



HAL
open science

Partitionnement dynamique à grain fin contre des attaques par canaux auxiliaires en cache

Nicolas Gaudin

► **To cite this version:**

Nicolas Gaudin. Partitionnement dynamique à grain fin contre des attaques par canaux auxiliaires en cache. Architectures Matérielles [cs.AR]. Université Bretagne Sud, 2024. Français. NNT: . tel-04920225

HAL Id: tel-04920225

<https://hal.science/tel-04920225v1>

Submitted on 30 Jan 2025

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

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



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE 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

Nicolas GAUDIN

Partitionnement dynamique à grain fin contre des attaques par canaux auxiliaires en cache

Thèse présentée et soutenue à Lorient, le 18 décembre 2024

Unité de recherche : Lab-STICC, UMR 6285

Thèse N° : 714

Rapporteurs avant soutenance :

David HÉLY Professeur à Grenoble INP, Grenoble, France
Damien COUROUSSÉ Ingénieur de Recherche HDR au CEA-List, Grenoble, France

Composition du Jury :

Présidente :	Emmanuelle ENCRENAZ	Professeure à Sorbonne Université, Paris, France
Examineurs :	Pascal BENOIT	Professeur à l'Université de Montpellier, Montpellier, France
	Guillaume HIET	Professeur à CentralSupélec, Rennes, France
	David HÉLY	Professeur à Grenoble INP, Grenoble, France
	Damien COUROUSSÉ	Ingénieur de Recherche HDR au CEA-List, Grenoble, France
Dir. de thèse :	Guy GOGNIAT	Professeur à l'Université Bretagne Sud, Lorient, France
Co-dir. de thèse :	Vianney LAPÔTRE	Maître de Conférences HDR à l'Université Bretagne Sud, Lorient, France
Enc. de thèse :	Pascal COTRET	Maître de Conférences à l'ENSTA Bretagne, Brest, France

Au mé pay, partit tau cèu trop de d'ore.

REMERCIEMENTS

Je tiens tout d'abord à remercier Vianney Lapôte pour son engagement et son accompagnement au quotidien. Ses discussions enrichissantes et son aide précieuse m'ont permis de progresser tant sur le plan scientifique qu'humain. J'exprime également ma gratitude à Guy Gogniat, dont la sagesse et la disponibilité ont été d'une grande valeur tout au long de ces trois années de thèse. Ses précieux conseils ont joué un rôle essentiel dans l'aboutissement de ces travaux. Je remercie Pascal Cotret pour son implication remarquable malgré la distance. Grâce à son soutien, parfois même tard le soir, les barrières liées à l'éloignement ont été largement surmontées. J'en profite également pour réitérer mes remerciements envers mes encadrants pour m'avoir offert l'opportunité de partager ces trois années enrichissantes en leur compagnie.

Je remercie Emmanuelle Encrenaz, Pascal Benoit et Guillaume Hiet d'avoir accepté d'examiner ma soutenance de thèse en faisant partie du jury. Je remercie également David Hély et Damien Couroussé pour leur avis éclairé et le temps qu'ils ont accordé dans le cadre de leur rôle de rapporteurs.

Je tiens également à remercier les membres du laboratoire qui m'ont accueilli dans cette région qui m'était inconnue. Mes premiers mots vont à Florence et Virginie qui, par leur gentillesse et leur bienveillance, ont rendu cette thèse bien plus facile. Je remercie M. Johann Laurent qui a veillé sans relâche au bon fonctionnement de mon cache. Enfin, mes remerciements vont aux collègues pour ces moments de partage passés lors des pauses café. Je remercie l'ensemble des doctorants pour ces trois années passées dans le bureau 213, pour leur encouragement mutuel, leur bonne humeur et leur entraide sur les aspects techniques. Je remercie les collègues de Bochum pour leur accueil chaleureux durant ce séjour de trois mois en Allemagne.

Bien qu'ils tentent encore de comprendre les tenants et les aboutissants de mon sujet de thèse, je remercie chaleureusement toute ma famille pour leur soutien indéfectible tout au long de mon parcours académique. À cela s'ajoute la source de motivation apportée par *lous maynats*, Anaë, Lucas et Gabriel, qui se demandent pourquoi leur tonton est encore à l'école.

Finalement, je remercie ma bande de copains qui m'ont permis de m'évader de mon exil en terre bretonne lors de nombreux week-ends aux quatre coins de la France.

SOMMAIRE

Introduction	13
Évolution des processeurs : d’hier à aujourd’hui	13
Projet SCRATCHS	15
Structure de la thèse	16
1 État de l’art	19
1.1 Microarchitecture	21
1.1.1 Pipeline	21
1.1.2 Caches	23
1.2 Fuites sur la microarchitecture	26
1.2.1 Analyse des fuites	26
1.2.2 Attaques par canaux auxiliaires temporels	31
1.3 Contremesures existantes	34
1.3.1 Détection des attaques	35
1.3.2 Contremesures basées sur l’ajout d’aléa	37
1.3.3 Contremesures basées sur le partitionnement	39
1.3.4 Autres solutions	41
1.4 Conclusion	42
2 Mécanisme de verrouillage de lignes au sein d’un cache de données	43
2.1 Modèle de menaces	45
2.2 Mécanisme de verrouillage des données au sein du cache	46
2.2.1 Motivations	46
2.2.2 Approche proposée	48
2.3 Solution d’implémentation Matérielle/Logicielle	50
2.3.1 Analyse des avantages et inconvénients	50
2.3.2 Extension du jeu d’instructions	53
2.3.3 Support matériel	55
2.3.4 Support logiciel	56
2.3.5 Spécification des instructions <code>lock</code> et <code>unlock</code>	56
2.4 Conclusion	58

3	Implémentation du mécanisme de verrouillage	59
3.1	Implémentation du mécanisme de sécurité	61
3.1.1	Utilisation par le logiciel	61
3.1.2	Implémentation matérielle	63
3.2	Évaluation	68
3.2.1	Évaluation de la surface matérielle	68
3.2.2	Évaluation de la sécurité	70
3.2.3	Évaluation des performances	72
3.3	Conclusion	77
4	Solution hybride du mécanisme de verrouillage	79
4.1	Motivations	81
4.2	Solution Proposée	82
4.2.1	Choix de la fonction de dérivation d'index	83
4.2.2	Choix de la politique de remplacement	84
4.3	Implémentation	85
4.3.1	Système considéré	85
4.3.2	Implémentation au sein du cache	87
4.3.3	Description détaillée des modules	89
4.4	Impact de l'hybridation sur la sécurité	90
4.5	Évaluation	90
4.5.1	Évaluation de la surface matérielle	91
4.5.2	Évaluation des performances	92
4.6	Conclusion	93
	Conclusion	95
	Synthèse	95
	Perspectives	96
	Listes des publications et des communications	99
	Bibliographie	101

TABLE DES FIGURES

1	Évolution des caractéristiques des processeurs sur 50 ans.	14
2	Schéma de principe du projet SCRATCHS.	15
1.1	Description matérielle d'un cœur de processeur 32-bit à 4 étages.	21
1.2	Architecture d'un cache associatif à N voies.	25
1.3	Représentation des fuites menant aux canaux auxiliaires.	28
1.4	Principe de fonctionnement de l'attaque PRIME+PROBE.	33
1.5	Schémas de principe d'architectures de caches basées sur l'aléa.	38
1.6	Schémas de principe d'architectures de caches basées sur le partitionnement.	40
2.1	Modèle de menaces.	45
2.2	Cas d'étude des failles induites par l'utilisation de PLcache.	47
2.3	Procédure d'accès au cache considérant le mécanisme de verrouillage.	49
2.4	Exemple pédagogique de mise à jour de la politique de remplacement.	49
3.1	Impact du placement mémoires des données à verrouiller.	62
3.2	Description matérielle du cœur CV32E40P et un niveau de mémoire cache de donnée.	64
3.3	Schéma bloc du cache implémentant le mécanisme de verrouillage.	65
3.4	Description matérielle de la politique de remplacement LRU.	66
3.5	Description matérielle des sous-modules utilisés dans la politique de remplacement.	67
3.6	Principe de l'algorithme de chiffrement AES-128.	70
3.7	Cartographie des résultats d'attaques en caches menées sur AES-128.	71
3.8	Impact sur la suite Embench-IoT 1.0 lorsqu'une application concurrente verrouille N_l lignes de cache.	75
4.1	Schéma de principe de notre architecture de mémoire cache hybride.	82
4.2	Cas d'étude de l'utilisation de SCARF avec plusieurs configurations.	83
4.3	Description matérielle du cœur CV32E40P et un niveau de mémoires caches.	86
4.4	Schéma bloc du cache hybride implémentant le mécanisme de verrouillage.	87
4.5	Description matérielle de la politique de remplacement VARP-64.	88
4.6	Description matérielle des sous-modules utilisés dans VARP-64.	89
4.7	Impact sur la suite Embench-IoT 1.0 lorsqu'une application concurrente verrouille N_l lignes de cache.	93

LISTE DES TABLEAUX

1.1	Liste non exhaustive d'attaques par canaux auxiliaires visant les caches.	31
1.2	Liste non exhaustive de contremesures logicielles et matérielles permettant de se prémunir des attaques en caches.	36
2.1	Bilan des avantages et des inconvénients de chacun des types de solutions.	52
2.2	Assignment des codes d'opérations des instructions de la base RISC-V.	54
2.3	Format de base des instructions RV32I incluant les variantes des immédiats.	55
2.4	Format des instructions <code>lock</code> et <code>unlock</code> se basant sur le type I.	57
3.1	Distribution des quatre modules de mises à jour en fonction de la voie accédée.	68
3.2	Résultats de surface post-implémentation sur FPGA Kintex-7.	69
3.3	Hausse de temps d'exécution introduit par l'utilisation du mécanisme de verrouillage sur Camellia et AES-128.	73
3.4	Évaluation des performances et du taux de <i>miss</i> sur Embench-IoT 1.0 lorsque AES-128 ou Camellia verrouille sa(ses) table(s).	76
4.1	Résultats de surface post-implémentation sur FPGA Kintex-7.	91

LISTE DES ACRONYMES

AES Advanced Encryption Standard
ALU Arithmetic Logic Unit
ASIC Application-Specific Integrated Circuit
BRAM Block Random-Access Memory
CPU Central Processing Unit
CSR Control and Status Register
DRAM Dynamic Random-Access Memory
E+T EVICT+TIME
FF Flip-Flop register
FPGA Field Programmable Gate Array
F+R FLUSH+RELOAD
HPC Hardware Performance Counter
IDF Index Derivation Function
IoT Internet of Things
ISA Instruction Set Architecture
L1D First Level Data cache
L1I First Level Instruction cache
L2 Second Level cache
LFSR Linear-Feedback Shift Register
LLC Last Level Cache
LRU Least Recently Used
LUT LookUp Table
LSU Load Store Unit
PMP Physical Memory Protection
P+P PRIME+PROBE
P+P+P PRIME+PRUNE+PROBE
RISC-V Reduced Instruction Set Computer V
RSA Rivest–Shamir–Adleman (name of creators)
VARP Variable Age Replacement Policy

INTRODUCTION

Les processeurs occupent une place essentielle dans notre quotidien et sont désormais omniprésents dans nos vies. Ils sont présents dans nos smartphones, ordinateurs, voitures, appareils électroménagers, et même dans les objets connectés de nos maisons, jouant un rôle crucial dans le fonctionnement de ces technologies modernes. Grâce à leurs optimisations, ils rendent nos appareils plus intelligents, plus rapides et plus performants. Nous octroyons de plus en plus de confiance en ces systèmes en les plaçant au centre de domaines critiques (transport, énergie, médical, etc). La construction de tels systèmes nécessite alors la prise en compte des contraintes de sécurité dès la conception.

Évolution des processeurs : d’hier à aujourd’hui

Les processeurs ou Unités Centrales de Calcul (CPU) sont, de nos jours, principalement basés sur l’architecture de Von Neumann. Ce type d’architecture est construit autour d’une mémoire permettant de stocker les instructions et les données. Les instructions composent l’algorithme compilé et sont amenées à manipuler les données durant leurs exécutions. Une interface mémoire dédiée aux instructions permet à l’Unité de Contrôle de rapatrier et de décoder les instructions afin de contrôler l’Unité Arithmétique et Logique (ALU). L’ALU, telle que décrite dans l’architecture de Von Neumann, englobe l’ensemble des opérations pouvant être exécutées au sein du processeur, en utilisant les données rapatriées de la mémoire via son interface dédiée et des entrées provenant de l’environnement dans lequel le processeur est utilisé. Une fois exécutées, ces données sont soit écrites en mémoire, soit permettent le contrôle des périphériques de sorties.

Aujourd’hui, les processeurs basés sur ce type d’architecture sont démocratisés au travers de l’usage des ordinateurs, des smartphones ou des tablettes. Afin de convenir aux nouveaux besoins et usages, les architectures des processeurs n’ont de cesse d’évoluer dans le but d’augmenter les performances. Une constante est commune à l’évolution des performances, il s’agit du nombre de transistors gravés sur les puces. En effet, comme représenté dans la Figure 1, nous observons l’évolution des processeurs sur 50 ans au travers de plusieurs métriques, à savoir : le nombre de transistors, les performances de chaque thread, la fréquence du système et le nombre de cœurs. Ainsi, en orange, nous observons que le nombre de transistors évolue de manière constante depuis les années 70. En corrélation avec la finesse de gravure, la fréquence du système (représentée en vert) continue d’augmenter jusqu’au début des années 2000. Outre la technologie des transistors, l’évolution de l’architecture permet de repousser sans cesse les performances (en bleu)

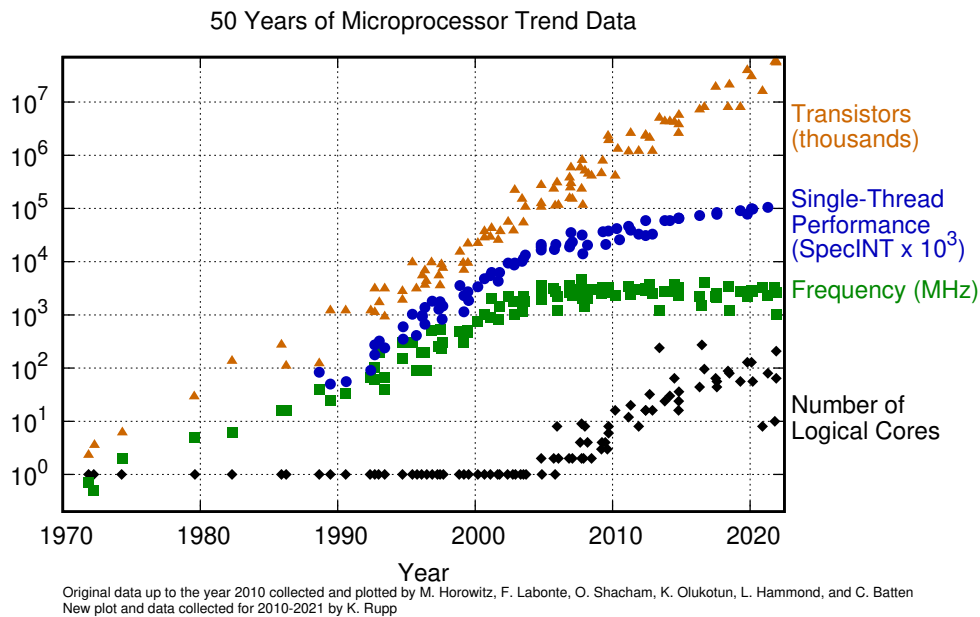


FIGURE 1 – Évolution des caractéristiques des processeurs sur 50 ans [1].

jusqu'à obtenir une limite physique ne permettant plus d'augmenter la fréquence d'utilisation. Ainsi, dès 2005, nous apercevons une multiplication du nombre de cœurs au sein des processeurs afin de contourner les limites physiques rencontrées. Cela permet de poursuivre l'évolution des performances puisque les tâches sont exécutées en parallèle sur les différents cœurs qui composent le processeur. Pendant ce temps, les performances de chaque cœur (en bleu) continuent d'augmenter grâce aux optimisations architecturales. Bien que la fréquence se soit stabilisée, la finesse de gravure continue de diminuer pour atteindre les 3 nm en 2024. Cette tendance permet de densifier les systèmes sur puces pour atteindre plusieurs dizaines de milliards de transistors (*e.g.* 20 milliards pour la puce M2 d'Apple [2], ou 146 milliards pour la puce Instinct MI300A d'AMD [3]). Cela permet également de limiter la consommation d'énergie, car plus un transistor est fin, moins il consomme d'énergie. Cependant, cette minimisation de la taille des transistors permet de développer des systèmes toujours plus complexes en densifiant ces transistors dans les puces. Plutôt que d'observer une baisse de la consommation énergétique des systèmes, nous observons plutôt une stabilisation, voire une hausse de la consommation d'énergie du fait d'une densité d'intégration plus élevée.

En parallèle des processeurs généralistes, il existe une autre gamme de processeurs avec des performances réduites. Il s'agit des processeurs embarqués ayant pour vocation d'être utilisés dans l'Internet des objets (IoT) où de fortes contraintes s'appliquent sur ces systèmes. En effet, ces objets à faible usage (*e.g.* montres connectées, capteurs divers, caméras de vidéosurveillance, enceintes connectées, ainsi que l'ensemble des objets pouvant être succédés des termes intelligents ou connectés) peuvent avoir des contraintes énergétiques, de dissipation thermique ou d'espace,

et n'ont généralement pas besoin de performances élevées. Le critère numéro un pour développer ce type de système est la faible consommation d'énergie. Pour ce faire, les architectes prennent avantages des optimisations issues des 50 ans de développement des processeurs généralistes. Ainsi, nous sommes capables de développer des processeurs embarqués engendrant une faible consommation d'énergie et avec des performances ajustées aux besoins du produit final.

Cette course aux performances d'une part et à la consommation d'énergie d'autre part se fait au détriment de la sécurité qui est bien souvent le dernier point considéré lors du développement de ces systèmes. En effet, les optimisations architecturales introduites avec un objectif de performances peuvent introduire des failles de sécurité. Ces problèmes de sécurité deviennent de plus en plus critiques à cause de la connectivité des objets et de la confidentialité des données qu'ils sont amenés à traiter (*e.g.* bancaire, santé, industrie 4.0, transport, énergie, etc). En conséquence, tout au long de ce manuscrit, nous nous focalisons sur un type de menace pouvant cibler les systèmes embarqués connectés.

Projet SCRATCHS

C'est dans ce contexte que le projet SCRATCHS - Side-Channel Resistant Applications Through Co-designed Hardware/Software - (2021-2025) est financé par le Labex Cominlabs, supporté par l'Agence Nationale de la Recherche française (ANR-10-LABX-07-01). Le projet SCRATCHS vise à proposer des solutions contre les menaces planant sur les systèmes embarqués. Les systèmes embarqués sont, dans la majorité des cas, très simples avec peu de tâches à accomplir. Ces objets connectés ont pour contrainte de consommer le moins possible, ce qui restreint la complexité du matériel utilisé.

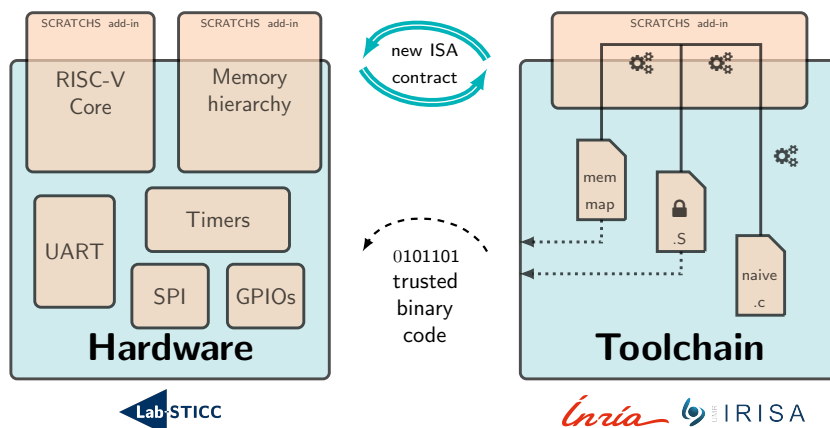


FIGURE 2 – Schéma de principe du projet SCRATCHS.

Cependant, ils sont de plus en plus nombreux et connectés, ce qui a pour conséquence directe d’augmenter leur surface d’attaque, notamment les attaques pouvant être menées à distance. Parmi ces menaces, le projet SCRATCHS se focalise sur les attaques utilisant le temps comme fuite d’information, également appelé canal auxiliaire. Afin de concevoir des contremesures, le projet se focalise sur un système composé d’un processeur en source ouverte (basé sur l’architecture RISC-V) accédant à une mémoire principale au travers d’une hiérarchie de mémoires caches. La réalisation de ces contremesures seront des mécanismes alliant implémentation matérielle et intégration logicielle afin de garantir un faible coût, au niveau de la surface, du programme binaire et des performances d’exécutions. Ainsi, le projet s’articule autour de deux équipes de recherches :

- UMR 6074 IRISA, INRIA / CentraleSupélec
- UMR 6285 Lab-STICC, Université Bretagne Sud / ENSTA Bretagne

Le but de cette collaboration est de pouvoir concevoir ensemble des mécanismes de sécurité alliant conjointement matériel et logiciel. Le développement et l’étude des mécanismes matériels est attribué à l’équipe de recherche du Lab-STICC et la partie plus orientée logicielle et preuves formelles du mécanisme à l’équipe de l’IRISA (comme illustré dans la Figure 2).

Structure de la thèse

Ce document est organisé en quatre chapitres. Le Chapitre 1 établit une revue des éléments microarchitecturaux présent dans un processeur. Ensuite, nous étudions l’interaction des processeurs avec leur environnement, ce qui nous permet de considérer les fuites par canaux auxiliaires temporels. Nous présentons un panel d’attaques considérant ce type de vulnérabilités sur les mémoires caches de données. Après avoir passé en revue les contremesures existantes, nous décidons de centrer nos travaux sur une solution basée sur le partitionnement à grain fin.

Le second Chapitre énonce les principes de notre solution. Dans un premier temps, une analyse souligne les limitations d’une solution existante basée sur un partitionnement à grain fin. Ensuite, nous proposons notre mécanisme de sécurité en apportant des garanties permettant de répondre au modèle de menaces. Deux nouvelles instructions sont spécifiées apportant un contrôle dynamique de notre mécanisme de sécurité.

Le Chapitre 3 est consacré à l’évaluation de notre solution. Après avoir décrit l’architecture de processeur, nous portons une évaluation sur la sécurité. Cette évaluation permet de souligner les garanties apportées contre les menaces considérées au bénéfice d’une application cryptographique. Ensuite, nous proposons une évaluation du surcoût au niveau des performances et de la surface matérielle engendré par l’implémentation de notre contremesure.

Le dernier Chapitre propose une solution originale en alliant notre mécanisme de sécurité à une solution existante. Cette solution hybride apporte un environnement sain et sécurisé pour

l'ensemble des applications avec des garanties supplémentaires apportées par notre mécanisme. Enfin, une évaluation de cette solution hybride appuie les résultats obtenus lors de l'évaluation du mécanisme seul.

Pour conclure, une synthèse des travaux et des perspectives sont présentées à la fin du manuscrit.

ÉTAT DE L'ART

SOMMAIRE

1.1	Microarchitecture	21
1.1.1	Pipeline	21
1.1.2	Caches	23
1.2	Fuites sur la microarchitecture	26
1.2.1	Analyse des fuites	26
1.2.2	Attaques par canaux auxiliaires temporels	31
1.3	Contremesures existantes	34
1.3.1	Détection des attaques	35
1.3.2	Contremesures basées sur l'ajout d'aléa	37
1.3.3	Contremesures basées sur le partitionnement	39
1.3.4	Autres solutions	41
1.4	Conclusion	42

1.1 Microarchitecture

Il est important de comprendre comment fonctionnent les processeurs et leurs composants microarchitecturaux afin de pouvoir étudier les fuites présentes dans la microarchitecture. C'est pourquoi, cette section présente les principaux composants d'un processeur. Pour finir, nous proposons une description plus approfondie des mémoires caches, un élément incontournable des processeurs modernes.

Au sein d'un processeur, l'unité de contrôle et l'unité arithmétique et logique sont découpées en sous étages qui, une fois insérés les uns à la suite des autres, forment un pipeline allant de 2 étages pour les plus petits processeurs embarqués à plusieurs dizaines d'étages pour les processeurs les plus complexes. La Figure 1.1 présente l'architecture d'un cœur de processeur ayant des caractéristiques proches de celui utilisé dans le cadre de nos travaux. Le pipeline du cœur de processeur, encadré en vert, est composé de 4 étages. Il interagit avec les instructions et les données au travers d'une hiérarchie de mémoires caches composée de deux niveaux via deux interfaces dédiées. Le processeur manipule des mots de données sur 32 bits.

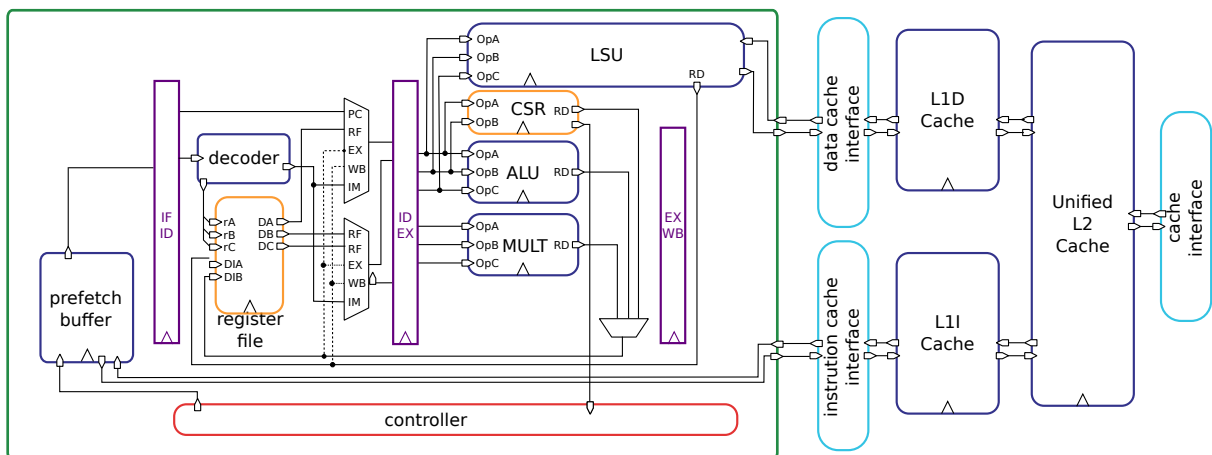


FIGURE 1.1 – Description matérielle des composants microarchitecturaux d'un cœur de processeur embarqué 32 bits à 4 étages et possédant une hiérarchie de mémoires caches (inspiré de [4]).

1.1.1 Pipeline

Notre processeur décrit dans la Figure 1.1 comprend un pipeline de 4 étages avec des missions bien définies pour chacun d'eux. Ils permettent entre autres de rapatrier les instructions, de les décoder, de fournir les variables, d'effectuer des calculs puis de retourner le résultat. Afin d'effectuer ces différentes tâches, chaque étage peut faire appel à des mécanismes d'optimisations pour améliorer les performances.

Instruction Fetch (IF) est l'étage connecté à l'interface d'instruction. Cet étage a la responsabilité de rapatrier les instructions dans le processeur en provenance de la mémoire. Pour ce faire, il utilise le registre PC (*Program Counter*) afin de connaître l'adresse de l'instruction courante. Ainsi, le module de *prefetch* effectue une requête de l'instruction suivante (PC+4) afin d'anticiper l'instruction qui sera exécutée après celle actuellement présente dans l'étage suivant. Le module *compress decoder* permet de transformer les instructions stockées en mémoire sur 16 bits (instructions compressées) en leur équivalent 32 bits afin que la suite du pipeline puisse interpréter les instructions.

Au sein de cet étage, différents mécanismes d'optimisations peuvent être implémentés, il s'agit principalement d'optimisations au niveau du *prefetcher*. Le *prefetcher* effectue des requêtes dont l'adresse est définie de manière plus intelligente en tenant compte de ce qui a déjà été exécuté par le cœur. En effet, lorsque l'instruction est un saut de programme ou un branchement, une table répertorie les issues de ce type d'instructions afin d'émettre une hypothèse sur la prochaine adresse à mettre en requête. Cette optimisation tire profit du fait que les programmes sont amenés à exécuter de manière redondante des blocs d'instructions (*e.g.* au travers de fonctions ou de boucles logicielles). Grâce à ce principe, il est plus intéressant d'émettre une hypothèse probablement vraie plutôt que d'attendre l'issue de chacune des instructions de ce type. Le prédicteur de branchement utilise les tables de Branch History Table (BHT) et de Branch Target Buffer (BTB) qui stockent respectivement un historique des branchements pris et non pris et d'autre part l'adresse ciblée lorsque le branchement est pris. De ce fait, lorsqu'un branchement apparaît, le *prefetcher* se réfère au BHT et au BTB afin d'émettre l'hypothèse de l'adresse de la prochaine requête.

Instruction Decode (ID) est l'étage en charge du décodage des instructions. En particulier, l'étage de décodage détermine, à partir des 32 bits composant l'instruction, quelle est la fonction de l'instruction, quelle est la valeur immédiate encodée et quels sont les registres à manipuler. Ainsi, l'étage ID pilote le banc de registres et les signaux de contrôle permettant d'exécuter l'opération au sein de l'étage suivant. Le banc de registre est une mémoire à faible capacité. Il permet de stocker temporairement les adresses liées à l'exécution du programme, mais également des adresses liées aux données à accéder. Il contient également les données qui attendent d'être traitées suite à une lecture mémoire ou à l'exécution d'une opération.

En tant qu'optimisation, une exécution dans le désordre peut être implémentée afin de contourner les dépendances de données et les dépendances de contrôle.

Execution (EX) est l'étage dans lequel est exécutée l'instruction précédemment décodée. À partir des signaux de contrôle et des signaux de données fournis par l'étage ID, l'étage EX est amené à propager ces signaux vers le module d'opération convenu. La majorité des opérations peuvent être exécutées par l'ALU, cependant des opérations plus spécifiques peuvent utiliser

des unités fonctionnelles dédiées. L'utilisation d'unités fonctionnelles permet, à l'aide d'une seule instruction, d'éviter le traitement de nombreuses instructions pour un résultat identique (*e.g.* opérations de multiplications, de divisions). Des accélérateurs peuvent être dédiés à des opérations relativement courantes (*e.g.* calculs matriciels) mais également à des opérations très spécifiques (*e.g.* applications cryptographiques, traitement du signal).

Dans notre cas, c'est également l'étage EX qui administre les Registres de Contrôles et de Statuts (CSRs) du cœur. À l'aide des CSR, nous pouvons configurer le cœur, la gestion des interruptions, ou par exemple gérer les modes de sécurité. Ces registres permettent d'obtenir un ensemble d'informations issues de l'exécution du cœur (*e.g.* compteur de cycle, identifiant du processus, identifiant du cœur, extensions supportées, etc). Ils permettent également de manipuler les Compteurs de Performances Matériels (HPCs) afin d'obtenir des analyses fines sur l'exécution des applications pour renseigner, par exemple, le système d'exploitation.

Le résultat de l'opération est ensuite retourné vers l'étage ID afin d'écrire sur le résultat dans le banc de registre.

Lorsque l'instruction décodée est un accès mémoire (*i.e.* lecture ou écriture), l'étage EX propage les signaux vers la Load Store Unit (LSU). La LSU supervise les requêtes vers l'interface des données. On peut notamment lui associer une Protection Physique de la Mémoire (PMP) à des fins de sécurité. La PMP permet de configurer des régions mémoires pour des applications spécifiques afin de restreindre la lecture, l'écriture ou l'exécution.

Write Back (WB) est l'étage acheminant le résultat vers le port d'écriture dédié du banc de registre situé à l'étage ID. Dans notre cas, le résultat des opérations exécutées est acheminé depuis l'étage EX. De ce fait, l'étage WB ne sert que pour les lectures mémoires asservies par la LSU.

Au sein du pipeline, plusieurs modules accèdent aux interfaces de données et d'instructions. Ces interfaces sont le lien entre le cœur et la hiérarchie de mémoires caches (partie droite de la Figure 1.1).

1.1.2 Caches

Les mémoires caches sont utilisées pour augmenter les performances des programmes s'exécutant sur le système en exploitant les localités spatiale et temporelle décelées lors des accès mémoire. En effet, les mémoires caches sont placées entre le cœur et la mémoire principale : cette dernière peut être spatialement éloignée (*i.e.* externe à la puce) et nécessite un long temps d'accès. En considérant une hiérarchie mémoire intégrant un seul niveau de cache, si la donnée est présente en cache, la requête est immédiatement traitée (*cache hit*); autrement, la requête est propagée à la mémoire principale (*cache miss*). Le CPU doit alors attendre la réponse de la mémoire principale avec la donnée accédée.

Les données en cache sont stockées sous forme de lignes de cache de B octets plus grande qu'un mot CPU. Afin de trouver une donnée pour une adresse mémoire spécifique, les $\log_2(B)$ bits les moins significatifs de l'adresse déterminent l'emplacement de la donnée accédée au sein de la ligne de cache.

Il existe trois grandes catégories d'architectures de caches : le cache à correspondance, le cache associatif et le cache associatif à N-voie.

Cache à correspondance

Le cache à correspondance (*Direct-mapped cache*) est une implémentation simple où chaque adresse se voit correspondre une ligne de cache en fonction d'une partie de son adresse, appelée index. Ainsi, lorsque le cache à correspondance comprend L lignes de cache, l'index est de taille $\log_2(L)$. Le reste des bits de l'adresse est stocké conjointement afin de déterminer si la ligne de cache accédée correspond à la bonne donnée. Cette partie de l'adresse est appelée tag. Ce type de cache a l'inconvénient de créer énormément de cache miss lorsque le programme accède régulièrement à des adresses correspondant à une même ligne de cache.

Cache associatif

Le cache entièrement associatif (*Fully associative cache*) permet pour une adresse en entrée de se trouver dans n'importe quelle ligne de cache. Une politique de remplacement de cache détermine dans quelle ligne de cache mettre la donnée parmi les L lignes de cache qui composent le cache associatif. Ce type d'architecture a l'avantage de provoquer moins de cache miss que la solution précédente, mais requiert une logique complexe et coûteuse pour chercher une donnée dans le cache. En effet, lors de chaque accès, le cache a besoin de comparer le tag d'adresse conjointement stocké dans les L lignes de caches.

Cache associatif à N-voie

L'architecture de cache la plus populaire est le cache associatif sur N voies (*N-way set associative cache*) et est illustrée dans la Figure 1.2. Ce type d'architecture propose un compromis entre les deux architectures de cache précédemment exposées. Cette architecture de mémoire cache est structurée sous la forme d'une matrice de S ensembles de cache avec N voies. Chaque adresse se voit correspondre un ensemble de cache en utilisant l'index de l'adresse. Dans le reste du manuscrit, un ensemble de cache sera appelé cache set. Une ligne de cache est rattachée à un cache set et peut être placée parmi les N voies qui composent ce set. Une adresse mémoire de A bits est découpée en différents champs : l'offset, l'index d'ensemble, et les bits de tag. Les $\log_2(B)$ bits les moins significatifs sont utilisés en tant qu'offset pour déterminer les octets à manipuler dans la ligne de cache. Les $\log_2(S)$ bits suivants déterminent le cache set ciblé. Et,

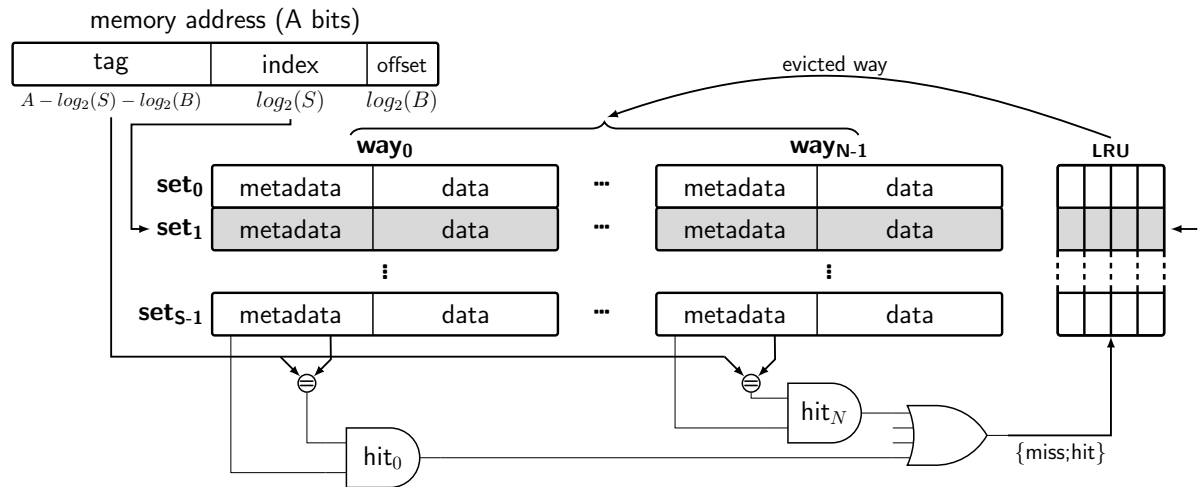


FIGURE 1.2 – Architecture d'un cache associatif à N voies.

les $A - \log_2(S) - \log_2(B)$ bits sont les bits de tag. Ces bits de tag permettent de différencier les différentes adresses se référant au cache set ciblé. Lorsque toutes les voies d'un cache set sont occupées, une politique de remplacement détermine quelle voie évincer afin de mettre en cache la donnée nouvellement accédée. Il est important de noter que la politique de remplacement joue un rôle clé pour les performances. Dans ces travaux de thèse, nous considérons la politique de remplacement Least Recently Used (LRU) qui évince la voie la moins récemment utilisée dans un cache set.

Un cache à correspondance peut être assimilé à un cache associatif à N voies de S cache sets à 1 voie. Enfin, un cache associatif peut être assimilé à un cache associatif à N voies de 1 cache set à N voies.

Hiérarchie de cache

Les différents types de mémoires caches décrits précédemment peuvent être utilisés pour construire une hiérarchie de mémoires caches se plaçant entre le cœur et la mémoire principale. Elle se compose dans la majorité des cas de deux ou trois niveaux de caches. Plus l'on s'élève dans la hiérarchie, plus la taille des caches est grande, plus la densité des transistors est élevée, et plus la latence d'accès est grande. Le premier niveau de cache est composé de deux caches dédiés exclusivement aux instructions (L1I) et aux données (L1D) respectivement. Ils ont généralement une petite capacité, mais permettent d'y effectuer une requête rapidement. Ensuite, le second niveau de cache L2 unifie généralement les instructions et les données, et permet le stockage d'une plus grande quantité que les caches L1.

Il existe différentes politiques d'inclusivité de cache, permettant de garantir une non-contention des données sur tous les étages. Ainsi, dans la majorité des cas, les caches L1I et L1D sont ex-

clusifs avec le cache L2, signifiant qu'une donnée ne peut pas être présente au sein du premier niveau de cache L1 et au sein du L2. Au contraire, le niveau de cache L3 est inclusif avec L1I, L1D et L2, signifiant que toutes les données présentes dans les trois caches de niveaux inférieurs sont présentes au sein du niveau L3. Dans notre cas, nous ne considérons aucune politique de cohérence, car nous ne considérons qu'un seul niveau de cache.

La politique d'écriture décrit le comportement lorsqu'une requête d'écriture survient. En effet, lorsqu'une donnée est modifiée en L1 il existe deux cas de figures pour propager l'écriture au sein de la hiérarchie mémoire. Si le système s'appuie sur la politique *write back*, alors l'écriture n'est effective que sur le cache L1, c'est ensuite lors de l'éviction de la donnée du cache L1 que l'écriture est propagée vers les autres niveaux de la hiérarchie. Si le système se base sur la politique *write through*, alors l'écriture est propagée immédiatement parmi tous les niveaux de la hiérarchie. Dans nos travaux, nous utilisons la politique d'écriture *write through*.

Au sein des architectures multicœurs, les cœurs et leurs hiérarchies de mémoires caches privées sont dupliquées. L'ensemble des caches L2 privés accèdent à un dernier niveau de cache, appelé Last Level Cache (LLC). Le LLC est partagé parmi tous les cœurs. Il est important de définir une politique de cohérence de caches dans les implémentations multicœurs, en effet plusieurs cœurs peuvent utiliser et manipuler une même donnée. La politique de cohérence s'assure que la donnée manipulée est intègre grâce à un signal alertant que la donnée a été mise à jour dans un autre cœur.

1.2 Fuites sur la microarchitecture

Les composants microarchitecturaux constituant l'ensemble d'un cœur de processeur peuvent échanger avec leur environnement. Des événements extérieurs peuvent venir perturber l'exécution du système, mais l'exécution du système peut également engendrer des perturbations sur l'environnement ou l'exécution du système lui-même.

1.2.1 Analyse des fuites

Dans cette section, nous identifions comment l'environnement extérieur peut interagir afin de modifier l'exécution d'un système. Dans un second temps, nous soulignons comment l'exécution de données dans un système peut engendrer des fuites d'informations au niveau de l'environnement extérieur et nous expliquons comment un attaquant peut inférer ces données ainsi que laisser fuiter sciemment des données.

Perturbations extérieures

Des perturbations extérieures peuvent modifier le comportement du circuit. Ces perturbations peuvent être involontaires via des rayonnements cosmiques [5, 6], ou totalement volontaires

dans le but de nuire à l'exécution d'une application. Les perturbations volontaires, également considérées comme des attaques par injections de fautes [7, 8], peuvent être menées par différents moyens en ayant un impact plus ou moins large et précis. Une injection de faute a pour but d'engendrer une modification du comportement du circuit entraînant ou non une modification du comportement du programme s'y exécutant. Par exemple, l'effet recherché de l'injection peut permettre le contournement des instructions activant un mécanisme de sécurité, modifier une donnée présente en registre, ou corrompre l'exécution du système. L'injection de faute peut également se localiser au niveau des mémoires afin de modifier les bits présents en mémoire ou au niveau du cœur de processeur afin de modifier l'état du pipeline. L'ensemble des attaques par injection de fautes requièrent un accès physique à la puce, cependant certaines injections nécessitent un accès physique au silicium et sont catégorisées comme invasives. Pour ce faire, le circuit intégré a besoin d'être décapsulé afin de pouvoir pénétrer physiquement le système. Nous retrouvons par exemple les injections de fautes par laser [9, 10], par sonde ionique [11, 12] où des ions sont injectés, ou bien optiques[13] où un flash lumineux est utilisé. Ensuite, d'autres attaques par injections de fautes sont considérées comme non invasives. Elles nécessitent seulement un accès à la puce sans avoir à l'altérer. Nous retrouvons les injections par rayons X [14], ou par rayonnement électromagnétique [15, 16] pouvant cibler une partie du circuit. Autrement, les injections par modulation du signal d'horloge [17], par modulation de la tension d'alimentation [18], ou par réchauffement du circuit [19, 20] visent le circuit dans sa globalité, même si les effets recherchés peuvent avoir un grain fin. Les injections de fautes peuvent avoir un effet irréversible sur le circuit en le rendant totalement hors d'usage.

Enfin, une dernière attaque possède un statut particulier parmi les attaques par injections de fautes. En effet, elle ne nécessite pas un accès au circuit intégré et ne tire pas profit d'une source extérieure pour injecter une faute. L'attaque RowHammer [21, 22] est menée par le logiciel s'exécutant sur le système en prenant avantage d'un phénomène physique lié aux mémoires DRAM. En accédant répétitivement à une cellule de la mémoire DRAM, RowHammer modifie l'état des cellules adjacentes.

Canaux auxiliaires

Les canaux auxiliaires sont des sources d'informations, provenant de l'exécution du programme dans le système, qui sont liées à l'observation d'effets sur son environnement.

La Figure 1.3 présente les différentes fuites issues du comportement d'un circuit apparaissant lors de l'exécution du programme au sein du système. Ces fuites sont utilisées comme canaux auxiliaires par des attaquants afin de retrouver tout ou partie des données manipulées par le programme s'exécutant sur le système. Bien que ce type de menace permette de récupérer tout type de données, les attaques par canaux auxiliaires sont généralement mises en œuvre pour déduire des valeurs secrètes (*e.g.* clés cryptographiques, mots de passe, etc). Ainsi, en

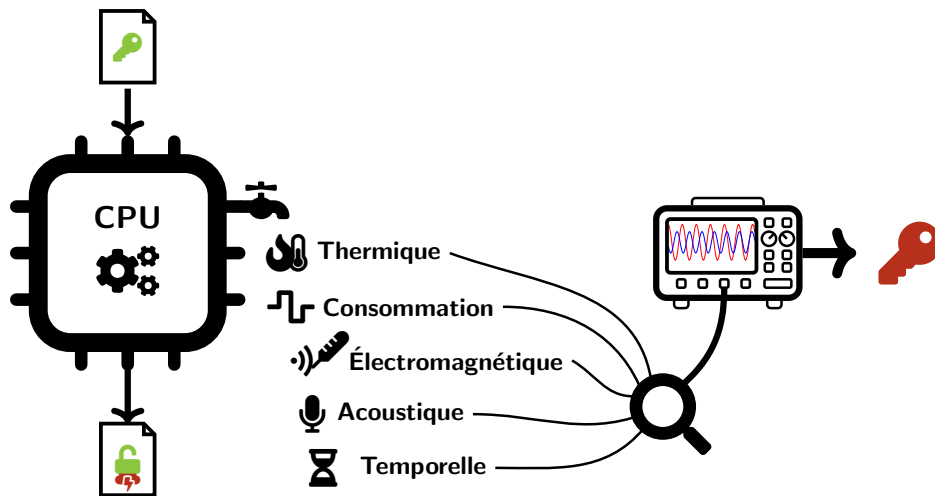


FIGURE 1.3 – Représentation des fuites menant aux canaux auxiliaires.

analysant de multiples traces dont les entrées diffèrent (*i.e.* texte clair, texte chiffré, clé, donnée d'entrée), l'attaquant peut déduire la valeur sensible manipulée par la cible. Nous pouvons également nous servir d'une analyse par canaux auxiliaires pour observer un évènement afin de déclencher une injection de faute. Il existe également un pan d'utilisation dédié à la rétro-ingénierie d'implémentation en boîte noire d'algorithmes de chiffrement [23]. Plus récemment et lié à l'essor de l'Intelligence Artificielle embarquée, les canaux auxiliaires sont également utilisés pour extraire les informations liées aux réseaux de neurones [24] (*i.e.* leurs entrées, leurs architectures, leurs poids).

La première fuite considérée dans la Figure 1.3 est liée à l'émanation thermique induite par l'activation et la désactivation des transistors. En effet, chaque commutation de l'état du transistor consomme de l'énergie qui est en partie libérée sous forme de chaleur. Ces émanations thermiques peuvent ensuite être mesurées à l'aide d'un thermomètre [19] pour, par exemple, identifier le poids de Hamming (*i.e.* nombre de bits à 1) de la donnée exécutée dans un microcontrôleur ATmega162. Les auteurs démontrent que les opérations nécessitant un long temps d'exécution sont propices à ce type d'attaque, par exemple l'exponentiation modulaire présente dans l'algorithme de chiffrement RSA. Une analyse comparative [25] des mesures de températures réalisées sur le Convertisseur Analogique vers Numérique (ADC) embarqué d'une carte de développement, a permis de retrouver la clé de chiffrement d'une implémentation non protégée de RSA. Dans ces mêmes travaux, les auteurs ont réussi à récupérer une partie du secret sur une implémentation protégée de RSA, allant même jusque la clé entière lorsque certains paramètres sont garantis.

En second lieu, les fuites liées à la consommation électrique sont largement connues et ont permis de développer de nombreuses attaques décorrélant la source d'analyse du bruit de consom-

mation observé lors de l'exécution du système. Des techniques basées sur l'analyse de la consommation [26] permettent de déduire les clés de chiffrement dans un objectif de cryptanalyse. La technique Simple Power Analysis (SPA) permet de déduire les opérations effectuées à l'aide d'une analyse individuelle de trace. De nombreux algorithmes de chiffrement sont ciblés dont notamment AES [27] mais également des implémentations supposées sécurisées [28]. La seconde technique se base sur une étude différentielle des traces de consommations, il s'agit de la Differential Power Analysis (DPA) [29]. L'analyse apportée par la DPA permet de se soustraire des limitations de la SPA grâce à l'apport d'études statistiques sur plusieurs milliers de traces de consommation. Cette technique, plus poussée, permet de venir à bout d'algorithmes et d'implémentations plus robustes [30].

Au sein du jeu d'instruction x86 d'Intel, dans [31], les auteurs manipulent les fuites de consommation à l'aide du logiciel grâce à un registre mesurant indirectement la consommation d'énergie du processeur. Ils démontrent l'étendue de leur attaque au travers de nombreuses vulnérabilités dont la récupération de clé RSA enclavée grâce à un attaquant avec privilège. Un attaquant sans privilège peut également retrouver les clés manipulées par les instructions AES-NI, casser le KASLR, ou bien établir une communication avec d'autres applications via un nouveau canal caché.

Tout comme la vulnérabilité précédente pouvant être menée à distance, les attaques par canaux auxiliaires mesurant la consommation électrique sont possibles dans le contexte des fournisseurs de services de FPGA dans le CaaS (Cloud as a Service), ou Faas (FPGA as a Service). Dans ce contexte-là, les FPGA permettent d'implémenter des accélérateurs matériels pilotés par des processeurs. En raison de la présence de multiples clients sur une même puce FPGA, les auteurs de [32] profitent de pouvoir implémenter des capteurs au sein du FPGA afin de mesurer la puissance consommée par un client ayant implémenté un algorithme RSA soit sur le CPU en logiciel, soit sur le FPGA en matériel. Ainsi, l'attaquant est capable de retrouver la clé manipulée par un autre processus.

Les deux sources de fuites précédentes sont engendrées par l'exécution du système dans sa globalité. La fuite se basant sur les émanations ÉlectroMagnétiques (EM) permettent de mesurer les fuites en EM sur une zone précise de la puce. En utilisant des techniques de rétro-ingénierie, il est possible de déterminer les zones dans lesquelles sont implémentées telles ou telles fonctionnalités du processeur (*i.e.* caches, ALU, décodage des instructions, etc). Il est également possible de cartographier l'ensemble de la puce en déplaçant la sonde EM pour chaque exécution de l'application. Ainsi, après avoir judicieusement choisi la zone nous intéressant, nous pouvons extraire les émanations électromagnétiques afin de réaliser une analyse pour récupérer des clés de chiffrement sur DES ou 3DES [33], AES [34], ou des informations sur l'architecture et les paramètres de réseaux de neurones [35].

Plus audacieusement, les canaux auxiliaires utilisant les fuites acoustiques sont réels et permettent d'extraire des clés de chiffrement [36]. Dans [36], les auteurs tirent parti des vibrations induites électromagnétiquement par certains composants électroniques du système. Afin de prouver cette source de bruit, les auteurs ont isolé les autres sources de bruits du système (*e.g.* ventilateurs, disques durs) et les émanations électromagnétiques que le micro pourrait mesurer. Autre source de canal auxiliaire acoustique, l'utilisation d'un périphérique peut être réalisée pour remonter à la source d'un secret. Par exemple, lorsqu'un utilisateur saisit un mot de passe sur un clavier [37]. Cependant, ce dernier canal n'est pas lié à l'exécution du système, mais à l'utilisation externe d'un périphérique par un utilisateur.

En dernier lieu, nous nous concentrons sur les canaux auxiliaires par fuite temporelle [38]. La technique la plus évidente pour tirer profit des fuites temporelles présentes sur le système est de mesurer le temps d'exécution de la victime [39-41]. Il est également possible de tirer profit du comportement des mécanismes d'optimisations et du partage des ressources avec l'ensemble des applications pour que l'exécution d'une application puisse dépendre des autres applications. Ainsi, l'attaquant peut utiliser le prefetcher matériel ou les mémoires caches afin de mener des attaques. Le prefetcher des instructions situé au niveau de l'étage IF du cœur d'un processeur est utilisé pour mener une attaque, notamment lorsque le BHT est sollicité [42, 43]. Concernant les fuites au niveau du cache, la fuite se base sur le temps qu'un accès mémoire va mettre pour être réalisé. En effet, le temps d'accès aux données dépend de l'emplacement de la donnée au sein de la hiérarchie mémoire. De là, sont apparues de nombreuses attaques en cache basées sur la manipulation de données, parmi lesquelles PRIME+PROBE [44] ou FLUSH+RELOAD [45] sont les plus connues. D'autres attaques se basent sur les mécanismes en mémoires caches conçus pour réduire la consommation d'énergie et améliorer les performances [46-48]. Dans [46], les auteurs tirent profit d'un mécanisme prédisant la voie accédée pour éviter la comparaison des N voies composant les caches associatifs à N voies. Concernant [47, 48], les auteurs tirent profit des prefetchers contrôlés respectivement par le logiciel et par le matériel pour mener leurs attaques.

Canaux cachés

Une fois la donnée ciblée ou le secret extrait par l'application attaquante, il faut exfiltrer l'information du système. Les canaux cachés se basent sur deux processus malicieux : un cheval de Troie et un espion. Le cheval de Troie extrait l'information depuis le même système que la cible et peut être introduit tant de manière logicielle [49] que matérielle [50, 51]. Cependant, l'espion n'a pas accès au secret. Les deux applications souhaitent communiquer entre elles afin que le cheval de Troie puisse communiquer le secret à l'espion, mais ne peuvent s'échanger les données de manière conventionnelle, car elles sont isolées l'une de l'autre. Elles vont ainsi utiliser les canaux cachés afin de communiquer le secret. Pour ce faire, le cheval de Troie va produire intentionnellement des fuites par canaux auxiliaires afin de communiquer les informations à l'espion.

Les communications par canaux cachés se basent sur les mêmes fuites que les canaux auxiliaires (*i.e.* en température [52-54], en consommation [31, 54], en émanations électromagnétiques [55, 56], et en temporelles [57-59]). Parmi les canaux temporels, les attaques par canaux axiliaires en caches peuvent être utilisées tant pour inférer des secrets (*i.e.* canaux auxiliaires) que pour les transmettre (*i.e.* canaux cachés). L'émergence des fournisseurs de services engageant un partage de ressources matérielles (*e.g.* serveurs, FaaS) entraîne une augmentation du risque de fuites d'informations par ces canaux.

1.2.2 Attaques par canaux auxiliaires temporels

Les fuites par canaux auxiliaires représentent une source largement exploitable pour retrouver un secret manipulé par un autre processus. Parmi ces canaux, les canaux auxiliaires temporels sont particulièrement puissants grâce à la facilité de mesurer le temps d'exécution sans exiger un accès physique au système. Les attaques par canaux auxiliaires se basant sur les caches sont largement répandues et visent de multiples configurations d'attaques (*e.g.* monocœur, multicœur, L1D, L2, LLC, etc).

TABLE 1.1 – Liste non exhaustive d'attaques par canaux auxiliaires visant les caches (inspirée de [38, 60]).

Attack Variant	Conditions								Type	Target
	ES	IS	SM	SR	HT	AS	CL			
PREFETCHER [47]	○	●	●	●	●	M	S	Control	None priv. checks on prefetch instr. Cache inclusivity + Shared libraries FLUSH+RELOAD optimizations Cache bank collisions NUCA and tile distance	
FLUSH+RELOAD [45]	○	●	●	●	●	D	S			
Amplifying F+R [61]	○	●	●	●	●	D	S			
CACHEBLEED [62]	○	○	●	●	○	D	P			
AotK [63]	○	○	●	●	●	D	S			
PRIME+PROBE [44, 57, 64, 65]	●	○	○	●	●	D	P,S	Eviction	Data eviction Metadata updates Data eviction on randomized cache Port Contention Cache eviction and repetability	
LRULEAKS [66]	●	○	○	●	●	M	P			
PRIME+PRUNE+PROBE [67]	●	○	○	●	●	D	P,S			
WRITE+WRITE [58]	●	○	○	●	●	D	S			
EVICT+TIME [64]	●	○	○	●	●	D	P,S			

ES : Eviction Set ; IS : Instruction Specific ; SM : Shared Memory ; SR : Shared cache Resources ; HT : High precision Timer ; ○ : not required ; ● : required ; ○ : optional ; AS : Aimed State (Data, Metadata) ; CL : Cache Level (Private, Shared, specified otherwise).

La Table 1.1 regroupe un ensemble non exhaustif d'attaques en caches dont les prérequis sont spécifiés. Parmi les attaques présentées, nous apportons une attention particulière sur le type d'attaques que nous considérons dans notre modèle de menace (Section 2.1). Pour chaque attaque, une spécification des conditions permettant de mener à bien l'attaque est fournie. Nous spécifions si l'attaque a besoin d'un ensemble d'éviction (ES), il s'agit d'un ensemble d'adresses permettant de manipuler l'état du cache. Des instructions spécifiques (IS) peuvent être nécessaires pour manipuler les données dans le cache (*e.g.* évincer une donnée ou rapatrier une donnée en avance). Nous spécifions également si l'attaque a besoin de partager l'espace d'adressage (SM) avec la victime. L'attaque peut tirer profit du partage des ressources matérielles (SR)

pour inférer les données. Les compteurs de haute précision (HT) peuvent être nécessaires pour mesurer les temps d'accès au cache ou le temps d'exécution d'un processus. Ensuite, la colonne AS détermine si ce sont les métadonnées (M) ou les données (D) qui sont visées/manipulées par l'attaque. Finalement, la colonne CL spécifie si l'attaque peut être menée sur les caches privés (P) ou partagés (S) de la hiérarchie de mémoires caches. Grâce à ces critères, nous pouvons catégoriser ces attaques parmi celles exploitant le contrôle des données et celles exploitant l'éviction des données. Enfin, la dernière colonne fournit des précisions supplémentaires sur chacune des attaques considérées.

Au travers de cette analyse, nous démontrons que le partage de ressources est la menace principale pesant sur les systèmes lorsque nous considérons ce type d'attaque. L'utilisation de compteur de haute précision facilite le succès des attaques. Par définition, les mémoires caches sont des ressources matérielles partagées entre les applications. Ainsi, l'ensemble des attaques présentées tirent profit des ressources matérielles partagées. Dans un premier cas, les attaques basées sur le contrôle nécessitent un accès partagé aux données. Cela permet de manipuler facilement le cache pour inférer les accès réalisés par une application tierce. Les instructions autorisant la manipulation (*i.e.* suppression `flush` ou rapatriement `prefetch`) facilitent grandement ce type d'attaque. Même lorsque l'espace d'adressage de la victime diffère de celui de l'attaquant, les attaques du type PRIME+PROBE se basent sur un ensemble d'évictions. Ensuite, l'attaque exploite les évictions de lignes de cache provoquées par la victime.

Exploitation du contrôle de données

La première attaque, PREFETCHER [47], tire parti de la non-vérification préalable du niveau de privilège lorsque des instructions de `prefetch` sont exécutées. Cela permet à un attaquant non-privilegié de contourner les protections classiques afin d'accéder à l'espace kernel en se défaisant du KASLR. Ainsi, les données se trouvant en cache dépendent des données privilégiées. C'est la seule attaque de notre sélection à ne pas manipuler directement l'état du cache pour mener son attaque.

Les attaques FLUSH+RELOAD [45] et une de ses améliorations [61] se basent sur l'utilisation de l'instruction `clflush` présente dans le jeu d'instructions x86. Cette instruction permet d'évincer la donnée ciblée du cache. Grâce à l'inclusivité des caches, évincer une donnée à l'aide de `clflush` permet de l'évincer dans ses caches privés, mais également de l'évincer au niveau du LLC et ainsi s'assurer d'être évincée parmi l'ensemble des caches privés du système. L'attaquant peut donc s'assurer qu'aucun cache ne contient un ensemble de données. Ensuite, lorsque la victime accède à des ressources partagées (*e.g.* bibliothèques de chiffrement partagées) dont l'adresse accédée dépend d'un secret. L'attaquant peut finalement accéder et mesurer le temps d'accès aux ressources partagées afin de déterminer ce que la victime a précédemment accédée. Cette attaque permet également de mener des communications par canaux cachés.

CACHEBLEED [62] utilise les collisions des banques de données en caches pour mener ses attaques contre l’algorithme de chiffrement RSA. En accédant répétitivement aux banques de données ciblées, l’attaquant observe une différence sur le temps d’accès lorsque la victime y accède également.

Attack of the Knights [63] s’attaque à des systèmes à haute densité de cœurs dans lesquelles les auteurs exploitent le principe de spatialité des cœurs au niveau des caches non-uniformes.

Exploitation de l’éviction de données

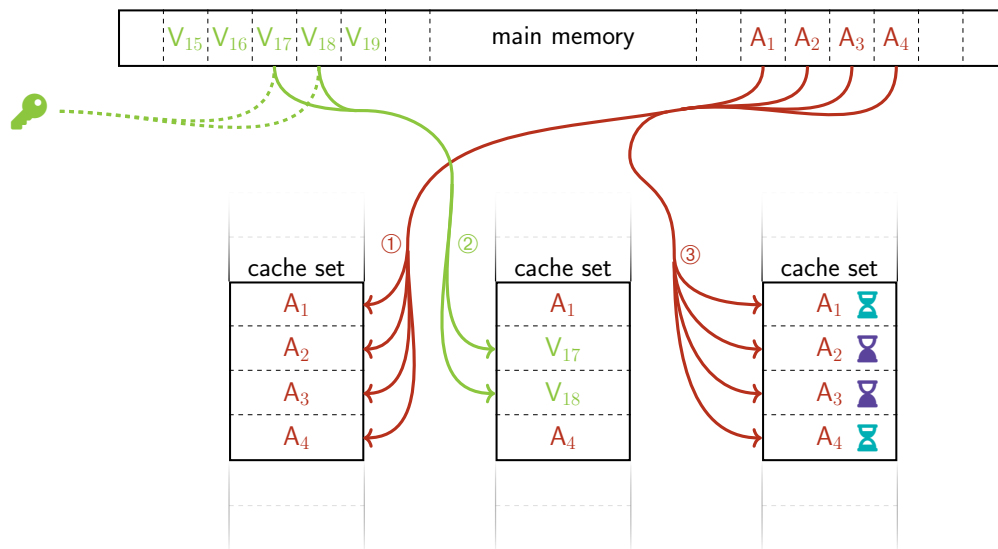


FIGURE 1.4 – Principe de fonctionnement de l’attaque PRIME+PROBE.

Le second type d’attaque se base sur l’éviction de données. Ce type d’attaque ne nécessite pas de partager les données avec la victime (colonne SM de la Table 1.1). L’attaque PRIME+PROBE [57, 64] est l’attaque en cache la plus courante avec de nombreuses déclinaisons pour tirer profit des cibles possibles (*e.g.* L1I, L1D, LLC, BHT, etc). Un exemple pédagogique d’une attaque de type PRIME+PROBE sur un cache set est illustré dans la Figure 1.4. L’attaque PRIME+PROBE se déroule en trois phases. Afin de mener cette attaque, l’attaquant doit composer un ensemble d’éviction s’indexant aux mêmes cache sets que les données de la victime. Une fois l’ensemble d’éviction construit, l’attaquant peut mener la première phase de l’attaque. Il réalise des accès sur son ensemble d’éviction (*i.e.* A₁₋₄) afin remplir le cache set avec ses données. Cela lui permet de connaître l’état du cache à la fin de cette étape et de s’assurer qu’aucune donnée de la victime ne s’y trouve. Ensuite, l’attaquant laisse la victime s’exécuter pour la seconde phase de l’attaque. Dans notre exemple, la victime a réalisé deux accès dépendants d’un secret sur les données V₁₇ et V₁₈ entraînant l’éviction des données A₂ et A₃ de l’attaquant. Lors de la troisième et dernière phase de l’attaque, l’attaquant accède de nouveau à son ensemble

d'éviction dans le but de déterminer le nombre d'évictions entraînées par l'exécution de la victime. Pour ce faire, l'attaquant mesure le temps d'accès de chacune de ses données. Si le temps d'accès est long (*i.e.* cache miss), cela signifie que la donnée a été évincée et nécessite de remonter la hiérarchie mémoire. Autrement, le temps d'accès est court (*i.e.* cache hit), cela signifie que la donnée n'a pas été évincée. Dans notre cas, l'attaquant est capable de déterminer que la victime a évincé deux de ses données. Les attaques PRIME+PROBE peuvent être mises en œuvre facilement et viser nombre d'algorithmes de chiffrement. En effet, tout algorithme s'adressant à la mémoire dépendamment du secret est sujet à ce type d'attaque. Par exemple, les implémentations AES avec tables de substitutions sont victimes de PRIME+PROBE. L'attaque LRULEAKS [66] manipule également un ensemble d'éviction non pas pour attaquer les données comme PRIME+PROBE mais pour inférer les métadonnées des politiques de remplacement.

Il existe également des déclinaisons de PRIME+PROBE pour outrepasser les contremesures. La déclinaison PRIME+PRUNE+PROBE [67] permet de construire un ensemble d'évictions en nécessitant moins d'accès que les techniques classiques de construction d'ensemble d'évictions. PRIME+PRUNE+PROBE permet de rendre considérablement moins efficace les contremesures basées sur l'ajout d'aléa dans les caches (voir Section 1.3). L'attaque WRITE+WRITE [58] permet également de construire un ensemble d'éviction rapidement en profitant de la latence générée lors d'écritures répétées sur le même cache set qu'une autre application.

Finalement, l'attaque EVICT+TIME [64] permet d'inférer de l'information en mesurant le temps d'exécution du programme victime. L'attaquant mesure le temps d'exécution de la victime avant de manipuler le cache. Puis, il mène la phase PRIME de l'attaque PRIME+PROBE, cela a pour effet d'évincer les données de la victime sur un cache set spécifique. Suite à la phase d'éviction, l'attaquant mesure le temps d'exécution du programme victime. Si l'attaquant observe une différence, cela signifie que la victime accède le cache set ciblé par la phase de PRIME. Bien que facile à mener, ce type d'attaque permet d'extraire difficilement des résultats à cause des variations temporelles provenant de l'ensemble de la couche logicielle.

Ces attaques par canaux auxiliaires temporels tirent parti de vulnérabilités introduites par le partage des ressources de caches. Par conséquent, un partitionnement strict des ressources de caches semble être nécessaire afin de contrer ces vulnérabilités. Une solution visant à diffuser les accès mémoires au sein des caches peut également être considéré afin d'éviter la collision entre deux données possédant le même index.

1.3 Contremesures existantes

La section précédente a permis d'identifier les vulnérabilités menant aux attaques par canaux auxiliaires en caches. Dans la suite du manuscrit, nous appellerons attaques en caches les attaques par canaux auxiliaires temporels exploitant les mémoires caches. Dans cette section,

nous étudions les contremesures développées afin de se prémunir des attaques en caches. Il existe différentes stratégies permettant de contrer ces menaces. La Table 1.2 catégorise différentes solutions permettant de détecter ou de contrer les attaques basées sur l’observation du temps d’exécution. Une attention particulière est portée sur les contremesures contre les attaques en caches. Au sein de cette table, les contremesures sont catégorisées en trois groupes principaux (*i.e.* Détection, Ajout d’aléa et Partitionnement). Un quatrième groupe englobe des solutions originales ne rentrant dans aucune des catégories précédemment énoncées. Au travers de chaque catégorie, chaque solution est assignée à une sous-catégorie permettant d’affiner les spécifications de la contremesure. Les spécifications sont enrichies par une courte description pour chacune d’elles. Lorsque renseignée, la colonne *Target* identifie la cible sur laquelle la contremesure est implémentée : *Prog* pour une contremesure logicielle, *LLC* pour une architecture de mémoire cache partagée, *Pr.C* pour une architecture de mémoire basée sur un cache privé. Le champ *Exec* est un cas particulier où la contremesure permet de détecter l’attaque pendant l’exécution du programme. Ensuite, la colonne *Support* renseigne quel type de support est nécessaire pour implémenter la contremesure (*i.e.* *Soft* logiciel et/ou *Hard* matériel). Finalement, la dernière colonne *Mitigations* renseigne le type de menace contre lequel la contremesure se prémunit. La colonne se décompose en trois critères : *F-b* correspond à la menace des attaques se basant sur les instructions `flush` ; *T-d* correspond aux attaques monitorant le temps d’exécution de la victime ; et *E-b* correspond aux attaques analysant les évictions en caches.

Nous nous concentrons à la fois sur des solutions réalisées sur des processeurs à hautes performances et sur des systèmes à ressources contraintes afin d’avoir une vue d’ensemble des solutions proposées. À la suite de cette analyse, nous pourrions identifier quels types de solutions mettre en place afin de protéger notre système embarqué contre les attaques par canaux auxiliaires temporels.

1.3.1 Détection des attaques

Plusieurs approches peuvent être mises en œuvre afin de détecter des attaques. Tout d’abord, au moment de la conception des programmes, des outils analysent les fuites temporelles présentes afin d’appliquer des contremesures lors de l’étape de la compilation. Ensuite, des outils d’analyse statique étudient le programme compilé afin de détecter les fuites au niveau des accès mémoires. Enfin, pendant l’exécution, des outils permettent de détecter la réalisation d’une attaque par un autre processus. Dans la suite, nous présentons ces différentes approches telles que présentées dans la Table 1.2.

Parmi les outils de détection intervenant à l’étape de la compilation, Winderix *et al.* [68] proposent d’analyser le flot de contrôle afin d’y appliquer des correctifs pour rendre l’exécution du programme en temps constant. Wu *et al.* [69] ajoutent une analyse des accès mémoire afin d’affiner l’exécution en temps constant. Ces techniques modifient uniquement le programme

TABLE 1.2 – Liste non exhaustive de contremesures logicielles et matérielles permettant de se prémunir des attaques en caches.

	Specification		Countermeasure	Target	Support		Mitigations		
					Soft	Hard	F-b	T-d	E-b
Detection	Constant time program	Control flow	Winderix <i>et al.</i> [68]	Prog	●	○	○	●	○
		Data & Control flow	Wu <i>et al.</i> [69]	Prog	●	○	○	●	○
	Code analyzing	Against cache-SCA	CacheAudit [70]	Prog	●	○	○	●	●
		Infer F+R, P+P patterns	CacheBar [71]	Prog	●	○	●	○	●
HPCs based	Runtime using ML	Nights-Watch [72]	Exec	●	○	●	○	●	
	Runtime using combined MLs	WHISPER [73]	Exec	●	○	●	○	●	
	Self execution HPCs	CacheShield [74]	Exec	●	○	●	○	●	
	Hardware monitoring	tākō [75]	Exec	●	●	○	○	●	
Randomization	Set associative	Permutation table	RPCache [76]		○	●	○	●	●
		Index Derivation Function	CEASER [77]		○	●	○	●	●
		Localized and dynamic	PhantomCache [78]		○	●	○	●	●
		Dynamic indexing	ScrambleCache [79]		○	●	○	●	●
		Procedural indexing	Song <i>et al.</i> [80]		○	●	○	●	●
Skewed	Keyed IDF	CEASER-S [81]		○	●	○	●	●	
	Keyed IDF	ScatterCache [82]		○	●	○	●	●	
	Introducing Time To Live	ClepsydraCache [83]		○	●	●	●	●	
	Dynamic Dyn. and core-dependant tags	IE-Cache [84] RECAST [85]	LLC	○	●	○	●	●	
Increased assoc.	Dynamic indexing	RollingCache [86]		○	●	○	●	●	
Partitioning	Temporal	<code>fence.t</code> for OoO multicores	<code>fence.t.s</code> [87]		●	●	○	●	○
		Mitigate hidden channels	Under the dome [88]		●	●	○	●	○
	Coarse grained	Page coloring	Chameleon [89]	LLC	●	○	●	○	●
		Page coloring	COLORIS [90]	LLC	●	○	●	○	●
		Lock memory pages	STEALTHMEM [91]	LLC	●	○	●	●	●
		OS-level monitoring	COTSknight [92]	LLC	●	●	●	○	●
		Way	NoMoCache [93]	Pr.C	○	●	○	●	●
		Dynamic way	SecDCP [94]	LLC	○	●	●	●	●
		Dynamic way	FairSDP [95]	LLC	○	●	●	●	●
	Fine grained	Reserve ways	HybCache [96]		●	●	●	○	●
Dynamic way		Vantage [97]	LLC	○	●	●	●	●	
Reserve cache line		PLCache [76]	Pr.C	●	●	○	●	●	
Other	Buffer last evictions	TreasureCache [98]	LLC	○	●	●	○	●	
	Preload secret using HTM	Cloak [99]		●	●	●	●	●	
	Hardware-prefetcher	Fang <i>et al.</i> [100]		●	●	●	○	●	

Mitigations : F-b : flush based, T-d : Time driven, E-b : Eviction based ; ○ : none ; ● : fully ; ● : potentially.

binaire, mais ne permettent pas de contrer les attaques en caches. Cependant, ces techniques de programmation en temps constant éliminent une des multiples sources de fuites temporelles pouvant être analysées par un attaquant afin de déduire un secret.

Ensuite, CacheAudit [70] et CacheBar [71] réalisent une analyse statique du programme afin de catégoriser les fuites au niveau des accès mémoires. Ces outils permettent d’avertir lorsque des accès mémoire peuvent être sujets à des attaques de manipulation de données ou à des attaques menées par l’observation du temps d’exécution. Ils peuvent ainsi garantir que le programme est protégé contre ces menaces. Cependant, le résultat obtenu via ce type d’outils dépend du modèle matériel et du modèle de menaces considérés. Ces analyses peuvent devenir obsolètes si les attaques récemment découvertes ne sont pas prises en compte dans le modèle.

Concernant les outils de détections basés sur les métriques issues de l'exécution d'un programme sur le système. Lorsqu'un évènement survient, le HPC dédié à l'évènement s'incrémente. De nombreux HPCs sont implémentés pour correspondre à plusieurs évènements (*i.e.* branchements pris, sauts de programme, lecture mémoire, cache miss, instructions exécutées, etc). Nights-Watch [72] propose plusieurs modèles de *Machine Learning* analysant en temps réel les données des HPCs dans le but de détecter des attaques en caches. Les mêmes auteurs proposent WHISPER [73] afin de combiner plusieurs modèles de *Machine Learning* pour avoir une analyse plus fine et obtenir un meilleur taux de détection. Tandis que Nights-Watch et WHISPER s'intéressent aux HPCs considérant les évènements de l'ensemble du système. CacheShield [74] ne s'intéresse quant à lui qu'aux évènements du programme à protéger. Ces outils de détections ciblent la protection des systèmes de chiffrements tels que RSA, AES, ou de signatures ECDSA. Enfin, tākō [75] propose d'étendre le contrat matériel/logiciel pour rendre la détection d'attaques en caches plus facile. En effet, les auteurs étendent les mémoires caches d'un module matériel dont le rôle est de surveiller les accès mémoires afin de détecter des motifs d'accès suspects pour en informer les programmes.

1.3.2 Contremesures basées sur l'ajout d'aléa

La seconde catégorie de contremesures regroupe des solutions permettant de contrer les attaques en caches grâce à l'insertion d'imprédictibilité. Cela a pour effet de diffuser les accès mémoires dans le cache afin de rendre plus difficile la construction d'un ensemble d'éviction. En effet, un cache associatif à N voies utilise une partie de l'adresse pour indexer le cache set. Grâce au comportement déterministe, il est très facile pour un attaquant de construire un ensemble d'évictions rentrant en collision avec les données de la victime, car il suffit de faire coïncider les champs d'index des adresses de son ensemble d'éviction avec ceux des données de la victime.

Permutation d'index

Ainsi, l'idée d'éviter les collisions lorsque deux indexs sont identiques semble intéressante pour éviter qu'un attaquant puisse construire un ensemble d'évictions. Cette technique doit respecter deux propriétés : a) déterministe : le même cache set est accédé pour tout accès avec une adresse mémoire identique ; b) diffusion : pour deux adresses différentes, dont les indexs correspondent, les cache sets accédés doivent être différents. Ce type de solution se basant sur la réindexation est illustré dans la Figure 1.5a dans laquelle le cache set accédé diffère de l'index provenant de l'adresse originelle (présentée en gris).

RPcache [76] utilise une table de permutation implémentée matériellement pour transformer l'index. Le défaut de cette implémentation est le manque de modularité pour renouveler l'aléa de la table de permutation. Au contraire, CEASER [77] utilise une fonction de dérivation d'index (IDF) afin d'assigner un nouvel index. La fonction réside en l'utilisation de l'adresse et d'une clé.

Ces implémentations augmentent considérablement le nombre d'accès nécessaires pour construire un ensemble d'éviction. Cependant, l'attaque PRIME+PRUNE+PROBE [67] propose une nouvelle méthodologie afin de se soustraire de cette couche de diffusion pour construire un ensemble d'éviction. Dans le but de maintenir un niveau de sécurité suffisant, les caches de ce type doivent donc actualiser leur fonction d'indexation pour rendre obsolète leur nouvel ensemble d'éviction. Concernant RCache, il faudrait que la permutation dépende d'une graine (seed) pouvant être renouvelée. Au sujet de CEASER, il suffit de renouveler la clé régulièrement. Ces opérations de renouvellement de l'aléa ont un coût important sur les performances puisque l'entièreté du cache doit être invalidée.

PhantomCache [78] propose une solution plus originale en assignant un groupe de cache sets et permet de mettre en difficulté PRIME+PRUNE+PROBE. Après avoir réindexé tel que réalisé dans CEASER, PhantomCache considère plusieurs cache sets au sein du voisinage de l'index. Cela permet d'augmenter artificiellement l'associativité du cache et d'augmenter la diffusion d'une adresse. ScrambleCache [79] automatise le renouvellement de la table de permutation en utilisant un générateur de nombre pseudo-aléatoire. Afin de ne pas subir de pertes de performances, ScrambleCache archive les précédentes permutations. Cette solution a pour avantage de renouveler dynamiquement la table de permutation, mais a l'inconvénient de manipuler toutes les permutations archivées. Song *et al.* [80] proposent une indexation dynamique basée sur la permutation. Lorsqu'un pattern d'attaque est identifié, le cache fait évoluer automatiquement sa fonction de permutation.

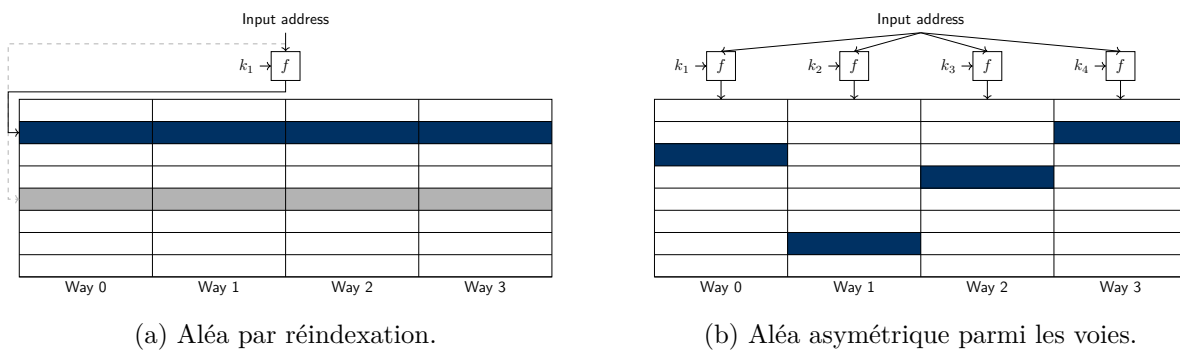


FIGURE 1.5 – Schémas de principe d'architectures de caches basées sur l'aléa.

Voies asymétriques

Une seconde sous-catégorie se base sur une gestion asymétrique des voies pour diffuser les accès mémoires au sein du cache. Ce type de solution est illustré dans la Figure 1.5b. Sez nec [101] énonce le principe dont le but est de dissocier l'indexation de chaque voie, ou de chaque groupe de voies. CEASER-S [81] et ScatterCache [82] proposent une réindexation basée sur des fonctions

IDF de la même manière que CEASER mais avec une gestion individuelle de chaque voie. À l'image de CEASER, l'approche PRIME+PRUNE+PROBE permet néanmoins de réduire la complexité de la construction d'un ensemble d'éviction. C'est pourquoi de nouvelles contremesures se sont inspirées de l'architecture proposée par CEASER-S [81] et ScatterCache [82] dans le but de prévenir l'approche de PRIME+PRUNE+PROBE. ClepsydraCache [83] suggère une utilisation novatrice des principes de la physique de l'électronique afin d'introduire le TimeToLive. Dans cette architecture de cache, le principe du TimeToLive réside dans le fait d'invalider une ligne de cache lorsqu'elle n'est pas accédée pendant un certain temps. Pour ce faire, les auteurs proposent de joindre une capacité à chaque ligne du cache. La charge de la capacité diminue avec le temps et indique implicitement le temps restant. Lors de chaque accès à la ligne en cache, la capacité se recharge afin d'accumuler de la charge. Une fois vidée de sa charge, la capacité invalide la ligne de cache. Ce principe novateur exploitant la physique des composants est facilement implémentable sur Application-Specific Integrated Circuit (ASIC) et permet de contrer les attaques du type PRIME+PRUNE+PROBE. IE-Cache [84] propose d'étendre le lien entre l'adresse accédée et la ligne de cache à évincer afin de contrer les attaques du type PRIME+PROBE. Enfin, RECAST [85] joint à la ligne de cache un tag généré pseudo-aléatoirement propre au cœur afin de contrer les attaques visant les caches partagés.

Finalement, RollingCache [86] propose d'indexer dynamiquement le cache set en le sélectionnant parmi une chaîne complexe de cache sets pointant les uns vers les autres. Ce principe permet de séparer physiquement les accès consécutifs en décalant l'associativité pour contrer les attaques en cache comme PRIME+PROBE et PRIME+PRUNE+PROBE.

La plupart de ces solutions utilisent une IDF pour indexer un cache set ou les voies dans le cas des caches à voies asymétriques. Les IDF peuvent utiliser des algorithmes de chiffrement par bloc qui se veulent être peu coûteux et à faible latence (*e.g.* PRESENT [102], MANTIS [103], QARMA [104], SCARF [105]). SCARF a l'avantage de regrouper ces propriétés en étant dédié pour cet usage d'IDF.

1.3.3 Contremesures basées sur le partitionnement

La troisième catégorie regroupe des solutions basées sur le partitionnement. Dans un premier temps, des solutions basées sur le partitionnement temporel sont étudiées, puis dans un second temps sur le partitionnement des ressources matérielles à différents niveaux de granularité.

Under the dome [88] et `fence.t.s` [87] proposent d'étendre le jeu d'instruction afin d'isoler à minima temporellement les applications. Ces instructions ont pour vocation d'être manipulées par le système d'exploitation, tout comme les quatre prochaines solutions basées sur le partitionnement logiciel. En effet, Chameleon [89] et COLORIS [90] proposent de gérer dynamiquement les pages mémoires en leur attribuant des zones d'adressages de mémoires virtuelles. Ces solutions attribuent de manière implicite des couleurs aux zones mémoires. Ensuite, des règles

définies par le système d'exploitation autorisent ou restreignent les interactions entre les zones de différentes couleurs. STEALTHMEM [91] verrouille virtuellement des pages mémoires au sein du LLC. Cela empêche les autres applications d'évincer les pages protégées par STEALTHMEM. COTSknight [92] surveille les accès mémoires au cache partagé pour réserver à la volée des voies pour une application. COTSknight protège contre les attaques du type PRIME+PROBE.

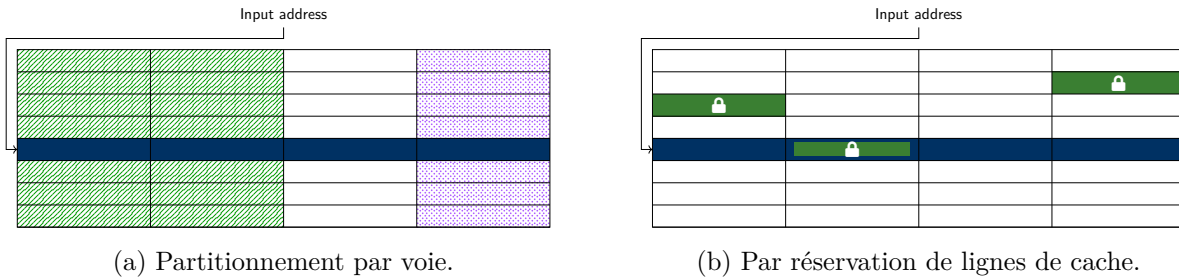


FIGURE 1.6 – Schémas de principe d'architectures de caches basées sur le partitionnement.

Ensuite, plusieurs solutions s'appuient sur un partitionnement par voie. Ce type de contre-mesure est illustré dans la Figure 1.6a. L'idée réside dans l'attribution d'une ou plusieurs voies à une application ou un groupe d'applications afin d'en isoler les données en cache. En se basant sur l'illustration, les accès mémoires d'une application quelconque ne pourra pas interagir avec les données présentes dans les voies 0 et 1 de la partition verte.

NoMocache [93] propose une architecture matérielle dans laquelle le cache est partitionné en plusieurs groupes de voies. Chaque groupe de voies est attribué à un thread du cœur au sein du cache L1. Cela permet de garantir qu'au moins une voie est disponible pour chaque thread. On pourrait imaginer étendre ce principe non pas à des threads physiques, mais aux applications qui s'y exécutent. Par exemple, SecDCP [94] et FairSDP [95] attribuent dynamiquement des partitions de voies à des groupes d'applications en fonction de leur niveau de sécurité. FairSDP [95] manipule des tags pour chacune des applications sécurisées afin de fournir une partition adaptée en fonction de l'occupation globale du cache et des besoins de l'application. Les deux solutions proposent une architecture de cache pour le LLC.

D'autres solutions proposent de partitionner le cache avec une granularité plus fine que les solutions précédentes. HybCache [96] attribue une faible portion des voies afin d'isoler les accès mémoires des applications sécurisées. Vantage [97] propose également de réserver des voies aux applications de manière dynamique en s'ajustant au plus près des besoins des applications.

Enfin, PLcache [76] propose de partitionner le cache au niveau de la plus petite partition possible (*i.e.* la ligne de cache), tel qu'illustré dans la Figure 1.6b. PLcache introduit de nouvelles instructions permettant de verrouiller et de déverrouiller une ligne de cache. Comme étudié plus tard dans ce manuscrit, il s'agit plutôt d'une réservation et d'une libération de lignes de cache. Cependant, une application tierce ne peut, en aucun cas, remplacer une ligne de cache

verrouillée. Cette solution, introduite par le biais d'une extension matérielle et logicielle, permet à l'utilisateur de s'octroyer le nombre de lignes de cache dont il a besoin, puis de les libérer dès lors qu'il a terminé son exécution.

Le partitionnement est une technique redoutable pour protéger contre les attaques en caches grâce à l'isolement des ressources matérielles. Cependant, elles peuvent nécessiter la manipulation du logiciel pour isoler efficacement les données. Également, en fonction de la granularité et des besoins de chaque application, les performances peuvent être grandement impactées.

1.3.4 Autres solutions

D'autres solutions ne rentrant dans aucune des catégories précédentes sont proposées pour prévenir les attaques en caches. Par exemple, TreasureCache [98] propose de stocker temporairement les dernières lignes de cache évincées dans une petite mémoire tampon. Cela permet de prévenir les attaques basées sur l'éviction en stockant les dernières données évincées dans la mémoire. Lors de l'exécution normale, cette architecture permet d'améliorer les performances grâce à l'augmentation du nombre de données pouvant être mises en cache. Par ailleurs, Cloak [99] et Fang *et al.* [100] proposent de détourner l'utilisation d'un mécanisme d'optimisation afin de prévenir les attaques en caches. Ils utilisent le prefetcher matériel afin de rapatrier les données en cache et éviter qu'un attaquant puisse inférer leur état au sein du cache.

Dans le cadre du projet SCRATCHS, nous souhaitons porter une solution permettant de se prémunir des attaques par canaux auxiliaires temporels exploitant les mémoires caches au sein d'un système embarqué. Ainsi, la solution retenue doit tenir compte des contraintes appliquées sur ce type de systèmes. De plus, nous visons une solution au niveau du contrat entre le logiciel et le matériel pour apporter des garanties de sécurité en engendrant de faibles surcoûts dans chaque partie du contrat. Dans un premier temps, les contremesures basées sur l'ajout d'aléa au sein des architectures de caches proposent de permuter l'index. Cependant, cette approche souffre de problème de maintien de sécurité en ayant besoin de renouveler régulièrement la permutation. D'autres approches permettent d'étendre le temps entre deux renouvellements de l'aléa grâce à une gestion asymétriques des voies et en utilisant les blocs de chiffrements légers à la place de simples permutations. L'impact en performance du renouvellement d'aléa est important. Cependant, ce type de solution ne propose pas une redéfinition du contrat matériel/logiciel. Par ailleurs, ce type de solution propose un même niveau de sécurité pour l'ensemble des applications quels que soient leurs besoins. D'autre part, des solutions partitionnent les ressources matérielles à gros grain en impliquant des baisses de performances non négligeables pour l'ensemble des applications. Dans le cadre du projet SCRATCHS, nous proposons un partitionnement à grain fin permettant la réservation de lignes de cache par une application. Cette réservation est réalisée par l'application au travers d'une extension du contrat matériel/logiciel supportée par de nouvelles

instructions. La solution mise en œuvre doit engendrer un faible impact sur les performances et la surface lors de son implémentation.

1.4 Conclusion

Dans ce chapitre, nous avons étudié le fonctionnement d'un cœur de processeur. Cela nous a permis de souligner que les optimisations architecturales créent des variations dans l'exécution du programme. Grâce aux canaux auxiliaires, un attaquant peut observer ces variations par différents canaux pour inférer un secret. En se concentrant sur les attaques par canaux auxiliaires temporelles, nous soulignons que le temps d'exécution des instructions et les mémoires caches sont des éléments clés menant à ces fuites. Pour atténuer les fuites temporelles sur le temps d'exécution des instructions, une solution au niveau de la chaîne de compilation suffit. D'autre part, les attaques par canaux auxiliaires exploitant les mémoires caches tirent profit des ressources matérielles partagées entre les applications. Ainsi, des contremesures basées sur l'ajout d'aléa diffusent les accès et rendent les ensembles d'évictions obsolètes. Cependant, pour maintenir un niveau de sécurité suffisant, ce type de contremesures nécessite de mettre à jour régulièrement l'aléa. Cette opération très coûteuse nous amène vers des contremesures basées sur le partitionnement. Le partitionnement consiste à séparer physiquement les applications. Cependant, des baisses de performances élevées sont observées pour des partitionnements à gros grains. De ce fait, dans la suite de ce manuscrit, nous nous concentrons sur une solution à grain fin permettant de partitionner dynamiquement les ressources des mémoires caches.

MÉCANISME DE VERROUILLAGE DE LIGNES AU SEIN D'UN CACHE DE DONNÉES

SOMMAIRE

2.1	Modèle de menaces	45
2.2	Mécanisme de verrouillage des données au sein du cache	46
2.2.1	Motivations	46
2.2.2	Approche proposée	48
2.3	Solution d'implémentation Matérielle/Logicielle	50
2.3.1	Analyse des avantages et inconvénients	50
2.3.2	Extension du jeu d'instructions	53
2.3.3	Support matériel	55
2.3.4	Support logiciel	56
2.3.5	Spécification des instructions <code>lock</code> et <code>unlock</code>	56
2.4	Conclusion	58

Après avoir discuté différentes contremesures permettant de se prémunir contre des attaques exploitant des failles présentes au sein de la microarchitecture, nous introduisons le modèle de menaces qui s'applique aux systèmes considérés dans ces travaux. Par la suite, nous spécifions les principes de notre mécanisme de sécurité afin de répondre à notre modèle de menaces. Enfin, les spécifications des nouvelles instructions sont définies. Ce chapitre reprend principalement les éléments de notre contribution publiée au workshop SILM co-localisé avec la conférence EuroS&P [106].

2.1 Modèle de menaces

Nous ciblons un modèle de menaces lié au monde des systèmes embarqués dans lequel un attaquant peut exécuter son propre code sur le système. Le but de l'attaquant est de récupérer des informations sur l'exécution d'applications tierces, considérées comme victimes, au travers de canaux auxiliaires utilisant les fuites temporelles du système. Nous ne nous focalisons pas sur la méthode utilisée pour extraire les informations déduites lors des attaques.

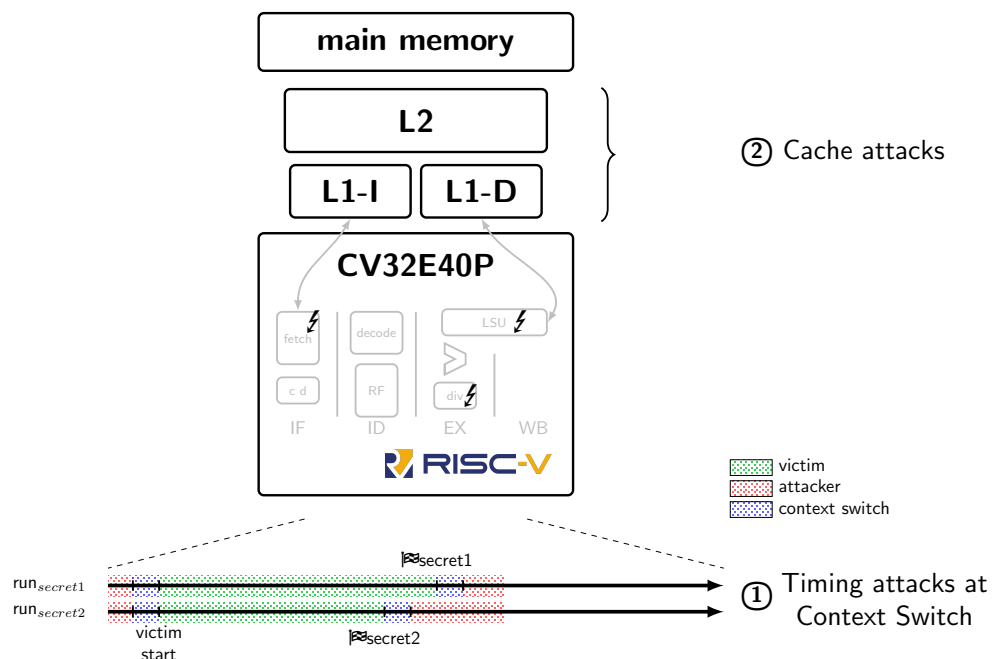


FIGURE 2.1 – Modèle de menaces.

Au travers de [107], les auteurs démontrent que les attaques par canaux auxiliaires temporels ciblant les mémoires caches peuvent être exploitées sur des processeurs monocœurs dans le but d'extraire des informations sensibles. Comme nous pouvons le voir dans la Figure 2.1, dans ces travaux, nous faisons l'hypothèse que deux applications s'exécutent sur un processeur monocœur avec une exécution dans l'ordre, lequel accède à une hiérarchie mémoire composée d'un niveau de cache partagé dédié aux données et d'une mémoire principale où sont stockées les instructions et les données. Ces applications ont des espaces mémoires non partagés. L'attaquant a connaissance du binaire en langage machine associé au programme de la victime (*e.g.* grâce aux bibliothèques cryptographiques en source ouverte [108, 109] ou grâce à la rétro-ingénierie [110-112]). L'attaquant peut inférer l'état du cache avant et après l'exécution de la victime, mais également après chaque changement de contexte. De plus, l'attaquant peut mesurer précisément le temps grâce aux registres de compteur de cycles du processeur, et ainsi déterminer si ses accès mémoire résultent en un cache hit ou un cache miss ②. Il peut également mesurer le temps d'exécution

de la victime ①. L'attaquant est également capable de se synchroniser avec l'application victime pour déclencher son exécution.

L'entièreté de la pile logicielle en charge de l'ordonnancement des applications est de confiance. Nous supposons que le flot de contrôle du programme de la victime ne laisse pas fuir d'information sensible (*e.g.* en utilisant des techniques de programmation en temps constant [68, 113]). Ainsi, seuls les accès à la mémoire liés aux données peuvent être dépendants d'un secret. Par conséquent, l'attaquant se concentre sur des scénarios basés sur la contention de données, telle que l'attaque `PRIME+PROBE` permettant de manipuler l'état du cache de données. Cette attaque est utilisée afin de tenter d'inférer des informations à propos des accès dépendants du secret de la victime. L'attaquant utilise également l'attaque `EVICT+TIME` pour tenter d'inférer des informations grâce aux temps provenant de multiples exécutions dans lesquelles l'attaquant modifie l'état du cache.

2.2 Mécanisme de verrouillage des données au sein du cache


Dans cette section, nous énonçons les principes de notre mécanisme de sécurité permettant de contrer des attaques en caches face au modèle de menaces précédemment énoncé. Nous entamons notre approche par une analyse de `PLcache` [76]. Cette analyse nous permet de souligner les limitations vis-à-vis de notre modèle de menaces et de motiver notre solution. Ensuite, nous spécifions les nouvelles instructions supports au mécanisme que nous proposons.

2.2.1 Motivations

L'utilisation du mécanisme de verrouillage est motivé par le besoin d'optimiser l'usage du cache tout en protégeant une part limitée de l'espace mémoire sur une période temporelle également limitée. Pour un développeur logiciel, la motivation réside dans le verrouillage d'un ensemble de données dans la mémoire cache dont les accès dépendent d'un secret. Bien que le retrait d'une partie du cache résulte en une baisse des performances pour l'exécution des processus concurrents, cette approche à la demande permet aux processus sécurisés de libérer les lignes de cache verrouillées après l'exécution de leurs sections de programmes critiques pour ensuite continuer leurs exécutions.

`PLcache` [76] propose une solution basée sur le partitionnement à grain fin au travers de deux instructions (`lock` et `unlock`). Cependant, `PLcache` souffre de lacunes qui ont été mises en évidence dans la première contribution du projet `SCRATCHS` [106] où, en plus des problèmes de sécurité démontrés dans [66], la spécification des instructions ne propose pas des garanties suffisantes pour protéger les accès réalisés sur un ensemble de données, supposés être secrets.

Dans le but d'illustrer les cas liés aux spécifications et à l'utilisation de `PLcache`, un cas d'étude est développé. Le cas d'étude consiste en une évaluation temporelle de l'état d'un cache

set composé de 4 voies. L'évaluation est réalisée sur 5 accès mémoires adressés par deux applications **A** et **B**. Cette étude est représentée dans la Figure 2.2. Les accès mémoires sont colorés en fonction de l'application l'exécutant et sont de type `lock`, `unlock` ou `load`. Chaque sous-ensemble (I à VI) représente l'état du cache avant et après l'exécution de l'instruction considérée. Ainsi, nous pouvons observer l'état de chaque voie (*i.e.* `wX` si la voie `X` est déverrouillée et  si la voie est verrouillée). Puis, l'état de la politique de remplacement (*e.g.* 4 pour la voie la moins récemment utilisée, et 1 pour la voie la plus récemment utilisée), un état **rouge** signifie qu'il fût mis à jour au travers de l'exécution précédente. La politique de remplacement LRU est considérée dans ce cas d'étude sur Plcache. Finalement, les données présentes dans chacune des voies sont colorées selon leur application de provenance.

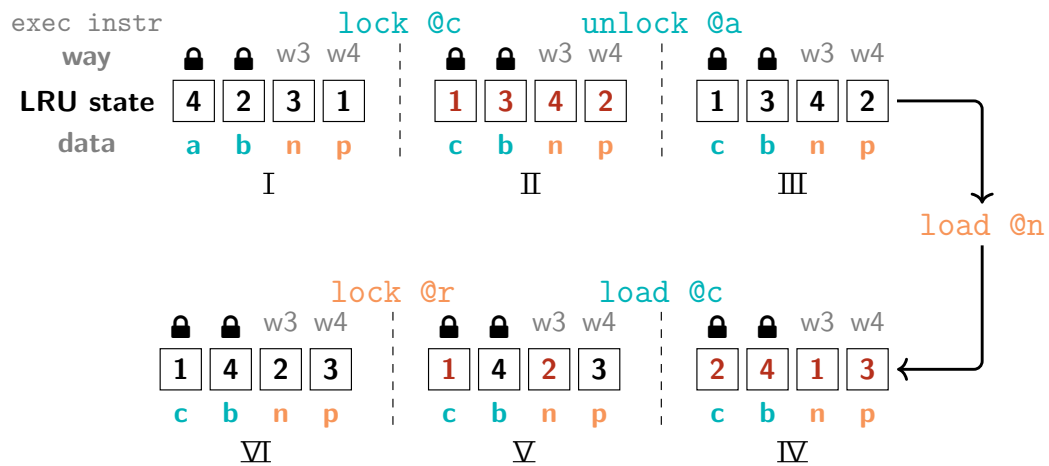


FIGURE 2.2 – Cas d'étude des failles induites par l'utilisation de PLcache [76].

Dans les faits, lors de l'exécution d'une instruction `lock`, PLcache propose de verrouiller la ligne de cache accédée et de l'assigner à l'application ayant fait la requête. Or, lors d'un accès mémoire résultant en un cache miss, si la politique de remplacement pointe vers la voie de la ligne de cache verrouillée appartenant à la même application, alors la donnée verrouillée est évincée au profit de la donnée nouvellement accédée (cas I à II de la Figure 2.2). Ainsi, tout futur accès à une donnée originellement verrouillée ne peut garantir que la donnée est présente en cache. Ce comportement entraîne une fuite d'information, étant donné que la donnée peut dorénavant se retrouver en cache dans une voie que l'attaquant peut manipuler. L'instruction de `lock` proposée par PLcache relève plutôt d'une réservation de ligne de cache pour une application que d'un réel verrouillage de données en cache.

D'autant plus, suite à son éviction, l'instruction `unlock` permettant de déverrouiller la donnée du cache n'aboutira pas, parce que l'adresse ciblée n'est plus présente dans la ligne de cache où elle fût verrouillée (cas II à III de la Figure 2.2). Au fur et à mesure de l'exécution d'applica-

tions utilisant le mécanisme de verrouillage de PLcache, les performances des applications vont s'amenuiser suite à la contention de ressources verrouillées « fantômes ».

En plus de la faille de sécurité pouvant conduire une donnée supposée sécurisée dans un espace manipulé par autrui, la proposition de PLcache souffre d'une autre faille de sécurité. En effet, les spécifications de l'instruction conduisent à une autre fuite d'information. Il se trouve qu'accéder à une ligne de cache présente dans un cache set où se trouve une ligne verrouillée met également à jour les métadonnées de la politique de remplacement de la voie verrouillée (cas III et IV de la Figure 2.2). De même, accéder à une ligne de cache verrouillée met à jour la politique de remplacement du cache set (cas IV et V de la Figure 2.2). Ainsi, l'attaquant peut manipuler les états de la politique de remplacement afin d'inférer les états des métadonnées de ses propres lignes de cache, mais également des lignes de cache verrouillées par la victime afin de mener un nouveau type d'attaque [66]. Ce problème est induit par le partage entre les applications des états de la politique de remplacement sur les données verrouillées.

En outre, la procédure d'accès de PLcache autorise un comportement atypique. En effet, en fonction de l'état du cache, un accès peut contourner celui-ci. Par exemple, si l'accès réalisé par une application A résulte en un cache miss et que la politique de remplacement pointe vers une voie verrouillée par l'application B, alors l'accès mémoire va contourner le cache (cas V et VI de la Figure 2.2). S'il s'agit d'une lecture mémoire, alors la mémoire hiérarchiquement supérieure retourne directement la donnée vers le processeur. S'il s'agit d'une écriture mémoire, alors la requête d'écriture est complétée sur le reste de hiérarchie mémoire.

2.2.2 Approche proposée

Dans un premier temps, notre objectif principal est de garantir qu'une adresse verrouillée en mémoire cache à l'aide de l'instruction `lock`, soit présente en cache jusqu'à l'exécution de l'instruction `unlock`. Ensuite, la ligne de cache adressée peut être évincée par la politique de remplacement. Cela permet de prévenir une attaque basée sur l'éviction. Par ailleurs, afin de prévenir une attaque manipulant les états de la politique de remplacement, tout accès réalisé sur une adresse verrouillée n'enclenche pas la mise à jour de la politique de remplacement. De même, tout accès réalisé sur des données non-verrouillées met à jour uniquement les états de la politique de remplacement des voies non-verrouillées. Ainsi, aucun accès ne peut interférer ou manipuler la voie dans laquelle se trouvent des données verrouillées, quel que soit le processus adressant le cache.

Dans le but de prévenir un accès outrepassant le cache dû au fait que toutes les voies du cache set aient été verrouillées, un minimum d'une voie est laissé disponible (*i.e.* non-verrouillée) afin de garantir un minimum de performance pour l'exécution des applications. Dans le cas contraire, *i.e.* lorsqu'une instruction `lock` adresse un cache set déjà complet, une exception matérielle est levée. Cela autorise de verrouiller jusqu'à $N - 1$ voies par cache set (où N est le nombre de

voies qui compose chacun des ensembles de cache). Une fois levée, l'exception doit être prise en compte par la pile logicielle pour mettre en pause ou arrêter le processus l'ayant causée.

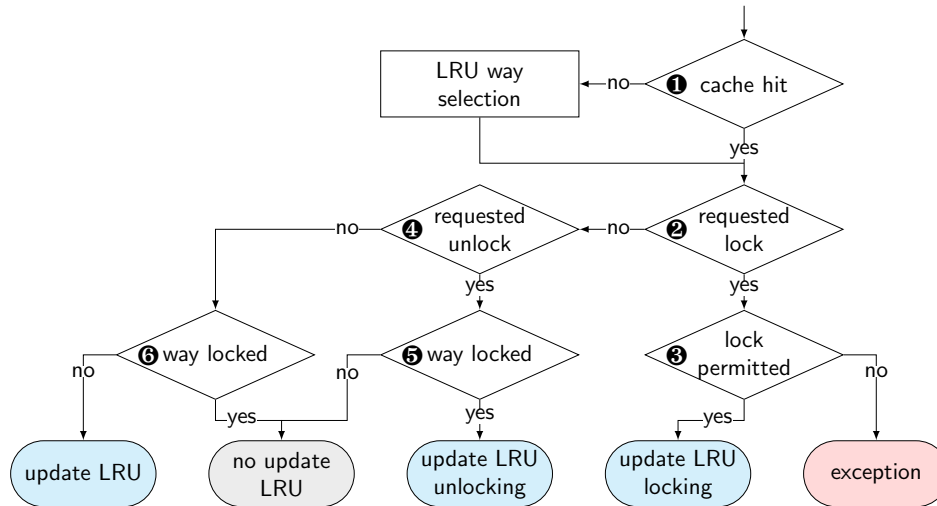


FIGURE 2.3 – Procédure d'accès au cache considérant le mécanisme de verrouillage.

La Figure 2.3 détaille la procédure d'accès au cache considérant le mécanisme de verrouillage. Premièrement, une condition vérifie si la donnée accédée est déjà en cache ❶ (cache hit) ; sinon, la politique de remplacement sélectionne la voie où mettre en cache la donnée. Ensuite, si la requête est un **lock** ❷, une condition détermine si l'état du cache permet de verrouiller la ligne de cache accédée ❸. Si une voie est disponible, la ligne de cache accédée est désormais marquée comme verrouillée et la politique de remplacement LRU est mise à jour pour les voies candidates à l'éviction. Autrement, une exception est levée. Si la requête est un **unlock** ❹, et si elle accède une ligne de cache verrouillée ❺, le déverrouillage est effectué et la ligne de cache peut dorénavant être évincée par la politique de remplacement. Autrement, la requête de déverrouillage concerne une ligne de cache non verrouillée. Les requêtes conventionnelles à la mémoire de types **load** ou **store** ❻ sont traitées de la manière suivante : si une ligne de cache verrouillée est accédée, la politique de remplacement n'est pas mise à jour. Autrement, la politique de remplacement est mise à jour pour toutes les voies non-verrouillées dans le cache set.

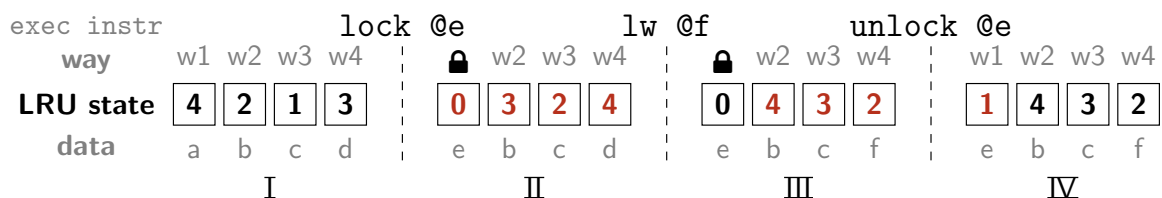


FIGURE 2.4 – Exemple pédagogique de mise à jour de la politique de remplacement.

Au travers d'un exemple pédagogique, la Figure 2.4 illustre l'évolution des états de la politique de remplacement LRU sur trois accès mémoire adressés sur un même cache set en considérant consécutivement un `lock`, un accès conventionnel `lw` et un `unlock`. Les métadonnées de la politique de remplacement (*i.e.* LRU states) sont représentées pour chaque voie du cache set. Le cache set est composé de $N=4$ voies (wX) et a déjà été accédé avant le premier accès simulé de cet exemple pédagogique. Les données `a`, `b`, `c` et `d` sont stockées parmi les quatre voies. Avant de considérer l'accès `lock @e`, nous pouvons remarquer que la voie la moins récemment utilisée est la voie `w1` (LRU state = N). Par ailleurs, aucune ligne de cache n'est verrouillée (*i.e.* $N_{lock} = 0$ où N_{lock} est le nombre de voies verrouillées dans le cache set) ainsi les états de la politique de remplacement valent au moins la valeur 1 pour toutes les voies (*i.e.* $LRU\ state \geq N_{lock} + 1$). Dans la seconde phase, suite à l'exécution de l'instruction `lock` accédant en mémoire à l'adresse de `e` (`lock @e`), la donnée `a` a été évincée et la voie `w1` est considérée comme verrouillée (LRU state = 0). Les états des autres voies non-verrouillées ont également été mises à jour, et il reste trois voies candidates pour la politique LRU dont leur LRU state $\geq N_{lock} + 1$ avec $N_{lock} = 1$. Étant verrouillée, la voie `w1` ne peut pas être sélectionnée par la politique de remplacement pour être évincée. Après avoir lu un mot à l'adresse de `f` (`lw @f`), la donnée `d` a été évincée par la donnée `f` et la voie `w4` devient la voie la plus récemment utilisée avec LRU state = $N_{lock} + 1$. Les états LRU state des voies `w2` et `w3` ont été mises à jour alors que la voie `w1` reste verrouillée. Le nombre de voies candidates à l'éviction reste le même (*i.e.* 3). Finalement, `w1` est déverrouillée grâce à l'exécution de l'instruction `unlock @e`, et a été réinsérée comme candidat à l'éviction et est considérée comme la voie la plus récemment utilisée.

2.3 Solution d'implémentation Matérielle/Logicielle

Dans cette section, nous discutons des différentes façons d'implémenter notre mécanisme de sécurité. Ensuite, nous explorons comment est défini un jeu d'instructions (Instruction Set Architecture (ISA)) dans le but de comprendre comment l'étendre afin de spécifier de nouvelles instructions et créer le support matériel et logiciel.

2.3.1 Analyse des avantages et inconvénients

Dans le but de concevoir notre mécanisme de protection contre les attaques en caches, nous étudions les différentes méthodes pour l'implémenter. De ce fait, nous étudions les avantages et inconvénients d'implémenter notre mécanisme dans une approche strictement matérielle, puis dans une approche strictement logicielle et dans un dernier temps, dans une approche combinant le logiciel et le matériel.

Solution strictement matérielle

Une solution strictement matérielle permet de répondre à un besoin (sécurité, performance) en modifiant la micro-architecture du processeur ou en ajoutant un coprocesseur / accélérateur matériel. La solution matérielle peut être implémentée et déployée immédiatement si le produit ciblé est un Field Programmable Gate Array (FPGA) ; sinon, il faut attendre la prochaine génération de matériel si le produit ASIC est ciblé. Ce type de solutions engendre un surcoût au niveau des ressources matérielles utilisées. Cela peut également entraîner une baisse de la fréquence du système dû à l'allongement du chemin critique. Cependant, lors du développement des solutions matérielles, il est préférable de ne pas affecter le chemin critique du système afin d'encourager l'utilisation de la solution développée. Pour cette raison, la plupart des solutions matérielles n'engendrent qu'une faible baisse de la fréquence du système. Ensuite, l'ajout de ressources matérielles sur le système engendre une augmentation de la consommation. Cependant, cette augmentation peut être négligeable au regard de la consommation de l'exécution complète du programme, notamment si la solution matérielle permet d'augmenter les performances du système. Cependant, cela peut engendrer de nouvelles failles de sécurité (par conception ou suite à une mauvaise utilisation) au travers de la création de nouveaux canaux auxiliaires, ou canaux cachés.

Solution strictement logicielle

D'autre part, une solution strictement logicielle permet de répondre à un besoin (sécurité, fonctionnalité, etc) en modifiant le programme s'exécutant sur le système. Ce type de solution est très modulaire et permet de déployer la contremesure en mettant simplement à jour les programme binaires sur les systèmes visés. Les correctifs logiciels engendrent une perte en performance qui peut être non négligeable et par effet de cause une augmentation de la consommation. Bien qu'aujourd'hui moins problématique que dans le passé, ces correctifs engendrent un surcoût au niveau de la taille du binaire dû à l'ajout d'instructions dans le programme. Ce type de contremesure permet de venir à bout de failles de sécurité récemment découvertes sur des matériels d'anciennes générations et de générations actuelles afin de pouvoir les corriger lors des prochaines générations matérielles si cela est pertinent (accélération importante, temps ingénierie, consommation, etc).

Solution matérielle/logicielle

Finalement, le dernier type de solution allie la contribution logicielle avec la contribution matérielle. Ce type de solution est motivé et souligné [114, 115] pour une conception optimisée et s'adaptant au mieux pour de nouveaux défis dans un monde en perpétuelle évolution où une intégration matérielle/logicielle semble être une approche répondant à ces nouveaux défis.

Cela consiste à étendre l'architecture matérielle avec de nouvelles fonctionnalités qui ont pour vocation d'être manipulées par le programme s'exécutant sur le système. De ce fait, la chaîne de compilation doit être au fait de l'existence des différentes fonctionnalités du matériel afin d'intégrer la manipulation de ces fonctionnalités à l'aide d'instructions dédiées ou bien d'instructions manipulant les Control and Status Registers (CSRs). Les instructions dédiées manipulant les éléments matériels conviennent parfaitement au jeu d'instructions RISC-V, puisque celui-ci intègre la possibilité d'étendre son jeu d'instructions grâce à des bits réservés qui permettent de rajouter des instructions tout en conservant une rétrocompatibilité avec les instructions existantes. D'autre part, la manipulation des éléments matériels à l'aide d'instructions manipulant des registres CSRs est moins invasive pour le jeu d'instructions sur lequel le système est basé. En effet, cette méthode ne requiert pas d'étendre l'ISA, puisque seules de nouvelles adresses de destinations pour les registres sont nécessaires pour contrôler et monitorer les mécanismes matériels. Cette intégration logicielle convient mieux aux jeux d'instructions propriétaires (ARM, x86) mais peut également être utilisée avec le jeu d'instructions RISC-V. Il convient, lors de la collaboration amenant au développement conjoint du matériel et du logiciel, de choisir par quel biais utiliser la fonctionnalité développée.

Ce type de solution subit les mêmes inconvénients qu'une solution strictement matérielle : *i.e.* déploiement long, utilisation de ressources matérielles, impact sur la consommation. Ces deux derniers inconvénients sont tout de même moins impactants qu'une solution strictement matérielle. En parallèle, il profite des avantages d'une solution matérielle, à savoir un faible impact sur les performances. Les solutions matérielles/logicielles tirent également leur avantage par l'agilité d'utilisation procurée par le contrôle logiciel exercé sur le mécanisme matériel.

TABLE 2.1 – Bilan des avantages et des inconvénients de chacun des types de solutions.

	Déploiement	Coût Logiciel	Coût Matériel	Impact Performances	Impact Consommation
Logiciel	aisé	élevé	∅	élevé	élevé
Matériel	complexe	∅	élevé	faible	faible
Matériel/Logiciel	complexe	faible	faible	modéré	modéré

La Table 2.1 récapitule les avantages et les inconvénients de chacune des solutions. L'approche matérielle/logicielle semble offrir le meilleur compromis. Bien qu'elle implique un déploiement contraignant et nécessite des modifications à la fois sur le plan matériel et logiciel, elle présente l'avantage de réduire l'impact sur les performances et la consommation. De plus, une approche matérielle/logicielle s'adapte particulièrement bien pour implémenter de nouveaux mécanismes de sécurité ou des adaptations sécurisées [88, 116]. Ce dernier point est lié à la possibilité de définir un nouveau contrat entre le logiciel et le matériel en tenant compte de la sécurité avec de fortes garanties [117]. Pour ces raisons, la mise en œuvre de notre mécanisme de verrouillage est portée par une solution alliant le matériel et le logiciel.

2.3.2 Extension du jeu d'instructions

Suite au choix d'implémenter nos nouvelles instructions `lock` et `unlock` en se basant sur une approche matérielle/logicielle, nous présentons le jeu d'instructions RISC-V. Nous introduisons tout d'abord la notion d'ISA et de comment l'étendre afin d'adapter le support matériel d'une part et le support logiciel d'autre part. Nous spécifions ensuite les instructions `lock` et `unlock` afin de permettre leur implémentation.

Le jeu d'instructions RISC-V

Le jeu d'instructions RISC-V a été initié dans les années 2010 par l'Université de Californie à Berkeley dans le but de proposer, contrairement à la plupart des ISA, une architecture de jeu d'instructions libre de droits et en licence ouverte. Il s'agit de la cinquième génération de ce type d'architecture de processeur à jeu d'instructions réduit (RISC) dont les deux premières générations ont été publiées par Patterson et Sequin [118]. Les spécifications de RISC-V sont disponibles en ligne pour une version non privilégiée [119] et privilégiée [120].

Le développement du RISC (en parallèle du MIPS) tient son leitmotiv du fait que la majorité des programmes n'utilisent qu'un nombre restreint d'instructions. C'est pourquoi la spécification RISC-V propose différentes bases pour se rapprocher le plus possible des besoins de chacun des systèmes développés. La base RV32I est l'architecture originelle et inclut le minimum d'instructions pour manipuler un programme et des données en 32 bits. Ensuite, nous pouvons également sélectionner la base RV32E permettant de s'ajuster aux besoins des systèmes embarqués (capteur, etc). Il existe également des bases de jeux d'instructions manipulant des données 64 bits et 128 bits pour des usages orientés vers des hautes performances (ordinateurs [121], serveurs [122], etc).

Après avoir sélectionné la base de l'architecture à développer, nous avons le choix de considérer différentes extensions afin de compléter l'offre d'instructions disponibles. Par exemple, l'extension A inclut les instructions atomiques qui sont requises pour implémenter un système d'exploitation. M pour les opérations de multiplication et de division, F pour supporter les opérations à virgules flottantes, ou encore C permettant de supporter les instructions compressées sur 16 bits. Bien que non indispensables à l'exécution des programmes, ces extensions permettent d'améliorer grandement les performances des programmes exécutés. En effet, le support logiciel permet d'utiliser astucieusement chacune des instructions présentes dans le système considéré via les extensions importées. Il existe également d'autres extensions pour des usages plus fins et étendant l'ISA avec peu d'instructions. Par exemple, Zifencei étend l'ISA avec l'instruction `fence.i` permettant de synchroniser plusieurs cœurs, ou bien l'extension Zicsr ajoutant plusieurs instructions manipulant les CSRs.

En choisissant sa base et ses extensions, chaque architecte peut ainsi construire son système correspondant au mieux à ses attentes.

Actuellement, RISC-V vit autour d'un écosystème d'entreprises et d'universités pour maintenir les spécifications à jour et lancer différents projets autour du processeur afin de mettre en lumière ce jeu d'instructions. Ces différentes initiatives sont importantes pour le monde de la recherche en tirant bénéfice de certaines compagnies et organisations qui mettent à disposition et en libre accès les implémentations matérielles de leur processeur se basant sur l'ISA RISC-V.

Support d'extension personnalisé

En plus des extensions officielles proposées par les spécifications RISC-V, il est possible d'étendre le jeu d'instructions avec ses propres instructions. En effet, le jeu d'instructions RISC-V a été construit dans ce but, où des codes d'opérations sont réservés pour que les architectes puissent développer leurs propres instructions, comme présenter dans la Table 2.2 issue de la spécification [119].

TABLE 2.2 – Assignation des codes d'opérations des instructions de la base RISC-V, valeur `inst[4:0]=11111` réservée pour des instructions de taille supérieure à 32 bits.

inst [6:0]		00011	00111	01011	inst [4:0] 01111	10011	10111	11011
inst [6:5]	00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32
	01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32
	10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2 /rv128
	11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3 /rv128

Grâce à cette table, nous prenons connaissance des codes d'opération (ou *opcodes*) associés à chaque type d'opération où 7 bits sont utilisés à cet effet. Parmi ces 7 bits, les deux bits de poids faibles valent 11 : en effet, toutes autres valeurs (00, 01, ou 10) signifient que l'instruction est compressée (construite sur 16 bits). Ensuite, la table est structurée de telle façon que les colonnes parcourent les 3 bits suivants et les lignes parcourent les 2 bits de poids fort du code d'opération. Non représentée sur la table, la colonne 11111 est réservée pour des formats d'instructions supérieurs à 32 bits (*i.e.* 48, 64 ou ≥ 80 bits). Par exemple, le champ AMO assignant les instructions de l'extension Atomique obtient la valeur binaire d'opcode 0101111. Au total, quatre champs sont réservés pour intégrer des instructions personnalisées afin d'étendre le jeu d'instructions RISC-V. Cela ne signifie pas que nous sommes limités à quatre instructions, mais à quatre groupes d'instructions.

Comme représenté dans la Table 2.3, il existe 4 types d'instructions permettant de représenter les instructions dans différents champs sur 32 bits. R, I, S et U sont les principaux types, B et J sont respectivement des variantes de S et U dans lesquelles la manipulation de certains champs est différente. Pour chacun des types, différents champs permettent de construire les 32 bits qui composent chaque instruction. Toutes ont pour base commune le champ opcode de 7 bits sur les poids faibles de l'instruction. Ce champ permet d'identifier le groupe de l'instruction. Ensuite, les instructions peuvent embarquer des champs `functX`, où X correspond à la taille du

TABLE 2.3 – Format de base des instructions RV32I incluant les variantes des immédiats.

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

champ **funct**, permettant d'affiner l'usage de l'instruction faisant partie du groupe opcode. Il y a également des champs permettant d'adresser les registres utilisés par l'instruction. Ils sont tous codés sur 5 bits pour adresser les 32 registres qui composent le banc de registres. Parmi eux, les champs **rs1** et **rs2** adressent les registres utilisés comme entrées de l'instruction. D'autre part, **rd** adresse le registre de destination où est écrit le résultat de l'instruction. Le champ **imm** de taille variable permet d'assigner une valeur encodée directement dans l'instruction. Ainsi, pour les codes d'opérations dont les instructions sont de type R, nous pouvons étendre le jeu d'instructions jusqu'à 1024 instructions grâce aux champs **funct3** et **funct7** respectivement codés sur 3 et 7 bits. En revanche, les types I, S et B permettent d'étendre uniquement de 8 instructions grâce au champ **funct3** sur 3 bits. Finalement, les types U et J n'autorisent qu'une instruction par valeur d'opcode mais avec un large champ d'immédiat offrant une plus grande liberté pour assigner une valeur directement via l'encodage de l'instruction (par exemple pour l'instruction **lui** permettant de charger 20 bits de poids forts dans le registre destination en encodant les 20 bits dans le champ **imm** de l'instruction).

2.3.3 Support matériel

Après avoir défini la nouvelle instruction dans l'ISA au travers de l'extension du jeu d'instructions, il faut implémenter matériellement les spécifications précédemment éditées. Dans un premier temps, l'étage de décodage du processeur doit pouvoir identifier l'instruction personnalisée en étendant le bloc de décodage. Pour ce faire, le bloc de décodage identifie le code d'opération puis, au travers des champs **funct**, identifie la fonction précise afin de retourner les données nécessaires via le banc de registres et d'assigner les signaux de contrôle vers l'étage suivant du processeur permettant la bonne exécution de l'instruction. Les signaux de contrôle se propagent jusqu'au module concerné où est implémentée l'opération définie dans les spécifications (*e.g.* une addition, $rd = rs1 + rs2$, récupère les registres sources **rs1** et **rs2** pour les propager jusqu'au module d'exécution des additions puis retourner le résultat dans le registre **rd**). Libre à l'architecte de concevoir le module d'exécution comme il le souhaite tant que le matériel développé est conforme aux spécifications.

2.3.4 Support logiciel

Maintenant que le matériel supporte l'ensemble des instructions souhaitées par le système, il est temps de générer le programme binaire s'exécutant sur le matériel. La chaîne de compilation prend plusieurs entrées afin de transformer le programme, généralement programmé en langage C, en un fichier binaire considéré comme code machine. Parmi les entrées, la chaîne de compilation doit connaître la base d'instructions (RV32I) et les extensions officielles (A, M, F, D, etc) qui composent le matériel pour lequel le programme est compilé. Si le jeu d'instructions est étendu à l'aide d'instructions personnalisées, la chaîne de compilation doit a minima connaître la représentation en langage d'assemblage de l'instruction pour pouvoir générer le code machine associé en binaire. À partir de là, il est possible d'utiliser l'instruction personnalisée dans un programme décrit en langage C au travers du mot clé `__asm__ volatile(...)` autorisant la description d'instruction en langage d'assemblage et permet d'insérer les instructions au sein du binaire généré.

2.3.5 Spécification des instructions `lock` et `unlock`

Après avoir défini le comportement souhaité des nouvelles instructions, il est temps de définir les spécifications de celles-ci au niveau de l'ISA.

L'instruction `lock` est décrite telle que `lock imm(rs1)` garantie l'accès en cache (cache hit) pour tout accès futur des données présentes dans la ligne de cache située à l'adresse `rs1+imm` où `rs1` est le contenu d'un registre lu dans le banc de registres, et `imm` est une valeur immédiate assemblée dans l'instruction. Ainsi, l'instruction peut être assimilée au pseudo-code `lock(adresse)` où *adresse* vaut `rs1+imm` et est l'adresse mémoire accédée et verrouillée.

Plusieurs cas de figures peuvent être rencontrés lorsqu'une requête de type `lock` est effectuée. En effet, lorsque la requête de verrouillage accède au cache : si la ligne de cache accédée n'est pas en cache (cache miss), alors la requête se propage aux étages supérieurs afin de rapatrier la ligne de cache au niveau du cache L1, ensuite même si la ligne de cache accédée est déjà présente en cache (cache hit) alors la politique de remplacement est mise à jour suivant la procédure d'accès décrite dans la Section 2.2.2. Enfin, si un accès de type `lock` est réalisé alors qu'il ne reste plus qu'une voie disponible (*i.e.* non-verrouillée) alors un signal d'exception est mis à l'état haut afin d'être propagé jusqu'au cœur et d'être traité grâce à une routine d'exception.

Par exemple, si les données A, B, C, et D sont adressées sur une même ligne de cache et qu'une instruction `lock` verrouille une des adresses répertoriées parmi les données considérées, alors tout accès mémoire fait sur les adresses de cette ligne de cache vont résulter en un cache hit, et ce quel que soit le type d'accès mémoire (`lw`, `sw`, `sb`, `lh`, etc). Bien qu'inutile, une instruction `lock` accédant une adresse déjà verrouillée résulte toujours en un cache hit. Cette propriété garantissant un cache hit est maintenue jusqu'à l'exécution de l'instruction `unlock` adressée à la ligne de cache (*i.e.* en adressant `@A`, `@B`, `@C` ou `@D` où `@X` est l'adresse de la donnée X).

Ensuite, la ligne de cache peut être à nouveau évincée dès lors que l'instruction `unlock` est exécutée.

En se basant sur la Table 2.3, les nouvelles instructions `lock` et `unlock` doivent s'intégrer parmi les types proposés. Au vu de la similarité des deux instructions, il est judicieux de choisir des types d'instructions identiques afin de faciliter leur intégration et leur utilisation tant d'un point de vue conception matérielle que développement logiciel. Premièrement, ces instructions de manipulation de données en mémoire requièrent une adresse dans le but d'adresser la bonne donnée à verrouiller en cache. Ainsi, notre format nécessite un champ dédié à un registre source, contraignant à l'utilisation des types R, I, S ou B.

Les instructions `lock` et `unlock` réalisent des accès mémoires, il est ainsi préférable d'utiliser le même type qu'utilisent les instructions de type `load` et `store` présentes dans le jeu d'instructions RISC-V. Ainsi, le choix se porte sur le type I, où nous avons à disposition un registre destination `rd`, un registre source `rs1` et un champ immédiat `imm` de 12 bits. Le format des instructions est décrit dans la Table 2.4. Les deux instructions utilisent le code d'opération `CUSTOM-0`. Le champ `funct3` permet de les différencier, l'instruction `lock` est associée à `000` et l'instruction `unlock` est associée à `001`. Le registre source permet de définir la base d'adresse et le champ `imm` est utilisé comme offset. Le registre de destination `rd` permet d'écrire dans le banc de registre le mot de donnée adressé à la mémoire.

TABLE 2.4 – Format des instructions `lock` et `unlock` se basant sur le type I.

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
offset[11:0]		base		LOCK		destination		CUSTOM-0	
x...x _b		x...x _b		000 _b		x...x _b		0001011 _b	
offset[11:0]		base		UNLOCK		destination		CUSTOM-0	
x...x _b		x...x _b		001 _b		x...x _b		0001011 _b	

Gestion des exceptions

L'utilisation des instructions `lock` et `unlock` peut mener à une exception levée lorsqu'une instruction `lock` est exécutée et l'état du cache ne permet pas de verrouiller une ligne de cache parmi le cache set ciblé. En effet, comme présenté dans la Section 2.2.2, un signal d'exception est levé par le cache puis est propagé vers le cœur afin de renseigner l'impossibilité de réaliser le processus de verrouillage. Lorsqu'un tel cas survient, nous proposons d'exécuter dans une routine d'exception afin de stopper l'exécution de l'application puis de donner la main au système d'exploitation. Le système d'exploitation peut restituer l'exécution à une autre application ayant déjà verrouillé ses données en caches afin qu'il termine son exécution pour les déverrouiller par la suite. Il peut également décider de déverrouiller un ensemble de données afin de restituer l'exécution à l'application dont l'instruction de `lock` a mené à l'exception.

Un autre comportement non-conventionnel mérite de générer une exception afin de laisser le système d'exploitation la piloter. En effet, déverrouiller une adresse dont la ligne de cache n'est pas verrouillée est un comportement anormal et peut révéler l'exécution d'une attaque contre notre système. De ce fait, nous proposons de renseigner et d'itérer un CSR dont les accès requièrent un privilège afin que le système d'exploitation puisse comptabiliser le nombre d'accès de ce type. En fonction du nombre d'accès lu dans ce registre, le système d'exploitation peut prendre une décision.

En l'état, ces exceptions ne sont ni monitorées par le cœur, ni par une couche logicielle dédiée. Seule une sortie booléenne du cache est mise à jour lorsque la première exception susmentionnée est rencontrée.

2.4 Conclusion

Dans ce chapitre, nous avons défini les garanties que nos instructions `lock` et `unlock` doivent respecter afin de se prémunir du modèle de menaces introduit dans le chapitre et de ne pas subir les mêmes failles de sécurité que la solution PLcache [76] également énoncé dans ce chapitre. Notre contribution propose d'utiliser explicitement le mécanisme de verrouillage afin de se prémunir des attaques `PRIME+PROBE` et `EVICT+TIME`. Nous avons défini les spécifications et les cas d'utilisation de notre mécanisme de verrouillage de données en cache afin de pouvoir étendre le jeu d'instructions RISC-V au travers d'une solution alliant le matériel et le logiciel.

IMPLÉMENTATION DU MÉCANISME DE VERROUILLAGE

SOMMAIRE

3.1	Implémentation du mécanisme de sécurité	61
3.1.1	Utilisation par le logiciel	61
3.1.2	Implémentation matérielle	63
3.2	Évaluation	68
3.2.1	Évaluation de la surface matérielle	68
3.2.2	Évaluation de la sécurité	70
3.2.3	Évaluation des performances	72
3.3	Conclusion	77

Dans ce chapitre, nous déterminons dans un premier temps comment implémenter notre solution de sécurité basée sur le verrouillage de données en cache dans le but de se prémunir contre les attaques en cache. Dans un second temps, nous implémentons la solution sur une cible FPGA afin de permettre la caractérisation de celle-ci au travers de diverses évaluations de coûts, de sécurité et de performances. L'ensemble de ces évaluations permettra de positionner notre solution par rapport aux contremesures existantes de l'état de l'art. Ce chapitre reprend principalement les éléments de notre contribution publiée à la conférence ISVLSI [123].

3.1 Implémentation du mécanisme de sécurité

Dans cette section, nous détaillons l'utilisation par le logiciel de notre mécanisme de sécurité et nous présentons également l'impact du placement mémoire des données à verrouiller sur les performances du système. Ensuite, nous détaillons l'implémentation matérielle du mécanisme de verrouillage de lignes de cache, introduisant les instructions `lock` et `unlock`.

3.1.1 Utilisation par le logiciel

Au travers de notre approche, le mécanisme de verrouillage est explicitement intégré par le développeur logiciel dans le but de protéger un ensemble de données. L'extrait de code dans le Listing 3.1 présente une fonction `fct` implémentant le mécanisme de verrouillage pour sécuriser les accès dépendants d'un secret effectués dans la fonction `algo` appelée dans `fct`. Ceci démontre l'utilisation et l'insertion des instructions `lock` et `unlock` qui sont insérées respectivement avant et après l'appel de `algo` en ligne 7. Concernant la phase de verrouillage, une boucle logicielle est utilisée dans les lignes 3-4 parcourant la table sensible octet par octet. Cependant, en s'assurant que les mots de données composant la table sont alignés en mémoires, l'incrément pourrait être de 4. De plus, en s'assurant que la table soit alignée en mémoire à la granularité des lignes de cache, l'incrément pourrait être équivalent à la taille d'une ligne de cache, dans notre cas $B = 16$ octets. Si la chaîne de compilation est capable de fournir des garanties sur l'alignement mémoire de la table à protéger, une baisse des dégradations des performances pourrait être observée. Dans le but de protéger entièrement la table, une macro en assembleur, présentée dans le Listing 3.2, est appelée à chaque ligne de cache (*i.e.* à chaque itération de la boucle `for`). Il est important de noter que dans cette phase, le nombre de lignes de cache verrouillées ne doit en aucun cas dépendre du secret manipulé par la fonction `algo`. À la suite de l'exécution de l'algorithme sensible, la table précédemment verrouillée est déverrouillée du cache (l. 10-11) grâce à une boucle `for` similaire à celle utilisée lors de la phase de verrouillage. Cependant, au lieu d'insérer la macro `lock_macro`, une macro similaire est utilisée où seule l'instruction `unlock` est utilisée à la place de l'instruction `lock` en ligne 4. Le registre destination des instructions `lock` et `unlock` est `x0`. Dans le banc de registre, le registre `x0` a la particularité de toujours valoir 0. Cette utilisation de `x0` est volontaire étant donné que nous ne souhaitons pas apporter de la contention sur le banc de registre par des données non utilisées immédiatement. Cependant, il peut être intéressant de retourner la donnée dans un registre général du banc de registre dans certains cas d'usage lorsque la donnée retournée est utilisée à la suite de son verrouillage en mémoire cache.

Le Listing 3.1 illustre comment un développeur logiciel peut explicitement verrouiller un ensemble de lignes de cache. C'est également de cette façon que le mécanisme est utilisé pour produire les résultats dans les sections suivantes. Il est important de noter que cette insertion

```

1 void fct(int* sensitive_table, int* input){
2     //lock phase
3     for(int i=0; i<sizeof(sensitive_table); i+=1)
4         lock_macro(&sensitive_table, i);
5
6     //algo accesses table depending on secret
7     algo(sensitive_table, input);
8
9     //unlock phase
10    for(int i=0; i<sizeof(sensitive_table); i+=1)
11        unlock_macro(&sensitive_table, i);
12 }

```

Listing 3.1 – Exemple d’utilisation du mécanisme de verrouillage.

```

1 c.mv t4,a4 # move &table in t4
2 c.mv t5,a5 # move i in t5
3 c.add t4,t5
4 lock x0,0(t4)

```

Listing 3.2 – Macro lock_macro en assembleur.

fastidieuse peut être automatisé en couplant des techniques d’annotations de programme [124, 125] et de propagations de teintes [126]. Cette automatisation conduirait à un impact plus faible sur le nombre d’instructions insérées. En effet, les instructions `c.mv` ne seraient plus nécessaires si l’adresse de la table `&table` et la valeur de l’incrément `i` sont déjà fournies dans le banc de registres. Par conséquent, moins d’instructions serait nécessaires tant dans le programme binaire que lors de l’exécution du programme. Cependant, ces travaux d’automatisation de l’intégration logicielle ne rentrent pas dans le cadre de ces travaux de thèse.

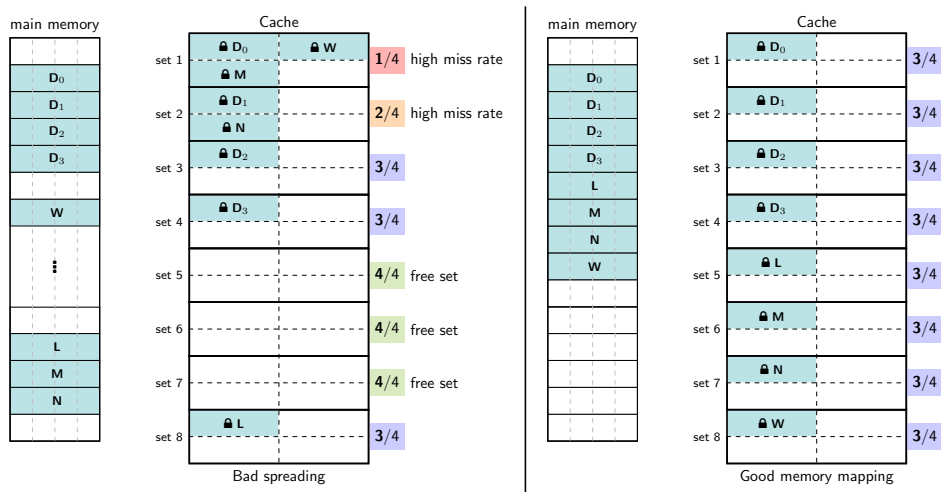


FIGURE 3.1 – Impact du placement mémoire des données à verrouiller sur la disponibilité du cache.

Dû au fait que la contention de ressource verrouillée laisse moins de ressources matérielles pour l'exécution d'autres applications, il est important d'utiliser intelligemment le mécanisme de verrouillage au sein du cache afin d'avoir le moins d'impact sur l'exécution d'applications tierces. La Figure 3.1 illustre cette problématique dans laquelle une application souhaite verrouiller différentes données en cache (D_{0-3} , L, M, N, et W) dans une mémoire cache à 4 voies indexée sur 8 sets. La partie gauche de la figure représente une mauvaise répartition des données en mémoire principale, entraînant une haute contention de verrous sur certains cache sets (**set 1** et **2**) et d'autres cache sets sont sous-utilisés (**set 5-7**). Ceci est problématique parce que les accès en cache indexés sur les cache sets **1** et **2** vont entraîner une augmentation du taux de *miss*, résultant en un plus grand nombre d'accès à la mémoire principale et donc en une baisse des performances et une augmentation de la consommation. Cependant, lorsque les données sont réparties en mémoire principale tel que représenté sur la partie droite de la figure, nous obtenons un taux de verrouillage équitable sur les cache sets. Cela a pour effet de laisser 3 voies disponibles sur chacun des cache sets réduisant le taux de cache miss lors de l'exécution de programmes ne dépendant pas des données verrouillées.

3.1.2 Implémentation matérielle

Dans cette section, nous présentons l'implémentation du mécanisme de verrouillage au sein du cache. Nous détaillons le module de politique de remplacement où nous avons choisi d'insérer le mécanisme de sécurité.

Systeme considéré

Nous basons notre architecture sur un cœur CV32E40P [127] interconnecté à la mémoire principale via un niveau de cache de données, une description est fournie dans la Figure 3.2. L'ensemble du système est décrit en langage de description matérielle SystemVerilog. Le cœur s'appuie sur l'ISA RISC-V, lequel implémente la base RV32I avec les extensions M, C, et Zicsr, étendant respectivement la base 32 bits des opérations de multiplications et de divisions, du support des instructions compressées et du support des registres de contrôle et de statut. Le cœur se compose de 4 étages et s'exécute dans l'ordre et n'implémente aucun mécanisme de prédiction de branchements. Le cœur est adressé physiquement sur 22 bits. Les modules modifiés pour permettre l'exécution en temps-constant des programmes manipulant un secret apparaissent en gris sur la Figure 3.2.

Dans notre démarche de permettre l'exécution en temps constant de programme manipulant un secret, nous identifions les instructions de divisions et modulus comme n'étant pas temps constant. Leurs temps d'exécutions dépendent de l'un des opérandes. En effet, le temps d'exécution de ces opérations dépend du premier bit de poids fort valant 1 sur le diviseur. Par exemple, une opération de division par $0x6000\ 0000$ nécessite 4 cycles, étant donné que le deuxième bit

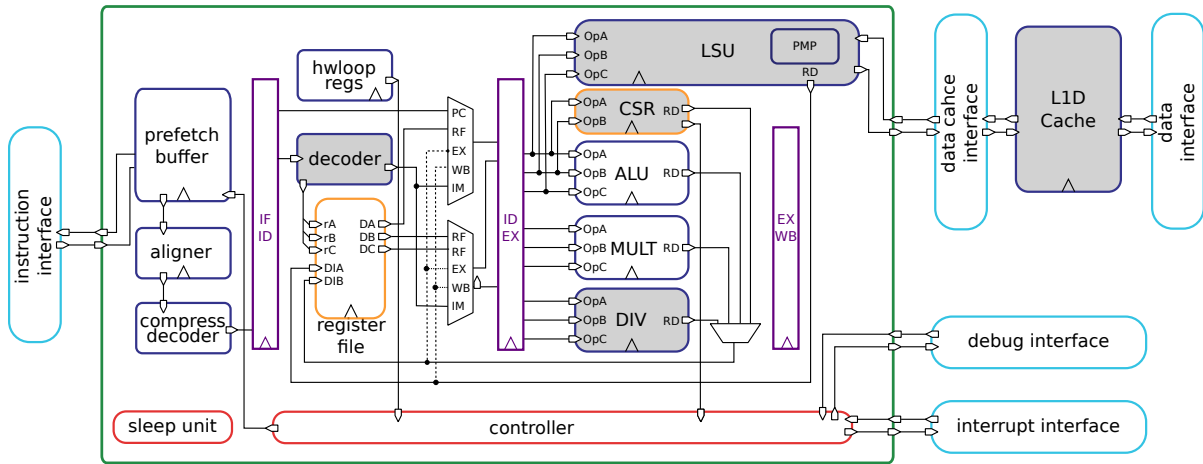


FIGURE 3.2 – Description matérielle du système incluant le cœur CV32E40P et un niveau de mémoire cache de donnée (inspiré de [4]).

de poids fort vaut 1, une opération de division par $0x0000\ 0003$ nécessite 33 cycles, puisque le 31ème bit de poids fort vaut 1. Dans le but de permettre l'exécution en temps constant de ces instructions et de rendre immune l'exécution de secret face aux fuites par canaux auxiliaires temporels, nous intégrons une contremesure simple dans le module de division (dans lequel sont opérées les instructions de division et de modulus). La contremesure consiste à réaliser une exécution au pire cas (*i.e.* 35 cycles) en ajoutant des états d'attente dans la machine à état qui compose le module de division. Dans le but de ne pas impacter l'ensemble des opérations de divisions et de modulus, une entrée permet de superviser ce mode d'exécution afin de permettre l'activation uniquement lorsque nécessaire. Ce mode est supervisé par un nouveau CSR dédié et est implémenté à l'adresse $0xD00$ du banc de registre des CSRs.

Le cache implémenté correspond à un cache de premier niveau dédié aux données. Il s'agit d'un cache associatif à N -voies où $N = 4$ voies sont utilisées parmi $S = 128$ cache sets. Les lignes de cache ont une taille de $B = 16$ octets, et permettent de stocker jusqu'à 4 mots de 32 bits. Ainsi, le cache a une capacité de 8 Ko. Lors d'un accès, l'adresse de 22 bits est utilisée de la manière suivante : les premiers $\log_2(B) = 4$ bits sont dédiés à la sélection des octets dans la ligne de cache, les $\log_2(S) = 7$ bits suivants indexent les cache sets, puis les 11 bits restants sont utilisés comme tag.

Implémentation au sein du cache

Nous proposons d'étendre les métadonnées de la politique de remplacement LRU et son processus de mise à jour pour supporter le mécanisme de verrouillage. En considérant un cache à ensemble associatif sur N voies et la politique de remplacement LRU originelle, les métadonnées associées à une voie sont assignées sur N états (*e.g.* pour un cache à ensemble associatif sur 4

voies, les états sont inclus entre 1 et 4). Cet état associé à chaque voie reflète l'âge de chacune des lignes de cache présentes dans le cache set, et plus précisément l'âge du dernier accès à cette donnée (e.g. l'état 1 en tant que plus récemment utilisée, et 4 en tant que moins récemment utilisée). Notre solution se base sur un état de LRU supplémentaire dédié aux lignes de cache verrouillées (l'état verrouillé vaut 0). Ainsi, $N + 1$ états doivent être considérés dans la politique de remplacement. Lorsqu'un accès mémoire est réalisé, les métadonnées de la LRU sont mises à jour selon les règles suivantes. Si une requête est associée à une instruction `lock` et est autorisée (voir Section 2.2.2), l'état de la voie sélectionnée est mis à jour en état verrouillé (pour rappel 0). Les métadonnées des états des voies non-verrouillées sont également mis à jour avec des valeurs d'états comprises entre $N_{lock} + 1$ et N , où $N_{lock} < N$, N_{lock} est le nombre de lignes de cache actuellement verrouillées dans le cache set (e.g. si $N_{lock} = 2$, deux états valent 0 et les deux états restants valent respectivement 3 et 4). Si la requête est associée à une instruction `unlock`, l'état de la voie sélectionnée est mis à jour à la valeur N_{lock} . Dans ce cas, les métadonnées des voies non-verrouillées n'ont pas besoin d'être mises à jour. Enfin, si un accès conventionnel est réalisé, les métadonnées de la LRU des voies non-verrouillées sont mises à jour avec des valeurs comprises entre N_{lock} et N .

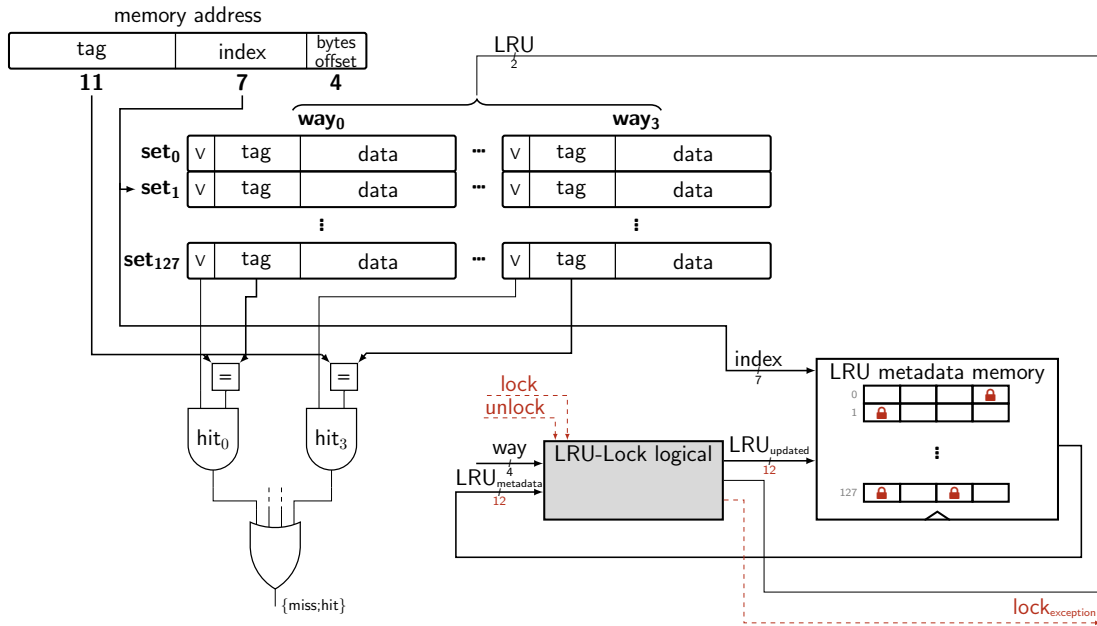


FIGURE 3.3 – Schéma bloc du cache implémentant le mécanisme de verrouillage.

Pour supporter les instructions `lock` et `unlock`, nous étendons la microarchitecture du cœur et du L1D associé ainsi que la chaîne de compilation. Dans cette section, nous nous concentrons sur l'extension matérielle. Les instructions `lock` et `unlock` sont principalement décodées comme des instructions `load` conventionnelles, mais suppléées par deux signaux de contrôle, indiquant

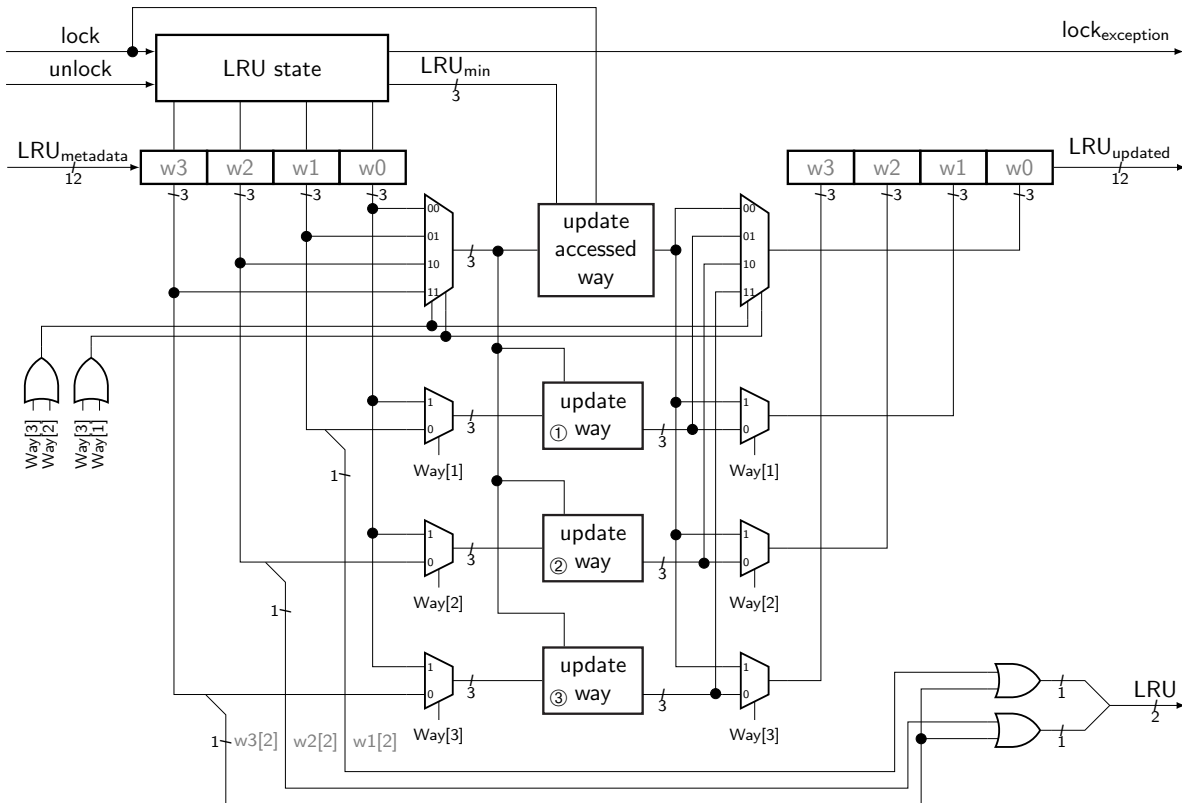


FIGURE 3.4 – **LRU-lock logical** - Description matérielle de la politique de remplacement LRU implémentant le mécanisme de verrouillage.

respectivement une opération de verrouillage ou de déverrouillage, générées et propagées vers le cache au travers du module de l'unité lecture et d'écriture (LSU). Dans le but d'implémenter le mécanisme de verrouillage présenté en Section 2.2.2, les métadonnées de la politique de remplacement sont étendues. Dans la Figure 3.3, le signal $LRU_{metadata}$ représente l'état des candidats de la politique de remplacement LRU du cache set sélectionné, et le signal $LRU_{updated}$ représente l'état des métadonnées mis à jour par le module *LRU-lock logical*. En considérant 4 voies, chaque candidat de la politique de remplacement LRU peut prétendre à 5 états : l'état 0 signifiant le statut verrouillé et les états 1, 2, 3, 4 signifiant les statuts de la politique de remplacement LRU. Ainsi, 3 bits sont nécessaires pour stocker l'état de chacune des voies, menant à 12 bits par cache set. Au total, 1536 bits (128 ensembles * 12 bits) sont nécessaires au stockage des états de la politique de remplacement LRU (e.g. stockés dans une demi-BRAM d'une puce FPGA de la série 7 de Xilinx). La Figure 3.4 détaille le module *LRU-lock logical* de la Figure 3.3 représentant la politique de remplacement du cache. Ce module met à jour les métadonnées lorsqu'un accès mémoire est réalisé et il fournit au cache la voie la moins récemment utilisée du cache set considéré, afin de réaliser l'éviction.

Description détaillée des modules

Le module *LRU state*, détaillé dans la Figure 3.5a, calcule la valeur d'état minimale LRU_{min} en prenant en compte le nombre actuel N_{lock} de lignes de cache verrouillées dans le cache set sélectionné (*i.e.* la valeur d'état minimale représente la valeur de l'état de la voie la plus récemment utilisée). La valeur du signal LRU_{min} dépend de l'instruction exécutée. Si une instruction `unlock` est traitée, alors LRU_{min} vaut N_{lock} . Autrement, pour les instructions `lock` et les instructions conventionnelles, LRU_{min} prend la valeur $N_{lock} + 1$. Le signal LRU_{min} est fourni en sortie du module *LRU state*.

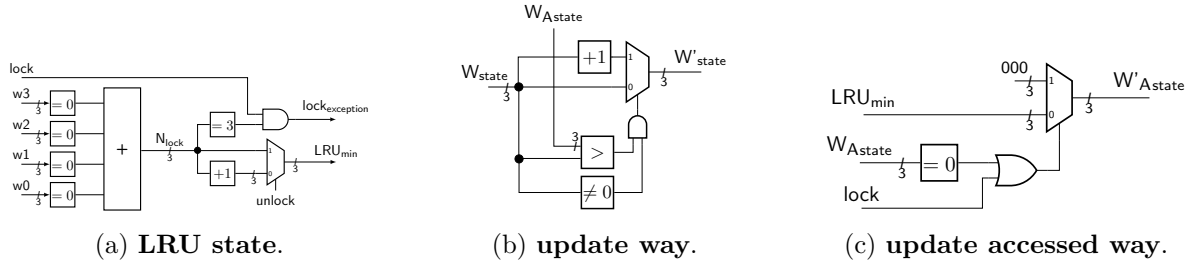


FIGURE 3.5 – Description matérielle des sous-modules utilisés dans la politique de remplacement.

Ce module lève également une exception lorsqu'une instruction `lock` est sollicitée au sein d'un cache set déjà entièrement verrouillé (*i.e.* lorsque 3 voies sont déjà verrouillées avec $N = 4$). L'exception est levée au travers du signal $lock_{exception}$. Dans la Figure 3.4, le signal $LRU_{metadata}$ contient les 12 bits des états de LRU pour l'ensemble des voies. À partir de ce signal, la voie la moins récemment utilisée est déterminée pour être propagée vers la sortie LRU . Le second objectif du module *LRU-lock logical* est de mettre à jour $LRU_{metadata}$ lorsqu'un accès est réalisé, comme exposé dans la Figure 2.3. L'état de la voie accédée est mis à jour grâce au module *update accessed way* nécessitant les signaux de l'état associé à la voie W_{Astate} , de la valeur LRU_{min} , et de $lock$. La description matérielle est détaillée dans la Figure 3.5c. Ainsi, si la voie accédée est déjà verrouillée $\boxed{=0}$ ou bien si une instruction `lock` est sollicitée, l'état de la voie accédée se voit attribuer la valeur 0, autrement il se voit attribuer la valeur LRU_{min} (*i.e.* lorsque la voie accédée n'est pas verrouillée). La sortie est alors allouée au sein du champ associé à la voie de la sortie $LRU_{updated}$ du module *LRU-lock logical*. Les trois autres champs de la sortie $LRU_{updated}$ sont mis à jour grâce aux trois modules *update way*, dont leur description est fournie dans la Figure 3.5b. Chacun des modules se voit attribuer ses entrées et distribuer ses sorties au travers d'un réseau de multiplexeur contrôlé par la voie accédée.

La Table 3.1 détaille comment sont attribués les trois modules *update way* et le module *update accessed way* en fonction de la voie accédée. Par exemple, lorsque la voie $w0$ est accédée (le signal *Way* égal à 0001_b), le module *update accessed way* met à jour l'état de la voie $w0$, le module *update way* ① met à jour l'état de la voie $w1$, le module *update way* ② met à jour l'état de la voie $w2$ tandis que le module *update way* ③ met à jour l'état de la voie $w3$.

TABLE 3.1 – Distribution des quatre modules de mises à jour en fonction de la voie accédée.

Way	0001 _b	0010 _b	0010 _b	1000 _b
Update accessed way	w0	w1	w2	w3
Update way ①	w1	w0	w1	w1
Update way ②	w2	w2	w0	w3
Update way ③	w3	w3	w3	w0

Le module *update way*, détaillé dans la Figure 3.5b, utilise l'état courant de la voie mise à jour ainsi que l'état courant de la voie accédée afin de mettre à jour l'état de la voie. Ce module fonctionne de la façon suivante : si la voie considérée n'est pas verrouillée $\boxed{\neq 0}$ et que la voie accédée est moins récemment accédée que la voie considérée (*i.e.* $W_{Astate} \boxed{>} W_{state}$), alors l'état de la voie considérée est incrémenté de 1 afin de renseigner la sortie du module. Autrement, l'état de la voie considérée dans ce module n'est pas incrémenté. Finalement, tous les états mis à jour sont propagés à la sortie $LRU_{updated}$ du module parent *LRU-lock logical*, afin d'être stockés dans la mémoire *LRU metadata memory*.

3.2 Évaluation

Dans cette section, nous évaluons le surcoût matériel de notre solution sur un processeur embarqué. Ensuite, nous évaluons la sécurité de notre mécanisme en s'assurant qu'il réponde bien au modèle de menace. Dans un dernier temps, nous évaluons les performances des applications utilisant notre solution. Nous évaluons également les performances des applications s'exécutant en parallèle de celles utilisant le mécanisme de verrouillage. L'évaluation de performance est complétée par un recensement du surcoût de la taille des programmes binaires pour deux algorithmes de chiffrement implémentant notre solution. Durant nos évaluations, nous nous sommes assurés que les données à verrouiller sont alignés en mémoire. Ainsi, le pas d'incrément utilisé lors de la phase de verrouillage et de déverrouillage est de $B=16$.

3.2.1 Évaluation de la surface matérielle

La Table 3.2 montre les résultats en surface post-implémentation pour une puce FPGA Xilinx de la famille Kintex-7 en utilisant le logiciel Vivado 2022.2. Les résultats de la politique de remplacement LRU sont extraits du cache et les résultats du cache et du cœur ont été extraits du CPU.

Les résultats montrent que notre solution n'impacte pas le nombre de BRAMs. Que ce soit dans le circuit de base ou le circuit protégé, 8 BRAMs sont implémentées pour la mémoire cache et une demie BRAM est attribuée au stockage des métadonnées de la LRU. Dans cette famille

TABLE 3.2 – Résultats de surface post-implémentation sur FPGA Kintex-7.

	without lock			with lock			Overhead	
	LUTs	FFs	BRAMs	LUTs	FFs	BRAMs	LUTs	FFs
→ LRU	26	24	0,5	50	34	0,5	+ 92,31%	+ 41,67%
→ Cache	980	1 065	8,5	1 007	1 077	8,5	+ 2,76%	+ 1,1%
→ Core	4 669	2 233	0	4 666	2 235	0	- 0,06%	+ 0,09%
CPU	5 661	3 467	8,5	5 683	3 481	8,5	+ 0,39%	+ 0,40%

de puce FPGA, une BRAM d'une taille de 36 ko peut en effet s'implémenter dans 2 BRAMs de 18 ko permettant d'optimiser l'utilisation de ce moyen de stockage pour des petites quantités de données. Attendant à ces résultats post-implémentation des BRAMs, le coût réel du nombre de bits stockés est caché derrière ce nombre de 8,5 BRAMs. Comme présenté dans l'équation 3.1, l'implémentation de base dans notre configuration de cache nécessite le stockage de 72 704 bits.

$$N * S * (8B + 11_{tag} + 1_{valid} + 2_{LRU}) = 72704 \quad (3.1)$$

où N=4, S=128, B=16

Concernant l'état de verrouillage inséré au sein des métadonnées, il est nécessaire d'ajouter 1 bit par ligne de cache. Dans notre configuration, cela ajoute un surcoût de 512 bits résultant en une augmentation de 0,7% des bits stockés pour une implémentation de type ASIC. Pour revenir sur les résultats, notre implémentation sur FPGA, présentée dans la Table 3.2, introduit une augmentation de 92,3% pour les LUTs et de 41,7% pour les registres Flip-Flops au sein de la politique de remplacement. Ce surcoût est important, mais est à relativiser au vu de la quantité de ressources impliquées. En effet, en observant le surcoût induit au niveau du cache, nous obtenons une augmentation de 2,8% pour les LUTs et de 1,1% pour les registres Flip-Flops au sein du cache. Concernant le cœur, l'ajout du mécanisme de sécurité a un effet négligeable malgré l'extension du module de décodage d'instruction et l'extension de la structure d'accès vers la mémoire. Nous observons même une baisse négligeable sur les LUTs, cette baisse est probablement induite par les optimisations de l'outil Vivado. Cependant, nous observons bien les 2 registres Flip-Flops supplémentaires pour stocker les états *lock* et *unlock* permettant la propagation des états au sein du pipeline jusqu'au cache. Concernant le CPU (englobant le cache et le cœur), le mécanisme de verrouillage mène à une augmentation de 0,4% pour les LUTs et pour les registres Flip-Flops. Il est intéressant de noter que cette faible augmentation a été introduite avec un processeur conçu à des fins de faible performance, tel que pour l'IoT. Ainsi, le coût d'étendre un processeur plus complexe avec notre mécanisme peut s'avérer encore plus faible. Au sujet de la fréquence maximale du système, une dégradation négligeable est observée passant de 105 à 104 MHz, cette baisse est considérée comme négligeable au vu de

l'incertitude de résultats que Vivado fournit sur cette métrique. Ainsi, notre solution mène à une faible augmentation de la surface et à un impact négligeable sur les performances temporelles du système.

3.2.2 Évaluation de la sécurité

Nous considérons l'algorithme de chiffrement AES-128 afin d'évaluer la sécurité. Il permet de chiffrer des blocs de 128 bits (16 octets) organisés en matrices d'octets de 4 par 4, appelées un état. L'algorithme de chiffrement AES-128 consiste en la concaténation de multiples rondes, 10, 12 ou 14 en fonction de la taille de la clé. Avant d'exécuter ces rondes, l'étape initiale `AddRoundKey` transforme le message clair en appliquant la clé grâce à l'opérateur XOR pour créer l'état d'entrée de la ronde 1.

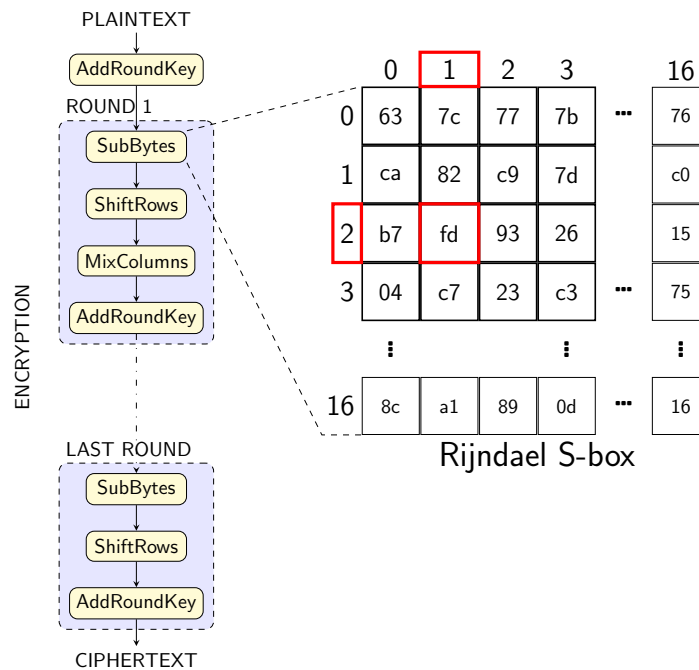


FIGURE 3.6 – Principe de l'algorithme de chiffrement AES-128 (inspirée de [128, 129]).

Dans notre cas, nous nous focalisons sur l'implémentation AES-128 logicielle sur 10 rondes avec une clé de 128 bits. Chacune de ces rondes se compose de 4 étapes :

1. `SubBytes` réalise une substitution octet par octet grâce à une table fixe.
2. `ShiftRows` réalise un décalage des colonnes qui composent la matrice (l'état).
3. `MixColumns` réalise un mélange des colonnes de la matrice (l'état).
4. `AddRoundKey` combine les octets de l'état avec une clé de ronde propre à chaque ronde.

Cela implique l'exécution de $10 - 1 = 9$ rondes auxquelles une ronde finale est appliquée. La ronde finale reprend les étapes 1, 2, et 4 des rondes classiques précédemment détaillées. L'ensemble des rondes qui compose l'algorithme de chiffrement est illustré dans la Figure 3.6.

Pour chaque ronde, lors de l'étape `SubBytes`, l'algorithme AES-128 procède à une substitution octet par octet au travers d'une table de 256 octets stockée en mémoire. Ainsi, pour substituer l'ensemble des octets de l'état courant, nous accédons à 16 reprises à la table de substitution (SBox). La taille de cette SBox est liée aux 256 valeurs que peut prendre un octet. Dans notre cas, nous utilisons la table de substitution de Rijndael.

Les accès répétés à la table stockée en mémoire passent au travers du cache, ainsi les lignes de cache associées aux octets de la SBox accédées se retrouvent en cache. Ce comportement rend sensible l'AES face à des attaques par canaux auxiliaires sur les mémoires caches. D'autant plus que lors de la première ronde, l'accès à la SBox dépend de la combinaison du bloc clair et de la clé (`AddRoundKey`). Ainsi, au travers d'une attaque en cache, un attaquant est capable de retrouver la clé, en considérant un message clair connu avec l'implémentation AES-128 dont les accès à la table de substitution dépendent du texte clair (P) et de la clé (K). Par conséquent, l'attaque `PRIME+PROBE` peut être utilisée pour déterminer quel index de la table de substitution a été accédé, et par conséquent, l'attaquant retrouve les valeurs de la clé K par association, car la ligne de cache accédée dépend des entrées $SBOX[P_i \oplus K_i]$ avec $0 \leq i \leq 16$.

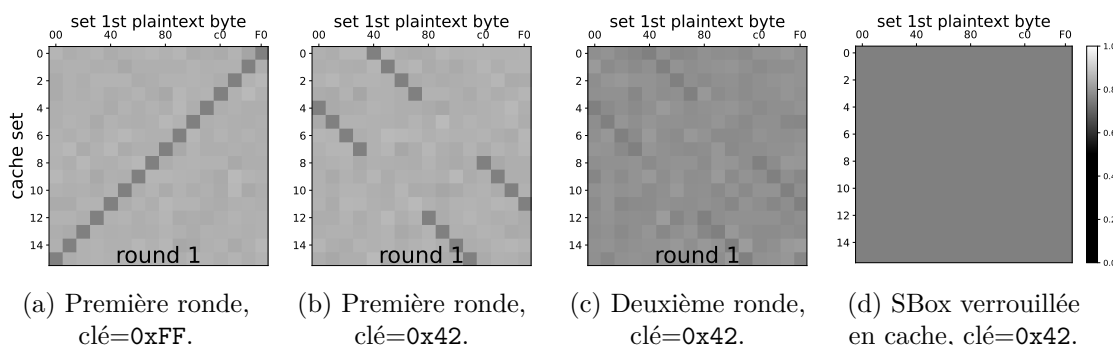


FIGURE 3.7 – Cartographie des résultats d'attaques en caches menées sur AES-128.

La Figure 3.7b illustre l'analyse de l'attaque après l'exécution d'une ronde de chiffrement AES non-protégée pour les 256 valeurs possibles du premier octet du message clair (les autres octets du message clair sont initialisés aléatoirement à chaque chiffrement) et la clé initialisée à la valeur `0x42`. À noter que la table de substitution est stockée dans 16 lignes de cache et est adressée sur 16 cache sets consécutifs. Pour la lisibilité, les figures se concentrent sur les 16 cache sets où la table de substitution est présente (parmi les 128 qui composent le cache). L'axe des ordonnées représente les lignes de cache observées lors de la phase `PROBE` de l'attaque, et l'axe des abscisses représente la valeur du premier octet du message clair. Les résultats observés sur ces cartes représentent le taux moyen de `hit` mesuré par l'attaquant sur 300 chiffrements. Dans

la Figure 3.7, plus le carré est sombre, plus la victime a évincé les données de l'attaquant sur ce cache set entre la phase PRIME et la phase PROBE de l'attaque. Nous pouvons distinguer un pattern induit par les accès à la table de substitution qui dépend du premier octet du message clair qui est fixé et connu de l'attaquant ainsi que de la valeur de la clé secrète. La Figure 3.7a met en lumière comment la valeur de la clé manipule la forme du pattern distingué lors de l'attaque. Au travers de la Figure 3.7c, nous observons comment le pattern se fond dans les accès des rondes suivantes, ce comportement est dû à la propriété de diffusion de l'algorithme AES. Cependant, il est encore possible de discerner un pattern, mais plus le nombre de rondes augmente, plus le pattern se dissipe. Par conséquent, dans la suite de l'analyse, nous considérons le pire scénario d'attaque où l'attaquant est capable de réaliser son analyse de PROBE après la première ronde AES.

Dans le but d'évaluer notre mécanisme de verrouillage, la Figure 3.7d montre qu'un attaquant ne peut déduire aucune information lorsque la victime accède à la table de substitution en mémoire cache. Une fois verrouillée en mémoire cache, l'attaquant n'est capable d'observer qu'un taux constant de `hit` de 75% au sein de la plage des cache sets comprenant la table de substitution en mémoire cache. Ce taux est propre à l'auto-évacuation de ses données permettant de remplir le cache afin de mener l'attaque PRIME+PROBE. En effet, l'attaquant utilise un ensemble d'évacuations composé de 4 adresses pour remplir chaque cache set. Durant la phase de PROBE, l'attaquant évince lui-même une de ses données puisque l'une des voies est verrouillée par le processus victime.

Le taux de 75% obtenu lorsque la victime verrouille la table de substitution est lié au fait qu'une ligne de cache soit verrouillée dans chaque cache set. De même, lorsque la victime verrouille deux lignes de cache au sein du cache set, l'attaquant observerait un taux de 50% et un taux de 25% pour 3 lignes verrouillées. Par ce biais, l'attaquant peut connaître le nombre de lignes de cache verrouillées dans le cache. Ainsi, le nombre de lignes de cache verrouillées dans le cache ne doit pas dépendre d'un secret. Dans un autre contexte, les instructions `lock` et `unlock` peuvent être utilisées pour établir une communication par canal caché entre deux applications ne partageant pas le même espace d'adressage. En se focalisant sur un cache set, la communication peut permettre l'envoi d'une information parmi N états sur chaque période pour $\{0 \dots N\}$ lignes de cache verrouillées.

3.2.3 Évaluation des performances

Cette section étudie l'impact du mécanisme de verrouillage en cache sur les performances. Premièrement, nous évaluons le surcoût introduit par l'utilisation des instructions `lock` et `unlock` sur le temps d'exécution de différents algorithmes de chiffrement. Ensuite, nous évaluons l'impact sur les performances de la rétention de ressources en cache par l'instruction `lock` sur les applications qui n'utilisent pas le mécanisme de verrouillage de lignes de cache.

Finalement, nous évaluons l’impact de l’utilisation des instructions `lock` et `unlock` sur la taille du programme binaire compilé.

Pour réaliser notre évaluation, nous nous basons sur une carte de développement Digilent Genesys 2 sur laquelle une puce FPGA est utilisée pour implémenter notre architecture de cœur et de cache. Le système, cadencé à 100 MHz, est implémenté avec le cœur CV32E40P basé sur l’ISA RISC-V et intégrant les instructions `lock` et `unlock`. Accolé au cœur, le système implémente un cache dont la configuration est présentée en Section 3.1.2. En plus du cache, la hiérarchie mémoire est complétée avec une mémoire DDR RAM externe dont le bus de données est de 32 bits. Lorsqu’un cache miss est obtenu, une latence de 36 cycles est observée pour récupérer une ligne de cache depuis la DDR RAM.

N’ayant pas encore de prototype fonctionnel pour mener notre évaluation sur carte FPGA, nous simulons notre système à l’aide du simulateur Verilator [130]. Verilator simule le modèle comportemental au cycle prêt de circuits décrit en langage de description matériel (*e.g.* SystemVerilog). Lors des simulations, nous enregistrons l’ensemble des accès à la mémoire principale afin d’appliquer le coût d’accès à la DDR RAM au nombre de cycles simulés par l’outil Verilator.

Impact sur les performances

TABLE 3.3 – Hausse de temps d’exécution (exprimé en %) introduit par l’utilisation du mécanisme de verrouillage sur Camellia et AES-128.

N_b	1	4	8	16	64	128	512	1024
Camellia	367,7	99,6	54,22	28,88	7,62	3,85	0,97	0,48
AES-128	2,77	0,71	0,35	0,18	0,04	0,02	-	-

Dans le but de mener l’évaluation, nous considérons deux algorithmes de chiffrements symétriques par blocs : AES-128 et Camellia. Les deux algorithmes chiffrent par blocs de 128 bits. Là où AES-128 utilise une table de substitution de 256 octets, Camellia requiert quatre tables de substitution de 1 024 octets chacune. L’évaluation compare le temps d’exécution entre un chiffrement conventionnel et un chiffrement protégé avec N_b chiffrements consécutifs. Alors que l’exécution conventionnelle comprend uniquement le chiffrement, l’exécution protégée comprend une exécution en trois étapes comme indiqué dans le Listing 3.1 : verrouiller les tables de substitution, chiffrer N_b blocs de message clairs, puis déverrouiller les tables de substitution. Durant cette évaluation, il est important de noter qu’aucune autre application ne vient perturber l’état du cache et du cœur. La Table 3.3 présente le coût supplémentaire en temps d’exécution (exprimé en pourcents) engendré par l’insertion des instructions `lock` et `unlock` par nombre de blocs N_b chiffrés pour les deux algorithmes de chiffrement AES-128 et Camellia. Concernant Ca-

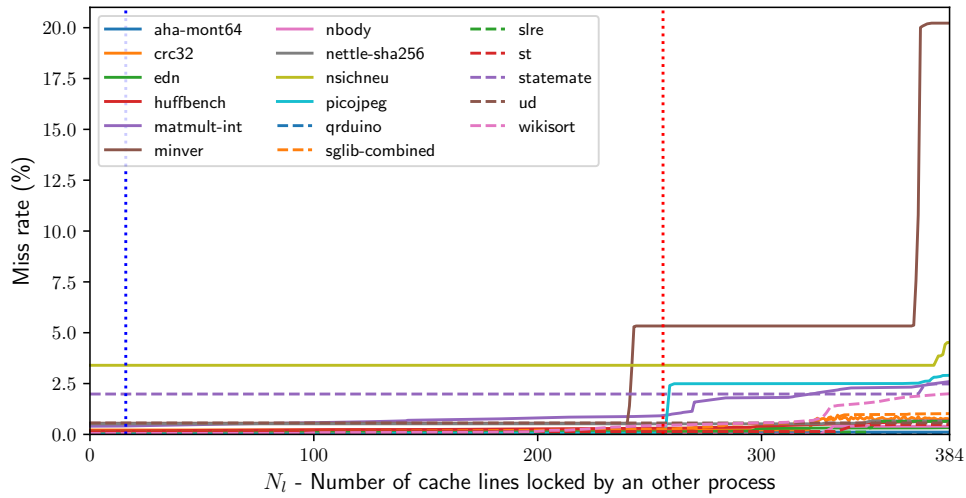
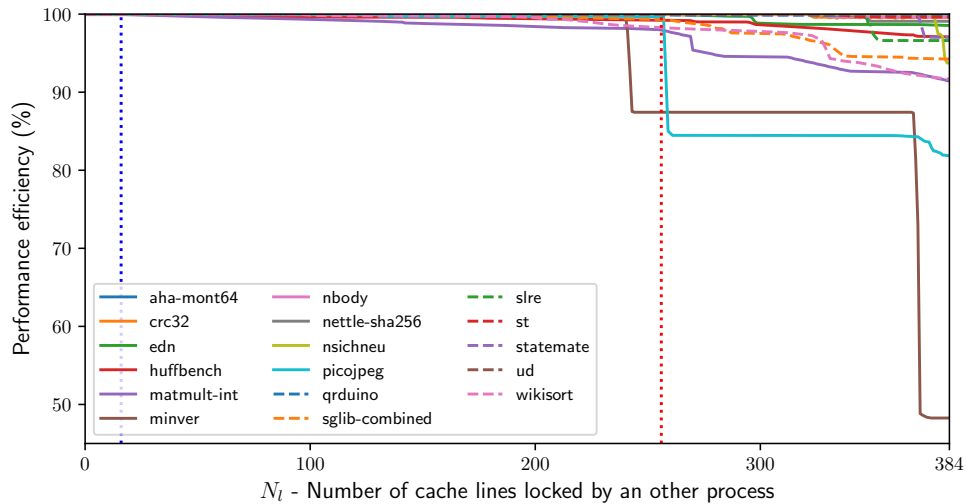
mellia, pour le chiffrement d'un seul bloc ($N_b = 1$), l'augmentation du temps de chiffrement est de l'ordre de 368%. Ce coût très élevé est causé par le faible nombre d'opérations nécessaire pour Camellia et que beaucoup d'accès mémoires sont nécessaires pour verrouiller les quatre tables de substitution de 1,024 octets chacune. Cependant, lors d'un usage plus réaliste, *i.e.* lorsqu'un plus grand nombre de blocs sont chiffrés consécutivement ; l'augmentation est en dessous de 8% dès 64 chiffrements et atteint moins de 1% pour 512 chiffrements consécutifs.

À propos de AES-128, une augmentation de 2,8% est observée sur le temps d'exécution d'un chiffrement, puis diminue à 0,7% pour 4 chiffrements et atteint moins de 0,2% pour 16 chiffrements consécutifs. Ces résultats montrent que l'utilisation du mécanisme de protection à un faible impact sur les performances et peut être considérée comme négligeable lorsque le nombre de blocs N_b à chiffrer devient suffisant.

Impact sur les applications concurrentes

Dans cette section, nous évaluons la baisse des performances introduite par la contention de lignes de cache verrouillées sur une application n'utilisant pas le mécanisme de verrouillage. L'évaluation est menée avec la suite de logiciels de référence du benchmark Embench-IoT 1.0 [131]. La Figure 3.8 montre les résultats en performances du benchmark à l'aide de deux métriques, le taux de *miss* et le temps d'exécution en cycles. Au sein de ces deux graphes, l'axe des abscisses représente le nombre de lignes de cache verrouillées N_l par un processus utilisant le mécanisme de verrouillage pour se prémunir contre des attaques en cache. La plage de l'axe des abscisses (allant de 0 à 384) est liée à la configuration de notre cache dans lequel 3 voies peuvent être verrouillées dans chacun des 128 cache sets, menant à la plage haute de $3 * 128 = 384$. Pour illustrer un usage selon des conditions réalistes du mécanisme de verrouillage, deux lignes verticales en pointillées ont été insérées dans la Figure 3.8. La ligne bleue, localisée à 16 lignes de cache verrouillées ($N_l = 16$), représente le nombre de lignes de cache à verrouiller pour la SBox de AES-128, tandis que la ligne rouge, localisée à 256 lignes de cache verrouillées, correspond au besoin des quatre SBox de Camellia. La Figure 3.8a montre le taux de *miss* pour toutes les applications du benchmark lorsqu'une application concurrente verrouille N_l lignes de cache. La plupart des applications n'utilisent pas l'entièreté du cache. Par conséquent, le taux de *miss* reste stable et bas au même niveau que lorsque le cache est entièrement disponible ($N_l = 0$). Cependant, quelques applications souffrent d'une augmentation du taux de *miss* lorsque le nombre de lignes de cache verrouillées se rapproche de 384. En effet, à 384 lignes de cache de verrouillées, le cache se comporte comme un cache directement adressé (Direct-Mapped cache) en raison de la contention de 3 voies parmi 4 dans chacun des cache sets dont il reste une seule voie disponible pour l'exécution d'autres applications.

Ce comportement est également observé en Figure 3.8b où est représentée le temps d'exécution en cycles par rapport à une exécution sans ligne de cache verrouillée. C'est pourquoi, lorsque

(a) Taux de cache *miss*.

(b) Temps d'exécution.

FIGURE 3.8 – Impact sur sur la suite Embench-IoT 1.0 lorsqu'une application concurrente verrouille N_l lignes de cache. La ligne verticale bleue en pointillés représente le besoin de l'application AES-128. La ligne verticale en pointillés rouge représente le besoin de l'application Camellia.

$N_l = 0$, l'efficacité vaut 1. Cependant, nous pouvons noter une dégradation non marginale des performances et du taux de *miss* pour le noyau `minver` dès lors qu'une application concurrente verrouille plus de 230 lignes de cache. À partir de ce seuil, la pile logicielle du noyau `minver` rentre en collision avec ses propres données et cause une auto-éviction dû à l'espace restreint disponible en cache. Ce même comportement est observé pour le noyau `picojpeg` lorsqu'une application concurrente excède 256 lignes de cache verrouillées.

TABLE 3.4 – Évaluation des performances et du taux de *miss* sur Embench-IoT 1.0 lorsque AES-128 ou Camellia verrouille sa(s) table(s).

Bench	Performance		Miss Rate		
	AES-128	Camellia	No lock	AES-128	Camellia
	$N_l = 16$ (%)	$N_l = 256$ (%)	$N_l = 0$ (%)	$N_l = 16$ (%)	$N_l = 256$ (%)
aha-mont64	100	100	0,11	0,11	0,11
crc32	100,04	99,95	0,22	0,22	0,22
edn	100	99,99	0,12	0,12	0,12
matmult-int	99,96	97,95	0,41	0,40	0,93
minver	100	87,43	0,55	0,55	5,33
nbody	99,92	99,92	0,01	0,01	0,01
nettle-sha256	100	100	0,52	0,52	0,52
nsichneu	100	100	3,40	3,40	3,40
picojpeg	100	99,60	0,01	0,01	0,07
qrduino	100	99,93	0,01	0,01	0,02
sglib-comb.	100	99,31	0,15	0,15	0,25
slre	100	100	0,12	0,12	0,12
st	100	100	0,13	0,13	0,13
statemate	100	100	1,99	1,99	1,99
ud	100	100	0,56	0,56	0,56
wikisort	100	98,28	0,04	0,04	0,45
Average	99,99	98,92	0,50	0,50	0,86

Le détail des temps d'exécution en cycles et le taux de *miss* induit si AES-128 (ligne pointillée bleue) ou Camellia (ligne pointillée rouge) verrouille leur(s) table(s) de substitution sont rapportés dans la Table 3.4. Les colonnes 1 et 2 présentent les résultats des performances, qui sont extraits de la Figure 3.8b lorsque respectivement AES-128 et Camellia utilise le mécanisme de verrouillage. Ensuite, les colonnes 3 à 5 présentent respectivement les résultats du taux de *miss*, extrait de la Figure 3.8a, lorsqu'aucune ligne de cache n'est verrouillée, lorsque AES-128 verrouille sa table de substitution, puis lorsque Camellia verrouille ses tables. Concernant les performances, en considérant la SBox de AES-128 verrouillée, nous observons une faible dégradation des performances moyennes (<0,01%). Seulement les applications `matmult-int` et `nbody` sont légèrement affectées. Nous observons une accélération du noyau `crc32`, cette accélération est due à l'utilisation d'entrées générées aléatoirement modifiant les patrons des accès mémoires. Concernant Camellia, la baisse de performance est en moyenne de 2% pour l'ensemble du benchmark. La plupart des applications sont légèrement impactées. Cependant, `minver` voit ses performances dégrader de 13%.

Au sujet du taux de *miss*, AES-128 n'affecte pas les applications concurrentes grâce à sa faible charge mémoire. Cependant, les 256 lignes de cache verrouillées par Camellia engendrent un taux de *miss* moyen de 0,86% pour l'ensemble des applications du benchmark, cela représente une hausse induite de 0,36% comparé au cas où aucune donnée n'est verrouillée (*No lock*). Notamment, nous observons que le noyau le plus impacté est également `minver` passant de 0,55% à 4,33%. Les autres applications sont, quant à elles, épargnées au niveau de cette métrique.

Taille du programme binaire

Finalement, nous évaluons le surcoût au niveau de la taille du programme binaire lorsque nous protégeons les algorithmes de chiffrement AES-128 et Camellia en insérant les instructions `lock` et `unlock` comme expliqué dans la Section 3.1.1. Chacun des binaires est généré avec la même configuration de la chaîne de compilation (*i.e.* sans option d'optimisation, en statique `-static`, en fournissant les spécifications de la cible `-mabi=ilp32 -march=rv32imc`, et en liant la librairie mathématiques `-lm`). Concernant AES-128, notre implémentation non protégée, décrite en langage C, produit un binaire de 71,7 ko. L'insertion des instructions `lock` et `unlock` produit un binaire de 71,9 ko, engendrant une augmentation de 0,28 % de la taille du programme binaire. Pour Camellia, la taille du binaire de notre implémentation non protégée est de 213,5 ko alors que celle de l'implémentation protégée est de 214,0 ko, introduisant une augmentation de 0,23 %.

3.3 Conclusion

Dans ce chapitre, nous avons décrit notre implémentation du mécanisme de verrouillage en cache au sein du premier niveau de cache de données. Puis, nous avons mené une évaluation de notre implémentation. Nous avons observé que l'implémentation engendre un faible coût sur le circuit. Notre implémentation est sécurisée et permet de répondre aux problématiques de sécurité du modèle de menaces. Dans un dernier temps, nous avons observé de faibles pertes de performances liées à l'utilisation de notre mécanisme. Ce faible surcoût est observé tant pour les applications qui se protègent contre les attaques en caches que pour les applications s'exécutant concurremment.

Suite à notre analyse portant sur la sécurité, nous avons souligné que le nombre de lignes de cache verrouillées peut être inféré par un processus attaquant. Pour rappel, afin de maintenir la sécurité lors de l'utilisation de notre mécanisme, nous alertons que la quantité de données à verrouiller ne doit pas dépendre du secret. Cependant, cette fuite d'information peut être volontairement utilisé pour établir un nouveau canal de communication caché entre deux processus.

SOLUTION HYBRIDE DU MÉCANISME DE VERROUILLAGE

SOMMAIRE

4.1	Motivations	81
4.2	Solution Proposée	82
4.2.1	Choix de la fonction de dérivation d'index	83
4.2.2	Choix de la politique de remplacement	84
4.3	Implémentation	85
4.3.1	Système considéré	85
4.3.2	Implémentation au sein du cache	87
4.3.3	Description détaillée des modules	89
4.4	Impact de l'hybridation sur la sécurité	90
4.5	Évaluation	90
4.5.1	Évaluation de la surface matérielle	91
4.5.2	Évaluation des performances	92
4.6	Conclusion	93

Dans ce chapitre, nous proposons une nouvelle solution originale basée sur l'hybridation de deux mécanismes de sécurité. D'une part, notre mécanisme de verrouillage de lignes de cache et d'autre part les architectures à indexation asymétrique des voies. Cette solution nous permet d'apporter de nouvelles garanties de sécurité et de flexibilité pour un faible coût au niveau du matériel et des performances. Notre choix de politique de remplacement a été motivée par l'analyse des politiques de remplacement sur des architectures de mémoires caches basées sur l'aléa. Cette analyse a été réalisée dans le cadre d'une mobilité de trois mois à la Ruhr University Bochum, j'ai été accueilli dans l'équipe du Pr. Güneysu. L'analyse a été publiée à la conférence ASIA CCS [132] : nous y avons participé en apportant des évolutions sur les spécifications des politiques de remplacement considérées, ainsi qu'en proposant des implémentations matérielles.

4.1 Motivations

Comme présenté dans la Section 1.3, les solutions basées sur une indexation asymétrique des voies se prémunissent des attaques en monitorant le temps d'exécution en comparant les traces avant et après l'éviction des données présentes dans un cache set. De plus, elles permettent de se prémunir des attaques inférant les évictions induites par le programme victime en monitorant l'état d'un cache set. Ce type de solutions est efficace grâce à l'ajout d'aléa entre l'adresse et les indexs accédés dans chaque voie. Cela permet de rendre inefficace les ensembles d'éviction classiques utilisés par l'attaquant pour évincer et monitorer les données en mémoire cache. En effet, l'attaquant doit réaliser un grand nombre d'accès mémoires afin de faire coïncider les nouvelles adresses de son ensemble d'éviction avec les données de la victime. Tant que l'ensemble d'éviction n'est pas construit, l'exécution des applications est immune face aux attaques par canaux auxiliaires exploitant les mémoires caches considérées. En fonction de la configuration du cache, cette phase peut nécessiter entre quelques milliers et plusieurs millions d'accès mémoires. C'est la raison pour laquelle un renouvellement de la génération d'aléa est nécessaire tous les n accès mémoires afin de maintenir un niveau de sécurité suffisant (n est le nombre d'accès mémoires nécessaires pour construire un ensemble d'éviction dans la configuration du cache). Le renouvellement de la génération d'aléa est propre à l>IDF utilisée. Cela peut être un renouvellement de clés si l>IDF manipule une clé, ou bien le renouvellement d'une seed utilisée pour les permutations. Dans notre solution, nous considérerons une IDF basée sur une clé, ainsi le terme de renouvellement de clé est utilisé pour parler de cette phase. Le renouvellement de clé engendre une perte de performances conséquente puisqu'elle induit l'invalidation de l'ensemble du cache.

D'autre part, le mécanisme de verrouillage de lignes de cache a été précédemment introduit dans ce manuscrit. À l'aide de deux nouvelles instructions étendant l'ISA RISC-V, nous proposons de verrouiller dynamiquement des lignes de cache au sein du cache de données. Cette solution permet de se prémunir des mêmes menaces que la solution précédente, mais en tirant profit d'un autre principe. Le principe consiste à séparer physiquement les ressources matérielles allouées aux applications. Ainsi, les accès mémoires sensibles sont réalisés sur des données verrouillées et ne peuvent plus être inférés par les accès mémoires d'un attaquant. De plus, le mécanisme apporte une garantie d'exécution en temps constant sur les accès mémoire réalisés sur des lignes de cache verrouillées. Cependant, ce mécanisme ne peut verrouiller qu'une quantité limitée de données en mémoire cache. Cette solution introduit également un nouveau canal caché permettant à deux applications ne partageant pas le même espace d'adressage de communiquer entre elles en monitorant le nombre de lignes de cache verrouillées.

À partir de ce constat, nous proposons une idée originale avec une architecture basée sur l'hybridation des deux mécanismes de sécurité. Cela nous garantit l'exécution en temps constant des sections de code critiques. De plus, cela empêcherait la fuite d'un secret lorsqu'une appli-

cation verrouille un nombre de lignes dépendant du secret. Ce cas apparaît lorsque l'utilisateur utilise explicitement les instructions `lock` et `unlock` en ne suivant pas les recommandations d'utilisation. Deuxièmement, cela réduirait le débit du canal caché provoqué par la contention des ressources induite par l'utilisation de l'instruction de `lock`.

Concernant les limitations de l'architecture à indexation asymétrique des voies, l'architecture hybride permettrait d'apporter une isolation supplémentaire pour les applications requérant un haut niveau de sécurité en verrouillant les lignes de cache dont les accès sont sensibles tout en apportant une diffusion des autres accès au travers de la réindexation.

4.2 Solution Proposée

La solution proposée s'appuie sur une architecture de mémoire cache à indexation asymétrique de voies (*e.g.* CEASER-S [81], ScatterCache [82]). Au-delà de l'approche basée sur l'insertion d'aléa, le verrouillage de lignes de cache permet de garantir à une application un cache hit sur les données verrouillées.

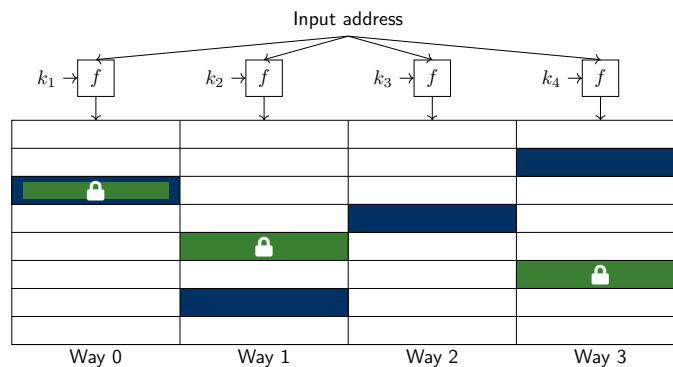


FIGURE 4.1 – Schéma de principe de notre architecture de mémoire cache hybride. Les lignes de cache apparaissant en vert sont verrouillées, celles en bleu sont sélectionnées par l'IDF.

La Figure 4.1 présente le principe de notre solution dans laquelle une gestion asymétrique des voies est couplée au mécanisme de verrouillage de lignes de cache. Une fonction de dérivation d'index (IDF) f est assignée à chaque voie du cache. Chaque IDF prend comme entrée tout ou partie de l'adresse ainsi qu'une clé. L'attribution d'une clé unique pour chaque IDF crée l'asymétrie dans le cache. Comme illustré dans la Figure 4.1, des lignes de cache verrouillées peuvent être accédées comme dans la précédente solution. Cependant, dès lors qu'une ligne de cache est verrouillée sur chaque voie, il est possible qu'un accès mémoire se voit attribuer quatre lignes de cache verrouillées par les IDFs. Bien que peu probable, ce cas existe et mérite d'être traité. Plus le cache est petit, plus la probabilité de rencontrer ce cas est grande. Pour se prémunir de ce cas particulier, nous pouvons limiter le verrouillage de lignes de cache à 3 voies (*e.g.* la voie

4 n'est jamais considérée lors d'une instruction `lock`). Un problème de cette restriction matérielle apparaît lorsqu'une instruction `lock` adresse une donnée déjà présente dans une ligne de cache résidant au sein de la voie non verrouillable. Il faudrait alors invalider la ligne de cache pour la déplacer dans une voie verrouillable. Un autre moyen de palier ce problème serait de lever une exception afin de donner la main au système d'exploitation. Malheureusement, ces problèmes ont été révélés tardivement lors de nos expérimentations. Ainsi, les résultats relevant de ce chapitre sont produits à titre expérimentaux et devront être reconsidérés compte tenu de ces limitations. Cependant, la démarche proposée ainsi que notre discussion à propos de la sécurité sont valides. Seule l'implémentation et les résultats qui en découlent peuvent être impactés.

Dans notre solution, les spécifications du mécanisme de verrouillage de lignes de cache restent inchangées par rapport aux chapitres précédents. Les choix de l'IDF et de la politique de remplacement sont motivés dans la suite de cette section.

4.2.1 Choix de la fonction de dérivation d'index

Il existe plusieurs IDF permettant de réindexer un cache set en utilisant l'adresse et une clé comme entrées. PRESENT [102], MANTIS [103], QARMA [104] et SCARF [105] sont des algorithmes de chiffrement par blocs à faible latence et à faible coût matériel. Notre choix porte sur SCARF en raison de sa très faible latence (divisé par 2 par rapport à l'état de l'art) et son faible coût en surface [105].

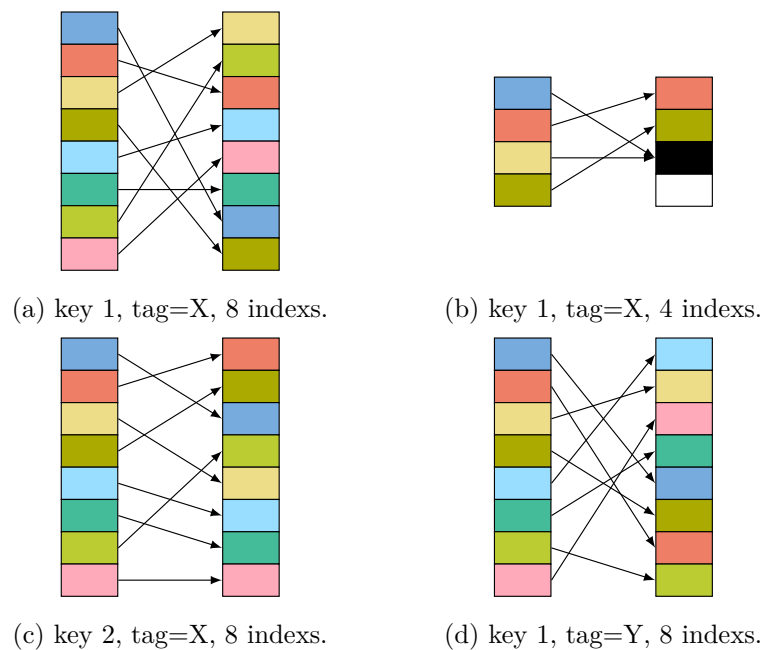


FIGURE 4.2 – Cas d'étude de l'utilisation de SCARF avec plusieurs configurations.

Le cas d'étude illustré dans la Figure 4.2 permet de présenter comment l'IDF SCARF se comporte au travers de plusieurs configurations. Dans le cas d'étude, nous considérons que le module SCARF est conçu pour un usage à 8 indexs. Au sein de ces figures, la colonne de gauche représente les entrées correspondant aux indexs, les flèches représentent l'attribution des entrées vers leurs sorties respectives, la colonne de droite représente les sorties. Par exemple, dans la Figure 4.2a, l'entrée orange correspond au champ d'index d'une adresse, la correspondance établie vers la sortie dépend de la clé et du tag de l'adresse. La Figure 4.2a présente comment les entrées se diffusent vers les sorties en maintenant une clé et un tag. La Figure 4.2b illustre le cas où l'implémentation recourt à un nombre réduit d'indexs. Nous observons que pour 4 index en entrées, seules 3 sorties sont considérées. Par ailleurs, une sortie (représentée en noir) est utilisée à deux reprises et une sortie n'est pas considérée (représentée en blanc). En effet, le module SCARF est conçu pour manipuler des entrées de 10 bits (3 dans notre cas d'étude présenté dans la Figure 4.2). De ce fait, la propriété de diffusion n'est plus garantie lorsque les entrées ou les sorties ne considèrent pas l'ensemble traité par le module SCARF. D'autre part, au travers des Figures 4.2c et 4.2d, nous observons que la diffusion proposée par le module SCARF est totalement modifiée lorsque l'un des deux paramètres est modifié (*i.e.* la clé pour la Figure 4.2c et le tag pour la Figure 4.2d). Concernant la Figure 4.2c, cela démontre que SCARF a un comportement déterministe et que seul le renouvellement de la clé permet de renouveler la diffusion. D'autre part, les modules SCARF ont besoin de manipuler des clés uniques lorsqu'il s'agit d'une architecture de cache à gestion asymétrique des voies. Concernant la Figure 4.2d, cela démontre que la correspondance change à chaque changement de tag. Cela signifie que deux zones mémoires distantes ne partagent pas la même diffusion.

Le module SCARF considéré dans notre implémentation manipule une clé de 240 bits avec un index d'entrée sur 10 bits et utilise également un tag de taille variable. En sortie, le module SCARF fournit un index de 10 bits. En fixant une clé et un tag, les $2^{10} = 1024$ entrées sont diffusées en 1024 sorties uniques. L'utilisation de SCARF en tant qu'IDF implique de considérer 1024 cache sets. Autrement, les propriétés de diffusion ne sont plus garanties.

4.2.2 Choix de la politique de remplacement

En complément du projet SCRATCHS, une contribution additionnelle a été effectuée lors d'une mobilité internationale de trois mois dans l'équipe de Tim Güneysu. Dans cette contribution publiée à la conférence ASIA CCS [132], nous analysons plusieurs politiques de remplacement dans le contexte des architectures de caches basées sur la réindexation asymétrique. Cette analyse a permis d'évaluer et de comparer les politiques de remplacement en termes de sécurité face à PRIME+PRUNE+PROBE [67], de performances et de surface matérielle. L'ensemble des politiques considérées obtiennent des résultats proches, ne permettant pas de distinguer la meilleure politique de remplacement grâce aux critères du coût en surface matérielle et de

performances. Le critère de la sécurité permet de les départager grâce à une analyse portant sur le nombre d'accès mémoire nécessaire pour construire un ensemble d'éviction dans un environnement sans bruit [133]. Suite à cette analyse, nous sélectionnons VARP-64 (Variable Age Replacement Policy) comme politique de remplacement. Contrairement à LRU, VARP-64 ne manipule pas des états qui indiquent l'ordre du dernier accès réalisé parmi les lignes de cache. VARP considère des états qui manipulent l'âge de la ligne de cache allant de 0 à 63 accès (*i.e.* dans le cas de VARP-64). L'âge s'incrémente à chaque accès lorsque la ligne de cache est considérée par les IDFs. Si la ligne de cache est accédée, alors son âge est réinitialisé à 0. La ligne de cache la plus âgée est sélectionnée pour l'éviction. Si plusieurs lignes de cache sont les plus âgées, alors le candidat à l'éviction est sélectionné aléatoirement parmi les plus âgées. Dans le cas de VARP-64, 6 bits sont nécessaires au stockage de l'âge pour chaque ligne de cache. Dans le but d'implémenter le mécanisme de verrouillage, un bit indiquant le verrouillage est ajouté aux métadonnées de chaque ligne de cache. Si la ligne de cache est verrouillée, l'incrémementation de l'âge de la ligne de cache est stoppée et la ligne de cache n'est pas considérée pour l'éviction. De plus, lorsqu'une ligne de cache verrouillée est accédée, l'âge des autres lignes de cache n'est pas incrémenté.

4.3 Implémentation

Dans cette section, nous détaillons l'implémentation matérielle de notre mécanisme de verrouillage au sein d'un cache à indexation asymétrique des voies. Une description détaillée des modules matériels est présentée avec une attention particulière sur le cache et sa politique de remplacement.

4.3.1 Système considéré

Le processeur considéré se base sur le cœur CV32E40P [127] qui, pour rappel, s'appuie sur les instructions RV32IMCZicsr. Le cœur accède à la mémoire principale au travers d'un niveau de mémoire cache L1 dans lequel les interfaces d'instructions et de données sont connectées à leurs caches respectifs. Les mémoires caches ont une architecture à voies asymétriques contrôlées par SCARF [105]. Ils utilisent la politique de remplacement VARP-64. De plus, la mémoire cache de données implémente le mécanisme de verrouillage. La description de l'architecture du processeur est présentée dans la Figure 4.3.

L'ensemble des modules liés à l'ajout de l'exécution en temps constant (*i.e.* instructions de divisions et modulus), du mécanisme de verrouillage et des mémoires caches sont colorés. Les modules colorés en gris correspondent aux modules implémentés. Les modules verts correspondent aux deux options que nous proposons pour renouveler les clés des modules SCARF. Le renouvellement des clés n'est pas implémenté dans notre solution évaluée. La première option est

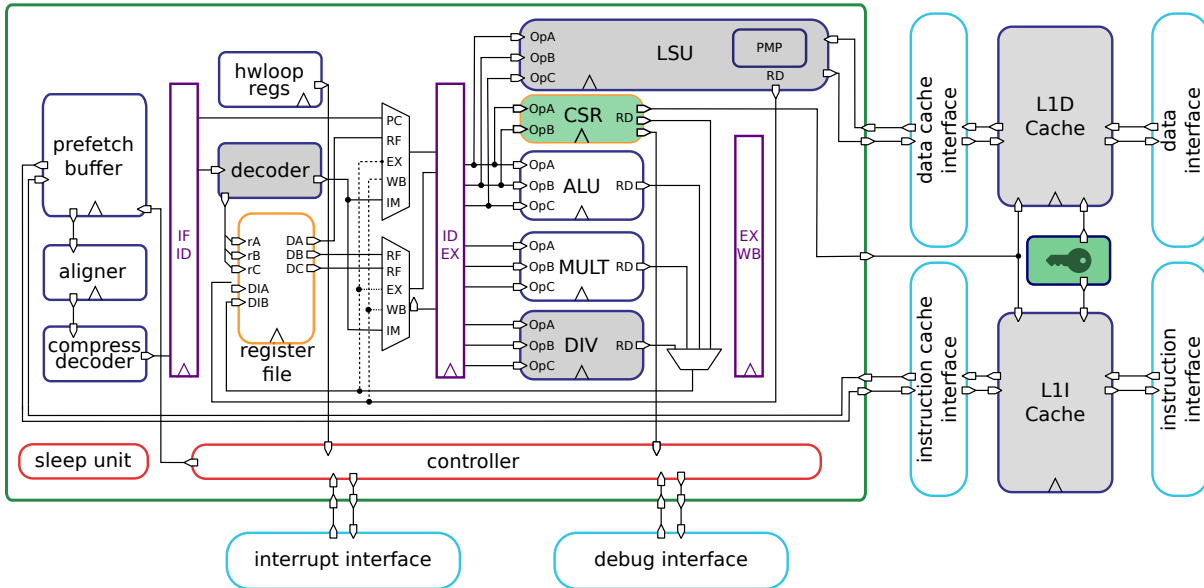


FIGURE 4.3 – Description matérielle du système incluant le cœur CV32E40P et un niveau de mémoires caches données et instructions (inspiré de [4]).

basée sur un support matériel utilisant un générateur de nombre pseudo aléatoire pour générer les clés de 240 bits nécessaires au SCARF. La seconde option est basée sur le support du logiciel grâce aux CSRs pour renseigner les clés à utiliser (*e.g.* le système d'exploitation). Cependant, elle implique également un support du matériel pour acheminer les clés vers le cache depuis l'étage d'exécution dans lequel les CSRs sont implémentés.

L'utilisation d'un cache dédié aux instructions permet d'harmoniser le premier niveau de mémoire cache. D'autre part, en considérant ce cache implémentant une architecture asymétrique, nous nous prémunissons des attaques par canaux auxiliaires contre les mémoires caches d'instructions. Ces attaques ciblent les branchements et sauts de programmes qui permettent d'obtenir des informations lorsqu'ils dépendent du secret. Ce type de vulnérabilités n'est pas considéré dans notre modèle de menace. Ainsi, nous n'évaluerons pas la sécurité sur cet aspect.

Induit par l'utilisation de la fonction IDF SCARF sur 10 bits, notre cache considère $N = 4$ voies pour $S = 1024$ cache sets. Les lignes de cache ont une taille de $B = 16$ octets. Au total, notre cache a une capacité de 64 Ko. La grande taille du cache de donnée associée à la faible charge mémoire des applications des benchmarks considérés ne permettent pas de rencontrer les problématiques énoncées dans la Section 4.2. La mémoire est adressée sur 22 bits et le cache utilise les $\log_2(B) = 4$ bits de poids faibles pour l'offset, les $\log_2(S) = 10$ bits suivants pour l'index et les 8 bits restant pour le tag.

4.3.2 Implémentation au sein du cache

L'architecture de cache considérée dans notre implémentation est représentée dans la Figure 4.4. Notre implémentation considère un cache de 4 voies avec 1024 cache sets. Chaque voie est dissociée et est accédée individuellement grâce à 4 modules SCARF manipulant des clés uniques de 240 bits. Les modules SCARF manipulent également le tag sur 8 bits afin de faire correspondre l'index sur 10 bits. L'éviction des voies et le mécanisme de verrouillage est considéré au niveau de la politique de remplacement VARP-64. Elle consiste en un module de logique *VARP-lock logical* et une mémoire dédiée aux métadonnées *VARP metadata memory*.

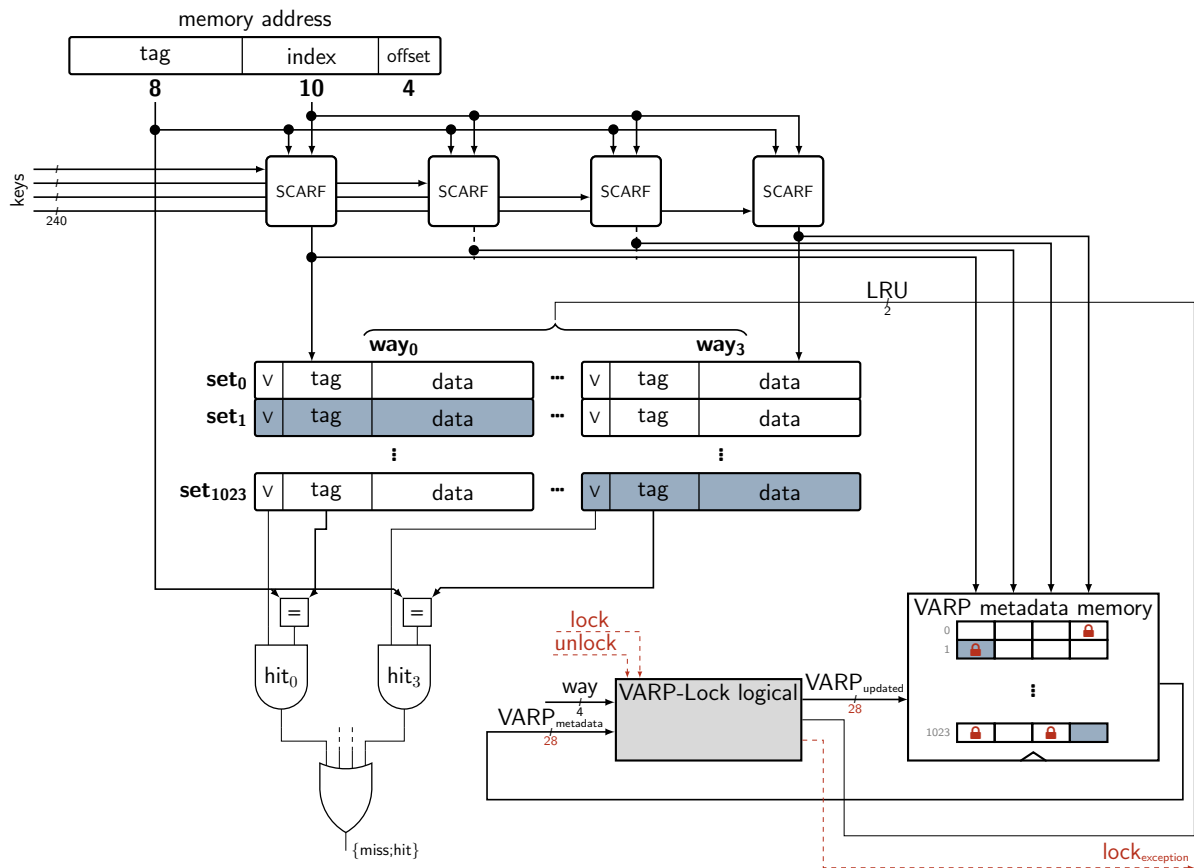


FIGURE 4.4 – Schéma bloc du cache hybride implémentant le mécanisme de verrouillage.

Le module de logique *VARP-lock logical* est représenté dans la Figure 4.5. Il utilise les entrées *lock* et *unlock* provenant directement du processeur permettant de désigner respectivement l'exécution d'une instruction *lock* et *unlock*. Ensuite, les entrées $VARP_{metadata}$ contiennent les états de chacune des voies accédées. Dans cette architecture, les métadonnées sont stockées dans 4 BRAMs dédiées à chacune des voies. Chaque BRAM stocke les métadonnées des 1024 lignes de cache associées. Les métadonnées de chaque ligne de cache sont encodées sur 7 bits

dont les 6 bits de poids faibles indiquent l'âge de la ligne (*i.e.* VARP-64) et 1 bit indique si la ligne de cache est verrouillée. Le module utilise une dernière entrée *Way* provenant du cache qui indique la voie accédée dans le but de mettre à jour les états de la politique de remplacement. Le module fournit en sortie la voie à évincer au travers du signal *LRU*. Ce signal indique la voie la moins récemment utilisée (*i.e.* la plus âgée). Il fournit également les métadonnées mises à jour au travers du signal *VARP_{updated}*. Un signal d'exception *lock_{exception}* est également généré et se comporte d'après la description des exceptions dans la Section 2.2.2.

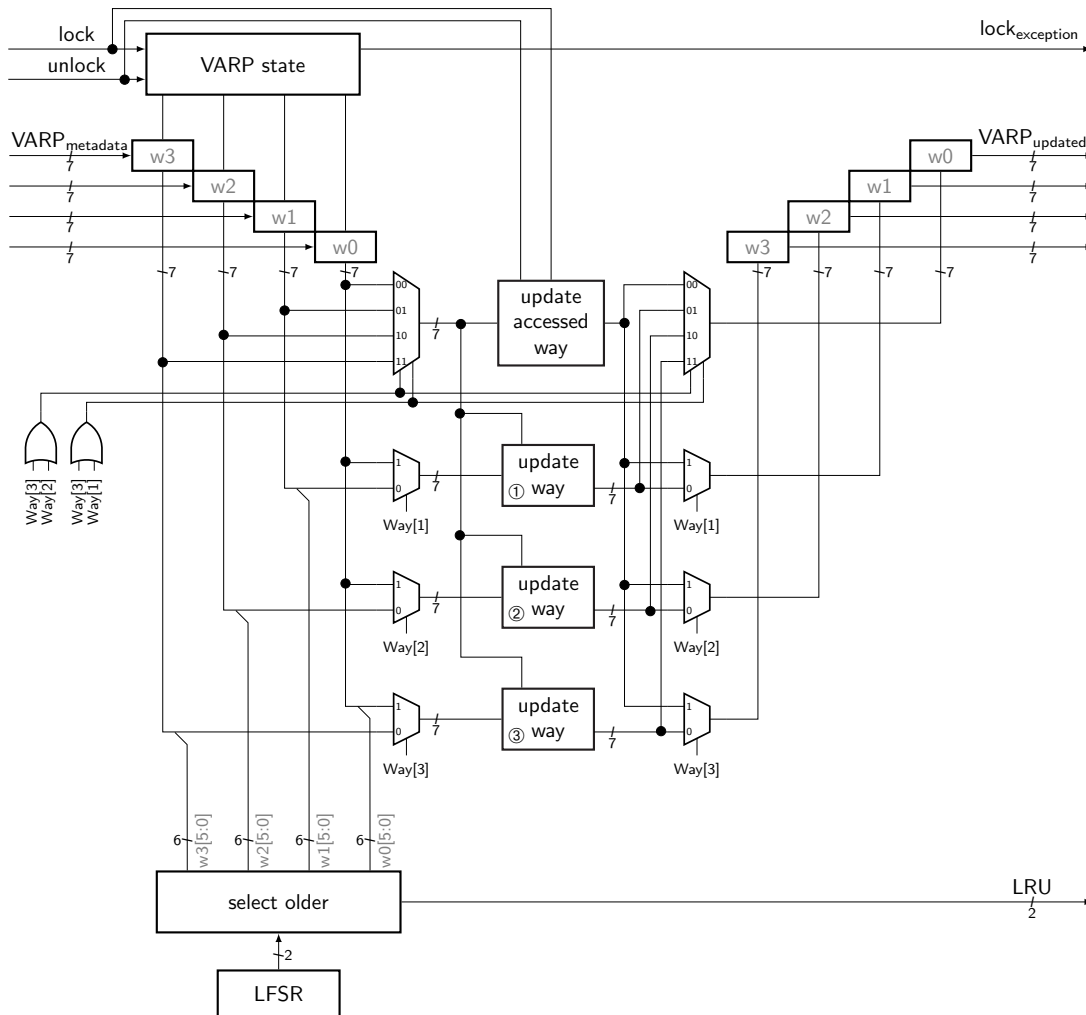


FIGURE 4.5 – **VARP-lock logical** - Description matérielle de la politique de remplacement VARP-64 implémentant le mécanisme de verrouillage.

Le module de politique de remplacement intègre plusieurs modules qui permettent respectivement de lever une exception (*VARP state*), de mettre à jour les états de VARP-64 (*update accessed way* et *update way*), et de sélectionner le candidat à l'éviction (*select older*). Ce dernier nécessite un générateur de nombre pseudo-aléatoire palliant le cas dans lequel plusieurs voies ont

l'âge le plus élevé parmi les candidats. Nous avons fait le choix d'utiliser un Registre à Décalage à Rétroaction Linéaire (LFSR) pour générer ce nombre pseudo-aléatoire sur 2 bits. Au sein du module *select older*, les 2 bits provenant du module *LFSR* permettent de contrôler un arbre de décision binaire en fonction des états les plus âgés.

4.3.3 Description détaillée des modules

Les modules de mises à jour des états de VARP-64 sont détaillés dans la Figure 4.6. Il existe deux modules dédiés à la mise à jour des états : *update accessed way* et *update way*. Le premier, détaillé dans la Figure 4.6a, permet de mettre à jour la voie accédée. Le second, détaillé dans la Figure 4.6b, permet de mettre à jour les voies candidates sélectionnées, mais non accédées. Il est présent en trois exemplaires. L'attribution des modules en fonction de la voie accédée suit l'attribution présentée dans la Section 3.1.2 au travers de la Table 3.1.

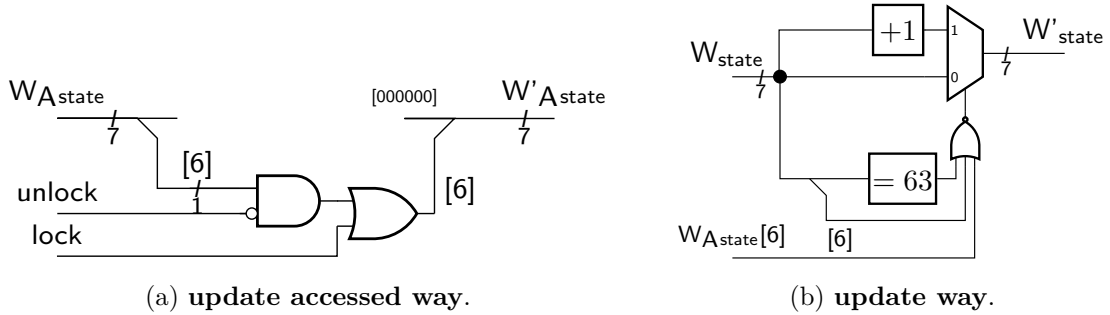


FIGURE 4.6 – Description matérielle des sous-modules utilisés dans VARP-64.

Le module *update accessed way* met à jour l'état des métadonnées de la voie accédée grâce à l'état courant des métadonnées de la voie W_{Astate} et des signaux *lock* et *unlock*. Concernant le nouvel âge de la voie, il devient ou reste 0 quel que soit le cas. Ainsi les 6 bits de poids faibles sont assignés à 0. Ensuite, le bit de poids fort [6] suit une logique lui permettant de faire correspondre l'état de verrouillage de la ligne de cache. Par conséquent, si la voie est déjà verrouillée et que l'accès ne correspond pas à une requête de déverrouillage (*i.e. unlock*) alors le bit vaut 1. C'est également le cas, lorsque l'accès correspond à une requête de verrouillage (*i.e. lock*). Autrement, le bit d'état vaut 0 si l'accès est un déverrouillage ou si la voie n'était pas déjà verrouillée.

Le module *update way* met à jour l'état des métadonnées d'une des voies non accédées et considérées comme candidates à l'éviction. Pour ce faire, le module utilise les métadonnées de la voie attribuée au module W_{state} ainsi que l'état de verrouillage de la voie accédée W_{Astate} [6]. Si la voie accédée ou la voie considérée dans le module est verrouillée ou bien si la voie considérée a atteint l'âge maximal (*i.e. 63*) alors W_{state} reste inchangé. Autrement, l'état W_{state} est incrémenté de 1.

4.4 Impact de l'hybridation sur la sécurité

Dans cette section, nous présentons l'apport en sécurité provenant de l'alliance des deux mécanismes. Chacune des solutions apporte ses avantages et ses inconvénients. Concernant la réindexation asymétrique des voies, la solution a l'avantage d'être transparente pour les applications. En effet, le mécanisme ne nécessite aucun support des applications s'exécutant sur le système. Cependant, les fonctions de réindexation requièrent un renouvellement de leurs clés pour maintenir un niveau suffisant de sécurité. Pour notre configuration de cache (*i.e.* 1024 cache sets, 4 voies, SCARF en IDF, VARP-64 en politique de remplacement), le renouvellement des clés est nécessaire tous les 6.8 millions d'accès mémoires. Cette période provient de l'analyse de Peters *et al.* [132] dans laquelle l'attaque PRIME+PRUNE+PROBE [67] cherche un ensemble d'éviction sur une configuration de cache similaire. Il est important de noter que l'analyse est réalisée avec une plateforme de simulation sans perturbations extérieures. Par ailleurs, de nouvelles techniques d'attaques inconnues de nos jours peuvent amener à une diminution du nombre d'accès à la mémoire pour construire des ensembles d'éviction. Ainsi, le niveau de renouvellement actuel peut devenir obsolète pour des applications critiques.

D'autre part, le mécanisme de verrouillage de lignes de cache a l'avantage de fournir des garanties de sécurité sur les accès aux données verrouillées. En effet, il permet de garantir un accès cache hit sur les lignes de cache verrouillées. Ainsi, un attaquant n'est pas capable de mesurer le temps d'accès aux données sensibles. De plus, l'attaquant n'est pas en mesure d'observer les accès réalisés sur un jeu de données verrouillées. Cependant, la solution fournit des garanties de sécurité uniquement aux applications utilisant explicitement le mécanisme de verrouillage.

En combinant les deux solutions, nous proposons de garder un environnement d'exécution sain et sécurisé au niveau des accès en mémoires caches pour l'ensemble des applications. Par ailleurs, les applications critiques ont la garantie d'accéder aux données en cache lorsque les données accédées sont explicitement et strictement verrouillées en mémoire cache. Cela permet notamment de se prémunir contre les attaques du type EVICT+TIME et PRIME+PROBE. Dans l'éventualité où un attaquant serait capable de construire un ensemble d'éviction plus rapidement qu'avec la technique PRIME+PRUNE+PROBE, les garanties de sécurité des données verrouillées sont maintenues.

4.5 Évaluation

Dans cette section, nous évaluons l'impact de notre solution sur le système au travers des critères de surface et de performances. Ces résultats présentent des résultats expérimentaux dans la mesure où une gestion par le système d'exploitation ou des restrictions matérielles doivent être implémentées pour obtenir une utilisation fiable de notre solution hybride.

4.5.1 Évaluation de la surface matérielle

L'évaluation en surface post-implémentation est menée dans des conditions similaires à l'évaluation réalisée dans la Section 3.2.1. Ainsi, nous considérons une puce FPGA de la famille Kintex-7 de Xilinx implémentée grâce à la suite logicielle Vivado 2022.2. Les éléments de l'architecture sont évalués au travers de trois critères : les LUTs, les Flip-Flops et les BRAMs. La Table 4.1 présente les résultats en surface de notre architecture. Nous sélectionnons les éléments principaux, à savoir le CPU englobant l'ensemble de notre système, le cœur et les mémoires caches dédiées respectivement aux instructions et aux données. Ensuite, nous détaillons le coût en surface de la politique de remplacement VARP-64 pour chacune des mémoires caches.

TABLE 4.1 – Résultats de surface post-implémentation sur FPGA Kintex-7.

	skewed alone			skewed with lock			Overhead	
	LUTs	FFs	BRAMs	LUTs	FFs	BRAMs	LUTs	FFs
→ VARP-64	94	85	2	100	89	2	+ 6,38%	+ 4,71%
→ Data Cache	3 301	1 297	18	3 308	1 302	18	+ 0,21%	+ 0,38%
→ VARP-64	94	85	2	96	89	2	+ 2,13%	+ 4,71%
→ Instr Cache	2 685	1 200	18	2 687	1 204	18	+ 0,07%	+ 0,33%
→ Core	4 655	2 251	0	4 619	2 252	0	- 2,18%	+ 0,04%
CPU	10 657	5 006	36	10 630	5 017	36	- 0,25%	+ 0,18%

Bien que les configurations des caches de données et d'instructions soient identiques, nous observons une différence des ressources associées. Cette différence est induite par le fait que le cache instruction ne supporte pas les instructions `lock` et `unlock`. De plus, la gestion relative aux écritures (*i.e.* requêtes de type `store`) n'ont pas besoin d'être supportées dans un cache instruction.

Chaque mémoire cache utilise 18 BRAMs pour stocker les données et les métadonnées. Concernant la politique de remplacement VARP-64, les métadonnées sont stockées dans 2 BRAMs. La technologie de la famille Kintex-7 permet de considérer une BRAM de 36 Ko en deux BRAMs indépendantes de 18 Ko. Ainsi, les métadonnées de chaque voie sont stockées dans une BRAM de 18 Ko. Au total, 4 BRAMs de 18 Ko sont nécessaires et sont réparties en 2 BRAMs pour chaque module de politique de remplacement. Après avoir soustrait les 2 BRAMs utilisées par les politiques de remplacement, chaque mémoire cache utilise 16 BRAMs pour stocker les métadonnées (*i.e.* tag, bit de validité) et les 16 octets de données. Pour ce faire, 4 BRAMs sont accédées individuellement par chaque voie.

Concernant une implémentation ASIC, il est intéressant d'évaluer les bits stockés en BRAMs. Notre politique de remplacement nécessite le stockage de 6 bits pour VARP-64 auxquels est accolé un bit pour indiquer l'état de verrouillage de chaque ligne de cache considérée. Au total, le stockage des états des 4 096 lignes de cache mène à 28 672 bits dont 4 096 sont nécessaires pour

intégrer le mécanisme de verrouillage (1 bit par ligne de cache). Cela engendre une augmentation de la surface de 16,7 %. Au niveau du cache, chaque ligne de cache stocke 128 bits de données ($B = 16$ octets) et 9 bits de métadonnées (*i.e.* 8 bits de tag, 1 bit de validité), menant à 137 bits par ligne de cache. Au total, 561 152 bits sont stockés dans les 16 BRAMs. En ajoutant les métadonnées de la politique de remplacement, le cache dans sa globalité nécessite le stockage de 589 824 bits, parmi lesquels 4 096 bits utilisés pour le verrouillage de chaque ligne de cache. L'introduction du mécanisme de verrouillage au sein de cette architecture de cache engendre une augmentation de 0,7 % du nombre de bits à stocker.

Actuellement, les clés sont attribuées physiquement aux modules SCARF lors de la génération du matériel. De ce fait, l'outil optimise le circuit en fonction de la clé fournie. Par ailleurs, en intégrant une gestion de clé dynamique (*i.e.* modules verts dans la Figure 4.3), le circuit devra supporter un choix exhaustif de clés. Cela engendrera une hausse des ressources matérielles nécessaires.

À titre de comparaison, nous comparons la solution hybride avec une solution intégrant uniquement la réindexation grâce aux colonnes grisées présentes dans la Table 4.1. Au niveau de la politique de remplacement, nous observons que le mécanisme de verrouillage introduit un coût additionnel de 6,4% pour les LUTs et 4,7% pour les registres. Au niveau du cache, cette augmentation est marginale (*i.e.* <0,4%). Pour une raison que nous ignorons, le cœur profite d'une baisse d'utilisation des LUTs ce qui engendre une baisse de 0,25% des LUTs au niveau du CPU pour une augmentation de 0,18% des registres. En définitive, l'apport du mécanisme de verrouillage engendre un impact négligeable au sein d'un processeur dont les mémoires caches se basent sur une réindexation asymétrique des voies.

4.5.2 Évaluation des performances

Cette section étudie l'impact du mécanisme de verrouillage en mémoire cache de données sur les performances dans un cache à gestion asymétrique des voies. Nous évaluons l'impact sur le taux de cache miss sur les applications du benchmark Embench-IoT [131] lorsqu'une application recourant au mécanisme de verrouillage verrouille N_l lignes de cache. En tenant compte des limitations référencées dans la Section 4.2, nous limitons notre étude à un nombre réduit de lignes de cache verrouillées dans le cache pour une application. En effet, l'application tierce se limite à verrouiller 1024 lignes de cache. Au-delà de 1024 lignes de cache verrouillées, des requêtes ne peuvent pas être validées en raison des exceptions levées.

La Figure 4.7 présente les taux de miss de l'ensemble des applications du benchmark Embench-IoT. Ce taux est évalué lorsque qu'une application tierce verrouille entre 0 et 1024 lignes de cache N_l par pas de 16. Pour un taux de 1024 lignes de cache, le cache est verrouillé à hauteur d'un tiers de ses capacités de verrouillage. Cela correspond au verrouillage de 16 Ko de données. Cette quantité de données verrouillées dépasse ce pourquoi le mécanisme de cache est conçu, à

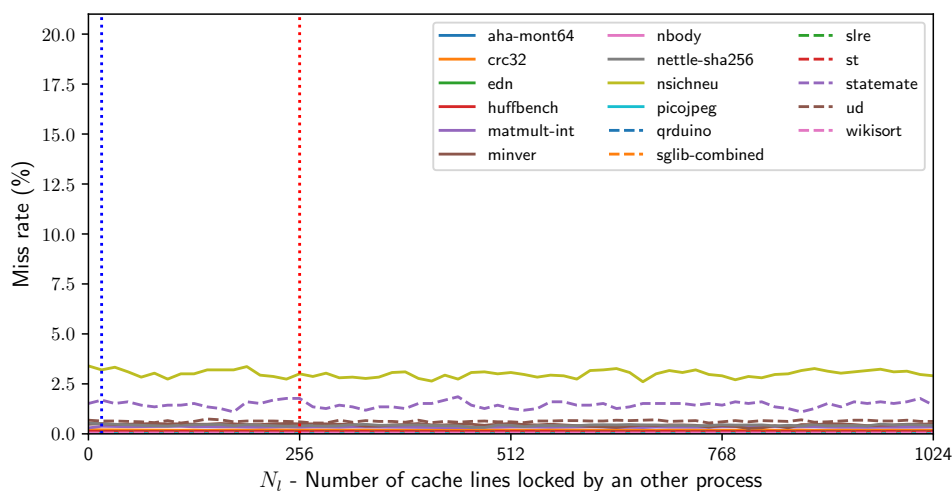


FIGURE 4.7 – Impact sur la suite Embench-IoT 1.0 lorsqu’une application concurrente verrouille N_l lignes de cache. La ligne verticale bleue en pointillés représente le besoin de l’application AES-128. La ligne verticale en pointillés rouge représente le besoin de l’application Camellia.

savoir une petite quantité de données verrouillées pendant une courte durée d’exécution. Afin d’illustrer deux applications pour lesquelles le mécanisme de verrouillage est conçu, deux lignes verticales sont insérées. Ces lignes indiquent le taux de miss induit lorsque AES-128 et Camellia verrouillent leurs tables de substitutions en mémoire cache de données. Elles sont insérées respectivement à 16 et 256 lignes de cache.

La figure révèle que le mécanisme de verrouillage n’influe pas sur le taux de cache miss des applications. Seules des fluctuations liées à la gestion asymétrique des voies sont observées pour les applications `nsichneu`, `statemate` et `ud`. Pour le reste des applications, les fluctuations sont marginales. La faible charge mémoire des applications couplée au nombre limité de lignes de cache verrouillées au sein d’un cache de donnée à grande capacité n’engendre pas un impact significatif sur les performances pour les applications du benchmark Embench-IoT.

4.6 Conclusion

Dans ce chapitre, nous proposons une solution originale en combinant deux mécanismes de sécurité. Grâce à cette solution hybride, nous proposons une mémoire cache sécurisée basée sur l’ajout d’aléa dans laquelle nous proposons d’intégrer notre mécanisme de verrouillage afin d’apporter une solution de sécurité stricte. En effet, cela permet aux applications critiques d’ajouter une couche de sécurité sur une architecture nécessitant une mise à jour régulière des primitives des fonctions d’aléa. Cette couche de sécurité supplémentaire tire profit d’une utilisation explicite des instructions `lock` et `unlock`, engendrant un faible surcoût.

CONCLUSION

Synthèse

Le travail présenté dans ce manuscrit fait suite au constat que la sécurité n'est pas le principal élément motivant la conception des systèmes embarqués. Par ailleurs, la connectivité des objets connectés facilite la mise en place d'attaques logicielles tirant profit du comportement fonctionnel du système. Par exemple, les attaques en mémoires caches que nous considérons examinent les canaux auxiliaires temporels provenant des variations du temps d'exécution. Ces variations sont induites par le partage des ressources matérielles des mémoires caches. De ce fait, l'exécution d'une application est amenée à modifier l'état du cache, impactant l'exécution des autres applications. De ce constat, différentes attaques exploitent ces vulnérabilités. Nous considérons principalement deux types d'attaques. D'une part, celles analysant le temps d'exécution de la victime après que l'attaquant ait manipulé l'état du cache (*i.e.* EVICT+TIME). D'autre part, celles étudiant l'état du cache pour inférer les accès mémoire que la victime réalise (*i.e.* PRIME+PROBE).

Il existe plusieurs solutions permettant de se prémunir de ces attaques. Parmi ces contre-mesures, nous concentrons nos études sur celles impliquant une modification du matériel. Le premier type de contre-mesure se base sur l'ajout d'aléa entre l'adresse de la requête et le cache set accédé. Cette catégorie propose un ensemble varié de contre-mesures avec des solutions de réindexation complexes dont une gestion asymétrique des voies. Les solutions sont intéressantes, car elles ne nécessitent aucun support de la part des applications. Cependant, elles ont besoin d'actualiser régulièrement leurs réindexations pour maintenir leur niveau de sécurité, ce qui impacte les performances. Le second type de contre-mesure se base sur le partitionnement de ressources en cache dans le but de séparer physiquement les ressources allouées aux applications. D'ordinaire, la réservation est réalisée pour une application ou un groupe d'applications critiques. En fonction de la granularité, les performances sont plus ou moins impactées. Une solution à grain fin permet d'ajuster les ressources réservées aux plus proches des besoins de l'application, ce qui minimise les pertes de performances pour l'application critique, mais également pour les autres applications. Ce type de solution est intéressant pour proposer des garanties de sécurité avec un faible coût matériel.

En collaboration avec les membres du projet SCRATCHS, nous proposons une solution permettant de verrouiller explicitement des lignes de cache au travers d'une extension du jeu d'instructions. Notre solution permet de protéger des sections critiques de programme en garantissant

un accès en temps constant aux données verrouillées. Les membres du projet SCRATCHS ont proposé des preuves formelles de la solution [134]. Suite à notre implémentation matérielle, nous avons vérifié la sécurité en considérant l’algorithme AES-128. En verrouillant la table de substitution, un attaquant n’est pas capable d’observer les accès réalisés sur la table lors du chiffrement. Cette implémentation engendre un faible surcoût au niveau de la surface avec une augmentation des ressources matérielles utilisées de moins de 3% pour une implémentation FPGA sur la famille Xilinx Kintex-7. Au regard des performances, l’introduction des instructions `lock` et `unlock` augmentent la taille du programme binaire de 0,28% et 0,23% respectivement pour les algorithmes AES-128 et Camellia. L’exécution de ses instructions engendre un surcoût de 3% sur le temps d’exécution pour un chiffrement de 128 bits avec AES-128 et devient négligeable dès 4 chiffrements consécutifs. Concernant Camellia, le temps d’exécution est multiplié par deux pour 4 chiffrements consécutifs de 128 bits, cependant ce coût se résorbe en augmentant le nombre de blocs chiffrés. Ce coût pour Camellia est lié à sa faible empreinte mémoire sur de grandes tables de substitution qui sont préalablement verrouillées. Dès 512 chiffrements, le coût de verrouillage devient inférieur à 1%. De plus, nous avons relevé un faible impact sur les performances du benchmark Embench-IoT lorsque AES-128 et Camellia verrouillent leur(s) table(s) de substitution. AES-128 a un impact négligeable et Camellia impacte les autres applications à hauteur de 1,1%.

Dans un second temps, nous avons proposé une solution originale en intégrant notre mécanisme de verrouillage au sein d’une architecture à gestion asymétrique des voies. Cependant, cette solution hybride implique des limitations au niveau de son utilisation. Ces dernières doivent être prises en compte pour une implémentation sortant d’un cadre expérimental. Une limitation matérielle du mécanisme de verrouillage peut résoudre ces problèmes. Du point de vue de la sécurité, la solution hybride propose une approche élégante avec une sécurité partagée parmi les utilisateurs du système à laquelle nous apportons une couche supplémentaire de sécurité pour les applications critiques. Par ce biais, nous prévenons tout attaquant capable de construire un ensemble d’éviction ciblant les données des applications critiques. L’implémentation engendre un faible impact sur les ressources matérielles ainsi qu’un impact négligeable sur les applications. À travers cette solution hybride expérimentale, nous démontrons que notre mécanisme de verrouillage peut être considéré au sein d’une architecture afin de consolider les besoins de sécurité grâce à un verrouillage strict des données en mémoires caches.

Perspectives

Ce manuscrit expose un nouveau mécanisme de verrouillage strict utilisant peu de ressources matérielles, et engendrant des baisses de performances minimales pour des applications recourant à notre mécanisme ou non. Les résultats s’intègrent dans une démarche expérimentale dans laquelle

nous maîtrisons de bout en bout l'exécution des applications au travers de simulations. De plus, nous nous sommes limités à des systèmes embarqués avec des besoins en performances limités et dont les applications nécessitent peu de ressources de calculs. C'est pourquoi, dans nos travaux futurs, nous souhaitons repousser les limitations que nous avons posées afin d'expérimenter notre solution dans des cas d'usages réels ou sur des systèmes à performances plus élevées.

Étendre la hiérarchie mémoire

Jusqu'à présent, nos solutions se basent sur une hiérarchie mémoire d'un niveau de mémoires caches en considérant un cache de données puis en étendant le premier niveau d'un cache instruction pour la solution hybride. Il serait intéressant de considérer un second de niveau de cache. Dans un premier temps, la question de l'inclusivité liée au verrouillage de données devra être traitée. Nous pouvons imaginer un verrouillage des données uniquement au niveau L2 afin de ne pas considérer le premier niveau de cache lors des accès sur des données verrouillées. Cela garantirait toujours un accès en temps constant à la donnée. De plus, les mémoires caches présentes dans ce niveau de hiérarchie ont une plus grande capacité de stockage. Ainsi, notre mécanisme de verrouillage bénéficierait d'un impact d'autant plus faible sur les performances.

En tenant compte de cette implémentation, nous imaginons également une hiérarchie dont les niveaux de caches profitent d'architectures différentes pour exploiter leurs avantages et en minimisant leurs inconvénients. Par exemple, notre solution de verrouillage propose un impact marginal sur les performances avec un surcoût faible au niveau des ressources matérielles lorsqu'il est considéré dans un cache de niveau L2. De plus, une approche basée sur de la réindexation est intéressante pour le niveau L1 du fait que ce type d'architecture ne convient pas au cache de très grande capacité et est limité par l'utilisation des IDFs.

Support d'un système d'exploitation

L'apport d'un prototype fonctionnel sur une carte FPGA avec un système d'exploitation embarqué permettra d'apporter une gestion des exceptions que le cache peut lever. En effet, plusieurs cas, en fonction des configurations de caches, peuvent engendrer une exception (*e.g.* une requête lock sur un cache set déjà plein, ou une requête se voyant pourvoir quatre lignes de cache verrouillées en considérant notre seconde contribution). Dans nos solutions, nous n'avons pas défini les routines d'exceptions. Dans la mesure où, dans notre première contribution, l'application connaît la configuration du cache, seul un usage du mécanisme de verrouillage par deux applications peut engendrer une exception. Sinon, il s'agit d'une application malveillante souhaitant exploiter un déni de service. Dans ce cas, le système d'exploitation peut gérer à la volée quelle application est autorisée à utiliser le mécanisme de verrouillage. Ce droit peut être réservé à des applications de confiance.

Impact de la microarchitecture sur le verrouillage

Le cœur de processeur que nous considérons est simple. Par ailleurs, nous posons une question ouverte sur l'impact des éléments microarchitecturaux sur l'utilisation du mécanisme de verrouillage. Notamment, lorsque le processeur considère une exécution dans le désordre ou avec des prédictions de branchements.

Verrouillage au sein d'un système multicœur

Dans ce manuscrit et au sein du projet SCRATCHS, nous avons pensé les instructions `lock` et `unlock` pour des processeurs monocœurs. Ainsi, la question du multicœur est posée. Premièrement, une mémoire cache de dernier niveau (LLC) est partagée avec tous les cœurs. Ainsi, il faut propager la gestion du verrouillage vers le LLC en provenance des caches privés d'un cœur. Il faut également considérer la gestion des ressources partagées pour la cohérence des données verrouillées entre les cœurs. Nous proposons de verrouiller les données au niveau du LLC afin que la donnée soit verrouillée en LLC et dans les caches privés du cœur ayant soumis le verrouillage.

Autres usages

Pour l'heure, seules des applications cryptographiques ont été considérées pour notre mécanisme de verrouillage. En effet, nous avons considéré les algorithmes de chiffrement AES et Camellia parce que ces algorithmes réalisent des accès mémoires dont les adresses dépendent d'un secret.

Cependant, les systèmes temps-réel peuvent tirer bénéfice de notre mécanisme. Ce type de système a besoin d'estimer le temps d'exécution au pire cas de chaque programme afin de planifier l'exécution de l'ensemble des programmes. Grâce au verrouillage de données en mémoires caches, nous pouvons leur garantir des accès en caches leur permettant de diminuer l'estimation du temps d'exécution au pire cas. Nous espérons que d'autres types d'applications peuvent considérer notre mécanisme de verrouillage.

LISTES DES PUBLICATIONS ET DES COMMUNICATIONS

Publications en conférences internationales avec actes et comités de relecture

- ▶ Nicolas GAUDIN, Jean-Loup HATCHIKIAN-HOUDOT, Frédéric BESSON, Pascal COTRET, Guy GOGNIAT, Guillaume HIET, Vianney LAPOTRE et Pierre WILKE, « Work in Progress : Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections », in : *Proc. IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, juill. 2023, DOI : 10.1109/EuroSPW59978.2023.00038
- ▶ Nicolas GAUDIN, Pascal COTRET, Guy GOGNIAT et Vianney LAPÔTRE, « A Fine-Grained Dynamic Partitioning Against Cache-Based Timing Attacks via Cache Locking », in : *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024, DOI : 10.1109/ISVLSI61997.2024.00041
- ▶ Moritz PETERS, Nicolas GAUDIN, Jan Philipp THOMA, Vianney LAPÔTRE, Pascal COTRET, Guy GOGNIAT et Tim GÜNEYSU, « On The Effect of Replacement Policies on The Security of Randomized Cache Architectures », in : *Proc. ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2024, p. 483-497, DOI : 10.1145/3634737.3637677

Communication en conférence nationale sans acte

- ▶ Nicolas GAUDIN, Pascal COTRET, Guy GOGNIAT et Vianney LAPÔTRE, « Verrouillage des lignes de cache pour la lutte contre les attaques par canaux auxiliaires exploitant les mémoires caches », in : *Cyber On Board*, 2024, URL : <https://hal.science/hal-04461273/>

Communications

- ▶ Nicolas GAUDIN, Pascal COTRET, Guy GOGNIAT et Vianney LAPÔTRE, « Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections », in : *CYBERUS summer school*, juill. 2023, URL : <https://hal.science/hal-04424956>
- ▶ Nicolas GAUDIN, Vianney LAPÔTRE, Pascal COTRET et Gogniat GUY, « Cache locking against cache-based side-channel attacks », in : *École d'hiver Francophone sur les Technologies de Conception des Systèmes Embarqués Hétérogènes (FETCH)*, fév. 2024, URL : <https://hal.science/hal-04446221>

Communications avec posters

- ▶ Journées CominLabs - *octobre 2022, septembre 2023, Rennes*
- ▶ International Winter School of Microarchitectural Security - *décembre 2022, Paris*
- ▶ Cyberus Summer School - *avril 2024, Lorient*

BIBLIOGRAPHIE

- [1] Karl RUPP, *50 Years of Microprocessor Trend Data*, Karl Rupp, URL : <https://github.com/karlrupp/microprocessor-trend-data>.
- [2] APPLE, *Apple Unveils M2 with Breakthrough Performance and Capabilities*, Apple, URL : <https://www.apple.com/fr/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities>.
- [3] AMD, *Fiche Produit de l'accélérateur AMD Instinct™ MI300A embarquant CPU, GPU et accélérateurs matériels*. AMD, URL : <https://www.amd.com/en/products/accelerators/instinct/mi300/mi300a.html>.
- [4] OpenHW GROUP, *Core-V CV32E40P User Manual*, OpenHW Group, URL : https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/cv32e40p_v1.8.3/intro.html.
- [5] J. F. ZIEGLER et W. A. LANFORD, « Effect of Cosmic Rays on Computer Memories », in : *Science* 206.4420 (1979), p. 776-788, DOI : 10.1126/science.206.4420.776.
- [6] Stefan K. HÖEFFGEN, Stefan METZGER et Michael STEFFENS, « Investigating the Effects of Cosmic Rays on Space Electronics », in : *Frontiers in Physics* 8 (2020), DOI : 10.3389/fphy.2020.00318.
- [7] H. BAR-EL, H. CHOUKRI, D. NACCACHE, M. TUNSTALL et C. WHELAN, « The Sorcerer's Apprentice Guide to Fault Attacks », in : *Proceedings of the IEEE* 94.2 (2006), p. 370-382, DOI : 10.1109/JPROC.2005.862424.
- [8] Jakub BREIER et Xiaolu HOU, « How Practical Are Fault Injection Attacks, Really ? », in : *IEEE Access* 10 (2022), DOI : 10.1109/ACCESS.2022.3217212.
- [9] Jean-Max DUTERTRE, Vincent BEROLLE, Philippe CANDELIER, Stephan DE CASTRO, Louis-Barthelemy FABER, Marie-Lise FLOTTES, Philippe GENDRIER, David HÉLY, Regis LEVEUGLE, Paolo MAISTRI, Giorgio DI NATALE, Athanasios PAPADIMITRIOU et Bruno ROUZEYRE, « Laser Fault Injection at the CMOS 28 nm Technology Node : an Analysis of the Fault Model », in : *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2018, DOI : 10.1109/FDTC.2018.00009.
- [10] Joaquín RODRIGUEZ, Alex BALDOMERO, Victor MONTILLA et Jordi MUJAL, « LLFI : Lateral Laser Fault Injection Attack », in : *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2019, DOI : 10.1109/FDTC.2019.00014.

-
- [11] Sara FAOUR, Mališa VUČINIĆ, Filip MAKSIMOVIC, David BURNETT, Paul MUHLETHALER, Thomas WATTEYNE et Kristofer PISTER, « Implications of Physical Fault Injections on Single Chip Motes », in : *Proc. IEEE World Forum on Internet of Things (WF-IoT)*, 2023, DOI : 10.1109/WF-IoT58464.2023.10539380.
- [12] Clemens HELFMEIER, Christian BOIT et Uwe KERST, « On charge sensors for FIB attack detection », in : *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2012, DOI : 10.1109/HST.2012.6224332.
- [13] Sergei P SKOROBOGATOV et Ross J ANDERSON, « Optical fault induction attacks », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, 2003, p. 2-12.
- [14] Stéphanie ANCEAU, Pierre BLEUET, Jessy CLÉDIÈRE, Laurent MAINGAULT, Jean-luc RAINARD et Rémi TUCOULOU, « Nanofocused X-ray beam to reprogram secure circuits », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, 2017, p. 175-188.
- [15] Thomas TROUCHKINE, Sébanjila Kevin BUKASA, Mathieu ESCOUTELOUP, Ronan LASHERMES et Guillaume BOUFFARD, « Electromagnetic fault injection against a complex CPU, toward new micro-architectural fault models », in : *Journal of Cryptographic Engineering (JCEN)* 11.4 (2021), p. 353-367, DOI : 10.1007/s13389-021-00259-6.
- [16] Mahmoud A. ELMOHR, Haohao LIAO et Catherine H. GEBOTYS, « EM Fault Injection on ARM and RISC-V », in : *Proc. International Symposium on Quality Electronic Design (ISQED)*, 2020, DOI : 10.1109/ISQED48828.2020.9137051.
- [17] Josep BALASCH, Benedikt GIERLICHs et Ingrid VERBAUWHEDE, « An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs », in : *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2011, DOI : 10.1109/FDTC.2011.9.
- [18] Claudio BOZZATO, Riccardo FOCARDI et Francesco PALMARINI, « Shaping the glitch : optimizing voltage fault injection attacks », in : *Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 2019.2 (), p. 199-224, DOI : 10.13154/tches.v2019.i2.199-224.
- [19] 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)*, 2014, DOI : 10.1007/978-3-319-08302-5_15.
- [20] Thomas KORAK, Michael HUTTER, Baris EGE et Lejla BATINA, « Clock Glitch Attacks in the Presence of Heating », in : *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2014, DOI : 10.1109/FDTC.2014.20.

-
- [21] Yoongu KIM, Ross DALY, Jeremie KIM, Chris FALLIN, Ji Hye LEE, Donghyuk LEE, Chris WILKERSON, Konrad LAI et Onur MUTLU, « Flipping bits in memory without accessing them : An experimental study of DRAM disturbance errors », in : *ACM SIGARCH Computer Architecture News* 42.3 (2014), p. 361-372, DOI : 10.1145/2678373.2665726.
- [22] Onur MUTLU et Jeremie S. KIM, « RowHammer : A Retrospective », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2020), p. 1555-1571, DOI : 10.1109/TCAD.2019.2915318.
- [23] Jan JANCAR, Vojtech SUCHANEK, Petr SVENDA, Vladimir SEDLACEK et Łukasz CHMIELEWSKI, « pyecsc : Reverse engineering black-box elliptic curve cryptography via side-channel analysis », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, 2024.
- [24] Hervé CHABANNE, Jean-Luc DANGER, Linda GUIGA et Ulrich KÜHNE, « Side channel attacks for architecture extraction of neural networks », in : *CAAI Transactions on Intelligence Technology* 6.1 (2021), p. 3-16, DOI : 10.1049/cit2.12026.
- [25] Abdullah ALJUFFRI, Marc ZWALUA, Cezar Rodolfo Wedig REINBRECHT, Said HAMDIOUI et Mottaqiallah TAOUIL, « Applying Thermal Side-Channel Attacks on Asymmetric Cryptography », in : *IEEE Transactions on Very Large Scale Integration Systems* 29.11 (2021), p. 1930-1942, DOI : 10.1109/TVLSI.2021.3111407.
- [26] Stefan MANGARD, Elisabeth OSWALD et Thomas POPP, *Power analysis attacks : Revealing the secrets of smart cards*, t. 31, Springer Science & Business Media, 2008, DOI : 10.1007/978-0-387-38162-6.
- [27] Stefan MANGARD, « A simple power-analysis (SPA) attack on implementations of the AES key expansion », in : *Information Security and Cryptology (ICISC)*, 2003, DOI : 10.1007/3-540-36552-4_24.
- [28] Christophe CLAVIER, Damien MARION et Antoine WURCKER, « Simple power analysis on AES key expansion revisited », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, 2014, p. 279-297.
- [29] P KOCHER, *Introduction to Differential Power Analysis and Related Attacks*, rapp. tech., Technical Report, 1998.
- [30] Stefan MANGARD, Norbert PRAMSTALLER et Elisabeth OSWALD, « Successfully Attacking Masked AES Hardware Implementations », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2005, DOI : 10.1007/11545262_12.

-
- [31] Moritz LIPP, Andreas KOGLER, David OSWALD, Michael SCHWARZ, Catherine EASDON, Claudio CANELLA et Daniel GRUSS, « PLATYPUS : Software-based Power Side-Channel Attacks on x86 », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, 2021, DOI : 10.1109/SP40001.2021.00063.
- [32] Mark ZHAO et G. Edward SUH, « FPGA-Based Remote Power Side-Channel Attacks », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, 2018, DOI : 10.1109/SP.2018.00049.
- [33] Timo KASPER, David OSWALD et Christof PAAR, « EM side-channel attacks on commercial contactless smartcards using low-cost equipment », in : *International Workshop on Information Security Applications*, 2009, DOI : 10.1007/978-3-642-10838-9_7.
- [34] Gregor HAAS et Aydin AYSU, « Apple vs. EMA : electromagnetic side channel attacks on apple CoreCrypto », in : *Proc. Design Automation Conference (DAC)*, 2022, DOI : 10.1145/3489517.3530437.
- [35] Honggang YU, Haocheng MA, Kaichen YANG, Yiqiang ZHAO et Yier JIN, « DeepEM : Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage », in : *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2020, DOI : 10.1109/HOST45689.2020.9300274.
- [36] Daniel GENKIN, Adi SHAMIR et Eran TROMER, « RSA key extraction via low-bandwidth acoustic cryptanalysis », in : *Advances in Cryptology - CRYPTO*, 2014, DOI : 10.1007/978-3-662-44371-2_25.
- [37] Alireza TAHERITAJAR et hrReza RAHAEIME, *Acoustic Side Channel Attack on Keyboards Based on Typing Patterns*, 2024, DOI : 10.48550/arXiv.2403.08740.
- [38] Jiliang ZHANG, Congcong CHEN, Jinhua CUI et Keqin LI, « Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures », in : *ACM Computing Surveys (CSUR)* (avr. 2024), DOI : 10.1145/3645109.
- [39] David BRUMLEY et Dan BONEH, « Remote timing attacks are practical », in : *Computer Networks* 48.5 (2005), p. 701-716, DOI : 10.1016/j.comnet.2005.01.010.
- [40] Jean-Francois DHEM, Francois KOEUNE, Philippe-Alexandre LEROUX, Patrick MESTRÉ, Jean-Jacques QUISQUATER et Jean-Louis WILLEMS, « A practical implementation of the timing attack », in : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*, 1998, DOI : 10.1007/10721064_15.
- [41] Werner SCHINDLER, « A timing attack against RSA with the chinese remainder theorem », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2000, p. 109-124, DOI : 10.1007/3-540-44499-8_8.

-
- [42] Dmitry EVTYUSHKIN, Ryan RILEY, Nael ABU-GHAZALEH et Dmitry PONOMAREV, « Branch-Scope : A New Side-Channel Attack on Directional Branch Predictor », in : *Proc. Architectural Support for Programming Languages and Operating Systems (APLOS)*, t. 53, 2, 2018, DOI : 10.1145/3296957.3173204.
- [43] Paul KOCHER, Jann HORN, Anders FOGH, Daniel GENKIN, Daniel GRUSS, Werner HAAS, Mike HAMBURG, Moritz LIPP, Stefan MANGARD, Thomas PRESCHER, Michael SCHWARZ et Yuval YAROM, « Spectre Attacks : Exploiting Speculative Execution », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, 2019, DOI : 10.1109/SP.2019.00002.
- [44] Fangfei LIU, Yuval YAROM, Qian GE, Gernot HEISER et Ruby B. LEE, « Last-Level Cache Side-Channel Attacks are Practical », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, 2015, DOI : 10.1109/SP.2015.43.
- [45] Yuval YAROM et Katrina FALKNER, « FLUSH+RELOAD : A High Resolution, Low Noise, L3 Cache Side-Channel Attack », in : *Proc. 23th USENIX Security Symposium (USENIX Security)*, 2014, URL : <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.
- [46] Moritz LIPP, Vedad HADŽIĆ, Michael SCHWARZ, Arthur PERAIS, Clémentine MAURICE et Daniel GRUSS, « Take A Way : Exploring the Security Implications of AMD's Cache Way Predictors », in : *Proc. ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2020, p. 813-825, DOI : 10.1145/3320269.3384746.
- [47] Daniel GRUSS, Clémentine MAURICE, Anders FOGH, Moritz LIPP et Stefan MANGARD, « Prefetch Side-Channel Attacks : Bypassing SMAP and Kernel ASLR », in : *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, DOI : 10.1145/2976749.2978356.
- [48] Daimeng WANG, Zhiyun QIAN, Nael ABU-GHAZALEH et Srikanth V. KRISHNAMURTHY, « PAPP : Prefetcher-Aware Prime and Probe Side-channel Attack », in : *Proc. Design Automation Conference (DAC)*, 2019, DOI : 10.1145/3316781.3317877.
- [49] Jiliang ZHANG, Chaoqun SHEN et Gang QU, « Mex+Sync : Software Covert Channels Exploiting Mutual Exclusion and Synchronization », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.12 (2023), p. 4491-4504, DOI : 10.1109/TCAD.2023.3291669.
- [50] Shivam BHASIN et Francesco REGAZZONI, « A survey on hardware trojan detection techniques », in : *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015, DOI : 10.1109/ISCAS.2015.7169073.

-
- [51] Zhe CHEN, Shize GUO, Jian WANG, Yubai LI et Zhonghai LU, « Toward FPGA Security in IoT : A New Detection Technique for Hardware Trojans », in : *IEEE Internet of Things Journal* 6.4 (2019), p. 7061-7068, DOI : 10.1109/JIOT.2019.2914079.
- [52] Taras IAKYMCHUK, Maciej NIKODEM et Krzysztof KĘPA, « Temperature-based covert channel in FPGA systems », in : *Proc. International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, 2011, DOI : 10.1109/ReCoSoC.2011.5981510.
- [53] Shanquan TIAN et Jakub SZEFER, « Temporal Thermal Covert Channels in Cloud FPGAs », in : *Proc. International Symposium on Field-Programmable Gate Arrays*, 2019, DOI : 10.1145/3289602.3293920.
- [54] Ilias GIECHASKIEL, Kasper Bonne RASMUSSEN et Jakub SZEFER, « C3APSULe : Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, 2020, DOI : 10.1109/SP40000.2020.00070.
- [55] Baki Berkay YILMAZ, Robert L. CALLAN, Milos PRVULOVIC et Alenka ZAJIĆ, « Capacity of the EM Covert/Side-Channel Created by the Execution of Instructions in a Processor », in : *IEEE Transactions on Information Forensics and Security* 13.3 (2018), p. 605-620, DOI : 10.1109/TIFS.2017.2762826.
- [56] Zihao ZHAN, Zhenkai ZHANG et Xenofon KOUTSOUKOS, « BitJabber : The World's Fastest Electromagnetic Covert Channel », in : *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2020, DOI : 10.1109/HOST45689.2020.9300268.
- [57] Colin PERCIVAL, « Cache missing for fun and profit », in : 2005.
- [58] Jan Philipp THOMA et Tim GÜNEYSU, « Write Me and I'll Tell You Secrets – Write-After-Write Effects On Intel CPUs », in : *Proc. International Symposium on Research in Attacks, Intrusions and Defenses(RAID)*, 2022, DOI : 10.1145/3545948.3545987.
- [59] Yanan GUO, Dingyuan CAO, Xin XIN, Youtao ZHANG et Jun YANG, « Uncore Encore : Covert Channels Exploiting Uncore Frequency Scaling », in : *Proc. International Symposium on Microarchitecture (MICRO)*, 2023, DOI : 10.1145/3613424.3614259.
- [60] Maria MUSHTAQ, Muhammad Asim MUKHTAR, Vianney LAPOTRE, Muhammad Khurram BHATTI et Guy GOGNIAT, « Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA », in : *Information Systems* 92 (2020), p. 101524, ISSN : 0306-4379, DOI : <https://doi.org/10.1016/j.is.2020.101524>.
- [61] Thomas ALLAN, Billy Bob BRUMLEY, Katrina FALKNER, Joop van de POL et Yuval YAROM, « Amplifying side channels through performance degradation », in : *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2016, DOI : 10.1145/2991079.2991084.

-
- [62] Yuval YAROM, Daniel GENKIN et Nadia HENINGER, « CacheBleed : a timing attack on OpenSSL constant-time RSA », in : *Journal of Cryptographic Engineering (JCEN)* 7 (2017), p. 99-112, DOI : 10.1007/s13389-017-0152-y.
- [63] Farabi MAHMUD, Sungkeun KIM, Harpreet Singh CHAWLA, Eun Jung KIM, Chia-Che TSAI et Abdullah MUZAHID, « Attack of the Knights : Non Uniform Cache Side Channel Attack », in : *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2023, DOI : 10.1145/3627106.3627199.
- [64] Dag Arne OSVIK, Adi SHAMIR et Eran TROMER, « Cache attacks and countermeasures : the case of AES », in : *Proc. Topics in Cryptology-CT-RSA 2006*, 2006, DOI : 10.1007/11605805_1.
- [65] Eran TROMER, Dag Arne OSVIK et Adi SHAMIR, « Efficient Cache Attacks on AES, and Countermeasures », in : *Journal of Cryptology* 23 (2010), p. 37-71, DOI : 10.1007/s00145-009-9049-y.
- [66] Wenjie XIONG, Stefan KATZENBEISSER et Jakub SZEFER, « Leaking Information Through Cache LRU States in Commercial Processors and Secure Caches », in : *IEEE Transactions on Computers* 70.4 (2021), p. 511-523, DOI : 10.1109/TC.2021.3059531.
- [67] Antoon PURNAL, Lukas GINER, Daniel GRUSS et Ingrid VERBAUWHEDE, « Systematic Analysis of Randomization-based Protected Cache Architectures », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, mai 2021, DOI : 10.1109/SP40001.2021.00011.
- [68] Hans WINDERIX, Jan Tobias MÜHLBERG et Frank PIESENS, « Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks », in : *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, sept. 2021, DOI : 10.1109/EuroSP51992.2021.00050.
- [69] Meng WU, Shengjian GUO, Patrick SCHAUMONT et Chao WANG, « Eliminating timing side-channel leaks using program repair », in : *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, DOI : 10.1145/3213846.3213851.
- [70] Goran DOYCHEV, Boris KÖPF, Laurent MAUBORGNE et Jan REINEKE, « CacheAudit : A Tool for the Static Analysis of Cache Side Channels », in : *ACM Transaction on Information and System Security* 18.1 (2015), DOI : 10.1145/2756550.
- [71] Ziqiao ZHOU, Michael K. REITER et Yinqian ZHANG, « A Software Approach to Defeating Side Channels in Last-Level Caches », in : *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, DOI : 10.1145/2976749.2978324.

-
- [72] Maria MUSHTAQ, Ayaz AKRAM, Muhammad Khurram BHATTI, Maham CHAUDHRY, Vianney LAPOTRE et Guy GOGNIAT, « NIGHTs-WATCH : a cache-based side-channel intrusion detector using hardware performance counters », in : *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2018, DOI : 10.1145/3214292.3214293.
- [73] Maria MUSHTAQ, Jeremy BRICQ, Muhammad Khurram BHATTI, Ayaz AKRAM, Vianney LAPOTRE, Guy GOGNIAT et Pascal BENOIT, « WHISPER : A Tool for Run-Time Detection of Side-Channel Attacks », in : *IEEE Access* (2020), DOI : 10.1109/ACCESS.2020.2988370.
- [74] Samira BRIONGOS, Gorka IRAZOQUI, Pedro MALAGÓN et Thomas EISENBARTH, « Cache-Shield : Detecting Cache Attacks through Self-Observation », in : *Proc. ACM Conference on Data and Application Security and Privacy*, 2018, DOI : 10.1145/3176258.3176320.
- [75] Brian C. SCHWEDOCK, Piratach YOOVIDHYA, Jennifer SEIBERT et Nathan BECKMANN, « tākō : a polymorphic cache hierarchy for general-purpose optimization of data movement », in : *Proc. International Symposium on Computer Architecture (ISCA)*, 2022, DOI : 10.1145/3470496.3527379.
- [76] Zhenghong WANG et Ruby B. LEE, « New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks », in : *Proc. International Symposium on Computer Architecture (ISCA)*, 2007, DOI : 10.1145/1250662.1250723.
- [77] Moinuddin K. QURESHI, « CEASER : Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping », in : *Proc. International Symposium on Microarchitecture (MICRO)*, 2018, DOI : 10.1109/MICRO.2018.00068.
- [78] Qinhan TAN, Zihua ZENG, Kai BU et Kui REN, « PhantomCache : Obfuscating Cache Conflicts with Localized Randomization. », in : *Proc. Network and Distributed System-Security Symposium (NDSS)*, 2020, DOI : 10.14722/ndss.2020.24086.
- [79] Amine JAAMOUM, Thomas HISCOCK et Giorgio Di NATALE, « Scramble Cache : An Efficient Cache Architecture for Randomized Set Permutation », in : *Proc. Design, Automation & Test in Europe Conference (DATE)*, 2021, DOI : 10.23919/DATE51398.2021.9473919.
- [80] Wei SONG, Zihan XUE, Jinchi HAN, Zhenzhen LI et Peng LIU, « Randomizing Set-Associative Caches Against Conflict-Based Cache Side-Channel Attacks », in : *IEEE Transactions on Computers* 73.4 (2024), p. 1019-1033, DOI : 10.1109/TC.2024.3349659.
- [81] Moinuddin K. QURESHI, « New Attacks and Defense for Encrypted-Address Cache », in : *Proc. International Symposium on Computer Architecture (ISCA)*, 2019, DOI : 10.1145/3307650.3322246.

-
- [82] Mario WERNER, Thomas UNTERLUGGAUER, Lukas GINER, Michael SCHWARZ, Daniel GRUSS et Stefan MANGARD, « ScatterCache : Thwarting Cache Attacks via Cache Set Randomization », in : *Proc. 28th USENIX Security Symposium (USENIX Security)*, 2019, URL : <https://www.usenix.org/conference/usenixsecurity19/presentation/werner>.
- [83] Jan Philipp THOMA, Christian NIESLER, Dominic A FUNKE, Gregor LEANDER, Pierre MAYR, Nils POHL, Lucas DAVI et Tim GÜNEYSU, « ClepsydraCache - Preventing Cache Attacks with Time-based Evictions », in : *Proc. 32th USENIX Security Symposium (USENIX Security)*, 2023, URL : <https://www.usenix.org/conference/usenixsecurity23/presentation/thoma>.
- [84] Muhammad Asim MUKHTAR, Muhammad Khurram BHATTI et Guy GOGNIAT, « IE-Cache : Counteracting Eviction-Based Cache Side-Channel Attacks Through Indirect Eviction », in : *Proc. ICT Systems Security and Privacy Protection*, t. 580, 2020, p. 32-45, DOI : 10.1007/978-3-030-58201-2_3.
- [85] Xingjian ZHANG, Haochen GONG, Rui CHANG et Yajin ZHOU, « RECAST : Mitigating Conflict-Based Cache Attacks Through Fine-Grained Dynamic Mapping », in : *IEEE Transactions on Information Forensics and Security* 19 (2024), p. 3758-3771, DOI : 10.1109/TIFS.2024.3368862.
- [86] Divya OJHA et Sandhya DWARKADAS, *RollingCache : Using Runtime Behavior to Defend Against Cache Side Channel Attacks*, 2024, DOI : 10.48550/arXiv.2408.08795.
- [87] Nils WISTOFF, Gernot HEISER et Luca BENINI, *fence.t.s : Closing Timing Channels in High-Performance Out-of-Order Cores through ISA-Supported Temporal Partitioning*, 2024, DOI : 10.48550/arXiv.2409.07576.
- [88] Mathieu ESCOUTELOUP, Ronan LASHERMES, Jacques FOURNIER et Jean-Louis LANET, « Under the Dome : Preventing Hardware Timing Information Leakage », in : *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*, 2021, DOI : 10.1007/978-3-030-97348-3_13.
- [89] Jicheng SHI, Xiang SONG, Haibo CHEN et Binyu ZANG, « Limiting Cache-based Side-Channel in Multi-tenant Cloud using Dynamic Page Coloring », in : *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, DOI : 10.1109/DSNW.2011.5958812.
- [90] Ying YE, Richard WEST, Zhuoqun CHENG et Ye LI, « COLORIS : A Dynamic Cache Partitioning System using Page Coloring », in : *Proc. International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, DOI : 10.1145/2628071.2628104.

-
- [91] Taesoo KIM, Marcus PEINADO et Gloria MAINAR-RUIZ, « STEALTHMEM : System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud », in : *Proc. 21th USENIX Security Symposium (USENIX Security)*, 2012, URL : <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim>.
- [92] Fan YAO, Hongyu FANG, Miloš DOROSLOVAČKI et Guru VENKATARAMANI, « COTSknight : Practical Defense against Cache Timing Channel Attacks using Cache Monitoring and Partitioning Technologies », in : *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2019, DOI : 10.1109/HST.2019.8740835.
- [93] Leonid DOMNITSER, Aamer JALEEL, Jason LOEW, Nael ABU-GHAZALEH et Dmitry PONOMAREV, « Non-Monopolizable Caches : Low-Complexity Mitigation of Cache Side Channel Attacks », in : *ACM Transactions on Architecture and Code Optimization* (jan. 2012), DOI : 10.1145/2086696.2086714.
- [94] Yao WANG, Andrew FERRAIUOLO, Danfeng ZHANG, Andrew C. MYERS et G. Edward SUH, « SecDCP : Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection », in : *Proc. Design Automation Conference (DAC)*, 2016, DOI : 10.1145/2897937.2898086.
- [95] Sercan SARI, Onur DEMIR et Gurhan KUCUK, « FairSDP : Fair and Secure Dynamic Cache Partitioning », in : *Proc. International Conference on Computer Science and Engineering (UBMK)*, 2019, DOI : 10.1109/UBMK.2019.8907000.
- [96] Ghada DESSOUKY, Tommaso FRASSETTO et Ahmad-Reza SADEGHI, « HybCache : Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments », in : *Proc. 29th USENIX Security Symposium (USENIX Security)*, 2020, URL : <https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky>.
- [97] Daniel SANCHEZ et Christos KOZYRAKIS, « Vantage : scalable and efficient fine-grain cache partitioning », in : *Proc. International Symposium on Computer Architecture (ISCA)*, 2011, DOI : 10.1145/2000064.2000073.
- [98] Mengming LI, Kai BU, Chenlu MIAO et Kui REN, « TreasureCache : Hiding Cache Evictions Against Side-Channel Attacks », in : *IEEE Transactions on Dependable and Secure Computing* 21.5 (2024), p. 4574-4588, DOI : 10.1109/TDSC.2024.3354991.
- [99] Daniel GRUSS, Julian LETTNER, Felix SCHUSTER, Olya OHRIMENKO, Istvan HALLER et Manuel COSTA, « Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory », in : *Proc. 26th USENIX Security Symposium (USENIX Security)*, 2017, URL : <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss>.

-
- [100] Hongyu FANG, Sai Santosh DAYAPULE, Fan YAO, Miloš DOROSLOVAČKI et Guru VENKATARAMANI, « Defeating Cache Timing Channels with Hardware Prefetchers », in : *IEEE Design & Test* 38.3 (2021), p. 7-14, DOI : 10.1109/MDAT.2021.3063313.
- [101] André SEZNEC, « A Case for Two-Way Skewed-Associative Caches », in : *Proc. International Symposium on Computer Architecture (ISCA)*, 1993, DOI : 10.1145/165123.165152.
- [102] Andrey BOGDANOV, Lars R KNUDSEN, Gregor LEANDER, Christof PAAR, Axel POSCHMANN, Matthew JB ROBshaw, Yannick SEURIN et Charlotte VIKKELSOE, « PRESENT : An ultra-lightweight block cipher », in : *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007, DOI : 10.1007/978-3-540-74735-2_31.
- [103] Christof BEIERLE, Jérémy JEAN, Stefan KÖLBL, Gregor LEANDER, Amir MORADI, Thomas PEYRIN, Yu SASAKI, Pascal SASDRICH et Siang Meng SIM, « The SKINNY family of block ciphers and its low-latency variant MANTIS », in : *Advances in Cryptology - CRYPTO*, 2016, DOI : 10.1007/978-3-662-53008-5_5.
- [104] Roberto AVANZI, « The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes », in : (2017), DOI : 10.13154/tosc.v2017.i1.4-44.
- [105] Federico CANALE, Tim GÜNEYSU, Gregor LEANDER, Jan Philipp THOMA, Yosuke TODO et Rei UENO, « SCARF : A Low-Latency Block Cipher for Secure Cache-Randomization », in : *Proc. 32th USENIX Security Symposium (USENIX Security)*, 2023, URL : <https://www.usenix.org/conference/usenixsecurity23/presentation/canale>.
- [106] Nicolas GAUDIN, Jean-Loup HATCHIKIAN-HOUDOT, Frédéric BESSON, Pascal COTRET, Guy GOGNIAT, Guillaume HIET, Vianney LAPOTRE et Pierre WILKE, « Work in Progress : Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections », in : *Proc. IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, juill. 2023, DOI : 10.1109/EuroSPW59978.2023.00038.
- [107] Ning ZHANG, Kun SUN, Deborah SHANDS, Wenjing LOU et Y. Thomas HOU, *TruSpy : Cache Side-Channel Information Leakage from the Secure World on ARM Devices*, 2016.
- [108] OPENSSL, *OpenSSL Project*, OpenSSL, URL : <https://openssl.org>.
- [109] Intel CORPORATION, *TinyCrypt Cryptographic Library*, Intel, URL : <https://github.com/intel/tinycrypt>.
- [110] National Security AGENCY, *Ghidra - Software Reverse Engineering Framework*, NSA, URL : <https://github.com/NationalSecurityAgency/ghidra>.

-
- [111] Hedi FENDRI, Marco MACCHETTI, Jérôme PERRINE et Mirjana STOJILLOVIĆ, « A Deep-Learning Approach to Side-Channel Based CPU Disassembly at Design Time », in : *Proc. Design, Automation & Test in Europe Conference (DATE)*, 2022, DOI : 10.23919/DATE541114.2022.9774531.
- [112] Muhui JIANG, Yajin ZHOU, Xiapu LUO, Ruoyu WANG, Yang LIU et Kui REN, « An empirical study on ARM disassembly tools », in : *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, DOI : 10.1145/3395363.3397377.
- [113] Sandrine BLAZY, David PICHARDIE et Alix TRIEU, « Verifying constant-time implementations by abstract interpretation », in : *Journal Comput. Secur.* 27 (2019), DOI : 10.3233/JCS-181136.
- [114] Gernot HEISER, « For Safety's Sake : We Need a New Hardware-Software Contract! », in : *IEEE Design & Test* (2018), DOI : 10.1109/MDAT.2017.2766559.
- [115] 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)*, 2018, DOI : 10.1145/3265723.3265724.
- [116] Yongseok LEE, Jonghee YOUN, Kevin NAM, Heon Hui JUNG, Myunghyun CHO, Jimyung NA, Jong-Yeon PARK, Seungsu JEON, Bo Gyeong KANG, Hyunyoung OH et Yunheung PAEK, « An Efficient Hardware/Software Co-Design for FALCON on Low-End Embedded Systems », in : *IEEE Access* 12 (2024), DOI : 10.1109/ACCESS.2024.3387489.
- [117] Mathieu ESCOUTELOUP, Ronan LASHERMES, Jean-Louis LANET et Jacques FOURNIER, « Recommendations for a radically secure ISA », in : *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2020, URL : <https://hal.science/hal-03128242/document>.
- [118] David A PATTERSON et Carlo H SEQUIN, « RISC I : A reduced instruction set VLSI computer », in : *25 years of the international symposia on Computer architecture (selected papers)*, 1998, p. 216-230.
- [119] « The RISC-V Instruction Set Manual, Volume I : User-Level ISA », in : *RISC-V International*, sous la dir. d'Andrew WATERMAN et Krste ASANOVIĆ, 2019.
- [120] « The RISC-V Instruction Set Manual, Volume II : Privileged Architecture », in : *RISC-V International*, sous la dir. d'Andrew WATERMAN et Krste ASANOVIĆ, 2019.
- [121] DEEPCOMPUTING, *DC-ROMA RISC-V LAPTOP II - A laptop based on a RISC-V CPU*, DeepComputing, URL : <https://deepcomputing.io/product/dc-roma-risc-v-laptop-ii>.

-
- [122] SCALEWAY, *Scaleway lance ses serveurs RISC-V dans le cloud, une première mondiale*, ScaleWay, URL : <https://www.scaleway.com/fr/news/scaleway-lance-ses-serveurs-risc-v-dans-le-cloud-une-premiere-mondiale-et-un-geste-fort-pour-lindependance-technologique>.
- [123] Nicolas GAUDIN, Pascal COTRET, Guy GOGNIAT et Vianney LAPÔTRE, « A Fine-Grained Dynamic Partitioning Against Cache-Based Timing Attacks via Cache Locking », in : *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024, DOI : 10.1109/ISVLSI61997.2024.00041.
- [124] Iulia BASTYS, Pauline BOLIGNANO, Franco RAIMONDI et Daniel SCHOEPE, « Automatic Annotation of Confidential Data in Java Code », in : *Foundations and Practice of Security*, 2022, DOI : 10.1007/978-3-031-08147-7_10.
- [125] Aakanksha SAHA, Tamara DENNING, Vivek SRIKUMAR et Sneha Kumar KASERA, « Secrets in Source Code : Reducing False Positives using Machine Learning », in : *International Conference on COMMunication Systems & NETWORKS (COMSNETS)*, 2020, DOI : 10.1109/COMSNETS48256.2020.9027350.
- [126] Edward J. SCHWARTZ, Thanassis AVGERINOS et David BRUMLEY, « All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) », in : *Proc. IEEE Symposium on Security and Privacy (SP)*, 2010, p. 317-331, DOI : 10.1109/SP.2010.26.
- [127] OpenHW GROUP, *OpenHW Group CORE-V CV32E40P RISC-V IP*, OpenHW Group, 2019, URL : <https://github.com/openhwgroup/cv32e40p>.
- [128] NEWAE, *Embedded Security Isn't Easy*. NewAE, URL : <https://www.newae.com/embedded-security-101>.
- [129] YOSSIEA, *AES S-box*. Wikimedia Commons, URL : https://commons.wikimedia.org/wiki/File:AES_S-box.png.
- [130] Wilson SNYDER, *Welcome to Verilator, the fastest Verilog/SystemVerilog simulator*. Veritool, URL : <https://www.veripool.org/verilator>.
- [131] EMBENCH, *Embench™ : Open Benchmarks for Embedded Platforms*, Embench, 2020, URL : <https://github.com/embench/embench-iot>.
- [132] Moritz PETERS, Nicolas GAUDIN, Jan Philipp THOMA, Vianney LAPÔTRE, Pascal COTRET, Guy GOGNIAT et Tim GÜNEYSU, « On The Effect of Replacement Policies on The Security of Randomized Cache Architectures », in : *Proc. ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2024, p. 483-497, DOI : 10.1145/3634737.3637677.

-
- [133] Amine JAAMOUM, Thomas HISCOCK et Giorgio DI NATALE, « Noise-Free Security Assessment of Eviction Set Construction Algorithms with Randomized Caches », in : *Applied Sciences* 12.5 (2022), DOI : 10.3390/app12052415.
- [134] Jean-Loup HATCHIKIAN-HOUDOT, Pierre WILKE, Frédéric BESSON et Guillaume HIET, « Formal Hardware/Software Models for Cache Locking Enabling Fast and Secure Code », in : *Proc. European Symposium on Research in Computer Security*, 2024, DOI : 10.1007/978-3-031-70896-1_8.
- [135] Nicolas GAUDIN, Pascal COTRET, Guy GOGNIAT et Vianney LAPÔTRE, « Verrouillage des lignes de cache pour la lutte contre les attaques par canaux auxiliaires exploitant les mémoires caches », in : *Cyber On Board*, 2024, URL : <https://hal.science/hal-04461273/>.
- [136] Nicolas GAUDIN, Pascal COTRET, Guy GOGNIAT et Vianney LAPÔTRE, « Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections », in : *CYBERUS summer school*, juill. 2023, URL : <https://hal.science/hal-04424956>.
- [137] Nicolas GAUDIN, Vianney LAPÔTRE, Pascal COTRET et Gogniat GUY, « Cache locking against cache-based side-channel attacks », in : *École d'hiver Francophone sur les Technologies de Conception des Systèmes Embarqués Hétérogènes (FETCH)*, fév. 2024, URL : <https://hal.science/hal-04446221>.

Titre : Partitionnement dynamique à grain fin contre des attaques par canaux auxiliaires en cache

Mot clés : Canaux auxiliaires en cache, RISC-V, Sécurité, Architecture de cache

Résumé : L'utilisation des systèmes embarqués ne cesse de croître et profite des avancées architecturales des processeurs modernes dans le but d'augmenter les performances tout en gardant une consommation d'énergie faible : nous pouvons citer, par exemple, les mémoires caches dans les systèmes embarqués. Elles permettent d'accélérer considérablement les accès mémoires en stockant temporairement les données au plus proche du cœur d'exécution. Par ailleurs, les données des différentes applications partagent les mêmes ressources matérielles. Cette particularité a pour effet qu'une application peut avoir un impact sur l'exécution des autres applications du système. Cette interaction est à la base des attaques par canaux auxiliaires exploitant les mémoires caches. Ce type de vulnérabilité est principalement ex-

ploité pour attaquer les applications cryptographiques dans le but d'extraire les données secrètes qu'elles manipulent. Ces menaces, bien connues des processeurs modernes, ont conduit à des contremesures complexes, souvent inapplicables aux systèmes embarqués ou engendrant un surcoût trop élevé. C'est la raison pour laquelle nous proposons, dans ces travaux, une contremesure basée sur un partitionnement à grain fin, qui permet à une application de verrouiller dynamiquement ses données en mémoire cache. Une fois les données verrouillées, aucune application ne peut inférer les accès réalisés sur celles-ci. Cette solution apporte des garanties de sécurité sur des sections de programmes critiques tout en engendrant un faible surcoût temporel (<4%) grâce à une solution hybride matérielle et logicielle.

Title: Fine-grained dynamic partitioning against cache-based side channel attacks

Keywords: Cache Side-Channel, RISC-V, Security, Cache Architecture

Abstract: The growth of embedded systems takes advantage of architectural advances from modern processors to increase performance while maintaining a low power consumption. Among these advances is the introduction of cache memory into embedded systems. These memories speed up the memory accesses by temporarily storing data close to the execution core. Furthermore, data from different applications share the same hardware resources, so the execution of one application affects the others. These interactions between applications give rise to cache-based side-channel attacks. This threat takes advantage of memory accesses to extract secret data executed by cryptographic

applications. These attacks are well known on modern processors and have led to countermeasures designed for modern processors. These solutions are either not feasible on embedded systems due to their requirements or result in high additional costs. In this context, we present a countermeasure based on a fine-grained partitioning, so that an application can dynamically lock its data into the cache. Once a data is locked, no application can infer information about the memory accesses made to it. It provides strong security guarantees for critical program sections while introducing a low performance overhead (<4%) through a new hardware/software contract.