b}}_1^0[0]$	$\hat{\mathbf{b}}_1^0[1]$	$\cdots$	$\hat{\mathbf{b}}_1^{15}[0]$	$\hat{\mathbf{b}}_1^{15}[1]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathtt{M}_{127}$	$\hat{\mathbf{b}}_{127}^0[0]$	$\hat{\mathbf{b}}_{127}^0[1]$	$\cdots$	$\hat{\mathbf{b}}_{127}^{15}[0]$	$\hat{\mathbf{b}}_{127}^{15}[1]$

FIGURE 4.7 – Représentation «*share slicing*» d'ordre 1 sur une architecture 32 bits. $\hat{\mathbf{b}}_i^j$ désigne un masquage du i-ième bit du j-ième bloc.

Implémentations «*share slicing*» de AES [Bar+17] ont proposé une multiplication masquée réalisable efficacement en représentation «*share slicing*». L'algorithme 9 décrit cette multiplication masquée à l'ordre 1. [Bar+17] ont montré que cette multiplication masquée pouvait être rendue SNI en rafraîchissant le masquage de sortie. Cette multiplication masquée fut ensuite améliorée dans [Gré+18 ; Gou+18 ; Bar+19a]. [Gré+18] ont proposé une implémentation «*share slicing*» de AES masquée aux ordres 3 et 7. Une implémentation «*share slicing*» masquée à l'ordre 31 a également été proposée dans [JS17].

Algorithme 9 Multiplication masquée à l'ordre 1 de [Bar+17].

Entrée: Masquage $\hat{\mathbf{x}} \in \mathrm{GF}(2)^2$ de x et masquage $\hat{\mathbf{y}} \in \mathrm{GF}(2)^2$ de y
Sortie: Masquage $\hat{\mathbf{z}} \in \mathrm{GF}(2)^2$ de z tel que $\hat{\mathbf{z}}[0] \oplus \hat{\mathbf{z}}[1] = x \otimes y$

$a_1 \xleftarrow{\$} \hat{\mathbf{x}} \otimes \hat{\mathbf{y}}$
$r \xleftarrow{\$} \mathbb{K}_2^2$
$b_1 \xleftarrow{\$} a_1 \oplus r$
$a_2 \xleftarrow{\$} \hat{\mathbf{x}} \otimes \mathsf{ROT}(\hat{\mathbf{y}}, 1)$
$b_2 \xleftarrow{\$} b_1 \oplus a_2$
$\hat{\mathbf{z}} \xleftarrow{\$} b_2 \oplus \mathsf{ROT}(r, 1)$

Avantages et inconvenients Les implémentations de [Gré+18 ; JS17] ont montré que la multiplication masquée de [Bar+17] est relativement efficace, tant en termes d'aléas que de cycles d'exécution. De plus, le «*share slicing*» nécessite moins de mots mémoire que le BS masqué, réduisant ainsi le coût lié à la gestion des différents blocs. En effet, le nombre de

blocs chiffrés en parallèle est divisé par $d + 1$. Une propriété intéressante du «*share slicing*» est qu'il évite les fuites liées aux transitions [Gao+19], puisque la position de chaque part dans un mot mémoire est toujours la même. Le «*share slicing*» évite ainsi les recombinaisons des parts lors des opérations *load*/*store*. Par contre, le placement de toutes les parts dans un même mot mémoire peut conduire à des recombinaisons liées à la micro-architecture des processeurs [Gao+19].

4.2.6 Comparaison des différentes solutions

Le tableau 4.2 présente les performances de différentes implémentations logicielles masquées du chiffrement AES. On constate que le masquage induit un coût important en terme de temps d'exécution. Le coût du masquage augmente de façon exponentielle avec l'ordre de masquage. Par exemple, les implémentations SW-M-RP masquées aux ordres 1, 2 et 3 engendrent respectivement un surcoût de 43, 90 et 157. L'implémentation masquée SW-M-HTABLE est beaucoup moins efficace que SW-M-RP. Par exemple, l'implémentation SW-M-HTABLE masquée à l'ordre 2 augmente le temps d'exécution d'un facteur 289 par rapport à l'implémentation non masquée SW-BW. Le «*share slicing*» semble être une solution efficace pour réduire le temps d'exécution et masquer à des ordres élevés. Une comparaison plus précise est réalisée au chapitre 6 en réimplémentant ces différentes implémentations sur un même cœur.

Implém.	Source	d	Cœur	Temps	
				Cycles	cmp
SW-BW	[RP10]	0	8051	3000	base
SW-M-RP		1	8051	129000	×43
SW-M-RP		2	8051	271000	×90
SW-M-RP		3	8051	470000	×157
SW-BW	[CRZ18]	0	Intel	1600	base
SW-M-HTABLE		2	Intel	463000	×289
SW-M-HTABLE		3	Intel	771000	×482
SW-TTABLE	[SS16]	0	Cortex-M4	662	base
SW-M-SS	[JS17]	31	Cortex-M4	2783510	×4207
SW-M-16S	[BS12]	0	NEON	306	base
SW-M-SS	[Gré+18]	3	NEON	8793	×29
SW-M-SS		7	NEON	26601	×87

TABLE 4.2 – Performances de différentes implémentations logicielles masquées du chiffrement AES. Le temps d'exécution est donné en nombre de cycles par bloc.

Chapitre 5

Extensions de jeu d'instructions

Différentes approches existent pour implémenter un cryptosystème. Une *extension de jeu d'instructions*, abrégée en ISE pour «*instruction set extension*» en anglais, ajoute de nouvelles instructions au jeu d'instructions de base pour faire certains calculs de façon plus efficace, augmenter la sécurité, ou réduire l'empreinte mémoire. Pour implémenter le masquage de façon efficace et sécurisée, des ISE dédiées au masquage ont récemment été proposées.

Dans ce chapitre, la section 5.1 introduit le concept d'architecture de jeu d'instructions. La section 5.2 décrit le jeu d'instructions RISC-V. La section 5.3 introduit le concept d'extension de jeu d'instructions. La section 5.4 établit un état de l'art des ISE dédiées au masquage.

5.1 Architecture de jeu d'instructions

Une *architecture de jeu d'instructions*, abrégée en ISA pour «*instruction set architecture*» en anglais, fait partie du modèle abstrait d'un processeur. L'ISA agit comme une interface entre le matériel et le logiciel, spécifiant ce que le processeur est capable de faire (voir figure 5.1). En général, une ISA définit les instructions prises en charge, les types de données, les registres, les modes d'adressage, la mémoire virtuelle et le modèle d'entrée/sortie.

Implémentation d'une ISA Une ISA peut être implémentée par des processeurs avec des micro-architectures très différentes. Toutes les implémentations d'une ISA sont capables d'exécuter n'importe quel exécutable respectant les contraires de l'ISA. Ainsi, un processeur moins performant et moins coûteux peut être remplacé par un processeur plus coûteux et plus performant implémentant la même ISA sans avoir à modifier le logiciel.

Classification des ISA La conception d'ISA est un problème complexe. Les ISA peuvent être classées en deux catégories en fonction de leur complexité architecturale. Les *jeux d'instructions complexes*, abrégés en CISC pour «*complex instruction set computer*» en anglais, comportent de nombreuses instructions spécialisées. Ce type de jeu d'instructions permet d'optimiser l'exécution de certaines opérations, d'améliorer l'utilisation de la mémoire et du cache et de simplifier la programmation. Cependant, le nombre important d'instructions spécialisées augmente la taille du processeur et impacte sa fréquence. De plus, de nombreuses instructions sont rarement utilisées en pratique conduisant à un faible taux d'utilisation des unités de calcul. Les *jeux d'instructions réduits*, abrégés en RISC pour «*reduced instruction set computer*» en anglais, réduisent le nombre d'instructions au minimum en implémentant uniquement les

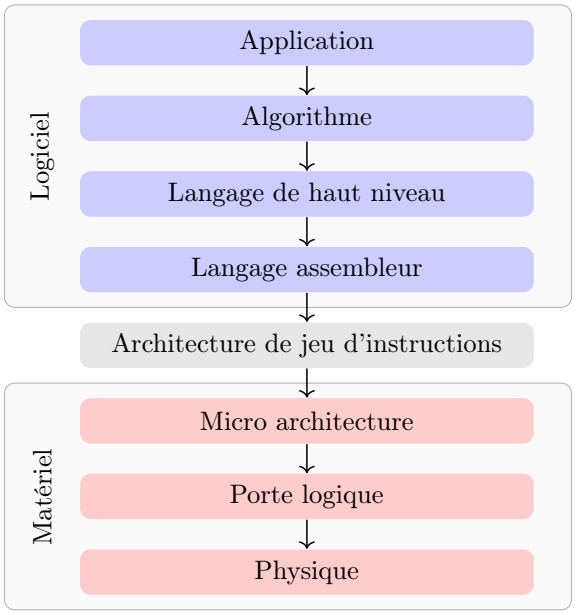


FIGURE 5.1 – Architecture de jeu d'instructions.

instructions les plus fréquemment utilisées dans les programmes. La surface et la consommation énergétique des processeurs RISC est généralement réduite par rapport aux processeurs CISC car le décodage des instructions est plus simple. En contrepartie, le nombre d'instructions exécutées est plus élevé et le code moins compact qu'avec des jeux d'instructions CISC. L'augmentation du nombre d'instructions exécutées peut être compensée par l'utilisation de la technique du *pipeline* qui permet d'augmenter la fréquence d'horloge.

5.2 Jeu d'instructions RISC-V

RISC-V est une ISA open source de type RISC qui peut être implémentée, modifiée ou étendue par toute personne sans exigence de licence ni de redevance (à la différence de MIPS, ARM et x86). RISC-V fut développé initialement en 2010 à l'université Berkeley en Californie pour soutenir la recherche et l'éducation en architecture des ordinateurs. Depuis 2010, l'ISA RISC-V a gagné en popularité dans le milieu académique et industriel. En 2015, la fondation RISC-V est créée pour maintenir la conception de RISC-V. La fondation RISC-V se compose d'universitaires et de grandes entreprises telles que Google, Intel, Nvidia, AMD, Qualcomm, et IBM. Cette importante communauté a permis le développement rapide d'une large gamme d'implémentations de processeurs RISC-V. RISC-V est également pris en charge par des systèmes d'exploitation et de nombreuses chaînes d'outils logiciels. Le jeu d'instructions RISC-V a une conception modulaire qui offre une grande flexibilité. Il se compose d'une ISA de base qui est réduite à un ensemble minimal d'instructions. L'ISA de base peut être complétée par un ensemble d'extensions, standards ou non-standards. Cette approche permet d'ajouter uniquement les extensions nécessaires pour répondre aux contraintes du concepteur. Nous allons à présent décrire plus précisément l'ISA de base RV32I. Une description plus complète du jeu d'instructions RISC-V est disponible dans [WA19].

5.2.1 ISA de base RV32I

L'ISA de base RV32I possède 32 registres qui ont tous une largeur de 32 bits. En tant qu'ISA de type RISC, RISC-V a une architecture *load-store*, c'est-à-dire que les opérandes et la destination d'une opération doivent être des registres. L'ISA de base RV32I implémente 47 instructions permettant d'obtenir les fonctionnalités de base avec des entiers 32 bits. Cette ISA comprend des instructions pour l'addition et la soustraction entières, les opérations binaires, les *load-store*, les sauts et les branchements. Les formats des instructions de RV32I sont de 6 types différents (voir table 5.1). Les instructions de type R lisent deux registres sources d'indices `rs1` et `rs2` dans le fichier de registres, et écrivent le résultat dans le registre destination d'indice `rd` dans le fichier de registres. Les autres types d'instructions utilisent des valeurs immédiates codées dans l'instruction (`imm`). Les types I et S sont utilisés respectivement pour les *load* et les *store*. Les branchements conditionnels ont un format de type B alors que les sauts inconditionnels ont un format de type J. Le type U permet un immédiat de plus grande taille. La position des champs de bits dans les différents types d'instructions a été choisie pour simplifier le décodage. Par exemple, les indices dans le fichier de registres des registres sources et destination sont maintenus à la même position dans tous les formats.

Type	31 30 25	24 21 20	19 15	14 12	11 8 7	6 0
R	`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`
I	`imm[11 : 0]`		`rs1`	`funct3`	`rd`	`opcode`
U	`imm[31 : 12]`				`rd`	`opcode`
S	`imm[11 : 5]`	`rs2`	`rs1`	`funct3`	`imm[4 : 0]`	`opcode`
B	`imm[12]` `imm[10 : 5]`	`rs2`	`rs1`	`funct3`	`imm[4 : 1]` `imm[11]`	`opcode`
J	`imm[20]` `imm[10 : 1]` `imm[11]`		`imm[19 : 12]`		`rd`	`opcode`

TABLE 5.1 – Formats d'instruction de RV32I, reproduit de [WA19].

5.2.2 Extensions standards

L'ISA de base RISC-V peut être étendue avec des extensions de sorte que les processeurs personnalisés soient adaptés aux besoins de l'application. La spécification RISC-V définit un certain nombre d'extensions standards. Parmi ces extensions standards, on peut citer l'extension M (multiplication), A (instructions atomiques), F (virgule flottante simple précision), C (instructions compressées), ou encore l'extension K (cryptographie scalaire). D'autres extensions ne sont pas encore validées par la fondation RISC-V et sont appelées non-standards. Le nom donné à l'ISA obtenue en ajoutant les extensions est le nom de l'ISA de base concaténé avec les lettres de chaque extension ajoutée, comme par exemple RV32IMAFD.

5.2.3 Implementations RISC-V

De nombreuses implémentations de cœurs RISC-V ont été proposées avec diverses caractéristiques et objectifs. On peut notamment citer VexRiscv, Rocket, PicoRV32 ou encore les cœurs développés par l'OpenHW Group [1].

1. Une liste complète des cœurs RISC-V est disponible à `https://riscv.org/risc-v-cores-and-soc-overview/`

OpenHW Group *OpenHW Group*[2] est une organisation mondiale à but non-lucratif dont font partie des acteurs majeurs de l'industrie. L'openHW Group a développé CORE-V, une série de cœurs open source basée sur RISC-V. La famille CORE-V[3] fournit des cœurs ayant des implémentations optimisées en silicium et FPGA et une conception conforme aux pratiques de l'industrie. La famille CORE-V se compose du CVA6[4], un cœur adapté aux applications hautes performances, du CV32E40P[5] pour les plates-formes embarquées, et du CVE2[6] pour les systèmes embarqués à fortes contraintes énergétiques. Un projet de vérification fonctionnelle a été développé pour vérifier le fonctionnement de la famille de cœurs CORE-V[7].

CV32E40P Nous avons choisi d'implémenter nos extensions sur le cœur CV32E40P. Le cœur CV32E40P est un processeur RISC-V 32 bits à 4 étages. Il implémente le jeu d'instructions RV32IMFC, ou l'extension F est optionnelle. L'ISA du CV32E40P a été étendue pour prendre en charge plusieurs instructions supplémentaires, notamment les boucles matérielles, les instructions de *load* et de *store* avec post-incrémentation, les instructions de manipulation de bits et les instructions SIMD qui ne font pas partie de l'ISA RISC-V standard.

5.3 Extension de jeu d'instructions

Une *extension de jeu d'instructions* (ISE) étend une ISA avec de nouvelles instructions ou d'autres fonctionnalités. Une implémentation de l'ISA étendue pourra toujours exécuter du code machine pour les versions de l'ISA sans ces extensions. Par contre, le code machine utilisant des extensions ne fonctionnera que sur les implémentations qui prennent en charge ces extensions. Les ISE permettent d'améliorer différents paramètres tels que le nombre de cycles, l'utilisation de la mémoire (mémoire de données et mémoire de programme), la flexibilité ou encore la consommation d'énergie. Par contre, l'implémentation des nouvelles instructions augmente la surface du processeur et peut impacter sa fréquence de fonctionnement. Le jeu d'instructions x86 fournit de nombreux exemples d'ISE développées notamment par Intel et AMD.

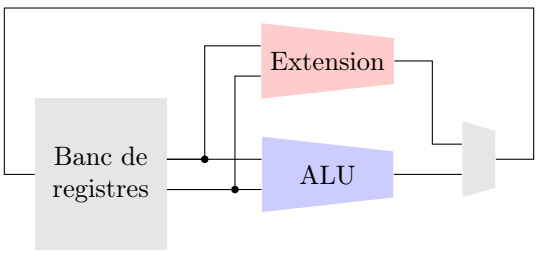


FIGURE 5.2 – Extension de jeu d'instructions.

2. https://www.openhwgroup.org/
3. Une liste complète des cœurs CORE-V est disponible à https://github.com/openhwgroup/core-v-cores
4. https://github.com/openhwgroup/cva6
5. https://github.com/openhwgroup/cv32e40p
6. https://github.com/openhwgroup/cve2
7. https://github.com/openhwgroup/core-v-cores

Comparaison avec un coprocesseur Une autre solution pour améliorer les performances d'une implémentation spécifique est d'ajouter un *coprocesseur* au processeur. Un modèle typique d'utilisation d'un coprocesseur se compose de quelques étapes simples : configurer le coprocesseur, charger les données d'entrée, lancer le coprocesseur et récupérer les résultats. Le transfert des données entre le processeur et le coprocesseur engendre un coût important en termes de nombre de cycles. Une ISE est un compromis entre une implantation logicielle et un coprocesseur. Elle fournit un couplage plus étroit du matériel personnalisé au *pipeline* du processeur. Les nouvelles instructions permettent de réduire le temps d'exécution par rapport à une implantation logicielle, et de réduire la surface par rapport à un coprocesseur.

5.4 ISE pour la contre-mesure de masquage

5.4.1 Pourquoi une ISE dédiée au masquage ?

Une ISE est une solution pertinente pour améliorer les performances du masquage et augmenter sa sécurité, tout en maintenant une utilisation simple et flexible.

Performance Les implémentations de masquage uniquement logiciel imposent un surcoût important en termes de temps d'exécution, de taille de code exécutable et de valeurs aléatoires de haute qualité. Par exemple, le schéma de masquage de [RP10] dont les performances sont décrites dans le tableau 5.2 induit un surcoût relativement important par rapport à la solution non masquée de [Ber+02]. Le nombre d'instructions est multiplié par 30, le nombre de cycles d'horloge est multiplié par 26, la taille du code est multipliée par 6, et l'usage de la mémoire est presque multiplié par 3. Le masquage est donc très coûteux en ressource, et cela, d'autant plus que l'ordre de masquage est élevé. Une ISE peut aider à réduire le surcoût du masquage en remplaçant un ensemble d'instructions par des instructions dédiées. Pour illustrer les gains apportés par les ISE, le tableau 5.2 présente les résultats d'implémentation de AES avec l'ISE non masquée de [Mar+20] et l'ISE masquée de [Gao+21]. L'ISE non masquée de [Mar+20] accélère le chiffrement AES d'un facteur 8, et réduit la taille du code par 3. L'ISE masquée de [Gao+21] accélère le chiffrement AES d'un facteur 60 par rapport à la version masquée sans ISE de [RP10].

Référence	Masqué	ISE	Temps		Taille de code (octet)	
			Instr.	Cycles	Instr.	Données
[Ber+02]	✗	✗	1932	2427	2148	524
[RP10]	✓	✗	59823	64200	14416	1356
[Mar+20]	✗	✓	238	291	730	10
[Gao+21]	✓	✓	1012	1113	968	84

TABLE 5.2 – Coût/Performance du masquage d'ordre 1 de AES extrait de [Gao+21].

Fuite micro-architecturale Les protections purement logicielles sont très limitées en raison de fuites dans la micro-architecture du processeur. Par exemple, [Gao+19] montre que les interactions de bits dans les instructions ARM des cœurs M0/M3 sont dévastatrices pour le masquage d'ordre supérieur. Cela oblige tout logiciel qui tente d'être sécurisé à être extrêmement

conservateur dans ses hypothèses, dégradant les performances en termes de temps d'exécution et de taille de code. Une ISE permet d'atténuer les fuites micro-architecturales en adoptant une approche similaire à l'ISA augmentée (ou aISA) de [GYH18] et en contraignant la micro-architecture pour répondre à des propriétés de sécurité plus fortes.

Flexibilité et simplicité d'utilisation La conception de schémas de masquage est une tâche relativement difficile. Une ISE dédiée au masquage peut fournir un moyen simple de masquer une implémentation. De plus, une ISE dédiée au masquage est une solution efficace pour augmenter la sécurité et les performances en gardant de la flexibilité.

5.4.2 ISE dédiées au masquage de l'état de l'art

Les ISE cryptographiques dédiées au masquage sont assez récentes. Les instructions masquées ajoutées aident à réduire le surcoût du masquage en remplaçant certaines séquences d'instructions (éventuellement longues) par un très petit nombre de séquences dédiées, et augmentent la sécurité en atténuant les fuites micro-architecturales.

Une première ISE pour le masquage d'ordre 1 a été proposée par [TG07]. Le processeur est divisé en une zone non sécurisée et une zone sécurisée. Les données critiques dans la zone non sécurisée sont toujours protégées par un masque booléen. Les opérations les plus critiques en matière de sécurité sont implémentées dans la zone sécurisée en utilisant un style logique résistant à la DPA. La structure de la zone sécurisée est illustrée à la figure 5.3. Les opérandes $op1_m$ et $op2_m$ sont d'abord démasquées et traitées par l'unité fonctionnelle FU, donnant le résultat res. Un générateur de nombres aléatoires produit un nouveau masque et l'applique au résultat avant qu'il ne quitte la zone sécurisée. Cette extension masquée conduit à un surcoût d'un facteur 3,5 en surface, 1,2 pour le chemin critique et 3,5 pour la consommation électrique. Par la suite, des ISE similaires à [TG07] ont été proposées telles que [Nak+11 ; BBT10 ; TKS10 ; TKS11].

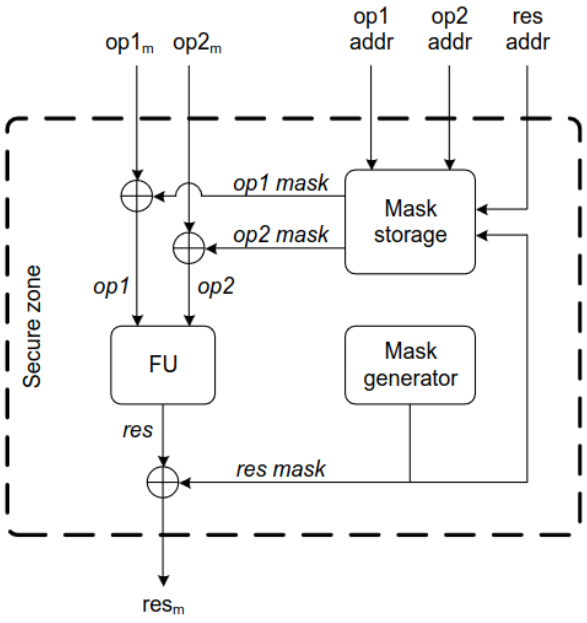


FIGURE 5.3 – Structure de la zone sécurisée de l'ISE de [TG07].

[Gro+16] présente une ISE dédiée au masquage, que nous nommons GJM, pour le cœur RISC-V V-scale et des ordres de masquage $d \in \{1, 2, 3, 4\}$ fixés au moment de la synthèse. La figure 5.4 est un schéma simplifié du cœur avec l'ISE dédiée au masquage de [Gro+16], dans lequel les flèches rouges représentent des variables masquées et `reg[x]` désigne la valeur du registre d'indice `x` dans le fichier de registres. Le fichier de registres est abrégé en RF pour «*register file*» en anglais. Le fichier de registres est dupliqué pour stocker les $d + 1$ parts d'une variable masquée. Cette solution est donc limitée à de petits ordres de masquage puisque la taille du fichier de registres est multipliée par $d+1$. Le chemin de données du processeur est séparé en une partie protégée et une partie non protégée. La partie non protégée applique les opérations de l'ALU de base sur les opérandes préalablement démasquées. La partie protégée contient une ALU masquée qui implémente les instructions masquées en utilisant la représentation DOM [GMK16]. Les instructions étant masquées avec DOM, les utilisateurs doivent rafraîchir les masquages à certains endroits bien choisis pour maintenir la sécurité.

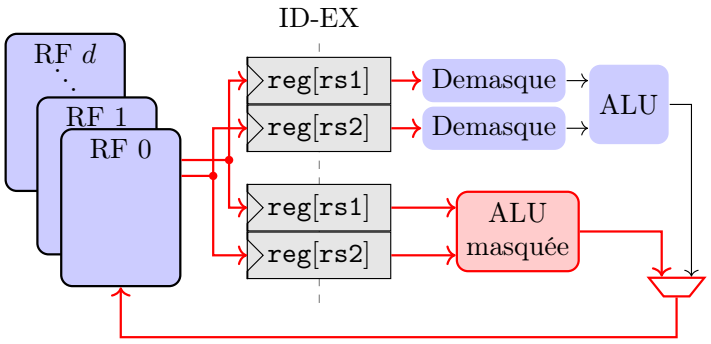


FIGURE 5.4 – Schéma de l'ISE de [Gro+16]. Les flèches rouges représentent des variables masquées. `reg[x]` désigne la valeur du registre d'indice `x` dans le fichier de registres.

[DGH19] présente une ISE dédiée au masquage d'ordre 1 pour un cœur RISC-V avec une protection SCA lors des accès mémoires. Un processeur RISC-V est complètement masqué avec le schéma de masquage TI [NRR06]. Pour protéger les accès mémoires, un masque supplémentaire est généré à partir d'un aléa et de l'adresse mémoire pour remasquer une variable avant de la stocker en mémoire. Le masque étant généré à la volée, il n'est pas nécessaire de le stocker en mémoire. Cela supprime le surcoût de mémoire associé au masquage et protège la variable masquée en mémoire. Il n'existe pas d'évaluation de la surface, mais on peut penser que le coût de cette ISE est important car les implémentations TI nécessitent un nombre de parts élevé (voir section 3.4).

[Kia+21] présente SKIVA, une ISE pour le cœur LEON3 (jeu d'instructions SPARC V8). Les instructions de Skiva accélèrent les implémentations BS masquées aux ordres 1 ou 3. Les nouvelles instructions permettent d'accélérer l'algorithme de multiplication masquée défini dans [Bar+17]. Il combine également des protections de masquage et de détection de fautes.

[Gao+21] présente une ISE dédiée au masquage d'ordre 1 pour le cœur SCARV RISC-V, que nous nommons GGM. Cette ISE utilise la représentation DOM [GMK16]. La figure 5.5 est un schéma simplifié du cœur avec l'ISE dédiée au masquage de [Gao+21]. Le fichier de registres est divisé en 2 parties (une partie pour chaque part). Cette solution est limitée aux petites ordres de masquage car une ISE dédiée au masquage d'ordre d diviserait le fichier de registres en $d + 1$ parties.

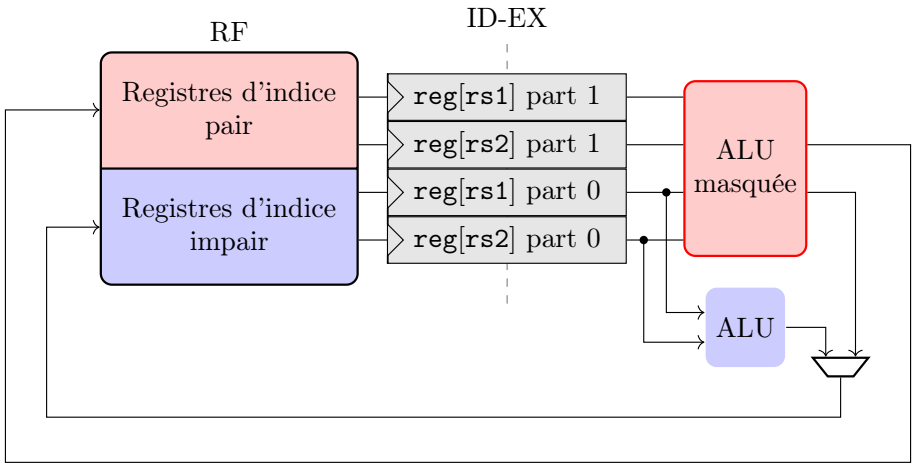


FIGURE 5.5 – Schéma de l'ISE de [Gao+21]. `reg[x]` désigne la valeur du registre d'indice `x` dans le fichier de registres.

[MP21] présente SME, une ISE dédiée au masquage pour le cœur RISC-V SCARV et pour des ordres de masquage $d \in \{1, 2, 3\}$ fixés au moment de la synthèse. Leur approche repose sur des similitudes entre les implémentations de schémas de masquage d'ordre supérieur et la programmation vectorielle. La figure 5.6 est un schéma simplifié du cœur avec l'ISE dédiée au masquage de [MP21]. La première part d'une variable masquée est stockée dans le fichier de registres. Les autres parts sont stockées dans des fichiers de registres dédiés, placées dans l'étage d'exécution du *pipeline*. Une ALU masquée implémente les instructions masquées en utilisant la représentation DOM [GMK16]. L'ISE intègre une version masquée de l'extension cryptographique scalaire officielle RISC-V pour AES [Mar+20].

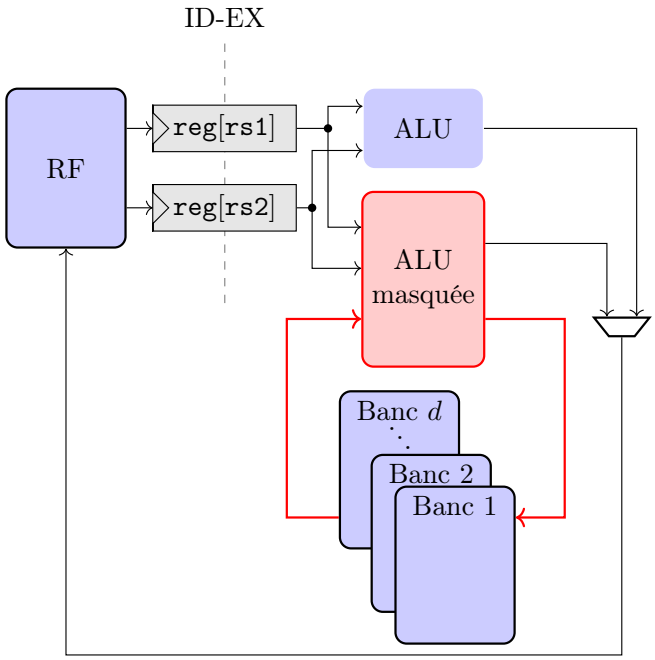


FIGURE 5.6 – Schéma de l'ISE de [MP21]. `reg[x]` désigne la valeur du registre d'indice `x` dans le fichier de registres.

5.4.3 Comparaison des ISE dédiées au masquage de l'état de l'art

Les tableaux 5.3 et 5.4 présentent les performances et surfaces de différentes ISE dédiées au masquage de l'état de l'art. HW-M-SKIVA, HW-M-GGM et HW-M-SME désignent respectivement des implémentations de AES masquées en utilisant les ISE SKIVA, GGM et SME. On constate que HW-M-SME et HW-M-GGM réduisent fortement le coût du masquage. Même à l'ordre 3, HW-M-SME induit seulement un surcoût de 1,5 par rapport à SW-TTABLE, ce qui est cohérent car HW-M-SME est optimisée pour le chiffrement AES. En revanche, HW-M-SKIVA engendre un surcoût de 4,0 à l'ordre 1 et de 13,2 à l'ordre 3 par rapport à SW-BS. HW-M-SKIVA est moins efficace que HW-M-SME, mais est applicable à toutes sortes de cryptosystèmes contrairement à HW-M-SME. En termes de surface, GGM est la plus petite extension à l'ordre 1 avec une augmentation de 1.25 en FFs et 1.82 en LUTs. GJM et SME induisent des coûts plus important en surface, puisque le fichier de registres est dupliqué.

Source	Implém.	Cœur	d	Temps	
				Cycles	cmp
[Kia+21]	SW-BS	Leon3	0	704	base
	HW-M-SKIVA		1	2816	×4,0
			3	9264	×13,2
[Gao+21]	SW-TTABLE	SCARV	0	1016	base
	HW-M-GGM		1	1113	×1,1
[MP21]	SW-TTABLE	SCARV	0	1016	base
	HW-M-SME		1	1142	×1,1
			2	1333	×1,3
			3	1524	×1,5

TABLE 5.3 – Performance de différentes ISE dédiées au masquage de l'état de l'art. Le temps d'exécution est donné en nombre cycles par bloc.

Source	ISE	Cœur	d	Surface & période		
				FF	LUT	P
[Gro+16]	Sans ISE	V-scale	0	base	base	base
	GJM		1	×1,9	×1,6	×0,8
			2	×2,6	×2,2	×0,8
			3	×3,6	×2,8	×0,8
			4	×4,6	×3,6	×1,1
[Gao+21]	Sans ISE	SCARV	0	base	base	base
	GGM		1	×1,3	×1,8	×1,1
[MP21]	Sans ISE	SCARV	0	base	base	base
	SME		1	×1,5	×1,6	×1,6
			2	×1,9	×1,9	×1,7
			3	×2,6	×2,2	×1,7

TABLE 5.4 – Surface et période de différentes ISE dédiées au masquage de l'état de l'art. FF, LUT et P sont données en surcoût.

Deuxième partie

Contributions

Chapitre 6

Environnement expérimental et reproduction de résultats de solutions de l'état de l'art

Une chaîne d'outils RISC-V a été mise en place pour compiler et simuler des codes sur le cœur CV32E40P. Des solutions de l'état de l'art ont ensuite été réimplémentées sur le cœur CV32E40P pour étudier leurs performances et les comparer à nos résultats d'implémentation dans la suite. Pour implémenter nos ISE, les nouvelles instructions sont ajoutées à la chaîne d'outils et implantées dans le cœur. Un générateur de traces simulées de consommation a été développé pour analyser la sécurité de nos implémentations contre certaines attaques SCA.

Dans ce chapitre, la section 6.1 donne un aperçu de la chaîne d'outils utilisée pour compiler et simuler un code sur une architecture RISC-V. La section 6.2 décrit les modifications nécessaires pour ajouter une ISE à un cœur RISC-V. La section 6.3 présente les résultats de la réimplantation de solutions de l'état de l'art sur le cœur CV32E40P. La section 6.4 décrit l'évaluation pratique de la résistance contre certaines attaques SCA.

6.1 Environnement expérimental

Pour réaliser nos ISE, un *environnement expérimental* a été mis en place (voir figure 6.1) en utilisant des outils libres de la communauté RISC-V pour compiler un code source, simuler son exécution sur une architecture RISC-V et synthétiser le cœur CV32E40P.

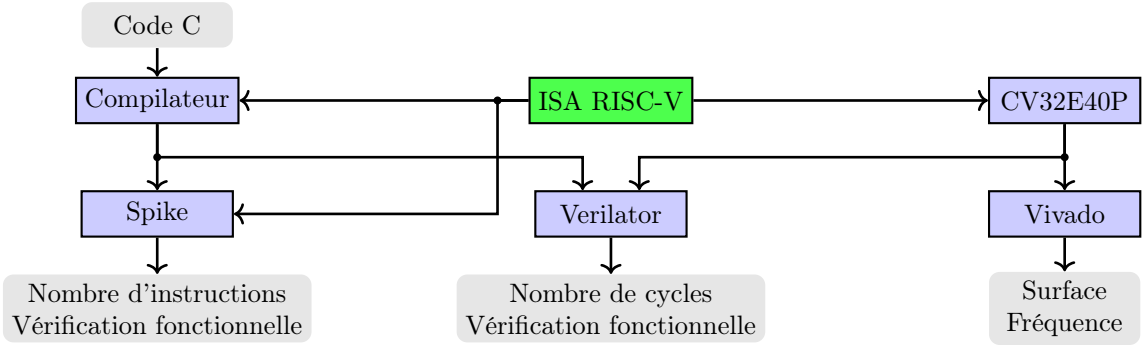


FIGURE 6.1 – Notre environnement expérimental.

6.1.1 Génération d'un exécutable

La chaîne d'outils GNU [1] pour RISC-V est utilisée pour générer des exécutables RISC-V. Elle dispose notamment d'un *compilateur croisé* GCC capable de créer un code assembleur pour une architecture RISC-V, ainsi qu'une collection d'outils binaires tels que l'*assembleur* et l'*éditeur de liens*. Un programme peut être compilé pour s'exécuter avec le système d'exploitation Linux. Les programmes exécutés sous Linux bénéficient d'une prise en charge complète du système. Cependant, la durée d'exécution est longue en raison de la nécessité de démarrer Linux dans l'environnement de simulation. Alternativement, un programme peut être compilé pour s'exécuter avec un «*proxy kernel*», abrégé en PK, sans prise en charge complète du système d'exploitation. Le PK RISC-V [2] fournit un environnement de démarrage Linux et transmet simplement les *appels systèmes* à l'hôte via l'*interface hôte-cible*, abrégée en HTIF pour «*host target interface*» en anglais. Le PK prend en charge uniquement les appels systèmes les plus couramment utilisés tels que `printf`. Une autre solution est de compiler un programme pour qu'il s'exécute en «*bare metal*». Les programmes exécutés en «*bare metal*» ne prennent pas en charge de bibliothèques C. Les appels systèmes doivent être gérés directement par le programme.

6.1.2 Simulation d'un code

Deux méthodologies ont été mises en place pour simuler l'exécution d'un programme sur une architecture RISC-V. La première consiste à exécuter un programme sur Spike, un simulateur du jeu d'instructions RISC-V. La seconde réalise une simulation précise au niveau du cycle du cœur CV32E40P en utilisant Verilator.

Simulation du jeu d'instructions RISC-V avec Spike Spike [3] est un *simulateur de jeu d'instructions*, abrégé en ISS pour «*instruction set simulator*» en anglais, permettant de simuler l'ISA RISC-V. En utilisant Spike, les binaires compilés pour une architecture RISC-V peuvent être exécutés sans processeur RISC-V. Les instructions RISC-V sont décodées et exécutées en utilisant le modèle fonctionnel d'un processeur RISC-V. L'utilisation d'un ISS présente l'avantage de s'abstraire de l'implémentation matérielle d'une ISA. En contrepartie, la simulation n'est pas précise au niveau du cycle (par exemple, le *pipeline* d'un processeur n'est pas simulé).

Simulation cycle précis avec Verilator Pour simuler un programme sur le cœur CV32E40P, nous avons utilisé le *simulateur précis au niveau du cycle* Verilator [4]. Un banc de test a été développé dans lequel le cœur CV32E40P est relié à une mémoire. La mémoire est initialisée avec le code à tester. Le simulateur Verilator évalue alors le circuit RTL à chaque cycle d'horloge. Contrairement à la simulation avec un ISS, la simulation avec Verilator prend en compte l'architecture du CV32E40P. Ce type de simulation est cependant moins rapide que la simulation avec un ISS.

1. `https://github.com/riscv-collab/riscv-gnu-toolchain`
2. `https://github.com/riscv-software-src/riscv-pk`
3. `https://github.com/riscv-software-src/riscv-isa-sim`
4. `https://www.veripool.org/verilator/`

6.1.3 Synthèse FPGA

Pour évaluer la surface et la fréquence de fonctionnement du CV32E40P, le cœur est synthétisé sur un FPGA. La synthèse est réalisée en utilisant le logiciel Vivado 2019.2 [5] de AMD-Xilinx. Le cœur CV32E40P est synthétisé sur le FPGA Digilent Arty A7 [6].

6.2 Conception et mise en œuvre d'une ISE

6.2.1 Conception d'une ISE

Pour concevoir une ISE, une exploration de l'espace de conception a été réalisée. Nous avons étudié les implémentations logicielles masquées du chiffrement AES sur le processeur cible afin d'identifier les principaux goulots d'étranglement. De nouvelles instructions ont alors été identifiées pour réduire le nombre d'instructions exécutées et augmenter la sécurité. Une exploration de différents types d'ISE dédiées au masquage a également été réalisée pour analyser les compromis entre performance et surface. La micro-architecture du processeur a été étudiée pour détecter l'origine de fuites qui peuvent conduire à une recombinaison des parts. Cette analyse nous a conduit à identifier de nouvelles instructions masquées. Chaque ISE proposée et les instructions correspondantes ajoutées seront détaillées dans les chapitres suivants. Nous présentons dans cette section uniquement la méthodologie commune aux différentes contributions.

6.2.2 Ajout des nouvelles instructions à l'assembleur

Lorsque de nouvelles instructions sont identifiées, il est nécessaire de les ajouter à la chaîne d'outils afin que des programmes informatiques puissent les utiliser. L'option la plus simple consiste à ajouter les nouvelles instructions dans l'assembleur, car celui-ci offre généralement une structure moins complexe que le compilateur. Les opcodes et d'autres informations sur les nouvelles instructions sont ajoutés à l'assembleur. Les nouvelles instructions peuvent ensuite être utilisées dans des programmes assembleurs ainsi que dans des langages de haut niveau tel que le langage C (en intégrant directement le code assembleur). L'assembleur modifié permet alors de traduire un programme informatique utilisant les nouvelles instructions en un exécutable.

Choix des opcodes Les opcodes utilisables pour de nouvelles instructions sont décrits par la spécification RISC-V [WA19]. Les deux premiers bits de l'opcode ont toujours pour valeur 1. Les opcodes se divisent en 5 catégories en fonction de la valeur de `opcode[6 : 2]` comme le montre le tableau 6.1. Les extensions standards utilisent uniquement des opcodes standards et ne doivent pas entrer en conflit les uns avec les autres dans leurs utilisations de ces opcodes. Les opcodes réservés ne sont actuellement pas définis, mais sont réservés pour les futures extensions standards. Les opcodes personnalisés sont réservés à un usage expérimental. Pour nos ISE, nous avons choisi d'utiliser l'opcode 1101011.

5. `https://www.xilinx.com/products/design-tools/vivado.html`
6. `https://digilent.com/reference/programmable-logic/arty-a7/start`

opcode[6 : 5] \ opcode[4 : 2]	000	001	010	011	100	101	110	111
00								
01								
10								
11								

TABLE 6.1 – Description des différents opcodes de la spécification RISC-V [WA19]. Opcodes standards en bleu, opcodes réservés aux extensions futures en rouge, opcodes personalisés en vert, opcodes réservés à RV128I en gris, opcodes réservés aux instructions de taille supérieure à 32 bits en jaune.

Utilisation logicielle d'une nouvelle instruction Avec les modifications apportées à l'assembleur, les nouvelles instructions peuvent être utilisées dans un langage C à l'aide d'un bloc `asm`. L'utilisation d'une nouvelle instruction `ise.instr` est décrite à la figure 6.2. On peut alors vérifier que l'instruction `ise.instr` est générée en observant le code assembleur de l'exécutable.

```
int main(void) {
    uint32_t a = 1;
    uint32_t b = 2;
    uint32_t c;

    asm volatile (
        "ise.instr  %[rd], %[rs1], %[rs2]\n\t"
        : [rd] "=r" (c)
        : [rs1] "r" (a), [rs2] "r" (b) );
}
```

FIGURE 6.2 – Utilisation logicielle d'une nouvelle instruction.

6.2.3 Ajout des nouvelles instructions à Spike

Pour pouvoir simuler les nouvelles instructions personnalisées ajoutées à l'assembleur, Spike doit être modifié. Les opcodes des nouvelles instructions sont d'abord ajoutés pour que Spike puisse les reconnaître. Ensuite, une description du comportement fonctionnel des nouvelles instructions est ajoutée. La simulation avec Spike permet déja de fournir des résultats en termes de performances et de besoins en mémoire, permettant une première sélection entre différentes solutions.

6.2.4 Ajout des nouvelles instructions au processeur

Les instructions personnalisées doivent ensuite être intégrées dans le processeur CV32E40P. Les modifications à apporter dépendent dans une large mesure de la nature des nouvelles instructions. Les modifications courantes englobent la logique de décodage et l'ALU. La logique de décodage doit reconnaître les nouveaux opcodes des instructions ajoutées et définir les signaux de contrôle en conséquence. Les nouvelles instructions sont implémentées dans l'ALU existante ou une nouvelle ALU. D'autres changements pourraient être l'ajout de registres personnalisés ou la modification du nombre de ports de lecture et d'écriture du fichier de registres. Un

programme qui utilise l'ISE peut être exécuté sur le processeur modifié afin de recueillir des données de performances précises. L'implantation matérielle du processeur modifié sur FPGA permet d'évaluer la surface et la période minimale du processeur.

6.3 Réimplantation de l'état de l'art

Une comparaison directe des solutions de protection par masquage de l'état de l'art n'est pas pertinente car les cœurs et les FPGA utilisés sont différents dans les papiers. Pour réaliser une comparaison cohérente, nous avons réimplantées plusieurs solutions de l'état de l'art, toutes pour AES, dans le cœur CV32E40P.

6.3.1 Évaluation des performances

Pour évaluer les performances des codes implémentés, nous utilisons les *registres de contrôle et d'état*, abrégés en CSR pour «*control and status registers*» en anglais. L'ISA RISC-V possède des CSR fournissant des informations sur l'état et les performances du processeur. Le registre `time` compte le temps écoulé. Le registre `cycle` contient le décompte du nombre de cycles d'horloge exécutés par le processeur. Le registre `instret` compte le nombre d'instructions exécutées par le processeur. Les CSR sont accessibles par des instructions spécifiques. Les valeurs des registres `time`, `cycle` et `instret` sont lues avant et après le code à évaluer, puis une soustraction permet d'obtenir le temps d'exécution ainsi que le nombre de cycles et d'instructions.

6.3.2 Solutions logicielles

Des solutions logicielles sans ISE de AES ont été réimplantées et sont étiquetées SW-. . . . Les performances sur le cœur RISC-V CV32E40P sont décrites dans le tableau 6.2 et la figure 6.3. SW-BW, SW-RW sont des implémentations de AES basées sur le calcul (voir sous-section 4.1.1) reproduites respectivement de [DR02] et de [Ber+02]. SW-TTABLE est une implémentation T-tables de AES [DR02] (voir sous-section 4.1.2). SW-BS et SW-16S sont respectivement des implémentations BS (voir sous-section 4.1.3) et 16-*slicing* (voir sous-section 4.1.4) générées en utilisant Usuba [MD19]. SW-M-RP et SW-M-CPRR sont des implémentations masquées basées respectivement sur le schéma de masquage RP [RP10] (voir sous-section 4.2.1) et CPRR [Cor+14] (voir sous-section 4.2.2). SW-M-HTABLE est une implémentation masquée de AES utilisant la méthode des tables masquées de [CRZ18] (voir sous-section 4.2.3). SW-M-BS est une implémentation BS masquée générée par Tornado [Bel+20] (voir sous-section 4.2.4). SW-M-SS est une implémentation «*share slicing*» (voir sous-section 4.2.5) adaptée de [JS17 ; Gré+18] et étendue pour masquer aux ordres $d \in \{1, 3, 7, 15, 31\}$. Nos résultats de réimplantation de la figure 6.2 sont cohérents avec ceux de l'état de l'art des tableaux 4.1 et 4.2. SW-TTABLE et SW-BS sont les implémentations non masquées les plus performantes. Les implémentations masquées les plus performances sont SW-M-BS et SW-M-SS. L'implémentation masquée SW-M-CPRR est légèrement plus performante que SW-M-RP. Comme constaté au chapitre 4, SW-M-HTABLE engendre des coûts beaucoup plus élevés que toutes les autres solutions masquées.

d	Implémentation	Temps		
		Cycles	cmp1	cmp2
0	SW-TTABLE	1266	base	n.a.
	SW-BW	5521	×4,4	
	SW-RW	4130	×3,3	
	SW-BS	2040	×1,6	
	SW-16S	3770	×2,9	
1	SW-M-BS	26001	×21	base
	SW-M-SS	43377	×34	×1,7
	SW-M-CPRR	168090	×133	×6,5
	SW-M-RP	264186	×209	×10,2
	SW-M-HTABLE	1039485	×821	×40,0
2	SW-M-BS	50830	×40	base
	SW-M-CPRR	298785	×236	×5,9
	SW-M-RP	455078	×360	×9,0
	SW-M-HTABLE	1810352	×1430	×35,6
3	SW-M-BS	98575	×78	base
	SW-M-SS	94351	×75	×1,0
	SW-M-CPRR	487092	×385	×4,9
	SW-M-RP	702266	×555	×7,1
	SW-M-HTABLE	2947581	×2328	×29,9
7	SW-M-BS	300414	×237	base
	SW-M-SS	410365	×324	×1,4
	SW-M-CPRR	1712402	×1353	×5,7
	SW-M-RP	2339654	×1848	×7,8
	SW-M-HTABLE	11120610	×8784	×37,0
15	SW-M-BS	1041285	×822	base
	SW-M-SS	1498669	×1184	×1,4
	SW-M-CPRR	6328095	×4998	×6,1
	SW-M-RP	8349458	×6595	×8,0
	SW-M-HTABLE	44908227	×35473	×43,1
31	SW-M-BS	3716920	×2936	base
	SW-M-SS	5566866	×4397	×1,5
	SW-M-CPRR	24740662	×19542	×6,7
	SW-M-RP	32177701	×25417	×8,7
	SW-M-HTABLE	182251172	×143958	×49,0

TABLE 6.2 – Résultats de réimplantations logicielles de plusieurs solutions de l'état de l'art sur le cœur CV32E40P pour différents ordres de masquage. Le temps d'exécution est donné en nombre de cycles pour chiffrer un bloc avec AES. cmp1 et cmp2 comparent respectivement le temps d'exécution par rapport à SW-TTABLE et SW-M-BS.

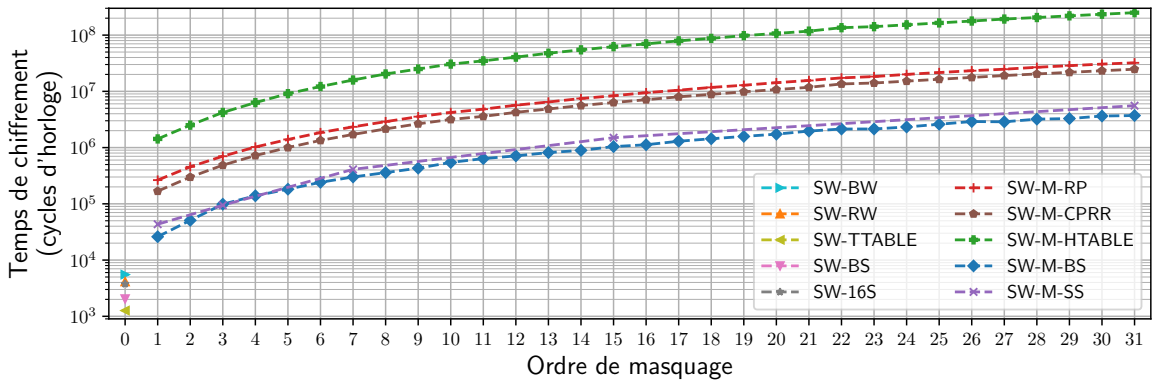


FIGURE 6.3 – Résultats de réimplantations logicielles de plusieurs solutions de l'état de l'art sur le cœur CV32E40P pour différents ordres de masquage. Le temps d'exécution est donné en nombre de cycles pour chiffrer un bloc avec AES.

6.3.3 ISE dédiées au masquage de l'état de l'art

Plusieurs ISE dédiées au masquage de l'état de l'art ont été réimplantées sur le cœur CV32E40P pour réaliser une comparaison plus objective des différentes solutions existantes. Comme nous nous intéressons aux ISE dédiées au masquage à des ordres élevés, nous avons réimplémenté uniquement SKIVA et SME. GJM n'a pas été réimplémentée car il n'y a pas d'implémentation logicielle disponible pour cette ISE. Les tableaux 6.3 et 6.4 présentent nos résultats d'implantations. Nos résultats d'implantations de SKIVA et SME sont cohérents avec ceux de l'état de l'art présentés dans les tableaux 5.3 et 5.4. Comme indiqué dans 5.4.3, HW-M-SME est très performant puisqu'il est optimisé pour le chiffrement AES, mais engendre une augmentation importante de la surface. À l'inverse, les gains de performance avec HW-M-SKIVA sont moins importants, mais le coût matériel est extrêmement faible.

d	Implémentation	Temps		
		Cycles	cmp1	cmp2
0	SW-TTABLE	1266	base	n.a.
1	SW-M-BS	26001	×21	base
	HW-M-SKIVA	14355	×11,3	/1,8
	HW-M-SME	1152	×0,9	/22,6
2	SW-M-BS	50830	×40	base
	HW-M-SME	1271	×1,0	/40.0
3	SW-M-BS	98575	×78	base
	HW-M-SKIVA	47062	×37,2	/2,1
	HW-M-SME	1417	×1,1	/69,6

TABLE 6.3 – Performance de différentes ISE dédiées au masquage de l'état de l'art réimplantées sur le cœur CV32E40P. Le temps d'exécution est donné en nombre de cycles pour chiffrer un bloc avec AES. cmp1 et cmp2 comparent respectivement le temps d'exécution par rapport à SW-TTABLE et SW-M-BS.

d	ISE	Surface & période				
		FF	cmp	LUT	cmp	P(ns)
0	Sans ISE	2141	base	4782	base	14,6
1	SKIVA	2167	×1,0	5040	×1,1	14,5
	SME	2974	×1,4	7015	×1,5	14,5
2	SME	3839	×1,8	8583	×1,8	16,5
3	SKIVA	2167	×1,0	5040	×1,1	14,5
	SME	5077	×2,4	10567	×2,2	16,2

TABLE 6.4 – Surface et période de différentes ISE dédiées au masquage de l'état de l'art réimplantées sur le cœur CV32E40P.

6.4 Analyse de la résistance contre les attaques SCA

6.4.1 Génération de traces simulées de consommation

Pour évaluer la résistance contre les attaques SCA, nous avons développé un *générateur de traces simulées de consommation*. Les traces simulées peuvent être réalisées en utilisant l'ISS Spike, ou la simulation cycle précis avec Verilator.

Génération de traces simulées avec Spike Pour générer des traces simulées de consommation avec Spike, nous utilisons le mode de débogage interactif qui permet de voir le contenu d'un registre ou d'un emplacement mémoire et d'exécuter des instructions une par une. L'état interne du banc de registres est récupéré à chaque instruction. La suite des états récupérés permet de calculer une trace d'activité théorique du processeur en utilisant la distance de Hamming comme modèle de consommation. Une fonction indique dans le code C à quel moment commencer et terminer l'acquisition de la trace. La figure 6.4 montre une trace simulée avec Spike pendant le chiffrement AES, dans laquelle on peut distinguer les différentes rondes. La trace simulée obtenue avec Spike ne correspond pas à l'activité réelle du processeur, mais permet une première estimation assez rapide des fuites.

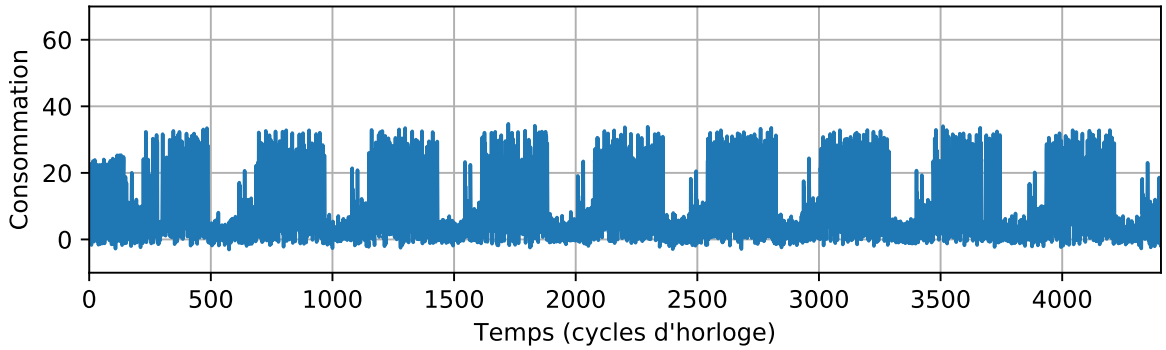


FIGURE 6.4 – Trace de consommation simulée avec Spike pendant le chiffrement AES.

Génération de traces simulées avec Verilator Pour générer des traces simulées avec Verilator, l'état interne du processeur est d'abord récupéré à chaque instruction en utilisant la «*netlist*» du CV32E40P. La consommation est ensuite estimée en calculant la distance de Hamming pour obtenir le nombre de transitions. Une trace d'activité théorique du processeur est alors obtenue. La figure 6.5 montre par exemple une trace simulée avec Verilator pendant le chiffrement AES. La trace simulée obtenue avec Verilator est plus précise qu'avec Spike car la micro-architecture du processeur est prise en compte, mais cette simulation est beaucoup plus lente.

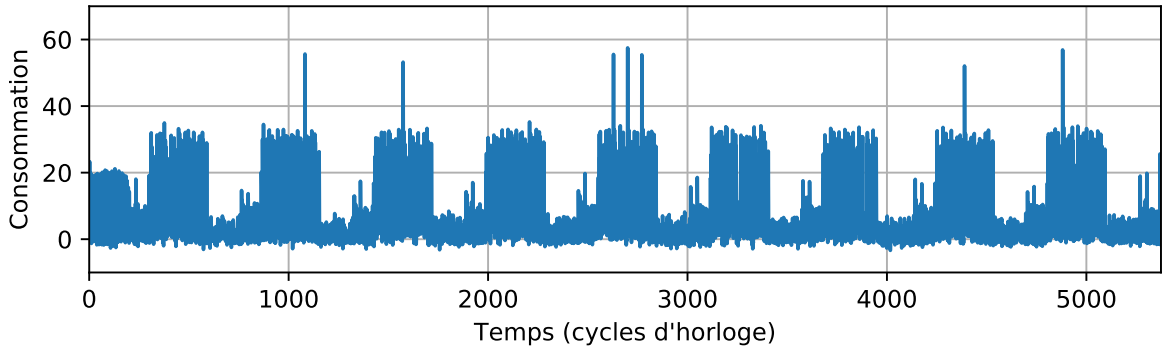


FIGURE 6.5 – Trace de consommation simulée avec Verilator pendant le chiffrement AES.

6.4.2 Analyse des traces simulées

Pour analyser la résistance de nos implémentions contre l'attaque SCA, nous avons réalisé des évaluations pratiques en utilisant des traces simulées.

Évaluation de la fuite Nous réalisons dans un premier temps une évaluation de la fuite d'information en utilisant la bibliothèque python SCALib [7]. Une évaluation pratique de la fuite peut être réalisée par un *t-test fixe contre aléatoire*, communément appelé TVLA [Coo+13 ; SM15]. Il s'agit d'une technique d'évaluation des fuites bien connue qui consiste à détecter si les fuites associées à deux groupes peuvent être distinguées. Un groupe contient les traces avec un message clair fixe et l'autre avec des messages clairs choisis au hasard, tandis qu'une clé constante et identique est utilisée dans les deux groupes. La figure 6.6 de gauche montre la valeur du t-test sur la première ronde de AES. On constate que des fuites sont détectées car le t-test dépasse une valeur seuil. Une autre solution pour détecter des fuites d'information est de calculer le *rapport signal sur bruit*, abrégé en SNR pour «*signal-to-noise ratio*» en anglais, entre les traces simulées et les valeurs intermédiaires. De manière informelle, le SNR permet de quantifier la quantité d'informations sur une variable aléatoire contenue dans la moyenne de la fuite. Un SNR élevé signifie que de l'information est détectée dans la moyenne de la fuite. La figure 6.6 de droite montre la valeur du SNR associée à chaque octet de la sortie de SubBytes de la première ronde de AES. Chaque pic de SNR indique qu'un octet a été détecté. L'évaluation de la fuite est une première étape pour analyser la sécurité d'une implémentation, mais n'est généralement pas suffisante pour affirmer sa résistance contre l'attaque SCA.

7. `https://scalib.readthedocs.io/en/stable/`

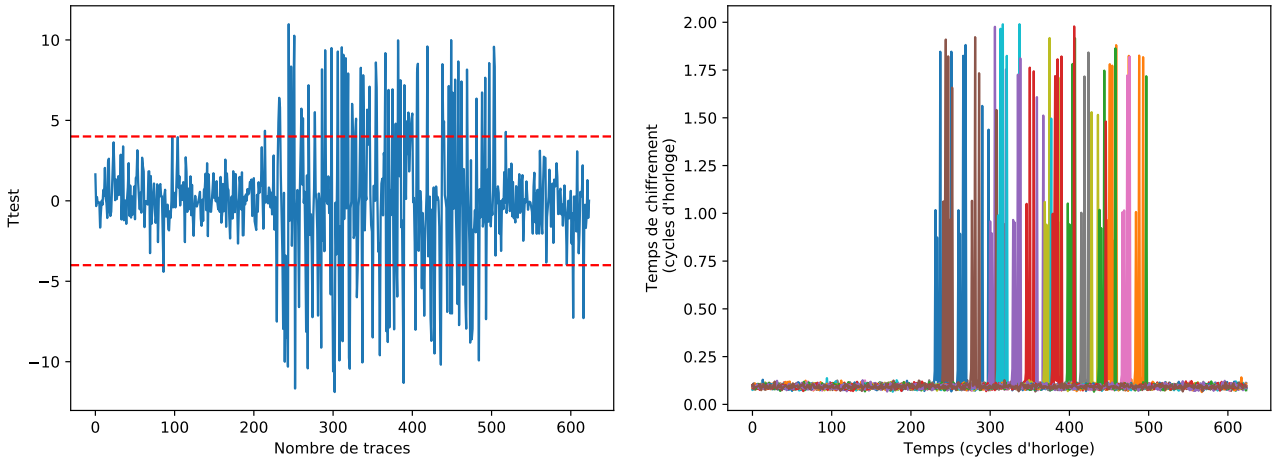


FIGURE 6.6 – Évaluation de la fuite durant la première ronde de l'implémentation SW-BW de AES. La figure de gauche est le t-test, celle de droite le SNR.

Réalisation d'attaques SCA Pour compléter l'évaluation de la sécurité d'une implémentation, nous réalisons des attaques SCA réputées de la littérature. Une première évaluation est effectuée en réalisant des attaques DPA, CPA et MIA avec la librairie python scared [8]. Par exemple, la figure 6.7 montre les résultats de la CPA ciblant le premier octet de la sortie de SubBytes de la première ronde de AES. La figure de gauche est le score du coefficient de Pearson pour chaque hypothèse sur le premier octet de la clé, et la figure de droite est la convergence du coefficient de Pearson en fonction du nombre de traces. On constate que la courbe correspondant à la valeur de clé correcte se détache dès 150 traces simulées.

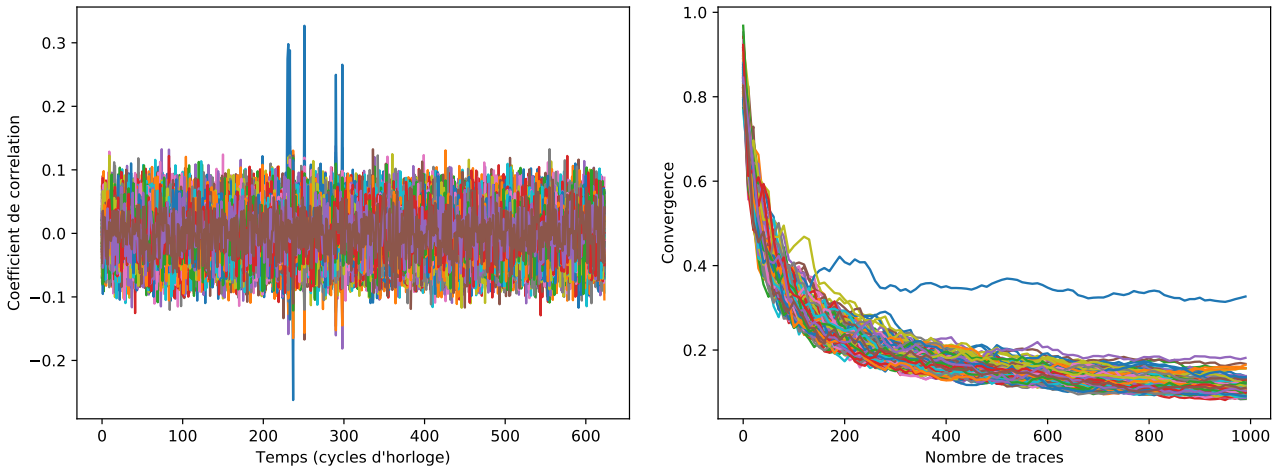


FIGURE 6.7 – Résultats de l'attaque CPA ciblant le premier octet de la sortie de SubBytes de la première ronde de l'implémentation SW-BW de AES. La figure de gauche est le score du coefficient de Pearson, celle de droite la convergence du coefficient de Pearson.

8. `https://eshard.gitlab.io/scared/index.html`

Nous réalisons également des *attaques par modèle* [Guo+20] en utilisant à nouveau la librairie python SCALib. Un ensemble de traces simulées est généré pour réaliser un modèle de la consommation. Ensuite, l'attaque utilise ce modèle pour déterminer la clé secrète. La figure 6.8 montre les résultats de l'attaque par modèle appliquée à la sortie de SubBytes de la première ronde de l'implémentation AES SW-BW. La figure de gauche est le rang de la clé correcte, et la figure de droite est le taux de succès. L'attaque par modèle détermine la clé en seulement quelques traces.

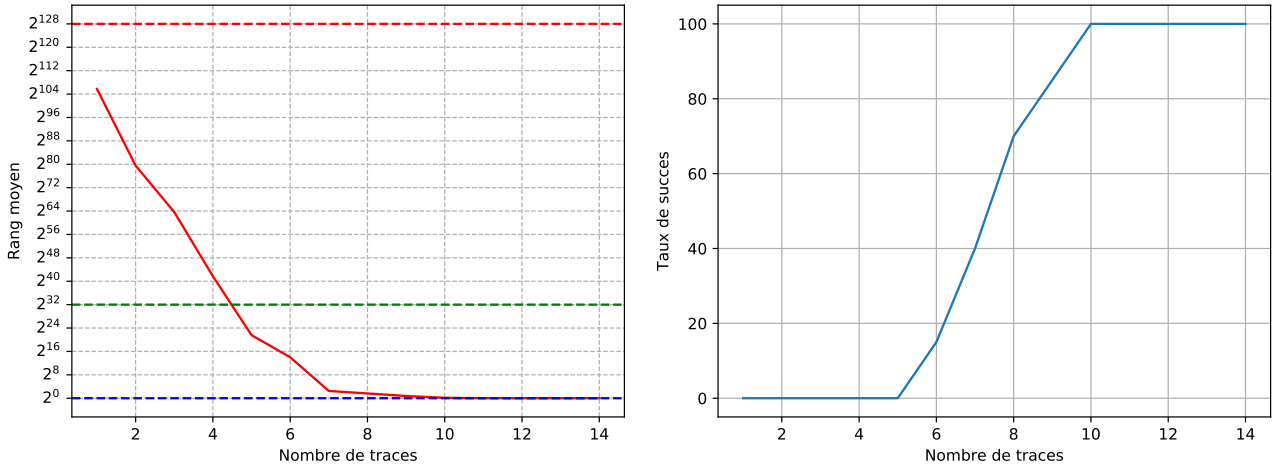


FIGURE 6.8 – Résultats de l'attaque par modèle ciblant la sortie de SubBytes de la première ronde de l'implémentation SW-BW de AES. La figure de gauche est le rang de la clé correcte, celle de droite le taux de succès.

Chapitre 7

Extension masquée pour les implémentations «bit slicing»

Le masquage d'ordre élevé est nécessaire pour sécuriser les implémentations exécutées sur des processeurs à faible niveau de bruit. [BS21] ont par exemple réalisé une attaque sur des implémentations purement logicielles (sans ISE) de AES masquées jusqu'à l'ordre 5. Les implémentations «*share slicing*» (voir sous-section 4.2.5) ont été utilisées pour masquer efficacement à des ordres élevés, allant de l'ordre 3 et 7 dans [Gré+18] jusqu'à l'ordre 31 dans [JS17]. Cependant, [Gao+19] ont montré que les implémentations purement logicielles du «*share slicing*» pouvaient conduire à une recombinaison des parts liée à la micro-architecture des processeurs. Pour limiter les fuites liées à la micro-architecture des processeurs, un support matériel dédié peut être mis en place, comme par exemple une ISE dédiée au masquage. Plusieurs ISE dédiées au masquage ont été proposées dans la littérature (voir sous-section 5.4.2), mais elles ont été implémentées seulement pour de petits ordres de masquage ($d \leq 4$). Dans ce chapitre, nous proposons une ISE dédiée au masquage, que nous nommons ISE1, permettant d'implémenter de façon sécurisée et efficace des implémentations «*share slicing*» de nombreux cryptosystèmes. L'ISE1 supporte des ordres de masquage élevés, allant jusqu'à l'ordre 31. Une implémentation masquée à l'ordre d utilisant les instructions de l'ISE1 est sécurisée contre un attaquant pouvant sonder d bits dans le circuit pendant l'exécution de l'implémentation masquée. L'ordre de masquage de l'ISE1 est fixé au moment de la synthèse. Une grande flexibilité est ainsi possible au moment de la conception pour divers compromis coût/performance. L'ISE1 a été implantée sur le processeur RISC-V CV32E40P pour réaliser des évaluations de performance et de surface.

Dans la section 7.1, nous décrivons les nouvelles instructions de l'ISE1. La section 7.2 présente la mise en œuvre de l'ISE1 ainsi que son implantation dans le cœur CV32E40P. Dans la section 7.3, nous évaluons les performances et la surface de l'ISE1 sur FPGA.

7.1 Spécification de l'ISE1

Dans cette section, nous décrivons tout d'abord les nouvelles instructions masquées que nous proposons. Puis, nous définissons des propriétés de composition permettant d'assurer la sécurité de séquences d'instructions masquées.

7.1.1 Description des instructions

Une implémentation BS (voir sous-section 4.1.3) permet de chiffrer 32 blocs en clair en parallèle. Le «*share slicing*» (voir sous-section 4.2.5) est une méthode de masquage des implémentations BS consistant à placer toutes les parts d'un bit masqué dans un même mot mémoire. Une implémentation «*share slicing*» masquée à l'ordre d permet alors de chiffrer $32/(d+1)$ blocs en clair en parallèle. Ce type d'implémentation permet de masquer efficacement à des ordres élevés [Gré+18 ; JS17]. Cependant, comme toutes les parts sont dans un même mot mémoire, [Gao+19] ont montré que la micro-architecture des processeurs pouvait produire une recombinaison des parts. Nous proposons une ISE dédiée au masquage, nommée ISE1, pour accélérer et sécuriser les implémentations «*share slicing*». L'ISE1 est composée de 4 instructions masquées `ise1.and`, `ise1.or`, `ise1.not` et `ise1.xor` réalisant respectivement les portes booléennes AND, OR, XOR et NOT masquées en respectant la représentation «*share slicing*». Le code de la figure 7.1 décrit les 4 nouvelles instructions en langage Python. En représentation «*share slicing*» d'ordre d, les registres sources `reg[rs1]` et `reg[rs2]` (du fichier de registres) contiennent les masquages d'ordre d de $32/(d+1)$ bits, chaque bit étant associé à un bloc chiffré différent. Par exemple, en représentation «*share slicing*» d'ordre 3, un mot mémoire de 32 bits contient 8 masquages de 4 parts. Nos instructions masquées appliquent une porte booléenne masquée à chaque masquage. mand, mxor et mnot désignent respectivement des circuits masqués des portes booléennes AND, XOR, NOT sécurisés dans le modèle de sondage robuste aux glitchs et/ou transitions. Pour terminer, le résultat de l'instruction masquée est écrit dans le fichier de registres. La figure 7.2 illustre l'instruction masquée `ise1.and`.

```
def ise1.and(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        reg[rd][i*n:(i+1)*n]=mand(reg[rs1][i*n:(i+1)*n],reg[rs2][i*n:(i+1)*n])

def ise1.xor(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        reg[rd][i*n:(i+1)*n]=mxor(reg[rs1][i*n:(i+1)*n],reg[rs2][i*n:(i+1)*n])

def ise1.not(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        reg[rd][i*n:(i+1)*n]=mnot(reg[rs1][i*n:(i+1)*n])

def ise1.or(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        tmp1 = mnot(reg[rs1][i*n:(i+1)*n])
        tmp2 = mnot(reg[rs2][i*n:(i+1)*n])
        tmp3 = mand(tmp1,tmp2)
        reg[rd][i*n:(i+1)*n]=mnot(tmp3)
```

FIGURE 7.1 – Code Python décrivant les nouvelles instructions de l'ISE1 masquée à un ordre $d \in \{1, \cdots, 31\}$. mand, mxor et mnot désignent respectivement des circuits masqués des portes booléennes AND, XOR, NOT. `reg[x]` désigne la valeur du registre d'indice `x` dans le banc de registres.

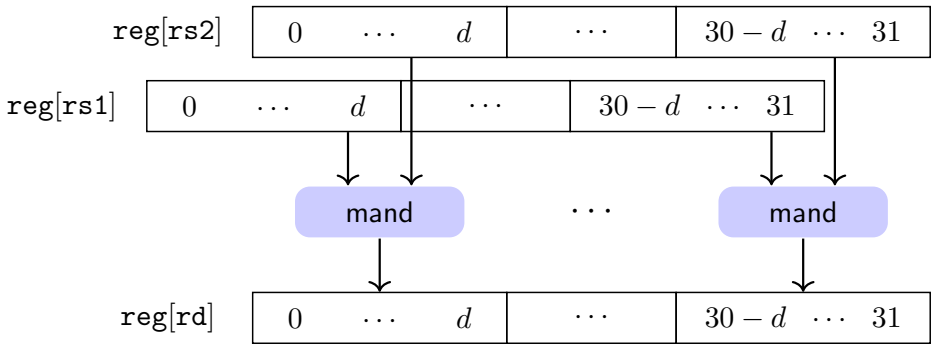


FIGURE 7.2 – Illustration de l'instruction `ise1.and` masquée à un ordre $d \in \{1, \cdots, 31\}$. mand désigne un circuit masqué à l'ordre d de la porte booléenne AND. `reg[x]` désigne la valeur du registre d'indice `x` dans le banc de registres.

7.1.2 Composition sécurisée d'instructions masquées

Pour assurer la sécurité de la composition de nos instructions masquées dans le modèle de sondage robuste aux glitchs et/ou transitions, leur implémentation doit être contrainte par des propriétés de sécurité plus fortes. Pour maintenir la sécurité de séquences d'instructions, la méthode utilisée dans les précédentes ISE de l'état de l'art consiste à contraindre les instructions linéaires à être d-NI et les instructions non-linéaires à être d-SNI dans le modèle de sondage robuste aux glitchs. Nous nommons cette méthode NI-SNI. Cependant, le programmeur doit alors appliquer des instructions de rafraîchissement à certains endroits bien choisis pour maintenir la sécurité de la composition (comme décrit à la sous-section 3.4.4). Pour composer simplement et de façon sécurisée les instructions de l'ISE1, nous proposons deux autres méthodes basées sur les propriétés de composition SNI et PINI.

Instructions SNI Pour assurer la sécurité d'une séquence d'instructions de l'ISE1, nous proposons de contraindre toutes les instructions masquées à être d-SNI dans le modèle de sondage robuste aux glitchs. Un programme utilisant uniquement les instructions de l'ISE1 est alors d-SNI, et donc sécurisé dans le modèle de sondage robuste aux glitchs. En effet, comme expliqué dans la sous-section 3.4.4, la composition d'implémentations masquées d-SNI est d-SNI. Cette méthode permet une composition simple et sécurisée des instructions de l'ISE1, mais impose un surcoût en temps d'exécution car les instructions linéaires doivent être d-SNI.

Instructions PINI Pour assurer la sécurité d'une séquence d'instructions de l'ISE1 tout en maintenant un masquage simple des instructions linéaires, nous proposons d'implémenter uniquement des instructions masquées d-PINI (resp. d-O-PINI). Un programme utilisant les instructions de l'ISE1 est alors d-PINI (resp. d-O-PINI) comme expliqué dans la sous-section 3.4.5, et donc sécurisé dans le modèle de sondage robuste aux glitchs (resp. aux glitchs et transitions). Les instructions PINI sont plus efficaces que les instructions SNI, car le masquage des instructions linéaires est simple. Les instructions PINI présentent également l'avantage de sécuriser contre les transitions de mémoire survenant durant des instructions *load*/*store* grâce à l'isolation des parts. En effet, les transitions survenant durant le remplacement de la valeur d'un registre n'apportent pas d'informations supplémentaires car les parts de même indice sont disposées à une même position dans un mot mémoire.

7.2 Implémentation de l'ISE1 sur le CV32E40P

Dans cette section, nous décrivons l'implémentation de l'ISE1 sur le cœur CV32E40P.

7.2.1 ALU masquée proposée

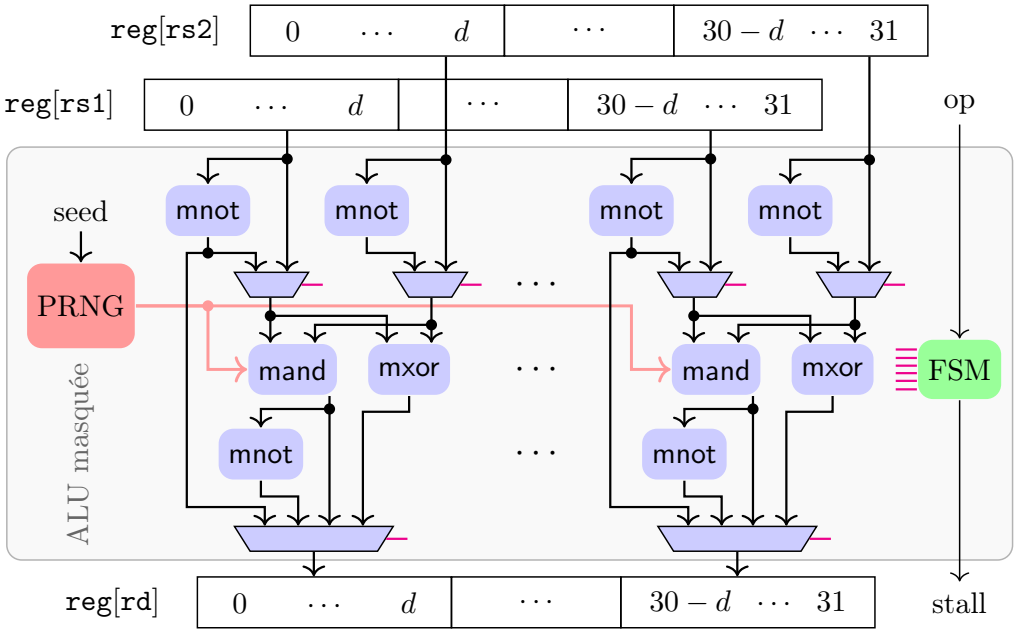


FIGURE 7.3 – Architecture de notre ALU masquée de l'ISE1.

L'ALU masquée décrite dans la figure 7.3 a été développée pour implémenter nos nouvelles instructions. Différentes versions de l'ALU masquée ont été réalisées pour chaque propriété de composition présentée à la section 7.1. Le tableau 7.1 décrit les caractéristiques des instructions masquées pour chaque version de l'ISE1. Dans la version NI-SNI, la porte masquée mand est implémentée avec le AND masqué d-SNI de [Fau+18] alors que les portes masquées mxor et mnot sont implémentées de façon directe. La version SNI est similaire à la précédente, à la différence qu'un circuit de rafraîchissement d-SNI est ajouté pour rendre les instructions `ise1.xor` et `ise1.not` d-SNI. La latence de ces instructions est alors de 3 cycles au lieu de 1 pour la version NI-SNI. Pour les instructions PINI, nous implémentons 3 versions différentes utilisant les portes masquées HPC1, HPC2, et HPC3 pour mand. Ces 3 versions permettent d'obtenir différents compromis entre la latence de l'instruction, le nombre d'aléas nécessaires, et la surface de l'implémentation. Par rapport à la version HPC1, les instructions `ise1.and` et `ise1.or` nécessitent deux fois moins d'aléas dans la version HPC2 et un cycle de moins dans la version HPC3. Dans ces 3 versions, les portes masquées mxor et mnot sont implémentées de façon directe, permettant d'exécuter les instructions `ise1.not` et `ise1.xor` en seulement 1 cycle. Pour les instructions O-PINI, nous implémentons 2 versions différentes utilisant les portes masquées O-PINI2 et HPC3$^+$ pour `mand`. La version HPC3$^+$ met un cycle de moins pour exécuter les instructions `ise1.and` et `ise1.or`, mais utilise deux fois plus d'aléas par instruction. Dans toutes ces versions, la porte masquée `mor` est implémentée en utilisant la porte masquée `mand` où ses opérandes et son résultat passent par des portes masquées `mnot` conformément aux lois

de de Morgan. Un *générateur de nombres pseudo-aléatoires*, abrégé en PRNG pour «*pseudo-random number generator*» en anglais, fournit des valeurs aléatoires pour les portes masquées. Il utilise une seule ronde de la permutation Keccak comme spécifié dans [Ber+10]. Les paramètres de la permutation ont été minimisés de façon à fournir suffisamment de bits aléatoires à chaque cycle d'horloge. Le PRNG est régulièrement réinitialisé à l'aide d'un *générateur de nombres aléatoires vrai*, abrégé en TRNG pour «*true random number generator*» en anglais, qui se trouve en dehors du cœur. [Yan+18] donne un exemple de TRNG qui peut être implémenté en pratique. Une FSM contrôle les multiplexeurs internes de l'ALU masquée en fonction de l'opération décodée. La FSM contrôle également le *pipeline* pour les instructions multi-cycles.

Version ISE1	Paramètre	`ise1.and`/`ise1.or`	`ise1.not`/`ise1.xor`
NI-SNI	Composition	SNI	NI
	Latence	3	1
	Aléas	$m \times \frac{d(d+1)}{2}$	0
SNI	Composition	SNI	SNI
	Latence	3	3
	Aléas	$m \times \frac{d(d+1)}{2}$	$m \times \frac{d(d+1)}{2}$
HPC1	Composition	PINI	PINI
	Latence	3	1
	Aléas	$m \times d(d+1)$	0
HPC2	Composition	PINI	PINI
	Latence	3	1
	Aléas	$m \times \frac{d(d+1)}{2}$	0
HPC3	Composition	PINI	PINI
	Latence	2	1
	Aléas	$m \times d(d+1)$	0
O-PINI2	Composition	O-PINI	O-PINI
	Latence	4	1
	Aléas	$m \times \left(\frac{d(d+1)}{2} + d\right)$	0
HPC3$^+$	Composition	O-PINI	O-PINI
	Latence	3	1
	Aléas	$m \times (d(d+1) + d)$	0

TABLE 7.1 – Caractéristiques des instructions masquées pour chaque version de l'ISE1. m est égal à $\mathsf{rnd}\left(\frac{32}{d+1}\right)$. La latence est exprimée en nombre de cycles et les aléas en nombre de bits par instruction.

7.2.2 Intégration dans le cœur CV32E40P

Le cœur CV32E40P ayant été conçu pour être modifié et étendu, l'intégration de l'ISE1 a été simple. Le cœur a été modifié comme illustré sur la figure 7.4. Le décodeur d'instructions modifié traite les nouvelles instructions et génère des signaux de contrôle pour les instructions multi-cycles. Pour éviter les problèmes de recombinaison des parts, nous avons conçu un chemin de données totalement séparé pour l'ALU masquée. En effet, [Gao+19] a montré que les calculs réalisés dans l'ALU de base peuvent engendrer une recombinaison des parts. La mise en place du chemin de données séparé nécessite d'étendre le registre de *pipeline* ID/EX avec 2 nouveaux registres 32 bits accessibles uniquement depuis l'ALU masquée. Selon l'instruction décodée, soit le chemin de données de masquage, soit le chemin de données normal est activé.

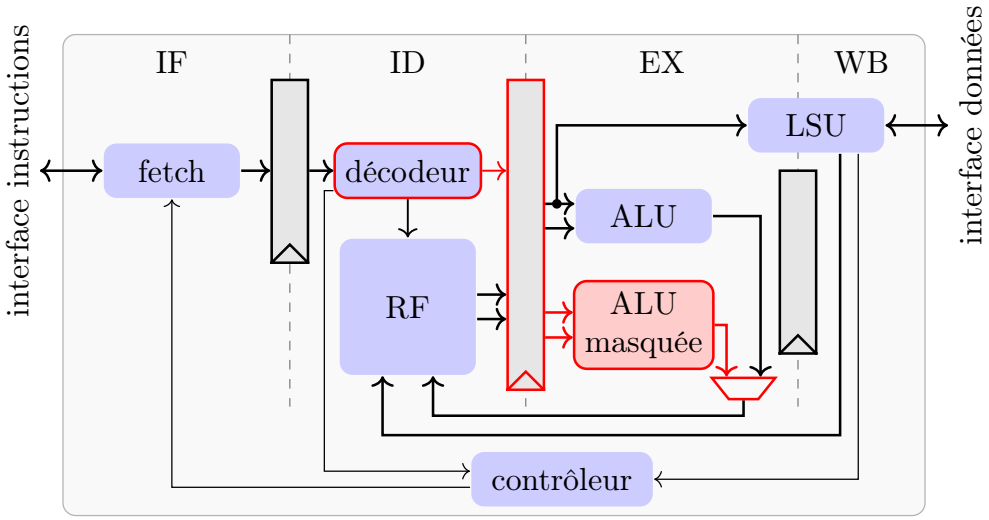


FIGURE 7.4 – Schéma simplifié du cœur modifié avec l'ISE1. Les parties rouges ont été ajoutées ou modifiées pour notre ISE.

7.2.3 Utilisation logicielle de l'ISE

Notre ISE peut être utilisée pour masquer divers cryptosystèmes. Nous utilisons Usuba [MD19] pour générer une implémentation BS en langage C. Pour la version NI-SNI de l'ISE, nous utilisons Tornado pour maintenir la sécurité en rafraîchissant le masquage à certains endroits. Les opérations élémentaires dans le langage C sont remplacées par les intrinsèques assembleurs de nos instructions masquées. Les blocs d'entrées sont préalablement masqués puis convertis en représentation BS à l'aide de décalages, d'opérations logiques et de valeurs aléatoires. À la toute fin, les blocs de sortie sont reconvertis en représentation standard puis démasqués à l'aide de décalages et d'opérations logiques.

7.3 Évaluation des performances et coûts

La figure 7.5 présente le temps d'exécution du chiffrement AES masqué en utilisant la version HPC3 de l'ISE1 (labellisée par HW-M-ISE1), ainsi que différentes solutions de la littérature que nous avons réimplémenté sur le cœur CV32E40P (voir section 6.3). Bien que nous nous sommes

focalisé sur le chiffrement AES, notre solution est applicable à toutes sortes de cryptosystèmes. On constate que HW-M-ISE1 surpasse l'implémentation BS masquée en logiciel SW-M-BS. Par exemple, la version HPC3 de l'ISE1 accélère SW-M-BS de 4,7 à l'ordre 1, de 9,0 à l'ordre 3, de 13,6 à l'ordre 7, de 23,8 à l'ordre 15, de 25,0 à l'ordre 23 et de 42,5 à l'ordre 31. Plus l'ordre de masquage est élevé, plus les gains sont importants. Concernant les solutions matérielles, l'ISE1 est légèrement plus rapide que Skiva mais plus lente que SME. Cela s'explique car SME est optimisé pour le chiffrement AES, mais est moins efficace pour implémenter d'autres cryptosystèmes.

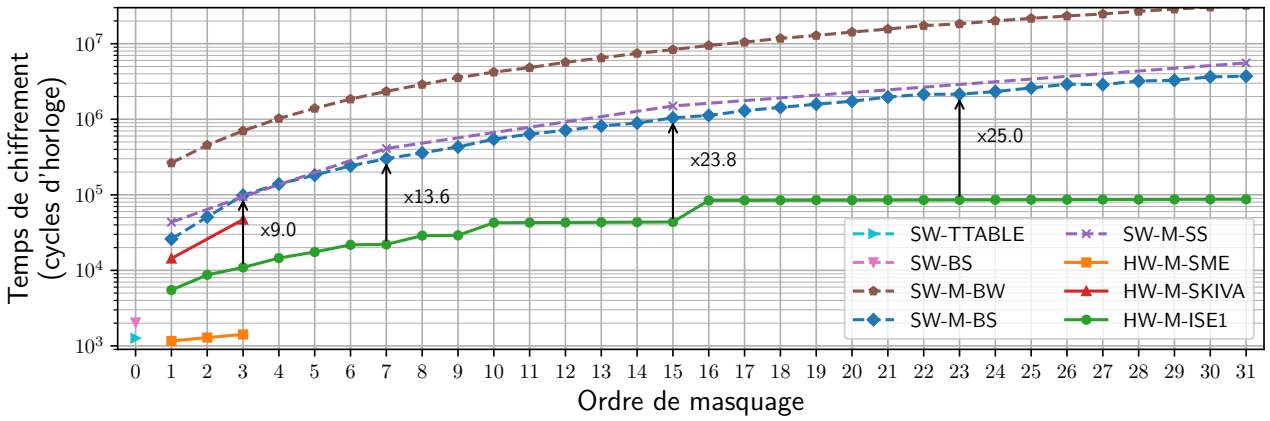


FIGURE 7.5 – Temps de chiffrement (en échelle log) de un bloc de AES avec la version HPC3 de l'ISE1 et diverses solutions de masquage logiciel et matériel.

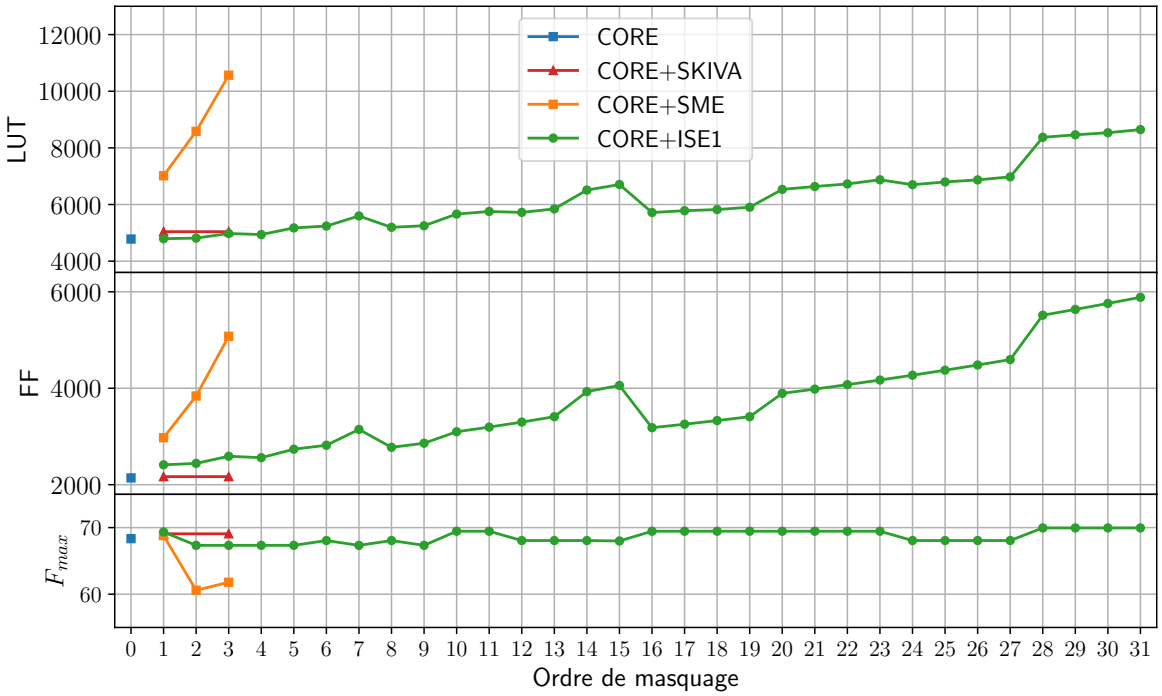


FIGURE 7.6 – Surface et fréquence maximale du cœur CV32E40P (CORE) sans ISE, avec Skiva (CORE+SKIVA), avec SME (CORE+SME) et avec la version HPC3 de l'ISE1 (CORE+ISE1).

La figure 7.6 présente la surface et la fréquence maximale du cœur CV32E40P (CORE) sans ISE, avec Skiva (CORE+SKIVA), avec SME (CORE+SME) et avec la version HPC3 de l'ISE1 (CORE+ISE1), pour des ordres de masquage d dans $\{1, 2, 3, \ldots 31\}$. Notre solution a un impact limité en surface, même pour des ordres de masquage élevés. Par exemple, le nombre de FF est augmenté par 1,13 à l'ordre 1, par 1,21 à l'ordre 3, par 1,47 à l'ordre 7, par 1,90 à l'ordre 15 et par 2,75 à l'ordre 31. L'ISE1 a une surface légèrement plus grande que Skiva à l'ordre 1 et 3, mais elle est plus rapide en temps de chiffrement. L'ISE1 masquée aux ordres 1, 2 et 3 est bien plus petite que SME. Bien que SME soit plus rapide en termes de temps d'exécution, cette ISE impose un coût important en termes de surface et il semble que SME ne puisse pas être implantée dans des petits FPGA pour des ordres de masquage élevés. De plus, l'ISE1 n'impacte pas la fréquence maximale du cœur CV32E40P (le chemin critique n'est pas dans l'ISE), contrairement à SME.

Comparaison des différentes versions de l'ISE Le tableau 7.2 présente les résultats d'implantation des différentes versions de l'ISE1 masquée aux ordres $d \in \{1, 3, 7, 15, 31\}$. La version NI-SNI est la plus petite et est assez efficace en termes de temps de chiffrement. Par contre, cette version est plus complexe à utiliser car l'utilisateur doit insérer des instructions de rafraîchissement dans le code pour maintenir la sécurité de la composition. La version SNI permet de composer directement les instructions masquées de façon sécurisée. Bien que le coût matériel reste faible, cette version est la moins efficace en termes de temps de chiffrement. Cette moindre efficacité s'explique par le fait que l'instruction `ise1.xor` s'exécute en 3 cycles d'horloge. Une autre solution pour composer directement les instructions masquées de façon sécurisée est d'utiliser des instructions PINI. On constate que la version HPC2 consomme plus de surface que HPC1 même si HPC1 utilise plus d'aléas par instruction. HPC3 est la version la plus efficace en temps de chiffrement, mais induit un léger surcoût en surface par rapport à HPC1. Concernant les versions O-PINI de l'ISE1, la version HPC3$^+$ est plus performante en temps de chiffrement et a une surface plus faible que la version O-PINI2. En résumé, la version NI-SNI induit le plus faible coût en matériel, mais est plus complexe à utiliser. La version HPC3 est la plus rapide en temps de chiffrement et est simple d'utilisation. La version HPC3$^+$ est la plus efficace pour protéger contre les glitchs et les transitions simultanément.

d	Implém.	Version ISE1	Temps			Surface & période		
			Cycles	cmp1	cmp2	FF	LUT	P(ns)
0	SW-TTABLE	Sans ISE	1266	base	n.a.	base	base	14.6
1	SW-M-BS	Sans ISE	26001	×21	base	×1,00	×1,00	14.6
	HW-M-ISE1	NI-SNI	5803	×4,6	/4,5	×1,10	×1,02	14,9
		SNI	8309	×6,6	/3,1	×1,14	×1,00	14,4
		HPC1	5803	×4,6	/4,5	×1,14	×1,02	14,9
		HPC2	5803	×4,6	/4,5	×1,16	×1,00	14,4
		HPC3	5483	×4,3	/4,7	×1,13	×1,00	14,4
		O-PINI2	6123	×4,8	/4,2	×1,19	×1,01	14,9
		HPC3$^+$	5803	×4,6	/4,5	×1,17	×1,02	14,9
3	SW-M-BS	Sans ISE	98575	×78	base	×1,00	×1,00	14.6
	HW-M-ISE1	NI-SNI	11567	×9,1	/8,5	×1,14	×1,03	14,9
		SNI	16579	×13,1	/5,9	×1,19	×1,01	14,4
		HPC1	11567	×9,1	/8,5	×1,22	×1,03	14,9
		HPC2	11567	×9,1	/8,5	×1,29	×1,05	14,9
		HPC3	10927	×8,6	/9,0	×1,21	×1,04	14,9
		O-PINI2	12207	×9,6	/8,1	×1,34	×1,06	14,9
		HPC3$^+$	11567	×9,1	/8,5	×1,29	×1,07	14,9
7	SW-M-BS	Sans ISE	300414	×237	base	×1,00	×1,00	14.6
	HW-M-ISE1	NI-SNI	23313	×18,4	/12,9	×1,27	×1,07	14,9
		SNI	33337	×26,3	/9,0	×1,35	×1,08	14,4
		HPC1	23313	×18,4	/12,9	×1,45	×1,15	14,9
		HPC2	23313	×18,4	/12,9	×1,60	×1,15	14,9
		HPC3	22033	×17,4	/13,6	×1,47	×1,17	14,9
		O-PINI2	24593	×19,4	/12,2	×1,63	×1,15	14,9
		HPC3$^+$	23313	×18,4	/12,9	×1,50	×1,18	14,9
15	SW-M-BS	Sans ISE	1041285	×822	base	×1,00	×1,00	14.6
	HW-M-ISE1	NI-SNI	46237	×36,5	/22,5	×1,48	×1,17	14,9
		SNI	66285	×52,4	/15,7	×1,63	×1,18	14,4
		HPC1	46237	×36,5	/22,5	×1,81	×1,31	14,7
		HPC2	46237	×36,5	/22,5	×2,17	×1,34	14,9
		HPC3	43677	×34,5	/23,8	×1,90	×1,40	14,7
		O-PINI2	48797	×38,5	/21,3	×2,20	×1,34	14,9
		HPC3$^+$	46237	×36,5	/22,5	×1,93	×1,38	14,9
31	SW-M-BS	Sans ISE	3716920	×2936	base	×1,00	×1,00	14.6
	HW-M-ISE1	NI-SNI	92609	×73,2	/40,1	×1,91	×1,36	14,9
		SNI	132705	×104,8	/28,0	×2,17	×1,41	14,4
		HPC1	92609	×73,2	/40,1	×2,55	×1,67	14,6
		HPC2	92609	×73,2	/40,1	×3,32	×1,74	14,6
		HPC3	87489	×69,1	/42,5	×2,75	×1,81	14,3
		O-PINI2	97729	×77,2	/38,0	×3,35	×1,74	14,6
		HPC3$^+$	92609	×73,2	/40,1	×2,78	×1,83	14,3

TABLE 7.2 – Résultats d'implantation de l'ISE1 masquée au ordres $d \in \{1, 2, 3, 7, 15, 31\}$. Le temps d'exécution est donné en nombre de cycles pour chiffrer un bloc avec AES. cmp1 et cmp2 comparent respectivement le temps d'exécution par rapport à SW-TTABLE et SW-M-BS. Les nombres de FF et LUT sont donnés en surcoût par rapport à ceux du cœur CV32E40P de base.

Chapitre 8

Extension masquée pour les implémentations m-slicing

L'ISE1 que nous avons présenté au chapitre 7 permet d'accélérer les implémentations masquées à des ordres élevés de divers cryptosystèmes en ayant un impact limité en surface. Néanmoins, lorsqu'on implémente un cryptosystème avec l'ISE1, $32/(d+1)$ blocs indépendants doivent être chiffrés en parallèle pour obtenir les meilleures performances. Par exemple, 16 blocs indépendants sont chiffrés en parallèle avec l'ISE1 masquée à l'ordre 1. Seule l'ISE1 masquée à un ordre supérieur à 16 permet de chiffrer un seul bloc à la fois. Le chiffrement de nombreux blocs en parallèle induit un coût lié à la gestion des différents blocs et augmente la pression sur les registres généraux du processeur. Cela limite également les modes de chiffrement utilisables à ceux permettant le chiffrement parallèle de différents blocs, tel que le mode CTR (voir sous-section 1.1.3). Les modes CBC, CFB et OFB peuvent être utilisés seulement pour chiffrer des flux de données complètement indépendants. Mais cette situation se produit rarement dans des systèmes embarquées où les flux de données sont plus faibles. Dans ce chapitre, nous proposons une nouvelle ISE dédiée au masquage, que nous nommons ISE2, qui étend l'ISE1 avec d'autres instructions pour masquer des implémentations *m-slicing* aux ordres $d \in \{1, 3, 7, 15\}$. L'ordre de masquage de l'ISE est fixé au moment de la synthèse. En masquant les implémentations 16-*slicing*, 8-*slicing*, 4-*slicing* et 2-*slicing* avec l'ISE2 masquée respectivement aux ordres 1, 3, 7, 15, nous obtenons des implémentations permettant de masquer efficacement un bloc à la fois. L'utilisation de tous les modes de chiffrement devient alors possible.

Dans la section 8.1, nous décrivons les nouvelles instructions de l'ISE2. La section 8.2 décrit 4 implémentations *m-slicing* de AES qui sont masquées avec l'ISE2. Deux de ces implémentations sont nouvelles. Dans la section 8.3, nous évaluons les performances et la surface de l'ISE2.

8.1 Spécification de l'ISE2

Une implémentation *m-slicing* (voir sous-section 4.1.4) permet de chiffrer $32/m$ blocs en clair en parallèle. En masquant une implémentation *m-slicing* à l'ordre d en utilisant la méthode du «*share slicing*» (voir sous-section 4.2.5), on réduit à $32/(m \times (d+1))$ le nombre de blocs en clair chiffrés en parallèle. Ainsi, une implémentation 16-*slicing* masquée à l'ordre 1 chiffre un seul bloc en utilisant le «*share slicing*». De même, en masquant respectivement les implémentations 8-*slicing*, 4-*slicing* et 2-*slicing* aux ordres 3, 7, et 15 avec le «*share slicing*», nous obtenons des implémentations permettant de chiffrer un bloc à la fois. Une implémentation *m-slicing*

utilise des instructions booléennes, mais également des instructions de décalage pour calculer sur les m bits d'un bloc présent dans un mot machine. Pour accélérer les implémentations m-*slicing* masquées aux ordres $d \in \{1, 3, 7, 15\}$ avec le «*share slicing*», nous avons donc ajouté des instructions masquées à l'ISE1. Les nouvelles instructions sont présentées dans la figure 8.1 et implémentent le décalage à droite et à gauche adapté à la représentation «*share slicing*».

```
def ise2.sll(rs1, rs2, rd):
    shift = int("".join(str(x) for x in reg[rs2][0:4]),2)
    reg[rd][0:32-(d+1)*shift]=reg[rs1][(d+1)*shift:32]
    reg[rd][32-(d+1)*shift:32]=(d+1)*shift*[0]

def ise2.srl(rs1, rs2, rd):
    shift = int("".join(str(x) for x in reg[rs2][0:4]),2)
    reg[rd][0:(d+1)*shift]=(d+1)*shift*[0]
    reg[rd][(d+1)*shift:32]=reg[rs1][0:32-(d+1)*shift]

def ise2.slli(rs1, imm, rd):
    shift = int("".join(str(x) for x in imm[0:4]),2)
    reg[rd][0:32-(d+1)*shift]=reg[rs1][(d+1)*shift:32]
    reg[rd][32-(d+1)*shift:32]=(d+1)*shift*[0]

def ise2.srli(rs1, imm, rd):
    shift = int("".join(str(x) for x in imm[0:4]),2)
    reg[rd][0:(d+1)*shift]=(d+1)*shift*[0]
    reg[rd][(d+1)*shift:32]=reg[rs1][0:32-(d+1)*shift]
```

FIGURE 8.1 – Code Python décrivant les nouvelles instructions de l'ISE2. `reg[x]` désigne la valeur du registre d'indice `x` dans le banc de registres.

8.2 Nouvelles implémentations m-slicing de AES

Nous avons développé des implémentations 16-*slicing*, 8-*slicing*, 4-*slicing* et 2-*slicing* de AES en utilisant Usuba (voir sous-section 4.1.3). Deux de ces implémentations sont nouvelles. En utilisant la méthode du «share slicing», ces implémentations de AES permettent de masquer un bloc à la fois avec l'ISE2 respectivement aux ordres 1, 3, 7, 15.

8.2.1 Implémentation 16-slicing de AES

	ligne 3								.....	ligne 0							
	colonne 0		colonne 1		colonne 2		colonne 3		.....	colonne 0		colonne 1		colonne 2		colonne 3	
	bloc 0	bloc 1	bloc 0	bloc 1	bloc 0	bloc 1	bloc 0	bloc 1	.....	bloc 0	bloc 1	bloc 0	bloc 1	bloc 0	bloc 1	bloc 0	bloc 1
M_0	b^0_{24}	b^1_{24}	b^0_{56}	b^1_{56}	b^0_{88}	b^1_{88}	b^0_{120}	b^1_{120}	.....	b^0_0	b^1_0	b^0_{32}	b^1_{32}	b^0_{64}	b^1_{64}	b^0_{96}	b^1_{96}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	.....	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
M_7	b^0_{31}	b^1_{31}	b^0_{63}	b^1_{63}	b^0_{95}	b^1_{95}	b^0_{127}	b^1_{127}	.....	b^0_7	b^1_7	b^0_{39}	b^1_{39}	b^0_{71}	b^1_{71}	b^0_{103}	b^1_{103}

FIGURE 8.2 – Notre représentation 16-*slicing* de 2 blocs dans 8 mots machines ($M_i, 0 \leq i \leq 7$) d'une architecture 32 bits. b^j_i désigne le i-ième bit du j-ième bloc.

Nous avons réimplémenté en utilisant Usuba l'implémentation 16-*slicing* de AES proposée dans [SS16] pour un processeur ARM Cortex-M3, puis portée sur une architecture RISC-V 32 bits dans [Sto19]. Cette implémentation permet de chiffrer en parallèle 2 blocs en clair en les disposant dans 8 mots machines comme illustré dans la figure 8.2. En masquant cette implémentation à l'ordre 1 avec le «*share slicing*», on peut alors chiffrer un seul bloc. Le code Usuba de cette implémentation se trouve dans la figure B.1 de l'annexe B. La sous-fonction MixColumns peut être calculée efficacement avec des instructions booléennes et de décalage. En effet, la multiplication par 2 est réalisée par un décalage vers la gauche et un masquage conditionnel avec $(00011011)_2$ chaque fois que le bit de poids fort (MSB) vaut 1. Puisque $\mathtt{M}_0$ contient le MSB de chaque octet, il suffit de l'ajouter aux quatre registres correspondants. De plus, comme les bits d'un bloc sont rangés par ligne puis colonne, l'ajout d'une ligne de l'état de AES à une autre se réalise simplement par une rotation et un XOR. MixColumns peut ainsi être réalisé de la manière suivante :

$$
\begin{aligned}
\mathtt{M}'_0 &= (\mathtt{M}_1 \oplus \mathtt{M}_1^{\gg 8}) \oplus \mathtt{M}_0^{\gg 8} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_0^{\gg 8})^{\gg 16} \\
\mathtt{M}'_1 &= (\mathtt{M}_2 \oplus \mathtt{M}_2^{\gg 8}) \oplus \mathtt{M}_1^{\gg 8} \oplus (\mathtt{M}_1 \oplus \mathtt{M}_1^{\gg 8})^{\gg 16} \\
\mathtt{M}'_2 &= (\mathtt{M}_3 \oplus \mathtt{M}_3^{\gg 8}) \oplus \mathtt{M}_2^{\gg 8} \oplus (\mathtt{M}_2 \oplus \mathtt{M}_2^{\gg 8})^{\gg 16} \\
\mathtt{M}'_3 &= (\mathtt{M}_4 \oplus \mathtt{M}_4^{\gg 8}) \oplus \mathtt{M}_3^{\gg 8} \oplus (\mathtt{M}_3 \oplus \mathtt{M}_3^{\gg 8})^{\gg 16} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_0^{\gg 8}) \\
\mathtt{M}'_4 &= (\mathtt{M}_5 \oplus \mathtt{M}_5^{\gg 8}) \oplus \mathtt{M}_4^{\gg 8} \oplus (\mathtt{M}_4 \oplus \mathtt{M}_4^{\gg 8})^{\gg 16} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_0^{\gg 8}) \\
\mathtt{M}'_5 &= (\mathtt{M}_6 \oplus \mathtt{M}_6^{\gg 8}) \oplus \mathtt{M}_5^{\gg 8} \oplus (\mathtt{M}_5 \oplus \mathtt{M}_5^{\gg 8})^{\gg 16} \\
\mathtt{M}'_6 &= (\mathtt{M}_7 \oplus \mathtt{M}_7^{\gg 8}) \oplus \mathtt{M}_6^{\gg 8} \oplus (\mathtt{M}_6 \oplus \mathtt{M}_6^{\gg 8})^{\gg 16} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_0^{\gg 8}) \\
\mathtt{M}'_7 &= (\mathtt{M}_0 \oplus \mathtt{M}_0^{\gg 8}) \oplus \mathtt{M}_7^{\gg 8} \oplus (\mathtt{M}_7 \oplus \mathtt{M}_7^{\gg 8})^{\gg 16}
\end{aligned}
$$

8.2.2 Implémentation 8-slicing de AES

	ligne 0							ligne 2						
	colonne 0			⋯	colonne 3			colonne 0			⋯	colonne 3		
	bloc 0	…	bloc 3	…	bloc 0	…	bloc 3	bloc 0	…	bloc 3	…	bloc 0	…	bloc 3
$\mathtt{M}_0$	b_0^0	⋯	b_0^3	⋯	b_{96}^0	⋯	b_{96}^3	b_8^0	⋯	b_8^3	⋯	b_{104}^0	⋯	b_{104}^3
⋮	⋮	…	⋮	…	⋮	…	⋮	⋮	…	⋮	…	⋮	…	⋮
$\mathtt{M}_7$	b_7^0	⋯	b_7^3	⋯	b_{103}^0	⋯	b_{103}^3	b_{15}^0	⋯	b_{15}^3	⋯	b_{111}^0	⋯	b_{111}^3

	ligne 1							ligne 3						
	colonne 0			⋯	colonne 3			colonne 0			⋯	colonne 3		
	bloc 0	…	bloc 3	…	bloc 0	…	bloc 3	bloc 0	…	bloc 3	…	bloc 0	…	bloc 3
$\mathtt{M}_8$	b_{16}^0	⋯	b_{16}^3	⋯	b_{112}^0	⋯	b_{112}^3	b_{24}^0	⋯	b_{24}^3	⋯	b_{120}^0	⋯	b_{120}^3
⋮	⋮	…	⋮	…	⋮	…	⋮	⋮	…	⋮	…	⋮	…	⋮
$\mathtt{M}_{15}$	b_{23}^0	⋯	b_{23}^3	⋯	b_{119}^0	⋯	b_{119}^3	b_{31}^0	⋯	b_{31}^3	⋯	b_{127}^0	⋯	b_{127}^3

FIGURE 8.3 – Notre représentation 8-*slicing* de 4 blocs dans 16 mots machines ($\mathtt{M}_i, 0 \leq i \leq 15$) d'une architecture 32 bits. b_i^j désigne le i-ième bit du j-ième bloc.

Nous avons développé une nouvelle implémentation 8-*slicing* de AES en utilisant Usuba. Cette implémentation permet de chiffrer en parallèle 4 blocs en clair en les disposant dans 16 mots machines comme illustré dans la figure 8.3. Cette implémentation masquée à l'ordre 1 ou 3 avec le «*share slicing*» permet de chiffrer respectivement 2 ou 1 blocs en parallèle. Le code Usuba de cette implémentation se trouve dans la figure B.2 de l'annexe B. La sous-fonction MixColumns peut être calculée efficacement avec des instructions booléennes et de décalage de la manière suivante :

$$
\begin{aligned}
\mathtt{M}'_0 &= \mathtt{M}_1 \oplus \mathtt{M}_9 \oplus \mathtt{M}_8 \oplus (\mathtt{M}_0 \oplus \mathtt{M}_8)^{\gg 16} \\
\mathtt{M}'_1 &= \mathtt{M}_2 \oplus \mathtt{M}_{10} \oplus \mathtt{M}_9 \oplus (\mathtt{M}_1 \oplus \mathtt{M}_9)^{\gg 16} \\
\mathtt{M}'_2 &= \mathtt{M}_3 \oplus \mathtt{M}_{11} \oplus \mathtt{M}_{10} \oplus (\mathtt{M}_2 \oplus \mathtt{M}_{10})^{\gg 16} \\
\mathtt{M}'_3 &= \mathtt{M}_4 \oplus \mathtt{M}_{12} \oplus \mathtt{M}_{11} \oplus (\mathtt{M}_3 \oplus \mathtt{M}_{11})^{\gg 16} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_8) \\
\mathtt{M}'_4 &= \mathtt{M}_5 \oplus \mathtt{M}_{13} \oplus \mathtt{M}_{12} \oplus (\mathtt{M}_4 \oplus \mathtt{M}_{12})^{\gg 16} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_8) \\
\mathtt{M}'_5 &= \mathtt{M}_6 \oplus \mathtt{M}_{14} \oplus \mathtt{M}_{13} \oplus (\mathtt{M}_5 \oplus \mathtt{M}_{13})^{\gg 16} \\
\mathtt{M}'_6 &= \mathtt{M}_7 \oplus \mathtt{M}_{15} \oplus \mathtt{M}_{14} \oplus (\mathtt{M}_6 \oplus \mathtt{M}_{14})^{\gg 16} \oplus (\mathtt{M}_0 \oplus \mathtt{M}_8) \\
\mathtt{M}'_7 &= \mathtt{M}_0 \oplus \mathtt{M}_8 \oplus \mathtt{M}_{15} \oplus (\mathtt{M}_7 \oplus \mathtt{M}_{15})^{\gg 16} \\
\mathtt{M}'_8 &= \mathtt{M}_9 \oplus (\mathtt{M}_1 \oplus \mathtt{M}_0 \oplus \mathtt{M}_8)^{\gg 16} \oplus \mathtt{M}_0 \\
\mathtt{M}'_9 &= \mathtt{M}_{10} \oplus (\mathtt{M}_2 \oplus \mathtt{M}_1 \oplus \mathtt{M}_9)^{\gg 16} \oplus \mathtt{M}_1 \\
\mathtt{M}'_{10} &= \mathtt{M}_{11} \oplus (\mathtt{M}_3 \oplus \mathtt{M}_2 \oplus \mathtt{M}_{10})^{\gg 16} \oplus \mathtt{M}_2 \\
\mathtt{M}'_{11} &= \mathtt{M}_{12} \oplus (\mathtt{M}_4 \oplus \mathtt{M}_3 \oplus \mathtt{M}_{11})^{\gg 16} \oplus \mathtt{M}_3 \oplus (\mathtt{M}_8 \oplus \mathtt{M}_0^{\gg 16}) \\
\mathtt{M}'_{12} &= \mathtt{M}_{13} \oplus (\mathtt{M}_5 \oplus \mathtt{M}_4 \oplus \mathtt{M}_{12})^{\gg 16} \oplus \mathtt{M}_4 \oplus (\mathtt{M}_8 \oplus \mathtt{M}_0^{\gg 16}) \\
\mathtt{M}'_{13} &= \mathtt{M}_{14} \oplus (\mathtt{M}_6 \oplus \mathtt{M}_5 \oplus \mathtt{M}_{13})^{\gg 16} \oplus \mathtt{M}_5 \\
\mathtt{M}'_{14} &= \mathtt{M}_{15} \oplus (\mathtt{M}_7 \oplus \mathtt{M}_6 \oplus \mathtt{M}_{14})^{\gg 16} \oplus \mathtt{M}_6 \oplus (\mathtt{M}_8 \oplus \mathtt{M}_0^{\gg 16}) \\
\mathtt{M}'_{15} &= \mathtt{M}_8 \oplus (\mathtt{M}_0 \oplus \mathtt{M}_7 \oplus \mathtt{M}_{15})^{\gg 16} \oplus \mathtt{M}_7
\end{aligned}
$$

8.2.3 Implémentation 4-slicing de AES

Nous avons réimplémenté en utilisant Usuba l'implémentation 4-*slicing* de AES proposée dans [AP20]. Cette implémentation permet de chiffrer en parallèle 8 blocs en clair en les disposant dans 32 mots machines comme illustré dans la figure 8.4. Cette implémentation masquée à l'ordre 1, 3 ou 7 avec le «*share slicing*» permet de chiffrer respectivement 4, 2 ou 1 blocs en parallèle. Le code Usuba de cette implémentation se trouve dans la figure B.3 de l'annexe B. Cette représentation permet de calculer efficacement ShiftRows car les lignes de l'état sont isolées dans des registres distincts. MixColumns ne nécessite plus de rotation mais uniquement des XOR puisque les différents octets au sein d'une colonne sont désormais stockés dans des registres distincts. MixColumns peut être calculé de la manière suivante :

$$
\begin{aligned}
\mathtt{M}'_{i} &= \mathtt{M}_{i+1} \oplus \mathtt{M}_{i+9} \oplus \mathtt{M}_{i+8} \oplus \mathtt{M}_{i+16} \oplus \mathtt{M}_{i+24} \\
\mathtt{M}'_{i+1} &= \mathtt{M}_{i+2} \oplus \mathtt{M}_{i+10} \oplus \mathtt{M}_{i+9} \oplus \mathtt{M}_{i+17} \oplus \mathtt{M}_{i+25} \\
\mathtt{M}'_{i+2} &= \mathtt{M}_{i+3} \oplus \mathtt{M}_{i+11} \oplus \mathtt{M}_{i+10} \oplus \mathtt{M}_{i+18} \oplus \mathtt{M}_{i+26} \\
\mathtt{M}'_{i+3} &= \mathtt{M}_{i+4} \oplus \mathtt{M}_{i+12} \oplus \mathtt{M}_{i+11} \oplus \mathtt{M}_{i+19} \oplus \mathtt{M}_{i+27} \oplus (\mathtt{M}_{i} \oplus \mathtt{M}_{i+8}) \\
\mathtt{M}'_{i+4} &= \mathtt{M}_{i+5} \oplus \mathtt{M}_{i+13} \oplus \mathtt{M}_{i+12} \oplus \mathtt{M}_{i+20} \oplus \mathtt{M}_{i+28} \oplus (\mathtt{M}_{i} \oplus \mathtt{M}_{i+8}) \\
\mathtt{M}'_{i+5} &= \mathtt{M}_{i+6} \oplus \mathtt{M}_{i+14} \oplus \mathtt{M}_{i+13} \oplus \mathtt{M}_{i+21} \oplus \mathtt{M}_{i+29} \\
\mathtt{M}'_{i+6} &= \mathtt{M}_{i+7} \oplus \mathtt{M}_{i+15} \oplus \mathtt{M}_{i+14} \oplus \mathtt{M}_{i+22} \oplus \mathtt{M}_{i+30} \oplus (\mathtt{M}_{i} \oplus \mathtt{M}_{i+8}) \\
\mathtt{M}'_{i+7} &= \mathtt{M}_{i} \oplus \mathtt{M}_{i+8} \oplus \mathtt{M}_{i+15} \oplus \mathtt{M}_{i+23} \oplus \mathtt{M}_{i+31}
\end{aligned}
$$

pour $i \in \{0, 8, 16, 24\}$ et où tous les indices doivent être considérés modulo 32.

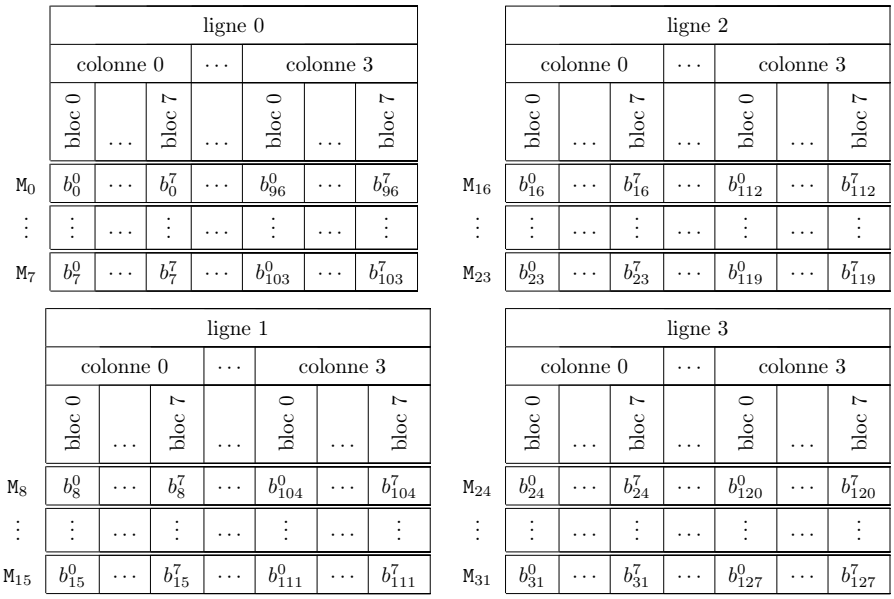


FIGURE 8.4 – Notre représentation 4-*slicing* de 8 blocs dans 32 mots machines ($\mathtt{M}_i, 0 \leq i \leq 31$) d'une architecture 32 bits. b_i^j désigne le i-ième bit du j-ième bloc.

8.2.4 Implémentation 2-slicing de AES

Nous avons développé une nouvelle implémentation 2-*slicing* de AES en utilisant Usuba. Cette implémentation permet de chiffrer en parallèle 16 blocs en clair en les disposant dans 64 mots machines comme illustré dans la figure 8.5. Cette implémentation masquée à l'ordre 1, 3, 7 ou 15 avec le «*share slicing*» permet de chiffrer respectivement 8, 4, 2 ou 1 blocs en parallèle. Le code Usuba de cette implémentation se trouve dans la figure B.4 de l'annexe B. La sous-fonction MixColumns peut être calculée efficacement avec des instructions booléennes et de décalage de la manière suivante :

$$
\begin{aligned}
\mathtt{M}'_i &= \mathtt{M}_{i+1} \oplus \mathtt{M}_{i+17} \oplus \mathtt{M}_{i+16} \oplus \mathtt{M}_{i+32} \oplus \mathtt{M}_{i+48} \\
\mathtt{M}'_{i+1} &= \mathtt{M}_{i+2} \oplus \mathtt{M}_{i+18} \oplus \mathtt{M}_{i+17} \oplus \mathtt{M}_{i+33} \oplus \mathtt{M}_{i+49} \\
\mathtt{M}'_{i+2} &= \mathtt{M}_{i+3} \oplus \mathtt{M}_{i+19} \oplus \mathtt{M}_{i+18} \oplus \mathtt{M}_{i+34} \oplus \mathtt{M}_{i+50} \\
\mathtt{M}'_{i+3} &= \mathtt{M}_{i+4} \oplus \mathtt{M}_{i+20} \oplus \mathtt{M}_{i+19} \oplus \mathtt{M}_{i+35} \oplus \mathtt{M}_{i+51} \oplus (\mathtt{M}_i \oplus \mathtt{M}_{i+16}) \\
\mathtt{M}'_{i+4} &= \mathtt{M}_{i+5} \oplus \mathtt{M}_{i+21} \oplus \mathtt{M}_{i+20} \oplus \mathtt{M}_{i+36} \oplus \mathtt{M}_{i+52} \oplus (\mathtt{M}_i \oplus \mathtt{M}_{i+16}) \\
\mathtt{M}'_{i+5} &= \mathtt{M}_{i+6} \oplus \mathtt{M}_{i+22} \oplus \mathtt{M}_{i+21} \oplus \mathtt{M}_{i+37} \oplus \mathtt{M}_{i+53} \\
\mathtt{M}'_{i+6} &= \mathtt{M}_{i+7} \oplus \mathtt{M}_{i+23} \oplus \mathtt{M}_{i+22} \oplus \mathtt{M}_{i+38} \oplus \mathtt{M}_{i+54} \oplus (\mathtt{M}_i \oplus \mathtt{M}_{i+16}) \\
\mathtt{M}'_{i+7} &= \mathtt{M}_i \oplus \mathtt{M}_{i+16} \oplus \mathtt{M}_{i+23} \oplus \mathtt{M}_{i+39} \oplus \mathtt{M}_{i+55}
\end{aligned}
$$

pour $i \in \{0, 8, 16, 24, 32, 40, 48, 56\}$ et où tous les indices doivent être considérés modulo 64.

	ligne 0					
	colonne 0			colonne 2		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_0$	b_0^0	$\cdots$	b_0^{15}	b_{64}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_7$	b_7^0	$\cdots$	b_0^{15}	b_{71}^0	$\cdots$	b_0^{15}

	ligne 0					
	colonne 1			colonne 3		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_8$	b_{32}^0	$\cdots$	b_0^{15}	b_{96}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{15}$	b_{39}^0	$\cdots$	b_0^{15}	b_{103}^0	$\cdots$	b_0^{15}

	ligne 1					
	colonne 0			colonne 2		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_{16}$	b_8^0	$\cdots$	b_0^{15}	b_{72}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{23}$	b_{15}^0	$\cdots$	b_0^{15}	b_{79}^0	$\cdots$	b_0^{15}

	ligne 1					
	colonne 1			colonne 3		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_{24}$	b_{40}^0	$\cdots$	b_0^{15}	b_{104}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{31}$	b_{47}^0	$\cdots$	b_0^{15}	b_{111}^0	$\cdots$	b_0^{15}

	ligne 2					
	colonne 0			colonne 2		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_{32}$	b_{16}^0	$\cdots$	b_0^{15}	b_{80}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{39}$	b_{23}^0	$\cdots$	b_0^{15}	b_{87}^0	$\cdots$	b_0^{15}

	ligne 2					
	colonne 1			colonne 3		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_{40}$	b_{48}^0	$\cdots$	b_0^{15}	b_{112}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{47}$	b_{55}^0	$\cdots$	b_0^{15}	b_{119}^0	$\cdots$	b_0^{15}

	ligne 3					
	colonne 0			colonne 2		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_{48}$	b_{24}^0	$\cdots$	b_0^{15}	b_{88}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{55}$	b_{31}^0	$\cdots$	b_0^{15}	b_{95}^0	$\cdots$	b_0^{15}

	ligne 3					
	colonne 1			colonne 3		
	bloc 0	$\cdots$	bloc 15	bloc 0	$\cdots$	bloc 15
$\mathtt{M}_{56}$	b_{56}^0	$\cdots$	b_0^{15}	b_{120}^0	$\cdots$	b_0^{15}
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\mathtt{M}_{63}$	b_{63}^0	$\cdots$	b_0^{15}	b_{127}^0	$\cdots$	b_0^{15}

FIGURE 8.5 – Notre représentation 2-*slicing* de 16 blocs dans 64 mots machines ($\mathtt{M}_i, 0 \leq i \leq 63$) d'une architecture 32 bits. b_i^j désigne le i-ième bit du j-ième bloc.

8.3 Évaluation des performances et coûts

8.3.1 Performance logiciel

Le tableau 8.1 présente les résultats d'implémentations de AES masquées aux ordres $d \in \{1, 3, 7, 15, 31\}$ sans ISE, avec l'ISE1 et avec l'ISE2. Les implémentations masquées avec l'ISE2 permettent de réduire le nombre de blocs chiffrés en parallèle par rapport à l'ISE1. Pour chaque ordre $d \in \{1, 3, 7, 15, 31\}$, nous avons une solution pour chiffrer un seul bloc à la fois, permettant l'utilisation de tous les modes de chiffrement. En termes de temps d'exécution, l'ISE2 impose un petit surcoût par rapport à l'ISE1. Par exemple à l'ordre 1, l'ISE2 induit une augmentation du temps d'exécution de 1,84 par rapport à l'ISE1. Néanmoins, le gain de temps reste important comparé à la solution SW-M-BS masquée sans ISE.

d	Implém.	Nb blocs chiffrés en parallèle	Temps		
			Cycles	cmp1	cmp2
0	SW-TTABLE	1	1266	base	n.a.
1	SW-M-BS	32	26001	×21	base
	HW-M-ISE1	16	5483	×4,3	/4,7
	HW-M-ISE2	8	8622	×6,8	/3,0
		4	8720	×6,9	/3,0
		2	7992	×6,3	/3,3
		1	10099	×8,0	/2,6
3	SW-M-BS	32	98575	×78	base
	HW-M-ISE1	8	10927	×8,6	/9,0
	HW-M-ISE2	4	17308	×13,7	/5,7
		2	17411	×13,8	/5,7
		1	15842	×12,5	/6,2
7	SW-M-BS	32	300414	×237	base
	HW-M-ISE1	4	22033	×17,4	/13,6
	HW-M-ISE2	2	34573	×27,3	/8,7
		1	34735	×27,4	/8,6
15	SW-M-BS	32	1041285	×822	base
	HW-M-ISE1	2	43677	×34,5	/23,8
	HW-M-ISE2	1	69001	×54,5	/15,1
31	SW-M-BS	32	3716920	×2936	base
	HW-M-ISE1	1	87489	×69,1	/42,5

TABLE 8.1 – Résultats d'implantations de l'ISE2 masquée aux ordres $d \in \{1, 2, 3, 7, 15, 31\}$. Le temps d'exécution est donné en nombre de cycles pour chiffrer un bloc avec AES. cmp1 et cmp2 comparent respectivement le temps d'exécution par rapport à SW-TTABLE et SW-M-BS.

8.3.2 Implémentation sur FPGA

Le tableau 8.2 présente les résultats de l'implémentation FPGA du cœur CV32E40P modifié pour prendre en charge l'ISE2. En termes de surface, l'ISE2 impose une très faible augmen-

tation comparée à l'ISE1. Par exemple, le nombre de LUT du cœur CV32E40P avec l'ISE2 augmente seulement d'un facteur 1,02 par rapport à l'ISE1. De plus, la période minimale de fonctionnement n'est pas impactée car le chemin critique n'est pas dans l'ISE.

d	ISE	Surface & période				
		FF	cmp	LUT	cmp	P (ns)
0	Sans ISE	2141	base	4782	base	14,6
1	ISE1	2412	×1,13	4794	×1,00	14,4
	ISE2	2417	×1,13	4912	×1,03	15.3
3	ISE1	2590	×1,21	4977	×1,04	14,9
	ISE2	2595	×1,21	5118	×1,07	14.9
7	ISE1	3146	×1,47	5596	×1,17	14,9
	ISE2	3151	×1,47	5721	×1,20	14.9
15	ISE1	4058	×1,90	6707	×1,40	14,7
	ISE2	4063	×1,90	6657	×1,39	14.3

TABLE 8.2 – Résultats d'implantation FPGA du cœur CV32E40P avec l'ISE2.

Chapitre 9

Extension masquée et sécurisée dans le modèle de sondage par régions

Une implémentation masquée utilisant les ISE précédentes est sécurisée contre un attaquant pouvant sonder d bits dans le circuit pendant l'exécution complète. Pour des implémentations de grande taille, le nombre de bits sondés est cependant faible par rapport au nombre de bits manipulés dans toute l'implémentation. Ce faible taux de fuite peut réduire la sécurité obtenue dans le modèle de fuite bruitée (voir sous-section 3.3.4). Pour améliorer le taux de fuite admis, nous proposons une nouvelle ISE, que nous nommons ISE3, qui modifie l'ISE1 pour obtenir une sécurité dans le modèle de sondage par régions, où chaque région comprend une instruction de l'ISE3. L'ISE3 masquée à l'ordre d permet alors à l'attaquant de sonder $d/2$ bits par instruction sans compromettre la sécurité du schéma de masquage.

Dans la section 9.1, nous décrivons les instructions de l'ISE3. La section 9.2 présente la mise en œuvre de l'ISE3 ainsi que son implantation dans le cœur CV32E40P. Dans la section 9.3, nous évaluons les performances et la surface de l'ISE3 sur FPGA.

9.1 Spécification de l'ISE3

Pour sécuriser contre un attaquant pouvant sonder $d/2$ bits par instruction, nous proposons d'implémenter des instructions masquées à l'ordre d comme décrit dans la figure 9.1. Le résultat de l'instruction masquée est obtenue par une opération d-NI dans le modèle de sondage robuste aux glitchs. Pour obtenir une isolation entre les différentes instructions, un circuit de rafraîchissement d-SNI est appliqué aux entrées de l'opération d-NI. Pour maintenir la sécurité dans notre modèle de sondage par régions, les registres d'indice `rs1` et `rs2` sont également rafraîchis avec des circuits de rafraîchissement d-SNI dans le modèle de sondage robuste aux glitchs. Sans rafraîchissement des registres d'indices `rs1` et `rs2`, un attaquant pourrait sonder $d/2$ bits à chaque utilisation d'un registre, réduisant ainsi le niveau de sécurité. Ces instructions masquées à l'ordre d permettent alors d'obtenir la sécurité contre un attaquant sondant $d/2$ bits par instruction :

Proposition 1. *Si un programme utilise uniquement des instructions masquées à l'ordre d implémentées comme dans la figure 9.1, alors le programme est sécurisé contre un attaquant sondant $d/2$ bits par instruction dans le modèle de sondage robuste aux glitchs.*

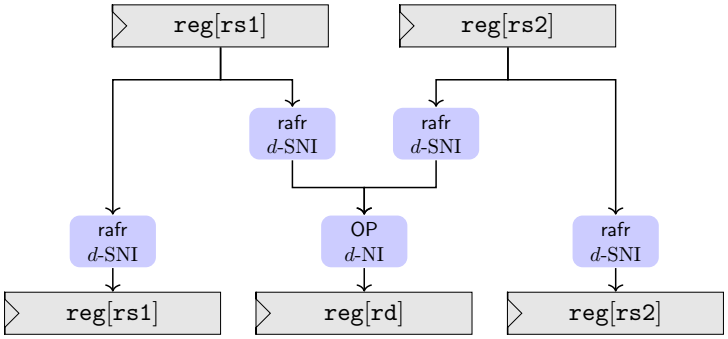


FIGURE 9.1 – Schéma d'une instruction masquée à l'ordre d et sécurisée dans le modèle de sondage par régions avec $d/2$ sonde par instruction. `reg[x]` désigne le registre d'indice `x` dans le banc de registres.

Démonstration. Considérons l'instruction masquée de la figure 9.2 dans laquelle les ensembles $\mathcal{O}_i$ désignent les sondes de sortie et $\mathcal{I}_i$ les sondes internes.

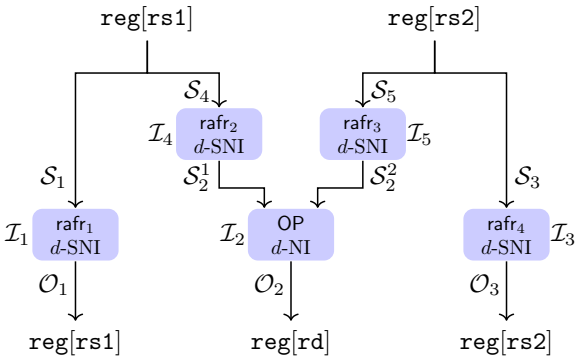


FIGURE 9.2 – Illustration d'une preuve de sécurité dans le modèle de sondage par régions.

Montrons dans un premier temps que si les sondes de sortie $\mathcal{O}_i, 1 \leq i \leq 3$ vérifient $|\mathcal{O}_i| \leq d/2$ pour tout i, et les sondes internes $\mathcal{I}_i, 1 \leq i \leq 5$ vérifient $\sum_i |\mathcal{I}_i| \leq d/2$, alors, les sondes de sortie et internes peuvent être simulées par un ensemble de sondes $\mathcal{A}$ de `reg[rs1]` vérifiant $|\mathcal{A}| \leq d/2$ et un ensemble de sondes $\mathcal{B}$ de `reg[rs2]` vérifiant $|\mathcal{B}| \leq d/2$. Pour cela, traitons chaque porte une par une :

- Comme $|\mathcal{O}_1| + |\mathcal{I}_1| \leq d/2 + d/2 \leq d$ et rafr_1 est d-SNI, alors les ensembles $\mathcal{O}_1$ et $\mathcal{I}_1$ peuvent être simulés avec l'ensemble $\mathcal{S}_1$ vérifiant $|\mathcal{S}_1| \leq |\mathcal{I}_1|$.
- De même, comme $|\mathcal{O}_3| + |\mathcal{I}_3| \leq d$ et rafr_4 est d-SNI, alors les ensembles $\mathcal{O}_3$ et $\mathcal{I}_3$ peuvent être simulés avec l'ensemble $\mathcal{S}_3$ vérifiant $|\mathcal{S}_3| \leq |\mathcal{I}_3|$.
- Comme $|\mathcal{O}_2| + |\mathcal{I}_2| \leq d/2 + d/2 \leq d$ et l'opération est d-NI, alors les ensembles $\mathcal{O}_2$ et $\mathcal{I}_2$ peuvent être simulés avec les ensembles $\mathcal{S}_2^1$ et $\mathcal{S}_2^2$ vérifiant $|\mathcal{S}_2^1| \leq |\mathcal{O}_2| + |\mathcal{I}_2|$ et $|\mathcal{S}_2^2| \leq |\mathcal{O}_2| + |\mathcal{I}_2|$.
- Comme $|\mathcal{S}_2^1| + |\mathcal{I}_4| \leq |\mathcal{O}_2| + |\mathcal{I}_2| + |\mathcal{I}_4| \leq d$ et rafr_2 est d-SNI, alors les ensembles $\mathcal{S}_2^1$ et $\mathcal{I}_4$ peuvent être simulés avec l'ensemble $\mathcal{S}_4$ vérifiant $|\mathcal{S}_4| \leq |\mathcal{I}_4|$.
- De même, comme $|\mathcal{S}_2^2| + |\mathcal{I}_5| \leq |\mathcal{O}_2| + |\mathcal{I}_2| + |\mathcal{I}_5| < d$ et rafr_3 est d-SNI, alors les ensembles $\mathcal{S}_2^2$ et $\mathcal{I}_5$ peuvent être simulés avec l'ensemble $\mathcal{S}_5$ vérifiant $|\mathcal{S}_5| \leq |\mathcal{I}_5|$.

L'ensemble des sondes peut donc être simulé avec un ensemble de parts $\mathcal{A} = \mathcal{S}_1 \cup \mathcal{S}_4$ de `reg[rs1]` et un ensemble de parts $\mathcal{B} = \mathcal{S}_3 \cup \mathcal{S}_5$ de `reg[rs2]`, vérifiant $|\mathcal{A}| \leq |\mathcal{S}_1| + |\mathcal{S}_4| \leq |\mathcal{I}_1| + |\mathcal{I}_4| \leq d/2$

et $|\mathcal{B}| \leq |\mathcal{S}_3| + |\mathcal{S}_5| \leq |\mathcal{I}_3| + |\mathcal{I}_5| \leq d/2$.

En considérant maintenant le programme comme un graphe dans lequel les différentes instructions sont des sous-graphes, nous pouvons simuler $d/2$ sondes par instruction à partir de $d/2$ parts de chaque masquage d'entrée en appliquant récursivement la propriété démontrée ci-dessus. □

Nous proposons une ISE dédiée au masquage, nommée ISE3, implémentant les instructions de l'ISE1 comme dans la figure 9.1 de façon à être sécurisées contre un attaquant pouvant sonder $d/2$ bits par instruction. Les instructions de l'ISE3 sont décrites dans la figure 9.3. mand, mxor et mnot désignent respectivement des circuits masqués d-NI dans le modèle de sondage robuste aux glitchs des portes booléennes AND, XOR, NOT. rafr désigne un circuit de rafraîchissement d-SNI dans le modèle de sondage robuste aux glitchs.

```
def ise_and(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        a=rafr(reg[rs1][i*n:(i+1)*n])
        b=rafr(reg[rs2][i*n:(i+1)*n])
        reg[rd][i*n:(i+1)*n]=mand(a,b)
        reg[rs1][i*n:(i+1)*n]=rafr(reg[rs1][i*n:(i+1)*n])
        reg[rs2][i*n:(i+1)*n]=rafr(reg[rs2][i*n:(i+1)*n])

def ise_xor(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        a=rafr(reg[rs1][i*n:(i+1)*n])
        b=rafr(reg[rs2][i*n:(i+1)*n])
        reg[rd][i*n:(i+1)*n]=mxor(a,b)
        reg[rs1][i*n:(i+1)*n]=rafr(reg[rs1][i*n:(i+1)*n])
        reg[rs2][i*n:(i+1)*n]=rafr(reg[rs2][i*n:(i+1)*n])

def ise_not(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        a=rafr(reg[rs1][i*n:(i+1)*n])
        b=rafr(reg[rs2][i*n:(i+1)*n])
        reg[rd][i*n:(i+1)*n]=mnot(a,b)
        reg[rs1][i*n:(i+1)*n]=rafr(reg[rs1][i*n:(i+1)*n])
        reg[rs2][i*n:(i+1)*n]=rafr(reg[rs2][i*n:(i+1)*n])

def ise_or(rs1, rs2, rd):
    n=d+1
    for i in range(32//n):
        a=rafr(reg[rs1][i*n:(i+1)*n])
        b=rafr(reg[rs2][i*n:(i+1)*n])
        tmp1 = mnot(a)
        tmp2 = mnot(b)
        tmp3 = mand(tmp1,tmp2)
        reg[rd][i*n:(i+1)*n]=mnot(tmp3)
        reg[rs1][i*n:(i+1)*n]=rafr(reg[rs1][i*n:(i+1)*n])
        reg[rs2][i*n:(i+1)*n]=rafr(reg[rs2][i*n:(i+1)*n])
```

FIGURE 9.3 – Code Python décrivant les nouvelles instructions de l'ISE3 masquée à l'ordre $d \in \{1, \cdots, 31\}$. mand, mxor et mnot désignent respectivement des circuits masqués d-NI des portes booléennes AND, XOR, NOT. rafr désigne un circuit de rafraîchissement d-SNI. reg[x] désigne le registre d'indice x dans le banc de registres.

9.2 Implémentation de l'ISE3 sur le CV32E40P

Dans cette section, nous décrivons l'implémentation de l'ISE3 sur le cœur CV32E40P.

9.2.1 ALU masquée proposée

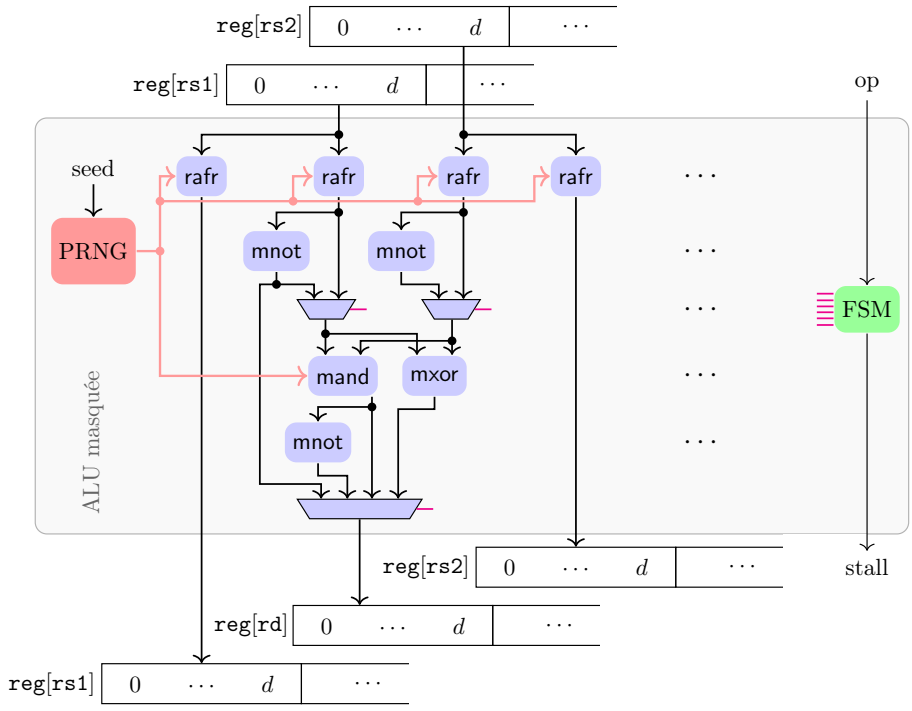


FIGURE 9.4 – Architecture de l'ALU masquée de l'ISE3.

L'ALU masquée décrite dans la figure 9.4 a été développée pour implémenter nos nouvelles instructions. Les portes masquées mand sont implémentées avec le AND masqué d-NI de [Fau+18] alors que les portes masquées mxor et mnot sont implémentées de façon directe. Les registres d'entrée sont préalablement rafraîchis avec un circuit de rafraîchissement d-SNI basé sur la porte AND masquée de [Fau+18]. L'ALU masquée génère également en sortie deux valeurs de 32 bits contenant les valeurs des registres `reg[rs1]` et `reg[rs2]` rafraîchies avec d'autres circuits de rafraîchissement d-SNI. Le tableau 9.1 décrit les caractéristiques des instructions masquées de l'ISE3.

Instructions	Latence	Aléas
`ise3.xor`	2	$m \times 4 \times \frac{d(d+1)}{2}$
`ise3.not`	2	$m \times 4 \times \frac{d(d+1)}{2}$
`ise3.and`	3	$m \times 4 \times \frac{d(d+1)}{2}$
`ise3.or`	3	$m \times 4 \times \frac{d(d+1)}{2}$

TABLE 9.1 – Caractéristiques des instructions masquées de l'ISE3. m est égal à rnd $\left(\frac{32}{d+1}\right)$. La latence est exprimée en nombre de cycles et les aléas en nombre de bits par instruction.

9.2.2 Intégration dans le cœur CV32E40P

L'ALU masquée a ensuite été intégrée au cœur CV32E40P, comme illustré sur la figure 9.5. Le décodeur d'instructions modifié traite les nouvelles instructions et génère des signaux de contrôle pour les instructions multi-cycles. Comme pour l'ISE1, nous avons conçu un chemin de données séparé pour l'ALU masquée afin d'éviter les problèmes de recombinaison des parts. La mise en place du chemin de données totalement séparé nécessite d'étendre le registre de pipeline ID/EX avec 2 nouveaux registres 32 bits accessibles uniquement depuis l'ALU masquée. À la différence de l'ISE1, nous avons ajouté deux nouveaux ports d'écriture au fichier de registres pour actualiser à chaque instruction les registres d'indices `rs1` et `rs2` avec leurs valeurs rafraîchies.

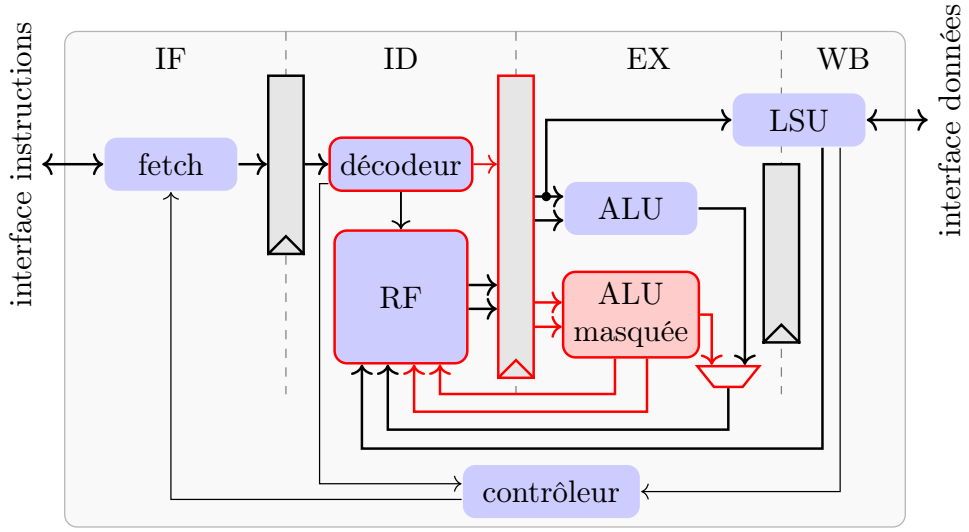


FIGURE 9.5 – Schéma simplifié du cœur modifié avec l'ISE3. Les parties rouges ont été ajoutées ou modifiées pour notre ISE.

9.3 Évaluation des performances et coûts

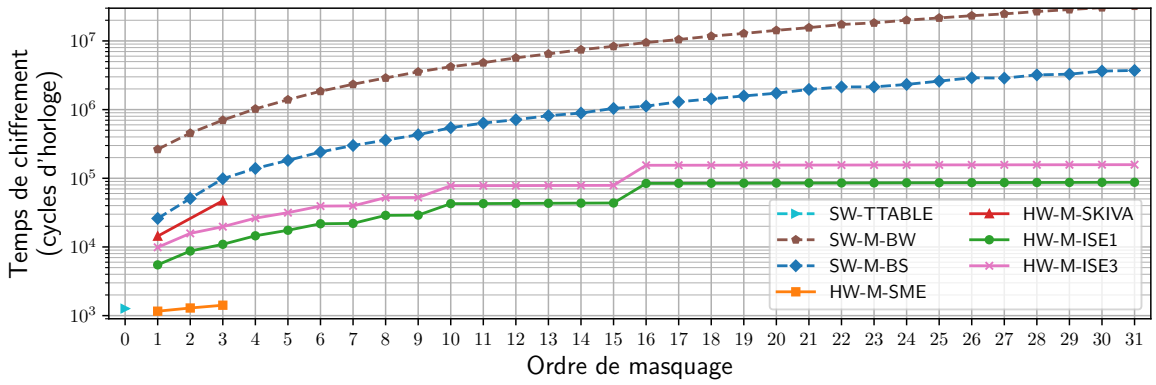


FIGURE 9.6 – Temps de chiffrement (en échelle log) pour un bloc de AES en utilisant l'ISE3 et diverses solutions de masquage logiciel et matériel.

La figure 9.6 présente le temps d'exécution du chiffrement AES masqué en utilisant l'ISE3 (labellisé par HW-M-ISE3), ainsi que différentes solutions de la littérature que nous avons réimplémenté sur le cœur CV32E40P (voir section 6.3). HW-M-ISE3 surpasse l'implémentation BS masquée en logiciel SW-M-BS. L'ISE3 induit un petit surcoût en termes de performance par rapport à l'ISE1, mais augmente la sécurité en permettant à un attaquant de sonder $d/2$ bits par instruction.

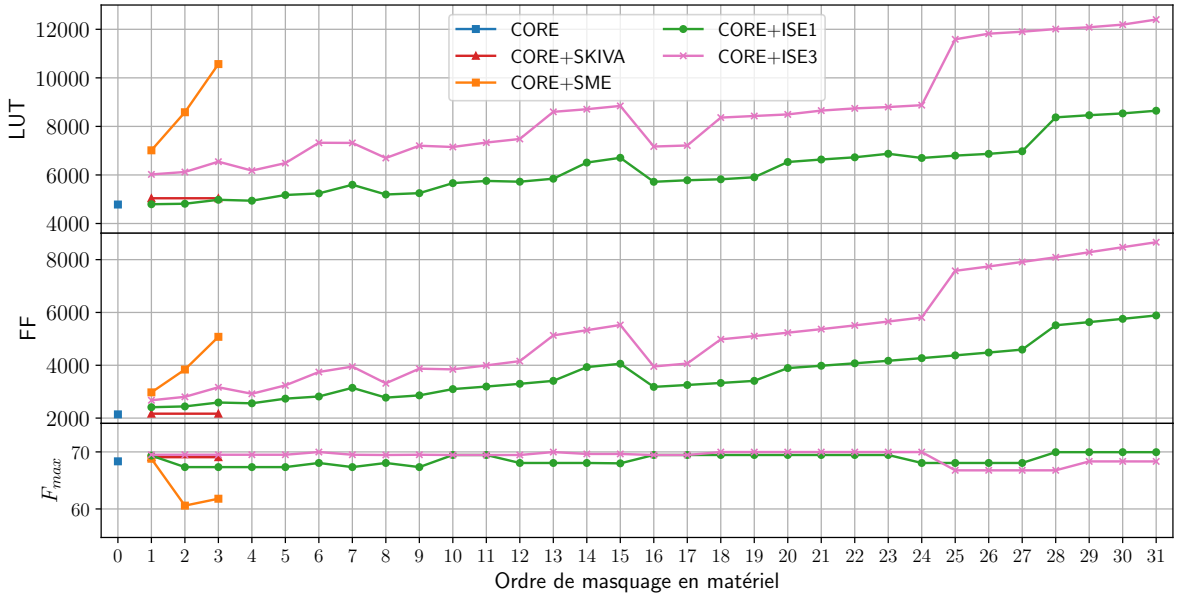


FIGURE 9.7 – Surface et fréquence maximale du cœur CV32E40P (CORE) sans ISE, avec Skiva (CORE+SKIVA), avec SME (CORE+SME), avec la version HPC3 de ISE1 (CORE+ISE1) et avec l'ISE3 (CORE+ISE3).

La figure 9.7 présente la surface et la fréquence maximale du cœur CV32E40P sans ISE (CORE), avec Skiva (CORE+SKIVA), avec SME (CORE+SME), avec la version HPC3 de l'ISE1 (CORE+ISE1) et avec l'ISE3 (CORE+ISE3), pour des ordres de masquage d dans $\{1, 2, 3, \ldots 31\}$. L'ISE3 masquée aux ordres 1, 2 et 3 est bien plus petite que SME. On constate cependant que l'ISE3 a un impact plus important en surface que l'ISE1. Cela s'explique par l'ajout de deux nouveaux ports d'écriture au fichier de registres.

Chapitre 10

Solution matérielle/logicielle flexible pour masquer à des ordres élevés

Pour maintenir la sécurité à long terme, il peut être essentiel de pouvoir adapter le niveau de sécurité de la protection de masquage. Contrairement à la mise en œuvre logicielle du masquage, les ISE masquées de l'état de l'art sont limitées aux petits ordres de masquage et ne sont pas assez flexibles pour adapter le niveau de sécurité sans changer le matériel (voir tableau 10.1). Nous proposons une solution de masquage *matériel et logiciel flexible*, que nous nommons HW-SW-ISE1, qui permet de masquer des implémentations BS à divers ordres élevés sans changer le matériel. Un premier niveau de masquage est implémenté en matériel en utilisant l'ISE1 (voir chapitre. 7) dont l'ordre de masquage d_{H} est fixé *au moment de la synthèse*. Un masquage logiciel, dont l'ordre d_{S} est fixé au *moment de la compilation logicielle*, utilise ensuite les instructions masquées de l'ISE1 pour atteindre divers ordres supérieurs. Notre solution matérielle/logicielle permet divers compromis entre le coût matériel, le temps d'exécution et la sécurité.

Référence	RISC-V	Ordre de masquage	Flexibilité au moment de la synthèse	Flexibilité au moment de la compilation
[Gro+16]	✓	$\{1,2,3,4\}$	✓	✗
[DGH19]	✓	1	✗	✗
[Gao+21]	✓	1	✗	✗
SKIVA [Kia+21]	✗	$\{1,3\}$	✗	✓
SME [MP21]	✓	$\{1,2,3\}$	✓	✗
HW-SW-ISE1	✓	$(d_{\mathrm{S}}+1)(d_{\mathrm{H}}+1)-1$, $d_{\mathrm{H}} \in \mathbb{N}$, $d_{\mathrm{S}} \in \mathbb{N}$	✓	✓

TABLE 10.1 – Caractéristiques des ISE masquées de l'état de l'art et de notre solution HW-SW-ISE1.

Dans la section 10.1, nous décrivons notre solution de masquage matériel/logiciel. Dans la section 10.2, nous évaluons les performances et la surface de notre solution sur FPGA.

10.1 Solution de masquage matériel/logiciel

10.1.1 Présentation de notre solution matérielle/logicielle

Notre solution de masquage Nous proposons une solution de masquage *matériel et logiciel*, que nous nommons HW-SW-ISE1, permettant de masquer des implémentations BS à des ordres de masquage beaucoup plus élevés avec une grande flexibilité au moment de la *synthèse* et au moment de la *compilation logicielle* pour divers compromis coût/performance. Un premier niveau de masquage est implémenté en matériel en utilisant l'ISE1 (voir chapitre 7) dont l'ordre de masquage matériel d_{H} est fixé au moment de la synthèse. Un masquage logiciel, dont l'ordre d_{S} est fixé au moment de la compilation logicielle, utilise ensuite de façon sécurisée les instructions masquées de l'ISE1 pour obtenir un ordre de masquage total plus élevé. L'ordre de masquage total obtenu est $d = (d_{\mathrm{S}} + 1)(d_{\mathrm{H}} + 1) - 1$.

Notre représentation masquée Notre solution de masquage matériel/logiciel utilise une nouvelle représentation masquée qui est un compromis entre le BS masqué et le «*share slicing*». Les bits sensibles sont d'abord masqués à l'ordre d_{S} en disposant les parts dans différents registres comme pour le BS masqué, puis chaque part est à nouveau masquée à l'ordre d_{H} en disposant les parts dans un même registre comme pour le «*share slicing*». Chaque bit est alors masqué en $(d_{\mathrm{S}} + 1)(d_{\mathrm{H}} + 1)$ parts et l'ordre de masquage obtenu est $d = (d_{\mathrm{S}} + 1)(d_{\mathrm{H}} + 1) - 1$. La figure 10.1 illustre notre représentation masquée avec $d_{\mathrm{H}} = 1$ et $d_{\mathrm{S}} = 2$.

	Bloc 0		Bloc 1		$\cdots$	Bloc 15	
M_0	$\hat{\mathbf{b}}_0^0[0][0]$	$\hat{\mathbf{b}}_0^0[1][1]$	$\hat{\mathbf{b}}_0^1[0][0]$	$\hat{\mathbf{b}}_0^1[1][1]$	$\cdots$	$\hat{\mathbf{b}}_0^{15}[0][0]$	$\hat{\mathbf{b}}_0^{15}[1][1]$
M_1	$\hat{\mathbf{b}}_0^0[1][0]$	$\hat{\mathbf{b}}_0^0[1][1]$	$\hat{\mathbf{b}}_0^1[1][0]$	$\hat{\mathbf{b}}_0^1[1][1]$	$\cdots$	$\hat{\mathbf{b}}_0^{15}[1][0]$	$\hat{\mathbf{b}}_0^{15}[1][1]$
M_2	$\hat{\mathbf{b}}_0^0[2][0]$	$\hat{\mathbf{b}}_0^0[2][1]$	$\hat{\mathbf{b}}_0^1[2][0]$	$\hat{\mathbf{b}}_0^1[2][1]$	$\cdots$	$\hat{\mathbf{b}}_0^{15}[2][0]$	$\hat{\mathbf{b}}_0^{15}[2][1]$
$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$	
M_{381}	$\hat{\mathbf{b}}_{127}^0[0][0]$	$\hat{\mathbf{b}}_{127}^0[0][1]$	$\hat{\mathbf{b}}_{127}^1[00[0]$	$\hat{\mathbf{b}}_{127}^1[0][1]$	$\cdots$	$\hat{\mathbf{b}}_{127}^{15}[0][0]$	$\hat{\mathbf{b}}_{127}^{15}[0][1]$
M_{382}	$\hat{\mathbf{b}}_{127}^0[1][0]$	$\hat{\mathbf{b}}_{127}^0[1][1]$	$\hat{\mathbf{b}}_{127}^1[1][0]$	$\hat{\mathbf{b}}_{127}^1[1][1]$	$\cdots$	$\hat{\mathbf{b}}_{127}^{15}[1][0]$	$\hat{\mathbf{b}}_{127}^{15}[1][1]$
M_{383}	$\hat{\mathbf{b}}_{127}^0[2][0]$	$\hat{\mathbf{b}}_{127}^0[2][1]$	$\hat{\mathbf{b}}_{127}^1[2][0]$	$\hat{\mathbf{b}}_{127}^1[2][1]$	$\cdots$	$\hat{\mathbf{b}}_{127}^{15}[2][0]$	$\hat{\mathbf{b}}_{127}^{15}[2][1]$

FIGURE 10.1 – Notre représentation masquée avec $d_{\mathrm{H}} = 1$ et $d_{\mathrm{S}} = 2$ sur une architecture 32 bits. $\hat{\mathbf{b}}_i^j$ désigne un masquage du i-ième bit du j-ième bloc. $\hat{\mathbf{b}}_i^j[k][l]$ désigne la l-ième part matérielle de la k-ième part logicielle du masquage $\hat{\mathbf{b}}_i^j$.

Flexibilité de notre solution Le niveau de sécurité de notre solution de masquage peut être adapté au moment de la compilation logicielle en modifiant l'ordre de masquage logiciel d_{S}. La flexibilité de notre solution permet également divers compromis coût/performance. Par exemple, pour des applications à surface limitée, une ISE masquée à l'ordre 1 peut être implémentée en matériel et utilisée en logiciel pour différentes valeurs de d_{S}, conduisant à un ordre de masquage total $d \in \{1, 3, 5, 7, 9, 11, \ldots\}$. Pour des applications de haute sécurité, une ISE masquée à un ordre plus élevé, comme par exemple 5, peut être implémentée en matériel puis utilisée en logiciel pour différentes valeurs de d_{S}, conduisant à un ordre de masquage total $d \in \{5, 11, 17, 23, 29, \ldots\}$.

10.1.2 Masquage logiciel/matériel avec la version NI-SNI de l'ISE1

Nous décrivons une première solution de masquage logiciel/matériel utilisant de façon sécurisée les instructions de la version NI-SNI de l'ISE1 masquée à l'ordre d_{H}. Pour manipuler des variables masquées avec notre représentation, nous avons développé des implémentations masquées à l'ordre $d = (d_{\mathrm{S}}+1)(d_{\mathrm{H}}+1)-1$ des portes booléennes XOR, NOT, AND, OR. Notre méthode consiste à prendre une implémentation masquée en logiciel à l'ordre d_{S} de la porte booléenne, puis de masquer à nouveau l'implémentation en matériel à l'ordre d_{H} en utilisant les instructions masquées de l'ISE1. Nous obtenons ainsi des implémentations d-NI des portes XOR, NOT et des implémentations d-SNI des portes AND, OR. Les implémentations BS peuvent alors être masquées à l'ordre d en utilisant nos portes booléennes masquées dans le code généré par Tornado. Nous allons à présent décrire chaque porte booléenne masquée et prouver leur sécurité dans le modèle de sondage robuste aux glitchs.

Algorithme 10 Porte XOR masquée à l'ordre d_{S} utilisant les instructions de l'ISE1 masquée à l'ordre d_{H}.

Entrée: $\hat{\mathbf{x}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$ et $\hat{\mathbf{y}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$
Sortie: $\hat{\mathbf{z}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$
  pour $i = 0$ à d_{S} **faire**
    $\hat{\mathbf{z}}[i] \leftarrow \mathrm{ise1.xor}(\hat{\mathbf{x}}[i], \hat{\mathbf{y}}[i])$
  fin pour

L'algorithme 10 est une implémentation logicielle masquée à l'ordre d_{S} de la porte XOR dans laquelle l'instruction `xor` est remplacée par l'instruction `ise1.xor` de l'ISE1 masquée à l'ordre d_{H}. Cette implémentation masquée est alors d-NI dans le modèle de sondage robuste aux glitchs :

Proposition 2. *Si l'instruction* ise1.xor *est d_{H}-NI dans le modèle de sondage robuste aux glitchs, alors la porte* XOR *masquée en logiciel à l'ordre d_{S} de l'algorithme 10 est d-NI dans le modèle de sondage robuste aux glitchs, avec $d = (d_{\mathrm{S}}+1)(d_{\mathrm{H}}+1)-1$.*

Démonstration. Voir la section C.1 de l'annexe C. □

Algorithme 11 Porte NOT masquée à l'ordre d_{S} utilisant les instructions de l'ISE1 masquée à l'ordre d_{H}.

Entrée: $\hat{\mathbf{x}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$
Sortie: $\hat{\mathbf{y}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$
  $\hat{\mathbf{y}}[0] \leftarrow \mathrm{ise1.not}(\hat{\mathbf{x}}[0])$
  pour $i = 0$ à d_{S} **faire**
    $\hat{\mathbf{y}}[i] \leftarrow \hat{\mathbf{x}}[i]$
  fin pour

L'algorithme 11 est une implémentation logicielle masquée à l'ordre d_{S} de la porte NOT dans laquelle l'instruction `not` est remplacée par l'instruction `ise1.not` de l'ISE1 masquée à l'ordre d_{H}. On peut montrer, de façon similaire à l'algorithme 10, que cette implémentation masquée est alors d-NI dans le modèle de sondage robuste aux glitchs, avec $d = (d_{\mathrm{S}}+1)(d_{\mathrm{H}}+1)-1$.

Algorithme 12 Porte AND masquée à l'ordre d_S et d_S-SNI utilisant les instructions de l'ISE1 masquée à l'ordre d_H et d_H-SNI.

Entrée: $\hat{\mathbf{x}} \in \mathrm{GF}(2)^{d_S+1}$ et $\hat{\mathbf{y}} \in \mathrm{GF}(2)^{d_S+1}$
Sortie: $\hat{\mathbf{z}} \in \mathrm{GF}(2)^{d_S+1}$
pour $i = 0$ à d_S **faire**
  $a_{i,i} \leftarrow \mathrm{ise1.and}(\hat{\mathbf{x}}[i], \hat{\mathbf{y}}[i])$
  $u_{i,i} \leftarrow a_{i,i}$
  pour $j = i + 1$ à d_S **faire**
    $r_{i,j} \xleftarrow{\$} \mathrm{GF}(32)$
    $a_{i,j} \leftarrow \mathrm{ise1.and}(\hat{\mathbf{x}}[i], \hat{\mathbf{y}}[j])$
    $u_{i,j} \leftarrow \mathrm{ise1.xor}(a_{i,j}, r_{i,j})$
    $a_{j,i} \leftarrow \mathrm{ise1.and}(\hat{\mathbf{x}}[j], \hat{\mathbf{y}}[i])$
    $u_{j,i} \leftarrow \mathrm{ise1.xor}(a_{j,i}, r_{i,j})$
  fin pour
fin pour
pour $i = 0$ à d_S **faire**
  $v_{i,0} \leftarrow u_{i,0}$
  pour $j = 1$ à d_S **faire**
    $v_{i,j} \leftarrow \mathrm{ise1.xor}(v_{i,j-1}, u_{i,j})$
  fin pour
  $\hat{\mathbf{z}}[i] \leftarrow v_{i,d_S}$
fin pour

L'algorithme 12 est une implémentation logicielle masquée à l'ordre d_S et d_S-SNI de la porte AND dans laquelle les instructions `xor` et `and` sont remplacées respectivement par les instructions `ise1.xor` et `ise1.and` de l'ISE1 masquée à l'ordre d_H. Cette implémentation masquée est alors d-SNI dans le modèle de sondage robuste aux glitchs.

Proposition 3. *Si l'instruction* ise1.xor *est d_H-NI et* ise1.and *est d_H-SNI dans le modèle de sondage robuste aux glitchs, alors la porte* AND *masquée à l'ordre d_S de l'algorithme 12 est d-SNI dans le modèle de sondage robuste aux glitchs, avec $d = (d_S + 1)(d_H + 1) - 1$.*

Démonstration. Voir la section C.2 de l'annexe C. □

Pour implémenter la porte OR masquée à l'ordre d, nous utilisons les portes masquées NOT et AND conformément aux lois de de Morgan.

10.1.3 Masquage logiciel/matériel avec une version PINI de l'ISE1

La solution de masquage logiciel/matériel de la sous-section précédente permet d'obtenir des implémentations NI et SNI de portes booléennes. Cette solution nécessite cependant d'insérer des codes de rafraîchissement à certains endroits bien choisis pour maintenir la sécurité de séquences d'instructions. Bien qu'il existe des outils pour le faire tel que Tornado, cela peut s'avérer compliqué et les codes de rafraîchissement insérés peuvent impacter les performances. Nous proposons une seconde solution de masquage logiciel/matériel utilisant de façon sécurisée les instructions des versions PINI de l'ISE1 masquée à l'ordre d_H. Nous avons développé des implémentations masquées à l'ordre $d = (d_S + 1)(d_H + 1) - 1$ des portes XOR, NOT, AND, OR

qui vérifient également la propriété de composition d-PINI. Grâce à la propriété de composition d-PINI, l'insertion de codes de rafraîchissement n'est plus nécessaire. Nous allons à présent décrire chaque porte booléenne masquée et prouver leur sécurité dans le modèle de sondage robuste aux glitchs.

Tout d'abord, l'algorithme 10 est d-PINI dans le modèle de sondage robuste aux glitchs si les instructions `ise1.xor` sont d_{H}-PINI dans le modèle de sondage robuste aux glitchs :

Proposition 4. *Si l'instruction* ise1.xor *est d_{H}-PINI dans le modèle de sondage robuste aux glitchs, alors la porte* XOR *masquée en logiciel à l'ordre d_{S} de l'algorithme 10 est d-PINI dans le modèle de sondage robuste aux glitchs, avec $d = (d_{\mathrm{S}} + 1)(d_{\mathrm{H}} + 1) - 1$.*

Démonstration. Voir la section C.3 de l'annexe C. □

De façon simialaire, on peut monter que l'algorithme 11 est d-PINI dans le modèle de sondage robuste aux glitchs si l'instruction `ise1.not` est d_{H}-PINI dans le modèle de sondage robuste aux glitchs.

Algorithme 13 Porte AND masquée à l'ordre d_{S} utilisant les instructions de l'ISE1 masquée à l'ordre d_{H} et d_{H}-PINI.

Entrée: $\hat{\mathbf{x}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$ et $\hat{\mathbf{y}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$
Sortie: $\hat{\mathbf{z}} \in \mathrm{GF}(2)^{d_{\mathrm{S}}+1}$

pour $i = 0$ à d_{S} **faire**
  $u_{i,i} \leftarrow \mathrm{ise1.and}(\hat{\mathbf{x}}[i], \hat{\mathbf{y}}[i])$
  pour $j = i + 1$ à d_{S} **faire**
    $r_{i,j} \xleftarrow{\$} \mathrm{GF}(32)$
    $a_{i,j} \leftarrow \mathrm{ise1.xor}(\hat{\mathbf{y}}[j], r_{i,j})$
    $b_{i,j} \leftarrow \mathrm{ise1.and}(\hat{\mathbf{x}}[i], a_{i,j})$
    $c_{i,j} \leftarrow \mathrm{ise1.not}(\hat{\mathbf{x}}[i])$
    $d_{i,j} \leftarrow \mathrm{ise1.and}(c_{i,j}, r_{i,j})$
    $u_{j,i} \leftarrow \mathrm{ise1.xor}(b_{i,j}, d_{i,j})$
    $a_{j,i} \leftarrow \mathrm{ise1.xor}(\hat{\mathbf{y}}[i], r_{i,j})$
    $b_{j,i} \leftarrow \mathrm{ise1.and}(\hat{\mathbf{x}}[j], a_{j,i})$
    $c_{j,i} \leftarrow \mathrm{ise1.not}(\hat{\mathbf{x}}[j])$
    $d_{j,i} \leftarrow \mathrm{ise1.and}(c_{j,i}, r_{i,j})$
    $u_{j,i} \leftarrow \mathrm{ise1.xor}(b_{j,i}, d_{j,i})$
  fin pour
fin pour
pour $i = 0$ à d_{S} **faire**
  $v_{i,0} \leftarrow u_{i,j}$
  pour $j = 1$ à d_{S} **faire**
    $v_{i,j} \leftarrow \mathrm{ise1.xor}(v_{i,j-1}, u_{i,j})$
  fin pour
  $\hat{\mathbf{z}}[i] \leftarrow v_{i,d_{\mathrm{S}}}$
fin pour

L'algorithme 13 est une implémentation logicielle masquée à l'ordre d_{S} et d_{S}-PINI de la porte AND dans laquelle les instructions `xor` et `and` sont remplacées respectivement par les ins-

tructions `ise1.xor` et `ise1.and` de l'ISE1 masquée à l'ordre d_H. Cette implémentation masquée est alors d-PINI dans le modèle de sondage robuste aux glitchs :

Proposition 5. *Si les instructions* ise1.xor *et* ise1.and *sont d_H-PINI dans le modèle de sondage robuste aux glitchs, alors la porte* AND *masquée à l'ordre d_S de l'algorithme 13 est d-PINI dans le modèle de sondage robuste aux glitchs, avec $d = (d_S + 1)(d_H + 1) - 1$.*

Démonstration. Voir la section C.4 de l'annexe C. □

Pour implémenter la porte OR masqué à l'ordre d, nous utilisons les portes masquées NOT et AND conformément aux lois de de Morgan.

10.2 Évaluation des performances et coûts

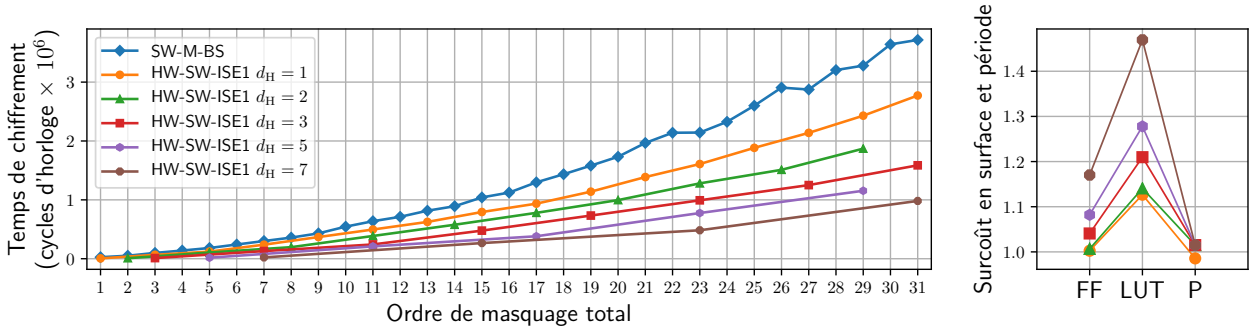


FIGURE 10.2 – Résultats d'implémentations du chiffrement AES avec notre solution de masquage matériel/logiciel utilisant la version HPC3 de l'ISE1 masquée aux ordres matériels $d_H \in \{1, 2, 3, 5, 7\}$ et pour différents ordres logiciels.

La figure 10.2 présente les résultats d'implémentations du chiffrement AES avec notre solution de masquage matériel/logiciel utilisant la version HPC3 de l'ISE1 masquée aux ordres matériels $d_H \in \{1, 2, 3, 5, 7\}$ et pour différents ordres logiciels. Notre solution de masquage matériel/logiciel permet divers compromis entre le coût matériel et le temps d'exécution. On constate en effet qu'augmenter l'ordre matériel de l'ISE1 permet de réduire le temps de chiffrement, mais augmente la surface du cœur. Les résultats d'implémentations présentés dans le tableau 10.2 illustrent les différents compromis possibles. Par exemple, notre solution matérielle/logicielle masquée aux ordres $(d_H = 1, d_S = 15)$, $(d_H = 3, d_S = 7)$, $(d_H = 7, d_S = 3)$, $(d_H = 15, d_S = 1)$ permet d'obtenir un ordre total $d = 31$. L'accélération obtenue par rapport à SW-M-BS est respectivement de 1,3 ; de 2,3 ; de 5,0 ; de 7,1 ; alors que le nombre de FF augmente respectivement de 1,13 ; de 1,21 ; de 1,47 ; de 1,90. Un autre bénéfice de notre solution de masquage matériel/logiciel est qu'elle permet de modifier l'ordre total via plusieurs codes compilés en continuant à utiliser l'ISE1 masquée à un ordre matériel fixé au moment de la synthèse. Par exemple, lorsque l'ISE1 est masquée à l'ordre $d_H = 3$, on peut obtenir un ordre total $d \in \{3, 7, 11, 15, 19, 23, 27, 31, \ldots\}$. On constate également que la solution utilisant la version NI-SNI de l'ISE1 est plus performante que la solution utilisant la version HPC3 de l'ISE1. Cela peut s'expliquer car l'algorithme 13 nécessite davantage d'opérations que l'algorithme 12.

Implémentation	Version ISE1	d	d_H	d_S	Temps			Surface	
					Cycles	cmp1	cmp2	FF	LUT
SW-TTABLE	Sans ISE	0	0	0	1266	base	n.a.	base	base
SW-M-BS	Sans ISE	3	0	3	98575	×78	base	×1,00	×1,00
HW-SW-ISE1	HPC3		1	1	66347	×52	/1,5	×1,13	×1,00
	NI-SNI				43307	×34	/2,3	×1,10	×1,02
	HPC3		3	0	10927	×8,6	/9,0	×1,21	×1,04
	NI-SNI				11567	×9,1	/8,5	×1,14	×1,03
SW-M-BS	Sans ISE	7	0	7	300414	×237	base	×1,00	×1,00
HW-SW-ISE1	HPC3		1	3	238560	×188	/1,3	×1,13	×1,00
	NI-SNI				224480	×177	/1,3	×1,10	×1,02
	HPC3		3	1	129688	×102	/2,3	×1,21	×1,04
	NI-SNI				85528	×67	/3,5	×1,14	×1,03
	HPC3		7	0	22033	×17,4	/13,6	×1,47	×1,17
	NI-SNI				23313	×18,4	/12,9	×1,27	×1,07
SW-M-BS	Sans ISE	15	0	15	1041285	×822	base	×1,00	×1,00
HW-SW-ISE1	HPC3		1	7	793304	×626	/1,3	×1,13	×1,00
	NI-SNI				724184	×572	/1,4	×1,10	×1,02
	HPC3		3	3	477371	×377	/2,2	×1,21	×1,04
	NI-SNI				449211	×354	/2,3	×1,14	×1,03
	HPC3		7	1	265732	×209	/3,9	×1,47	×1,17
	NI-SNI				173572	×137	/6,0	×1,27	×1,07
	HPC3		15	0	43677	×34,5	/23,8	×1,90	×1,40
	NI-SNI				46237	×36,5	/22,5	×1,48	×1,17
SW-M-BS	Sans ISE	31	0	31	3716920	×2936	base	×1,00	×1,00
HW-SW-ISE1	HPC3		1	15	2772567	×2190	/1,3	×1,13	×1,00
	NI-SNI				2470487	×1951	/1,5	×1,10	×1,02
	HPC3		3	7	1586573	×1253	/2,3	×1,21	×1,04
	NI-SNI				1448333	×1144	/2,6	×1,14	×1,03
	HPC3		7	3	742167	×586	/5,0	×1,47	×1,17
	NI-SNI				456468	×360	/8,1	×1,27	×1,07
	HPC3		15	1	526524	×415	/7,1	×1,90	×1,40
	NI-SNI				342204	×270	/10,7	×1,48	×1,17
	HPC3		31	0	87489	×69,1	/42,5	×2,75	×1,81
	NI-SNI				92609	×73,2	/40,1	×1,91	×1,36

TABLE 10.2 – Résultats d'implémentations du chiffrement AES avec notre solution matérielle/logicielle masquée aux ordres matériels $d_H \in \{1, 2, 3, 5, 7\}$ et pour différents ordres logiciels. cmp1 et cmp2 comparent respectivement le temps d'exécution par rapport à SW-TTABLE et SW-M-BS. Les nombres de FF et LUT sont donnés en surcoût par rapport à ceux du cœur CV32E40P de base.

Conclusion

Durant cette thèse, différentes *extensions de jeux d'instructions* (ISE) ont été développées pour masquer des implémentations «*bit slicing*» (BS) à des ordres élevés. Une première solution est donnée par l'ISE1 qui permet d'implémenter de façon sécurisée et efficace des implémentations BS masquées en utilisant la représentation «*share slicing*». L'ISE1 a été implémentée pour des *ordres de masquage élevés*, allant jusqu'à l'ordre 31. L'ordre de masquage de l'ISE1 est fixé au moment de la *synthèse*. Une grande *flexibilité* est ainsi possible au moment de la conception pour divers compromis coût/performance. Pour réduire le nombre de blocs indépendants chiffrés en parallèle, nous avons ensuite proposé l'ISE2 qui étend l'ISE1 pour masquer des implémentations *m-slicing* à des ordres $d \in \{1, 3, 7, 15\}$. De nouvelles implémentations *m-slicing* de AES ont été décrites permettant de chiffrer un bloc à la fois. Pour augmenter la sécurité apportée par le masquage, nous avons proposé l'ISE3 qui modifie l'ISE1 pour obtenir une sécurité dans le *modèle de sondage par régions*, où chaque région comprend une instruction de l'ISE3. L'ISE3 masquée à l'ordre d permet alors à l'attaquant de sonder $d/2$ bits par instruction sans compromettre la sécurité du schéma de masquage. Pour terminer, nous avons proposé une solution de *masquage matériel et logiciel flexible* qui permet de masquer des implémentations BS à *divers ordres élevés et variés* sans changer le matériel. Un premier niveau de masquage est implémenté en matériel en utilisant l'ISE1 dont l'ordre de masquage est fixé au *moment de la synthèse*. Un *masquage logiciel* utilise ensuite les instructions masquées de l'ISE pour atteindre *divers ordres supérieurs*. Notre solution offre ainsi une *flexibilité au moment de la synthèse*, pour répondre à diverses contraintes de performance et de budget, et au *moment de la compilation logicielle* pour augmenter la sécurité au fil du temps. Notre solution de masquage matériel/logiciel a été l'objet de notre publication [LT23] dans la conférence MWSCAS.

Comme futurs travaux, d'autres cryptosystèmes symétriques peuvent être implémentés avec nos solutions. Nous envisageons également d'implémenter les *chiffrements post-quantiques* Kyber [1] et Saber [2] avec des adaptations de nos ISE dédiées au masquage en utilisant par exemple les implémentations BS développées dans [BC22]. Un autre axe d'étude est d'ajouter de nouvelles instructions masquées à nos ISE pour accélérer des cryptosystèmes spécifiques tel que AES.

1. `https://pq-crystals.org/kyber/`
2. `https://www.esat.kuleuven.be/cosic/pqcrypto/saber/`

Annexe A

Conversion en représentation BS

L'algorithme de transposition de la figure 4.3 peut être amélioré [Knu98] en observant que la transposée d'une matrice peut s'écrire récursivement comme :

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} A^T & C^T \\ B^T & D^T \end{bmatrix}$$

jusqu'à obtenir des matrices de taille 2×2 (sur une matrice 32×32, cela prend 5 itérations). Cet algorithme effectue les opérations en même temps sur A et B, et sur C et D, réduisant le nombre d'opérations par rapport à l'algorithme de transposition simple. Cet algorithme est illustré par le code de la figure A.1.

```c
unsigned int mask_l[5] = {  0xaaaaaaaa,
                            0xcccccccc,
                            0xf0f0f0f0,
                            0xff00ff00,
                            0xffff0000
                         };
unsigned int mask_r[5] = {  0x55555555,
                            0x33333333,
                            0x0f0f0f0f,
                            0x00ff00ff,
                            0x0000ffff
                         };

void to_bitslice32x32(uint32_t data[]) {
  for (int i = 0; i < 5; i ++) {
    int n = (1UL << i);
    for (int j = 0; j < 32; j += (2 * n)){
      for (int k = 0; k < n; k ++) {
        unsigned int u = data[j + k] & mask_r[i];
        unsigned int v = data[j + k] & mask_l[i];
        unsigned int x = data[j + n + k] & mask_r[i];
        unsigned int y = data[j + n + k] & mask_l[i];
        data[j + k] = u | (x << n);
        data[j + n + k] = (v >> n) | y;
      }
    }
  }
}
```

FIGURE A.1 – Code C de la transposition récursive de 32 blocs de 32 bits en représentation BS.

Annexe B

Codes d'implémentations m-slicing de AES

```
table SBOX (input:v8) returns (output:v8) {
    99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,
    ...
    140,161,137,13,191,230,66,104,65,153,45,15,176,84,187,22}

node SubBytes (inputSR:u16x8) returns (out:u16x8) let
    out[0..7]=SBOX(inputSR[0..7]); tel

node ShiftRows (inputSR:u16x8) returns (out:u16x8) let
    forall i in [0,7] {
        out[i] = Shuffle(inputSR[i],[ 0,  1,  2,  3,
                                      5,  6,  7,  4,
                                     10, 11,  8,  9,
                                     15, 12, 13, 14] )
    } tel

node RL4 (input:u16) returns (out:u16) let
    out = input <<< 4; tel

node RL8 (input:u16) returns (out:u16) let
    out = input <<< 8; tel

node MixColumn (a:u16x8) returns (b:u16x8) let
    b[0]=a[1]^RL4(a[1])^RL4(a[0])^RL8(a[0]^RL4(a[0]));
    b[1]=a[2]^RL4(a[2])^RL4(a[1])^RL8(a[1]^RL4(a[1]));
    b[2]=a[3]^RL4(a[3])^RL4(a[2])^RL8(a[2]^RL4(a[2]));
    b[3]=a[4]^RL4(a[4])^a[0]^RL4(a[0])^RL4(a[3])^RL8(a[3]^RL4(a[3]));
    b[4]=a[5]^RL4(a[5])^a[0]^RL4(a[0])^RL4(a[4])^RL8(a[4]^RL4(a[4]));
    b[5]=a[6]^RL4(a[6])^RL4(a[5])^RL8(a[5]^RL4(a[5]));
    b[6]=a[7]^RL4(a[7])^a[0]^RL4(a[0])^RL4(a[6])^RL8(a[6]^RL4(a[6]));
    b[7]=a[0]^RL4(a[0])^RL4(a[7])^RL8(a[7]^RL4(a[7])); tel

node AddRoundKey(i:u16x8,key:u16x8) returns (r:u16x8) let
    r = i ^ key; tel

node AES128(plain:u16x8,key:u16x8[11]) returns (cipher:u16x8)
vars tmp : u16x8[10] let
    tmp[0]=AddRoundKey(plain, key[0]);
    forall i in [1,9] {
    tmp[i]=AddRoundKey(MixColumn(ShiftRows(SubBytes(tmp[i-1]))),key[i]);
    }
    cipher=AddRoundKey(ShiftRows(SubBytes(tmp[9])),key[10]); tel
```

FIGURE B.1 – Code Usuba de notre implémentation 16-slicing de AES.

```
table SBOX (input:v8) returns (output:v8) {
    99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,
    ...
    140,161,137,13,191,230,66,104,65,153,45,15,176,84,187,22}

node SubBytes (inputSR:u8x16) returns (out:u8x16) let
    out[0..7]=SBOX(inputSR[0..7]);
    out[8..15]=SBOX(inputSR[8..15]); tel

node ShiftRows (inputSR:u8x16) returns (out:u8x16) let
    forall i in [0,7] {
        out[i] = Shuffle(inputSR[i],[ 0,  1,  2,  3,
                                      6,  7,  4,  5] )}
    forall i in [8,15] {
        out[i] = Shuffle(inputSR[i],[ 1,  2,  3,  0,
                                      7,  4,  5,  6] )
    } tel

node RL4 (input:u8) returns (out:u8) let
    out = input <<< 4; tel

node MixColumn (a:u8x16) returns (b:u8x16) let
    b[0]=a[1]^a[9]^a[8]^RL4(a[0])^RL4(a[8]);
    b[1]=a[2]^a[10]^a[9]^RL4(a[1])^RL4(a[9]);
    b[2]=a[3]^a[11]^a[10]^RL4(a[2])^RL4(a[10]);
    b[3]=a[4]^a[12]^a[11]^RL4(a[3])^RL4(a[11])^a[0]^a[8];
    b[4]=a[5]^a[13]^a[12]^RL4(a[4])^RL4(a[12])^a[0]^a[8];
    b[5]=a[6]^a[14]^a[13]^RL4(a[5])^RL4(a[13]);
    b[6]=a[7]^a[15]^a[14]^RL4(a[6])^RL4(a[14])^a[0]^a[8];
    b[7]=a[0]^a[8]^a[15]^RL4(a[7])^RL4(a[15]);
    b[8]=a[9]^RL4(a[1])^RL4(a[0])^RL4(a[8])^a[0];
    b[9]=a[10]^RL4(a[2])^RL4(a[1])^RL4(a[9])^a[1];
    b[10]=a[11]^RL4(a[3])^RL4(a[2])^RL4(a[10])^a[2];
    b[11]=a[12]^RL4(a[4])^RL4(a[3])^RL4(a[11])^a[3]^a[8]^RL4(a[0]);
    b[12]=a[13]^RL4(a[5])^RL4(a[4])^RL4(a[12])^a[4]^a[8]^RL4(a[0]);
    b[13]=a[14]^RL4(a[6])^RL4(a[5])^RL4(a[13])^a[5];
    b[14]=a[15]^RL4(a[7])^RL4(a[6])^RL4(a[14])^a[6]^a[8]^RL4(a[0]);
    b[15]=a[8] ^RL4(a[0])^RL4(a[7])^RL4(a[15])^a[7]; tel

node AddRoundKey(i:u8x16,key:u8x16) returns (r:u8x16) let
    r = i ^ key; tel

node AES128(plain:u8x16,key:u8x16[11]) returns (cipher:u8x16)
vars tmp : u8x16[10] let
    tmp[0]=AddRoundKey(plain, key[0]);
    forall i in [1,9] {
    tmp[i]=AddRoundKey(MixColumn(ShiftRows(SubBytes(tmp[i-1]))),key[i]);
    }
    cipher=AddRoundKey(ShiftRows(SubBytes(tmp[9])),key[10]); tel
```

FIGURE B.2 – Code Usuba de notre implémentation 8-slicing de AES.

```
table SBOX (input:v8) returns (output:v8) {
    99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,
    ...
    140,161,137,13,191,230,66,104,65,153,45,15,176,84,187,22}

node SubBytes (inputSR:u4x32) returns (out:u4x32) let
    out[0..7]=SBOX(inputSR[0..7]);
    out[8..15]=SBOX(inputSR[8..15]);
    out[16..23]=SBOX(inputSR[16..23]);
    out[24..31]=SBOX(inputSR[24..31]); tel

node ShiftRows (inputSR:u4x32) returns (out:u4x32) let
    forall i in [0,7] {
        out[i]=inputSR[i];
        out[i+8]=inputSR[i+8] <<< 1;
        out[i+16]=inputSR[i+16] <<< 2;
        out[i+24]=inputSR[i+24] <<< 3;
    } tel

node MixColumn (a:u4x32) returns (b:u4x32) let
    forall i in [0,3] {
        b[i*8]=a[(i*8+1)%32]^a[(i*8+9)%32]^a[(i*8+8)%32]^a[(i*8+16)%32]^a[(i*8+24)%32];
        b[i*8+1]=a[(i*8+2)%32]^a[(i*8+10)%32]^a[(i*8+9)%32]^a[(i*8+17)%32]^a[(i*8+25)%32];
        b[i*8+2]=a[(i*8+3)%32]^a[(i*8+11)%32]^a[(i*8+10)%32]^a[(i*8+18)%32]^a[(i*8+26)%32];
        b[i*8+3]=a[(i*8+4)%32]^a[(i*8+12)%32]^a[(i*8+11)%32]^a[(i*8+19)%32]^a[(i*8+27)%32]
                ^(a[(i*8)%32]^a[(i*8+8)%32]);
        b[i*8+4]=a[(i*8+5)%32]^a[(i*8+13)%32]^a[(i*8+12)%32]^a[(i*8+20)%32]^a[(i*8+28)%32]
                ^(a[(i*8)%32]^a[(i*8+8)%32]);
        b[i*8+5]=a[(i*8+6)%32]^a[(i*8+14)%32]^a[(i*8+13)%32]^a[(i*8+21)%32]^a[(i*8+29)%32];
        b[i*8+6]=a[(i*8+7)%32]^a[(i*8+15)%32]^a[(i*8+14)%32]^a[(i*8+22)%32]^a[(i*8+30)%32]
                ^(a[(i*8)%32]^a[(i*8+8)%32]);
        b[i*8+7]=(a[(i*8)%32]^a[(i*8+8)%32])^a[(i*8+15)%32]^a[(i*8+23)%32]^a[(i*8+31)%32];
    } tel

node AddRoundKey(i:u4x32,key:u4x32) returns (r:u4x32) let
    r = i ^ key; tel

node AES128 (plain:u4x32,key:u4x32[11]) returns (cipher:u4x32)
vars tmp : u4x32[10] let
    tmp[0]=AddRoundKey(plain, key[0]);
    forall i in [1,9] {
    tmp[i]=AddRoundKey(MixColumn(ShiftRows(SubBytes(tmp[i-1]))),key[i]);
    }
    cipher=AddRoundKey(ShiftRows(SubBytes(tmp[9])),key[10]); tel
```

FIGURE B.3 – Code Usuba de notre implémentation 4-slicing de AES.

```
table SBOX (input:v8) returns (output:v8) {
    99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,
    ...
    140,161,137,13,191,230,66,104,65,153,45,15,176,84,187,22}

node SubBytes (inputSR:u2x64) returns (out:u2x64) let
    out[0..7]=SBOX(inputSR[0..7]);
    out[8..15]=SBOX(inputSR[8..15]);
    out[16..23]=SBOX(inputSR[16..23]);
    out[24..31]=SBOX(inputSR[24..31]);
    out[32..39]=SBOX(inputSR[32..39]);
    out[40..47]=SBOX(inputSR[40..47]);
    out[48..55]=SBOX(inputSR[48..55]);
    out[56..63]=SBOX(inputSR[56..63]); tel

node ShiftRows (inputSR:u2x64) returns (out:u2x64) let
    forall i in [0,7] {
        out[i]=inputSR[i];
        out[i+8]=inputSR[i+8];
        out[i+16]=inputSR[i+24];
        out[i+24]=inputSR[i+16]<<<1;
        out[i+32]=inputSR[i+32]<<<1;
        out[i+40]=inputSR[i+40]<<<1;
        out[i+48]=inputSR[i+56]<<<1;
        out[i+56]=inputSR[i+48];
    } tel

node MixColumn (a:u2x64) returns (b:u2x64) let
    forall i in [0,7] {
        b[i*8]=a[(i*8+1)%64]^a[(i*8+17)%64]^a[(i*8+16)%64]^a[(i*8+32)%64]
               ^a[(i*8+48)%64];
        b[i*8+1]=a[(i*8+2)%64]^a[(i*8+18)%64]^a[(i*8+17)%64]^a[(i*8+33)%64]
                 ^a[(i*8+49)%64];
        b[i*8+2]=a[(i*8+3)%64]^a[(i*8+19)%64]^a[(i*8+18)%64]^a[(i*8+34)%64]
                 ^a[(i*8+50)%64];
        b[i*8+3]=a[(i*8+4)%64]^a[(i*8+20)%64]^a[(i*8+19)%64]^a[(i*8+35)%64]
                 ^a[(i*8+51)%64]^(a[(i*8)%64]^a[(i*8+16)%64]);
        b[i*8+4]=a[(i*8+5)%64]^a[(i*8+21)%64]^a[(i*8+20)%64]^a[(i*8+36)%64]
                 ^a[(i*8+52)%64]^(a[(i*8)%64]^a[(i*8+16)%64]);
        b[i*8+5]=a[(i*8+6)%64]^a[(i*8+22)%64]^a[(i*8+21)%64]^a[(i*8+37)%64]
                 ^a[(i*8+53)%64];
        b[i*8+6]=a[(i*8+7)%64]^a[(i*8+23)%64]^a[(i*8+22)%64]^a[(i*8+38)%64]
                 ^a[(i*8+54)%64]^(a[(i*8)%64]^a[(i*8+16)%64]);
        b[i*8+7]=(a[(i*8)%64]^a[(i*8+16)%64])^a[(i*8+23)%64]^a[(i*8+39)%64]
                 ^a[(i*8+55)%64];
    } tel

node AddRoundKey(i:u2x64,key:u2x64) returns (r:u2x64) let
    r = i ^ key; tel

node AES128 (plain:u2x64,key:u2x64[11]) returns (cipher:u2x64)
vars tmp : u2x64[10] let
    tmp[0]=AddRoundKey(plain, key[0]);
    forall i in [1,9] {
    tmp[i]=AddRoundKey(MixColumn(ShiftRows(SubBytes(tmp[i-1]))),key[i]);
    }
    cipher=AddRoundKey(ShiftRows(SubBytes(tmp[9])),key[10]); tel
```

FIGURE B.4 – Code Usuba de notre implémentation 2-slicing de AES.

Annexe C

Preuves de sécurité du chapitre 10

C.1 Preuve de la propriété 2

Démonstration. Soit $\Omega = \mathcal{I} \cup \mathcal{O}$ un ensemble de bits sondés vérifiant $|\Omega| \leq d$, ou $\mathcal{I}$ désigne les sondes internes et $\mathcal{O}$ les sondes de sortie.

Désignons par $\mathcal{I}_i$ les sondes internes dans l'instruction $\hat{\mathbf{z}}[i] \leftarrow \texttt{ise1.xor}(\hat{\mathbf{x}}[i], \hat{\mathbf{y}}[i])$ et par $\mathcal{O}_i$ les sondes de sortie sur $\hat{\mathbf{z}}[i]$. Si $|\mathcal{I}_i \cup \mathcal{O}_i| \leq d_{\mathrm{H}}$, alors comme `ise1.xor` est d_{H}-NI dans le modèle de sondage robuste aux glitchs, les sondes $\mathcal{I}_i \cup \mathcal{O}_i$ peuvent être simulées par un ensemble de bits $\mathcal{X}_i$ de $\hat{\mathbf{x}}[i]$ vérifiant $|\mathcal{X}_i| \leq |\mathcal{I}_i \cup \mathcal{O}_i|$ et un ensemble de bits $\mathcal{Y}_i$ de $\hat{\mathbf{y}}[i]$ vérifiant $|\mathcal{Y}_i| \leq |\mathcal{I}_i \cup \mathcal{O}_i|$. Sinon, on a $|\mathcal{I}_i \cup \mathcal{O}_i| > d_{\mathrm{H}}$ et les sondes $\mathcal{I}_i \cup \mathcal{O}_i$ peuvent être simulées par l'ensemble de bits $\mathcal{X}_i$ contenant les d_{H} parts de $\hat{\mathbf{x}}[i]$ et l'ensemble de bits $\mathcal{Y}_i$ contenant les d_{H} parts de $\hat{\mathbf{y}}[i]$. $\mathcal{X}_i$ et $\mathcal{Y}_i$ vérifie alors $|\mathcal{X}_i| \leq d_{\mathrm{H}} \leq |\mathcal{I}_i \cup \mathcal{O}_i|$ et $|\mathcal{Y}_i| \leq d_{\mathrm{H}} \leq |\mathcal{I}_i \cup \mathcal{O}_i|$.

On en déduit que l'ensemble des bits sondés $\Omega = \mathcal{I} \cup \mathcal{O}$ peut être simulé par l'ensemble de bits $\mathcal{X} = \cup_i \mathcal{X}_i$ de $\hat{\mathbf{x}}$ vérifiant $|\mathcal{X}| \leq \sum_i |\mathcal{X}_i| \leq \sum_i |\mathcal{I}_i \cup \mathcal{O}_i| \leq |\Omega|$ et par l'ensemble de bits $\mathcal{Y} = \cup_i \mathcal{Y}_i$ de $\hat{\mathbf{y}}$ vérifiant $|\mathcal{Y}| \leq \sum_i |\mathcal{X}_i| \leq \sum_i |\mathcal{I}_i \cup \mathcal{O}_i| \leq |\Omega|$. L'algorithme 10 est donc d-NI dans le modèle de sondage robuste aux glitchs. □

C.2 Preuve de la propriété 3

Démonstration. Pour prouver la d-SNI de l'algorithme 12, nous construisons un simulateur. Soit $\Omega = \mathcal{I} \cup \mathcal{O}$ un ensemble d'au plus d bits sondés dans l'implémentation, ou $\mathcal{I}$ désigne l'ensemble des sondes internes et $\mathcal{O}$ désigne l'ensemble des sondes de sortie. On note $t_i = |\mathcal{I}|$ et $t_o = |\mathcal{O}|$. Désignons par $\mathcal{I}_x$ les sondes internes dans l'instruction générant la variable x. Notre objectif est de déterminer un ensemble de bits $\mathcal{X}$ de $\hat{\mathbf{x}}$ vérifiant $|\mathcal{X}| \leq t_i$ et un ensemble de bits $\mathcal{Y}$ de $\hat{\mathbf{y}}$ vérifiant $|\mathcal{Y}| \leq t_i$, tel que l'ensemble des bits sondés $\Omega = \mathcal{I} \cup \mathcal{O}$ puisse être simulé par $\mathcal{X}$ et $\mathcal{Y}$.

Nous définissons dans un premier temps un ensemble $\mathcal{K}$ tel que les bits sondés $\mathcal{I}_{r_{i,j}}$, $\mathcal{I}_{u_{i,j}}$ et $\mathcal{I}_{v_{i,j}}$ puissent être simulés avec les ensembles de bits $\mathcal{A}_{i,j} = \{a_{i,j}[k], 0 \leq k \leq d_{\mathrm{H}}\}$. Comme l'instruction `ise1.xor` est d_{H}-NI et donc implémentée de façon directe, les seuls bits qui peuvent être sondés dans l'instruction `ise1.xor` sont les bits d'entrées. Si $a_{i,j}[k]$, $u_{i,j}[k]$ ou $r_{i,j}[k]$ est sondé, on ajoute (i, j, k) à $\mathcal{K}$. Si $v_{i,j}[k]$ avec $j < d_{\mathrm{S}}$ est sondé, on ajoute (i, i, k) à $\mathcal{K}$. L'ensemble $\mathcal{K}$ vérifie alors $|\mathcal{K}| \leq \sum_{i,j} |\mathcal{I}_{r_{i,j}}| + |\mathcal{I}_{u_{i,j}}| + |\mathcal{I}_{v_{i,j}}|$.

Nous allons maintenant construire un ensemble de bits $\mathcal{X}$ de $\hat{\mathbf{x}}$ vérifiant $|\mathcal{X}| \leq t_i$ et un ensemble de bits $\mathcal{Y}$ de $\hat{\mathbf{y}}$ vérifiant $|\mathcal{Y}| \leq t_i$ tel que les bits $\mathcal{A}_{i,j}$ et $\mathcal{I}_{a_{i,j}}$ puissent être simulés

avec $\mathcal{X}$ et $\mathcal{Y}$. Comme l'instruction `ise1.and` est d_{H}-SNI, alors pour tout $0 \leq i, j \leq d_{\mathrm{S}}$, les ensembles de bits $\mathcal{A}_{i,j}$ et $\mathcal{I}_{a_{i,j}}$ peuvent être simulés avec un ensemble de bits $\mathcal{X}_{i,j}$ de $\hat{\mathbf{x}}$ vérifiant $|\mathcal{X}_{i,j}| \leq |\mathcal{I}_{a_{i,j}}| + |\mathcal{A}_{i,j}|$ et un ensemble de bits $\mathcal{Y}_{i,j}$ de $\hat{\mathbf{y}}$ vérifiant $|\mathcal{Y}_{i,j}| \leq |\mathcal{I}_{a_{i,j}}| + |\mathcal{A}_{i,j}|$. L'ensemble $\mathcal{X} = \cup_{i,j}\mathcal{X}_{i,j}$ vérifie alors $|\mathcal{X}| \leq \sum_{i,j} |\mathcal{X}_{i,j}| \leq |\mathcal{I}_{a_{i,j}}| + |\mathcal{A}_{i,j}| \leq \sum_{i,j} |\mathcal{I}_{a_{i,j}}| + |\mathcal{I}_{r_{i,j}}| + \mathcal{I}_{u_{i,j}} + \mathcal{I}_{v_{i,j}} \leq t_i$. De même, l'ensemble $\mathcal{Y} = \cup_{i,j}\mathcal{Y}_{i,j}$ vérifie $|\mathcal{Y}| \leq t_i$.

Nous vérifions à présent que les ensembles de bits $\mathcal{X}$ et $\mathcal{Y}$ permettent de simuler l'ensemble des bits sondés $\Omega = \mathcal{I} \cup \mathcal{O}$. Par construction, les ensembles de bits $\mathcal{A}_{i,j}$ et $\mathcal{I}_{a_{i,j}}$ peuvent être simulés avec les ensembles de bits $\mathcal{X}$ et $\mathcal{Y}$. Nous attribuons une valeur aléatoire à chaque $r_{i,j}[k]$ entrant dans le calcul de n'importe quel bit sondé. Le bit sondé $u_{i,j}[k]$ peut être simulé avec $a_{i,j}[k] \in \mathcal{K}$. Supposons maintenant que $v_{i,j}[k]$ soit sondé. $v_{i,j}[k]$ est la somme de $u_{i,J}[k]$ pour $0 \leq J \leq j$. Si $(i, J, k) \in \mathcal{K}$, alors $u_{i,J}[k]$ peut être simulé en utilisant $a_{i,J}[k]$. Autrement, le calcul de $u_{i,J}[k]$ utilise le bit aléatoire $r_{i,J}[k]$ si $i < J$ ou $r_{J,i}[k]$ si $i > J$ qui n'entre dans le calcul d'aucun autre bit sondé. Une valeur aléatoire peut donc être attribuée à $u_{i,J}[k]$. □

C.3 Preuve de la propriété 4

Démonstration. Soit $\mathcal{I}$ un ensemble de t_{int} sondes internes et $\mathcal{O}$ un ensemble de t_{out} sondes de sortie, tel que $t_{int} + t_{out} \leq d$. Déterminons un ensemble $\mathcal{A}$ d'au plus t_{int} indices de parts tels que l'ensemble de sondes $\mathcal{I} \cup \mathcal{O}$ puisse être simulé avec les parts d'entrée dont les indices sont $\mathcal{A} \cup \mathrm{Ind}(\mathcal{O})$, où $\mathrm{Ind}(\cdot)$ désigne les indices des parts d'un ensemble de sonde. Désignons par $\mathcal{I}_i$ les sondes internes dans l'instruction $\hat{\mathbf{z}}[i] \leftarrow \texttt{ise1.xor}(\hat{\mathbf{x}}[i], \hat{\mathbf{y}}[i])$ et par $\mathcal{O}_i$ les sondes de sortie sur $\hat{\mathbf{z}}[i]$. Comme `ise1.xor` est d_{H}-PINI dans le modèle de sondage robuste aux glitchs, il existe un ensemble d'indices de parts $\mathcal{A}_i$ vérifiant $|\mathcal{A}_i| \leq |\mathcal{I}_i|$ tel que l'ensemble de sondes $\mathcal{I}_i \cup \mathcal{O}_i$ puisse être simulé avec les parts d'entrée dont les indices sont $\mathcal{A}_i \cup \mathrm{Ind}(\mathcal{O})$. On en déduit que l'ensemble des bits sondés $\Omega = \mathcal{I} \cup \mathcal{O}$ peut être simulé par l'ensemble des parts d'entrée dont les indices sont $\mathcal{A} \cup \mathrm{Ind}(\mathcal{O})$ avec $\mathcal{A} = \cup_i \mathcal{A}$ et vérifie $|\mathcal{A}| \leq \sum_i |\mathcal{A}_i| \leq \sum_i |\mathcal{I}_i| \leq t_{int}$. L'algorithme 10 est donc d-PINI dans le modèle de sondage robuste aux glitchs. □

C.4 Preuve de la propriété 5

Démonstration. Soit $\mathcal{I}$ un ensemble de t_i sondes internes et $\mathcal{O}$ un ensemble de t_o indices de parts de sortie, tels que $t_i + t_o \leq d$.

Désignons par $\mathcal{I}_i$ l'ensemble des sondes internes dans la i-ième instruction masquée de l'algorithme 13. Comme l'instruction est d_{H}-PINI, alors l'ensemble de sonde $\mathcal{I}_i$ peut être simulé par l'ensemble des parts matérielles d'indices $\mathcal{X}_i$ dans les entrées de l'instruction, avec $|\mathcal{X}_i| \leq |\mathcal{I}_i|$. D'après la propriété PINI, il est équivalent de sonder $\mathcal{I}_i$ ou les parts matérielles d'indices $\mathcal{X}_i$ dans la sortie de l'instruction.

Simulons maintenant l'ensemble des parts de sortie d'indices $\mathcal{O}$, et les ensembles de parts d'indices $\mathcal{X}_i$ dans la variable de sortie de la i-ième instruction masquée. Soit $0 \leq k \leq d_{\mathrm{H}}$. Désignons par $\mathcal{Y}_k$ l'ensemble des parts matérielles sondées d'indices k. Si $|\mathcal{Y}_k| \leq d_{\mathrm{S}}$, alors il existe un ensemble $\mathcal{A}_k$ vérifiant $|\mathcal{A}_k| \leq |\mathcal{Y}_k|$ tel que $\mathcal{Y}_k$ puisse être simulé par les parts d'entrée d'indices dans $\mathcal{A}_k$. On en déduit que l'ensemble des parts sondées peuvent être simulé par les parts d'entrée d'indices $\cup_{0 \leq k \leq d_{\mathrm{H}}} \mathcal{A}_k$ vérifiant $|\cup_{0 \leq k \leq d_{\mathrm{H}}} \mathcal{A}_k| \leq \sum_{0 \leq k \leq d_{\mathrm{H}}} |\mathcal{Y}_k| \leq t_i + t_o \leq d$. □

Bibliographie

[ADF16] Marcin Andrychowicz, Stefan Dziembowski et Sebastian Faust. "Circuit Compilers with $\mathcal{O}(1/log(n))$ Leakage Rate". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2016, p. 586-615. doi : 10.1007/978-3-662-49896-5_21.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç et Jean-Pierre Seifert. "Predicting Secret Keys Via Branch Prediction". In : *Proc. Cryptographers' Track RSA Conference (CT-RSA)*. Springer, fév. 2007, p. 236-249. doi : 10.1007/11967668_15.

[AP20] Alexandre Adomnicai et Thomas Peyrin. "Fixslicing AES-like Ciphers : New bitsliced AES speed records on ARM-Cortex M and RISC-V". In : *Transactions on CHES* (déc. 2020), p. 402-425. doi : 10.46586/tches.v2021.i1.402-425.

[Bal+14] Josep Balasch et al. "On the Cost of Lazy Engineering for Masked Software Implementations". In : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, nov. 2014, p. 64-81. doi : 10.1007/978-3-319-16763-3_5.

[Bal+15] Josep Balasch et al. "DPA, Bitslicing and Masking at 1 GHz". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2015, p. 599-619. doi : 10.1007/978-3-662-48324-4_30.

[Bar+15] Gilles Barthe et al. "Verified Proofs of Higher-Order Masking". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, avr. 2015, p. 457-485. doi : 10.1007/978-3-662-46800-5_18.

[Bar+16] Gilles Barthe et al. "Strong Non-Interference and Type-Directed Higher-Order Masking". In : *Proc. Conference on Computer and Communications Security (CCS)*. ACM, oct. 2016, p. 116-129. doi : 10.1145/2976749.2978427.

[Bar+17] Gilles Barthe et al. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, avr. 2017, p. 535-566. doi : 10.1007/978-3-319-56620-7_19.

[Bar+19a] Gilles Barthe et al. "Improved parallel mask refreshing algorithms : generic solutions with parametrized non-interference and automated optimizations". In : *Journal of Cryptographic Engineering (JCEN)* (jan. 2019), p. 17-26. doi : 10.1007/s13389-018-00202-2.

[Bar+19b] Gilles BARTHE et al. "maskVerif : Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In : *Proc. European Symposium on Research in Computer Security (ESORICS)*. Springer, sept. 2019, p. 300-318. DOI : 10.1007/978-3-030-29959-0_15.

[Bat+16] Alberto BATTISTELLO et al. "Horizontal Side-Channel Attacks and Countermeasures on the ISW Masking Scheme". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2016, p. 23-39. DOI : 10.1007/978-3-662-53140-2_2.

[BBT10] Lyonel BARTHE, Pascal BENOIT et Lionel TORRES. "Investigation of a Masking Countermeasure against Side-Channel Attacks for RISC-based Processor Architectures". In : *Proc. International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, août 2010, p. 139-144. DOI : 10.1109/FPL.2010.35.

[BC22] Olivier BRONCHAIN et Gaëtan CASSIERS. "Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs". In : *Transactions on CHES* (août 2022), p. 553-588. DOI : 10.46586/tches.v2022.i4.553-588.

[BCO04] Eric BRIER, Christophe CLAVIER et Francis OLIVIER. "Correlation Power Analysis with a Leakage Model". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2004, p. 16-29. DOI : 10.1007/978-3-540-28632-5_2.

[Bel+16] Sonia BELAÏD et al. "Randomness Complexity of Private Circuits for Multiplication". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2016, p. 616-648. DOI : 10.1007/978-3-662-49896-5_22.

[Bel+17] Sonia BELAÏD et al. "Private Multiplication over Finite Fields". In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 2017, p. 397-426. DOI : 10.1007/978-3-319-63697-9_14.

[Bel+20] Sonia BELAÏD et al. "Tornado : Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2020, p. 311-341. DOI : 10.1007/978-3-030-45727-3_11.

[Ber+02] Guido BERTONI et al. "Efficient Software Implementation of AES on 32-Bit Platforms". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2002, p. 159-171. DOI : 10.1007/3-540-36400-5_13.

[Ber+10] Guido BERTONI et al. "Sponge-Based Pseudo-Random Number Generators". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2010, p. 33-47. DOI : 10.1007/978-3-642-15031-9_3.

[Ber05] Daniel J. BERNSTEIN. *Cache-timing attacks on AES*. 2005. URL : https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[BGR18] Sonia Belaïd, Dahmun Goudarzi et Matthieu Rivain. "Tight Private Circuits : Achieving Probing Security with the Least Refreshing". In : *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, déc. 2018, p. 343-372. DOI : 10.1007/978-3-030-03329-3_12.

[BGW88] Michael Ben-Or, Shafi Goldwasser et Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation". In : *Proc. Symposium on Theory of computing (STOC)*. ACM, jan. 1988, p. 1-10. DOI : 10.1145/62212.62213.

[Bih97] Eli Biham. "A fast new DES implementation in software". In : *Proc. Fast Software Encryption (FSE)*. Springer, jan. 1997, p. 260-272. DOI : 10.1007/BFb0052352.

[Bil+14a] Begül Bilgin et al. "A More Efficient AES Threshold Implementation". In : *Proc. International Conference on Cryptology in Africa (AFRICACRYPT)*. Springer, mai 2014, p. 267-284. DOI : 10.1007/978-3-319-06734-6_17.

[Bil+14b] Begül Bilgin et al. "Higher-Order Threshold Implementations". In : *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, déc. 2014, p. 326-343. DOI : 10.1007/978-3-662-45608-8_18.

[BM06] Joseph Bonneau et Ilya Mironov. "Cache-Collision Timing Attacks Against AES". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, oct. 2006, p. 201-215. DOI : 10.1007/11894063_16.

[BP12] Joan Boyar et René Peralta. "A Small Depth-16 Circuit for the AES S-Box". In : *Proc. International Information Security Conference (SEC)*. Springer, juin 2012, p. 287-298. DOI : 10.1007/978-3-642-30436-1_24.

[BS12] Daniel J. Bernstein et Peter Schwabe. "NEON Crypto". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2012, p. 320-339. DOI : 10.1007/978-3-642-33027-8_19.

[BS21] Olivier Bronchain et François-Xavier Standaert. "Breaking Masked Implementations with Many Shares on 32-bit Software Platforms". In : *Transactions on CHES* (juill. 2021), p. 202-234. DOI : 10.46586/tches.v2021.i3.202-234.

[BS91] Eli Biham et Adi Shamir. "Differential cryptanalysis of DES-like cryptosystems". In : *Journal of Cryptology (JoC)* (jan. 1991), p. 3-72. DOI : 10.1007/BF00630563.

[BS97] Eli Biham et Adi Shamir. "Differential fault analysis of secret key cryptosystems". In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 1997, p. 513-525. DOI : 10.1007/BFb0052259.

[Buc+06] Marco Bucci et al. "Three-Phase Dual-Rail Pre-charge Logic". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2006, p. 232-241. DOI : 10.1007/11894063_19.

[Can05] David Canright. "A Very Compact S-Box for AES". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2005, p. 441-455. DOI : 10.1007/11545262_32.

[Cas+20] Gaëtan CASSIERS et al. "Hardware Private Circuits : From Trivial Composition to Full Verification". In : *IEEE Transactions on Computers* (sept. 2020), p. 1677-1690. DOI : 10.1109/TC.2020.3022979.

[Cha+99] Suresh CHARI et al. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 1999, p. 398-412. DOI : 10.1007/3-540-48405-1_26.

[CK09] Jean-Sébastien CORON et Ilya KIZHVATOV. "An Efficient Method for Random Delay Generation in Embedded Software". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2009, p. 156-170. DOI : 10.1007/978-3-642-04138-9_12.

[Cla+11] Christophe CLAVIER et al. "Square Always Exponentiation". In : *Proc. International Conference on Cryptology in India (INDOCRYPT)*. Springer, déc. 2011, p. 40-57. DOI : 10.1007/978-3-642-25578-6_5.

[Cnu+16] Thomas de CNUDDE et al. "Masking AES with $d + 1$ Shares in Hardware". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2016, p. 194-212. DOI : 10.1007/978-3-662-53140-2_10.

[Coo+13] Jeremy COOPER et al. *Test vector leakage assessment (TVLA) methodology in practice*. 2013. URL : http://icmc-2013.org/wp/wp-content/uploads/2013/09/Rohatgi_Test-Vector-Leakage-Assessment.pdf.

[Cor+12] Jean-Sébastien CORON et al. "Conversion of Security Proofs from One Leakage Model to Another : A New Issue". In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, mai 2012, p. 69-81. DOI : 10.1007/978-3-642-29912-4_6.

[Cor+14] Jean-Sébastien CORON et al. "Higher-Order Side Channel Security and Mask Refreshing". In : *Proc. Fast Software Encryption (FSE)*. Springer, mars 2014, p. 410-424. DOI : 10.1007/978-3-662-43933-3_21.

[Cor+16] Jean-Sébastien CORON et al. "Faster Evaluation of SBoxes via Common Shares". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2016, p. 498-514. DOI : 10.1007/978-3-662-53140-2_24.

[Cor14] Jean-Sébastien CORON. "Higher Order Masking of Look-Up Tables". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2014, p. 441-458. DOI : 10.1007/978-3-642-55220-5_25.

[CRZ18] Jean-Sébastien CORON, Franck RONDEPIERRE et Rina ZEITOUN. "High Order Masking of Look-up Tables with Common Shares". In : *Transactions on CHES* (fév. 2018), p. 40-72. DOI : 10.13154/tches.v2018.i1.40-72.

[CS20] Gaëtan CASSIERS et François-Xavier STANDAERT. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference". In : *Transactions on Information Forensics and Security (TIFS)* (fév. 2020), p. 2542-2555. DOI : 10.1109/TIFS.2020.2971153.

[CS21] Gaëtan Cassiers et François-Xavier Standaert. "Provably Secure Hardware Masking in the Transition- and Glitch-Robust Probing Model : Better Safe than Sorry". In : *Transactions on CHES* (fév. 2021), p. 136-158. doi : 10.46586/tches.v2021.i2.136-158.

[DDF14] Alexandre Duc, Stefan Dziembowski et Sebastian Faust. "Unifying Leakage Models : From Probing Attacks to Noisy Leakage". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2014, p. 423-440. doi : 10.1007/978-3-642-55220-5_24.

[De +15] Thomas De Cnudde et al. "Higher-Order Threshold Implementation of the AES S-Box". In : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, nov. 2015, p. 259-272. doi : 10.1007/978-3-319-31271-2_16.

[De +17] Thomas De Cnudde et al. "Does Coupling Affect the Security of Masked Implementations ?" In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, avr. 2017, p. 1-18. doi : 10.1007/978-3-319-64647-3_1.

[DGH19] Elke De Mulder, Samatha Gummalla et Michael Hutter. "Protecting RISC-V against Side-Channel Attacks". In : *Proc. Design Automation Conference (DAC)*. ACM, juin 2019, p. 1-4. doi : 10.1145/3316781.3323485.

[DH76] Whitfield Diffie et Martin Hellman. "New directions in cryptography". In : *Transactions on Information Theory* (nov. 1976), p. 644-654. doi : 10.1109/TIT.1976.1055638.

[Dhe+98] Jean-François Dhem et al. "A Practical Implementation of the Timing Attack". In : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, sept. 1998, p. 167-182. doi : 10.1007/10721064_15.

[DR02] Joan Daemen et Vincent Rijmen. *The Design of Rijndael*. 1re éd. Springer, 2002. isbn : 978-3-540-42580-9. doi : 10.1007/978-3-662-04722-4.

[Fau+18] Sebastian Faust et al. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model". In : *Transactions on CHES* (août 2018), p. 89-120. doi : 10.13154/tches.v2018.i3.89-120.

[Fri72] Jeffrey Friedman. *TEMPEST : A Signal Problem*. National Security Agency. 1972. url : https://www.nsa.gov/portals/75/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf.

[Gao+19] Si Gao et al. "Share-slicing : Friend or Foe ?" In : *Transactions on CHES* (nov. 2019), p. 152-174. doi : 10.13154/tches.v2020.i1.152-174.

[Gao+21] Si Gao et al. "An Instruction Set Extension to Support Software-Based Masking". In : *Transactions on CHES* (août 2021), p. 283-325. doi : 10.46586/tches.v2021.i4.283-325.

[GBT07] Benedikt Gierlichs, Lejla Batina et Pim Tuyls. *Mutual Information Analysis – A Universal Differential Side-Channel Attack*. IACR Cryptology ePrint Archive. Jan. 2007. url : https://eprint.iacr.org/2007/198.

[GIB18] Hannes Gross, Rinat Iusupov et Roderick Bloem. "Generic Low-Latency Masking in Hardware". In : *Transactions on CHES* (mai 2018), p. 1-21. DOI : 10.13154/tches.v2018.i2.1-21.

[GM17] Hannes Gross et Stefan Mangard. "Reconciling $d + 1$ Masking in Hardware and Software". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2017, p. 115-136. DOI : 10.1007/978-3-319-66787-4_6.

[GM18] Hannes Gross et Stefan Mangard. "A unified masking approach". In : *Journal of Cryptographic Engineering (JCEN)* (mars 2018), p. 109-124. DOI : 10.1007/s13389-018-0184-y.

[GMK16] Hannes Gross, Stefan Mangard et Thomas Korak. *Domain-Oriented Masking : Compact Masked Hardware Implementations with Arbitrary Protection Order*. IACR Cryptology ePrint Archive. Nov. 2016. URL : https://eprint.iacr.org/2016/486.

[GMK17] Hannes Gross, Stefan Mangard et Thomas Korak. "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order". In : *Proc. Cryptographers' Track RSA Conference (CT-RSA)*. Springer, fév. 2017, p. 95-112. DOI : 10.1007/978-3-319-52153-4_6.

[GMO01] Karine Gandolfi, Christophe Mourtel et Francis Olivier. "Electromagnetic Analysis : Concrete Results". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, mai 2001, p. 251-261. DOI : 10.1007/3-540-44709-1_21.

[Gou+18] Dahmun Goudarzi et al. "Secure Multiplication for Bitslice Higher-Order Masking : Optimisation and Comparison". In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, avr. 2018, p. 3-22. DOI : 10.1007/978-3-319-89641-0_1.

[GP99] Louis Goubin et Jacques Patarin. "DES and Differential Power Analysis The Duplication Method". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 1999, p. 158-172. DOI : 10.1007/3-540-48059-5_15.

[GR17] Dahmun Goudarzi et Matthieu Rivain. "How Fast Can Higher-Order Masking Be in Software ?" In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, avr. 2017, p. 567-597. DOI : 10.1007/978-3-319-56620-7_20.

[Gré+18] Benjamin Grégoire et al. "Vectorizing Higher-Order Masking". In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, avr. 2018, p. 23-43. DOI : 10.1007/978-3-319-89641-0_2.

[Gro+15] Vincent Grosso et al. "LS-Designs : Bitslice Encryption for Efficient Masked Software Implementations". In : *Proc. Fast Software Encryption (FSE)*. Springer, mars 2015, p. 18-37. DOI : 10.1007/978-3-662-46706-0_2.

[Gro+16] Hannes Gross et al. “Concealing Secrets in Embedded Processors Designs”. In : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, nov. 2016, p. 89-104. DOI : 10.1007/978-3-319-54669-8_6.

[GST14] Daniel Genkin, Adi Shamir et Eran Tromer. “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis”. In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 2014, p. 444-461. DOI : 10.1007/978-3-662-44371-2_25.

[Guo+20] Qian Guo et al. “Modeling Soft Analytical Side-Channel Attacks from a Coding Theory Viewpoint”. In : *Transactions on CHES* (août 2020), p. 209-238. DOI : 10.13154/tches.v2020.i4.209-238.

[GYH18] Qian Ge, Yuval Yarom et Gernot Heiser. “No Security Without Time Protection : We Need a New Hardware-Software Contract”. In : *Proc. Asia-Pacific Workshop on Systems (APSys)*. ACM, août 2018, p. 1-9. DOI : 10.1145/3265723.3265724.

[HH98] Helena Handschuh et Howard M. Heys. “A Timing Attack on RC5”. In : *Proc. Selected Areas in Cryptography (SAC)*. Springer, août 1998, p. 306-318. URL : https://dl.acm.org/doi/10.5555/646554.694436.

[HS13] Michael Hutter et Jörn-Marc Schmidt. “The Temperature Side Channel and Heating Fault Attacks”. In : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, nov. 2013, p. 219-235. DOI : 10.1007/978-3-319-08302-5_15.

[ISW03] Yuval Ishai, Amit Sahai et David Wagner. “Private Circuits : Securing Hardware against Probing Attacks”. In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 2003, p. 463-481. DOI : 10.1007/978-3-540-45146-4_27.

[JS17] Anthony Journault et François-Xavier Standaert. “Very High Order Masking : Efficient Implementation and Security Evaluation”. In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2017, p. 623-643. DOI : 10.1007/978-3-319-66787-4_30.

[Ker83] Auguste Kerckhoffs. “La cryptographie militaire”. In : *Journal des Sciences Militaires* (jan. 1883), p. 161-191. URL : https://www.petitcolas.net/kerckhoffs/crypto_militaire_1.pdf.

[Kia+21] Pantea Kiaei et al. “Custom Instruction Support for Modular Defense Against Side-Channel and Fault Attacks”. In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, avr. 2021, p. 221-253. DOI : 10.1007/978-3-030-68773-1_11.

[KJJ99] Paul Kocher, Joshua Jaffe et Benjamin Jun. “Differential Power Analysis”. In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 1999, p. 388-397. DOI : 10.1007/3-540-48405-1_25.

[KM22] David Knichel et Amir Moradi. “Low-Latency Hardware Private Circuits”. In : *Proc. Conference on Computer and Communications Security (CCS)*. ACM, nov. 2022, p. 1799-1812. DOI : 10.1145/3548606.3559362.

[Knu98] Donald E. KNUTH. *The Art of Computer Programming, Volume 3 : Sorting and Searching*. 2e éd. Addison Wesley Longman Publishing Co., 1998. ISBN : 978-0-201-89685-5. DOI : 10.1007/978-0-387-38162-6.

[Koc+11] Paul KOCHER et al. "Introduction to differential power analysis". In : *Journal of Cryptographic Engineering (JCEN)* (mars 2011), p. 5-27. DOI : 10.1007/s13389-011-0006-y.

[Koc96] Paul C. KOCHER. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 1996, p. 104-113. DOI : 10.1007/3-540-68697-5_9.

[Kön08] Robert KÖNIGHOFER. "A Fast and Cache-Timing Resistant Implementation of the AES". In : *Proc. Cryptographers' Track RSA Conference (CT-RSA)*. Springer, avr. 2008, p. 187-202. DOI : 10.1007/978-3-540-79263-5_12.

[Krä+13] Juliane KRÄMER et al. "Differential Photonic Emission Analysis". In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, mars 2013, p. 1-16. DOI : 10.1007/978-3-642-40026-1_1.

[KS09] Emilia KÄSPER et Peter SCHWABE. "Faster and Timing-Attack Resistant AES-GCM". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2009, p. 1-17. DOI : 10.1007/978-3-642-04138-9_1.

[KSM20] David KNICHEL, Pascal SASDRICH et Amir MORADI. "SILVER – Statistical Independence and Leakage Verification". In : *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, déc. 2020, p. 787-816. DOI : 10.1007/978-3-030-64837-4_26.

[KSM21] David KNICHEL, Pascal SASDRICH et Amir MORADI. "Generic Hardware Private Circuits : Towards Automated Generation of Composable Secure Gadgets". In : *Transactions on CHES* (nov. 2021), p. 323-344. DOI : 10.46586/tches.v2022.i1.323-344.

[LT23] Fabrice LOZACHMEUR et Arnaud TISSERAND. "A RISC-V Instruction Set Extension for Flexible Hardware/Software Protection of Cryptosystems Masked at High Orders". In : *Proc. International Midwest Symposium on Circuits and Systems (MWSCAS)*. Phoenix, Arizona, USA : IEEE, août 2023. URL : https://www.mwscas2023.org/.

[Mar+20] Ben MARSHALL et al. "The design of scalar AES Instruction Set Extensions for RISC-V". In : *Transactions on CHES* (déc. 2020), p. 109-136. DOI : 10.46586/tches.v2021.i1.109-136.

[Mat93] Mitsuru MATSUI. "Linear Cryptanalysis Method for DES Cipher". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 1993, p. 386-397. DOI : 10.1007/3-540-48285-7_33.

[MD19] Darius MERCADIER et Pierre-Evariste DAGAND. "Usuba : High-Throughput and Constant-Time Ciphers, by Construction". In : *Proc. Conference on Programming Language Design and Implementation (PLDI)*. ACM, juin 2019, p. 157-173. DOI : 10.1145/3314221.3314636.

[MDP19] Loïc Masure, Cécile Dumas et Emmanuel Prouff. "A Comprehensive Study of Deep Learning for Side-Channel Analysis". In : *Transactions on CHES* (nov. 2019), p. 348-375. DOI : 10.13154/tches.v2020.i1.348-375.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish et Robert H. Sloan. "Investigations of Power Analysis Attacks on Smartcards". In : *Proc. Workshop on Smartcard Technology (WOST)*. Usenix, mai 1999, p. 17. URL : https://dl.acm.org/doi/10.5555/1267115.1267132.

[Mes00a] Thomas S. Messerges. "Securing the AES Finalists Against Power Analysis Attacks". In : *Proc. Fast Software Encryption (FSE)*. Springer, avr. 2000, p. 150-164. DOI : 10.1007/3-540-44706-7_11.

[Mes00b] Thomas S. Messerges. "Using Second-Order Power Analysis to Attack DPA Resistant Software". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2000, p. 238-251. DOI : 10.1007/3-540-44499-8_19.

[Moo+19] Thorben Moos et al. "Glitch-Resistant Masking Revisited : or Why Proofs in the Robust Probing Model are Needed". In : *Transactions on CHES* (fév. 2019), p. 256-292. DOI : 10.13154/tches.v2019.i2.256-292.

[MOP07] Stefan Mangard, Elisabeth Oswald et Thomas Popp. *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. 1re éd. Springer, 2007. ISBN : 978-0-387-38162-6. DOI : 10.1007/978-0-387-38162-6.

[Mor+11] Amir Moradi et al. "Pushing the Limits : A Very Compact and a Threshold Implementation of AES". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2011, p. 69-88. DOI : 10.1007/978-3-642-20465-4_6.

[MP21] Ben Marshall et Dan Page. *SME : Scalable Masking Extensions*. IACR Cryptology ePrint Archive. Oct. 2021. URL : https://eprint.iacr.org/2021/1416.

[MPG05] Stefan Mangard, Thomas Popp et Berndt M. Gammel. "Side-Channel Leakage of Masked CMOS Gates". In : *Proc. Cryptographers' Track RSA Conference (CT-RSA)*. Springer, fév. 2005, p. 351-365. DOI : 10.1007/978-3-540-30574-3_24.

[MPO05] Stefan Mangard, Norbert Pramstaller et Elisabeth Oswald. "Successfully Attacking Masked AES Hardware Implementations". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2005, p. 157-171. DOI : 10.1007/11545262_12.

[MPZ21] Maria Chiara Molteni, Jürgen Pulkus et Vittorio Zaccaria. "On robust strong-non-interferent low-latency multiplications". In : *IET Information Security* (nov. 2021), p. 127-132. DOI : 10.1049/ise2.12048.

[MS06] Stefan Mangard et Kai Schramm. "Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementations". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, oct. 2006, p. 76-90. DOI : 10.1007/11894063_7.

[Nak+11] Daisuke NAKATSU et al. "Combination of SW Countermeasure and CPU Modification on FPGA against Power Analysis". In : *Proc. World Conference on Information Security Applications (WISA)*. Springer, août 2011, p. 258-272. DOI : 10.1007/978-3-642-17955-6_19.

[NIS01] NIST. "Advanced Encryption Standard (AES)". In : *Federal Information Processing Standards Publications (FIPS PUBS)*. Nov. 2001. URL : https://csrc.nist.gov/publications/detail/fips/197/final.

[NRR06] Svetla NIKOVA, Christian RECHBERGER et Vincent RIJMEN. "Threshold Implementations Against Side-Channel Attacks and Glitches". In : *Proc. International Conference on Information and Communications Security (ICICS)*. Springer, déc. 2006, p. 529-545. DOI : 10.1007/11935308_38.

[NRS11] Svetla NIKOVA, Vincent RIJMEN et Martin SCHLÄFFER. "Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches". In : *Journal of Cryptology (JoC)* (avr. 2011), p. 292-321. DOI : 10.1007/s00145-010-9085-7.

[OST05] Dag Arne OSVIK, Adi SHAMIR et Eran TROMER. "Cache Attacks and Countermeasures : The Case of AES". In : *Proc. Cryptographers' Track RSA Conference (CT-RSA)*. Springer, fév. 2005, p. 1-20. DOI : 10.1007/11605805_1.

[Osw+05] Elisabeth OSWALD et al. "Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers". In : *Proc. Cryptographers' Track RSA Conference (CT-RSA)*. Springer, fév. 2005, p. 192-207. DOI : 10.1007/11605805_13.

[PM05] Thomas POPP et Stefan MANGARD. "Masked Dual-Rail Pre-charge Logic : DPA-Resistance Without Routing Constraints". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2005, p. 172-186. DOI : 10.1007/11545262_13.

[PP10] Christof PAAR et Jan PELZL. *Understanding Cryptography : A Textbook for Students and Practitioners*. Springer, 2010. DOI : 10.1007/978-3-642-04101-3.

[PR13] Emmanuel PROUFF et Matthieu RIVAIN. "Masking against Side-Channel Attacks : A Formal Security Proof". In : *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, mai 2013, p. 142-159. DOI : 10.1007/978-3-642-38348-9_9.

[PRB09] Emmanuel PROUFF, Matthieu RIVAIN et Regis BEVAN. "Statistical Analysis of Second Order Differential Power Analysis". In : *IEEE Transactions on Computers* (jan. 2009), p. 799-811. DOI : 10.1109/TC.2009.15.

[QS01] Jean-Jacques QUISQUATER et David SAMYDE. "ElectroMagnetic Analysis (EMA) : Measures and Counter-measures for Smart Cards". In : *Proc. International Conference on Research in Smart Cards (E-smart)*. Springer, sept. 2001, p. 200-210. DOI : 10.1007/3-540-45418-7_17.

[Rep+15] Oscar REPARAZ et al. "Consolidating Masking Schemes". In : *Proc. Annual Cryptology Conference (CRYPTO)*. Springer, août 2015, p. 764-783. DOI : 10.1007/978-3-662-47989-6_37.

[Rep15] Oscar Reparaz. *A note on the security of Higher-Order Threshold Implementations*. IACR Cryptology ePrint Archive. Jan. 2015. URL : https://eprint.iacr.org/2015/001.

[RP10] Matthieu Rivain et Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2010, p. 413-427. DOI : 10.1007/978-3-642-15031-9_28.

[RPD09] Matthieu Rivain, Emmanuel Prouff et Julien Doget. "Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2009, p. 171-188. DOI : 10.1007/978-3-642-04138-9_13.

[Sha49] Claude Shannon. "Communication theory of secrecy systems". In : *The Bell System Technical Journal* (oct. 1949), p. 656-715. DOI : 10.1002/j.1538-7305.1949.tb00928.x.

[Sha79] Adi Shamir. "How to Share a Secret". In : *Communications of the ACM* (nov. 1979), p. 612-613. DOI : 10.1145/359168.359176.

[SM15] Tobias Schneider et Amir Moradi. "Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2015, p. 495-513. DOI : 10.1007/978-3-662-48324-4_25.

[Sok+04] Danil Sokolov et al. "Improving the Security of Dual-Rail Circuits". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, août 2004, p. 282-297. DOI : 10.1007/978-3-540-28632-5_21.

[SS16] Peter Schwabe et Ko Stoffelen. "All the AES You Need on Cortex-M3 and M4". In : *Proc. Selected Areas in Cryptography (SAC)*. Springer, août 2016, p. 180-194. DOI : 10.1007/978-3-319-69453-5_10.

[Sto19] Ko Stoffelen. "Efficient Cryptography on the RISC-V Architecture". In : *Proc. International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*. Springer, oct. 2019, p. 323-340. DOI : 10.1007/978-3-030-30530-7_16.

[TB07] Michael Tunstall et Olivier Benoit. "Efficient Use of Random Delays in Embedded Software". In : *Proc. International Workshop on Information Security Theory and Practices (WISTP)*. Springer, mai 2007, p. 27-38. DOI : 10.1007/978-3-540-72354-7_3.

[TG07] Stefan Tillich et Johann Grossschädl. "Power Analysis Resistant AES Implementation with Instruction Set Extensions". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2007, p. 303-319. DOI : 10.1007/978-3-540-74735-2_21.

[TKS10] Stefan Tillich, Mario Kirschbaum et Alexander Szekely. "SCA-Resistant Embedded Processors : The next Generation". In : *Proc. Advances in Computer Systems Architecture (ACSAC)*. ACM, déc. 2010, p. 211-220. DOI : 10.1145/1920261.1920293.

[TKS11] Stefan TILLICH, Mario KIRSCHBAUM et Alexander SZEKELY. "Implementation and Evaluation of an SCA-Resistant Embedded Processor". In : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, sept. 2011, p. 151-165. DOI : 10.1007/978-3-642-27257-8_10.

[Tsu+03] Yukiyasu TSUNOO et al. "Cryptanalysis of DES Implemented on Computers with Cache". In : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, sept. 2003, p. 62-76. DOI : 10.1007/978-3-540-45238-6_6.

[UHA17] Rei UENO, Naofumi HOMMA et Takafumi AOKI. "Toward More Efficient DPA-Resistant AES Hardware Architecture Based on Threshold Implementation". In : *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*. Springer, avr. 2017, p. 50-64. DOI : 10.1007/978-3-319-64647-3_4.

[Vey+12] Nicolas VEYRAT-CHARVILLON et al. "Shuffling against Side-Channel Attacks : A Comprehensive Study with Cautionary Note". In : *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, déc. 2012, p. 740-757. DOI : 10.1007/978-3-642-34961-4_44.

[WA19] Andrew WATERMAN et Krste ASANOVIĆ. *The RISC-V Instruction Set Manual Volume I : Unprivileged ISA*. RISC-V Foundation. Déc. 2019. URL : https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[Yan+18] Bohan YANG et al. "ES-TRNG : A High-throughput, Low-area True Random Number Generator based on Edge Sampling". In : *Transactions on CHES* (août 2018), p. 267-292. DOI : 10.13154/tches.v2018.i3.267-292.

Table des figures

Liste des tableaux

Liste des algorithmes

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

Titre : Extensions cryptographiques pour processeurs embarqués

Mot clés : processeur sécurisé, masquage logiciel/matériel, cryptographie symétrique

Résumé : Les implémentations de cryptosystèmes, même robustes mathématiquement, sont susceptibles de divulguer des informations via l'observation de canaux auxiliaires (par exemple la consommation d'énergie). Les attaques par analyse des canaux auxiliaires (SCA) exploitent de potentielles corrélations entre des valeurs physiques mesurées et des opérations et opérandes traitées dans le circuit pour récupérer des données sensibles. Le masquage est une contre-mesure couramment utilisée pour protéger des systèmes logiciels et/ou matériels contre certaines attaques SCA. Cependant, la mise en œuvre purement logiciel du masquage impose un surcoût important en termes de tailles de code et temps de calcul, et n'atteint pas toujours le niveau de sécurité attendu en raison de fuites liées à la micro-architecture des processeurs. Dans cette thèse, nous avons développé différentes extensions du jeu d'instructions RISC-V afin de protéger les cryptosystèmes contre des attaques SCA en utilisant un masquage d'ordre élevé. Nous proposons également une solution de masquage matériel/logiciel pour masquer à des ordres élevés avec une grande flexibilité au moment de la synthèse et de la compilation logicielle pour divers compromis coût/performance. Nos solutions ont été implémentées sur le processeur RISC-V CV32E40P, puis validées et évaluées sur FPGA.

Title: Cryptographic extensions for embedded processors

Keywords: secure processor, software/hardware masking, symmetric cryptography

Abstract: Implementations of cryptosystems, even mathematically robust ones, are likely to leak information through the observation of side channels (e.g., energy consumption). Side channel analysis (SCA) attacks exploit potential correlations between measured physical values and operations or operands processed in the circuit to recover sensitive data. Masking is a common countermeasure to protect software or hardware systems against SCA attacks. However, pure software masking leads to important overheads in terms of code size and execution time while it does not always acheive the expected security level due to leaks in the processor micro-architecture. In this thesis, we developed several extensions of the RISC-V instruction set to protect cryptosystems against SCA attacks using high-order masking. We also proposed a hardware/software masking solution at high orders with flexibility at both hardware synthesis time and software compilation time for various cost/performance tradeoffs. Our solutions have been implemented on the RISC-V CV32E40P processor, then validated and evaluated on FPGA.