

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :

l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *1 juillet 2022* **par :**

Yuxiao MAO

Détection dynamique d'attaques logicielles et matérielles basée sur l'analyse de signaux microarchitecturaux

---

JURY

Lilian Bossuet	Professeur des universités	Président du jury
Guy Gogniat	Professeur des universités	Rapporteur
Sylvain Guilley	Professeur des universités	Rapporteur
Karine Heydemann	Maître de conférences	Rapporteure
Maria Méndez Real	Maître de conférences	Examinatrice
Benoît Morgan	Maître de conférences	Examinateur
Vincent Migliore	Maître de conférences	Directeur de thèse
Vincent Nicomette	Professeur des universités	Directeur de thèse

---

École doctorale et spécialité :
MITT : Informatique et Télécommunications

Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes

Directeur(s) de Thèse :
Vincent Migliore et *Vincent Nicomette*

Rapporteurs :
Guy Gogniat, *Sylvain Guilley* et *Karine Heydemann*

Remerciements

Les travaux présentés dans ce manuscrit ont été effectués dans l'équipe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF) au sein du Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie M. Liviu Nicu et M. Mohamed Kaâniche, précédent et actuel directeur du laboratoire, de m'avoir accueillie au sein du laboratoire et d'avoir assuré la direction du LAAS depuis mon arrivée.

Je remercie également M. Mohamed Kaâniche et Mme Hélène Waeselynck, précédent et actuelle responsable de l'équipe TSF, de m'avoir accueillie au sein de l'équipe pour accomplir mes travaux de recherche dans de parfaites conditions de travail.

Je remercie sincèrement mes deux directeurs de thèse, qui m'ont accompagnée pendant toute la durée de la thèse : M. Vincent Nicomette, pour m'avoir apporté beaucoup de soutien, non seulement scientifique et technique, mais aussi linguistique et personnel, ce qui nécessite une grande gentillesse et une grande énergie ; M. Vincent Migliore, pour m'avoir donné beaucoup de bonnes idées tout en me laissant prendre mes propres décisions, pour m'avoir permis de toujours garder une vision globale avec du recul, ce qui m'a empêché à de multiples reprises de me perdre dans les détails techniques.

Je remercie mes rapporteurs, Mme Karine Heydemann, M. Guy Gogniat et M. Sylvain Guilley, d'avoir accepté de rapporter ma thèse et de participer à mon jury. Je remercie mes examinateurs, M. Lilian Bossuet, Mme Maria Mendez Real et M. Benoît Morgan d'avoir accepté de participer à mon jury. Je remercie particulièrement M. Lilian Bossuet d'avoir présidé mon jury. Les échanges avec les membres du jury ainsi que les questions posées lors de la soutenance ont été très enrichissants.

Je remercie Paul Florence et Lucas Georget, pour m'avoir apporté une aide précieuse lors de la réalisation du prototype.

Je remercie également toute l'équipe TSF, les permanents, les doctorants et les docteurs, l'ambiance de l'équipe est toujours très accueillante, malgré l'arrivée de la pandémie de COVID-19, qui a malheureusement perturbé cet environnement. Je remercie particulièrement Guillaume Averlant, avec qui j'ai fait mon stage de fin d'étude et avec qui j'ai découvert la recherche et l'équipe ; Romain Cayre, arrivé en même temps que moi en stage et en thèse, et qui a soutenu à un jour près ; Yassir Id Messaoud, mon co-bureau.

Pour finir, je remercie ma famille et mes amis, et surtout mon copain, qui m'ont accompagnée et supportée pendant les périodes difficiles. Sans cela, je n'aurais pas pu arriver jusqu'au bout.

Table des matières

Liste des figures viii

Liste des tableaux ix

Introduction 1

1 Contexte 5
- 1.1 Ordinateurs et exécution des logiciels . . . 6
  - 1.1.1 Présentation générale . . . 6
  - 1.1.2 Couches d'abstraction . . . 7
  - 1.1.3 Exécution des logiciels et menaces . . . 11
- 1.2 Terminologie de la sécurité informatique . . . 12
  - 1.2.1 Sûreté de fonctionnement . . . 12
  - 1.2.2 Sécurité-immunité . . . 14
- 1.3 Classification des attaques . . . 15
  - 1.3.1 Critères de classification . . . 15
  - 1.3.2 Exemples d'attaques . . . 16
- 1.4 Sécurité couche basse . . . 20
  - 1.4.1 Empêcher la fuite d'information . . . 21
  - 1.4.2 Empêcher la descente d'information . . . 22
  - 1.4.3 Empêcher la remontée d'information . . . 24
  - 1.4.4 Détecter la présence de malveillance . . . 25
- 1.5 Conclusion et objectif de la thèse . . . 26

2 Contexte technique et état de l'art 29
- 2.1 Détection au niveau architecture (non reconfigurable) . . . 30
  - 2.1.1 Contexte technique . . . 30
  - 2.1.2 Détection basée sur des systèmes commerciaux . . . 30
  - 2.1.3 Détection basée sur des systèmes non commerciaux . . . 31
  - 2.1.4 Conclusion . . . 32
- 2.2 Détection au niveau microarchitecture (non reconfigurable) . . . 33
  - 2.2.1 Contexte technique . . . 33
  - 2.2.2 Compteurs matériels de performance . . . 34
  - 2.2.3 Supervision de signaux . . . 37
  - 2.2.4 Conclusion . . . 39
- 2.3 Matériel reconfigurable pour la détection . . . 40
  - 2.3.1 Contexte technique . . . 41
  - 2.3.2 État de l'art . . . 43
  - 2.3.3 Conclusion . . . 43
- 2.4 Conclusion . . . 44

3 Détection d'attaques par caractérisation d'empreintes microarchitecturales 47
3.1 Modèle de menace et hypothèses 47
3.2 Architecture 48
3.2.1 Partie matérielle 49
3.2.2 Partie logicielle 53
3.2.3 De l'intégration à l'utilisation 54
3.3 Conception et intégration de l'architecture de détection 55
3.3.1 Processus d'intégration 55
3.3.2 Choix de la configuration 56
3.4 Utilisation du système 59
3.4.1 Vue globale 59
3.4.2 Détection dynamique 60
3.5 Conclusion 64

4 Implémentation 65
4.1 Présentation générale 65
4.1.1 Plateforme d'expérimentation 66
4.1.2 Vue d'ensemble 67
4.2 Matériel 68
4.2.1 Système de base 68
4.2.2 Signaux supervisés et connexions 70
4.2.3 Module de détection matériel 73
4.3 Logiciel 75
4.3.1 Configuration du noyau 75
4.3.2 Module de détection logiciel 75
4.4 Conclusion 78

5 Étude de cas 81
5.1 Étude de cas : CSCA 81
5.1.1 Contexte technique 82
5.1.2 Méthode d'évaluation 86
5.1.3 Conception du détecteur et évaluation 88
5.1.4 Conclusion 99
5.2 Étude de cas : Attaques ROP 100
5.2.1 Contexte technique 100
5.2.2 Méthode d'évaluation 102
5.2.3 Conception du détecteur et évaluation 103
5.2.4 Conclusion 107
5.3 Conclusion 107

6 Conclusion 111
6.1 Conclusion 111
6.2 Perspectives 112

Bibliographie **115**

Liste des figures

1.1 Modèle de couches d'abstraction pour l'exécution des logiciels sur les ordinateurs. 8
1.2 Vocabulaire de la sûreté de fonctionnement. 13
1.3 Attaques représentatives de chaque catégorie pour l'exécution des logiciels. Abréviation : CA : canal auxiliaire. 17
1.4 Chemin de fuite de donnée entre deux logiciels à travers la micro-architecture. 21

2.1 Exemple d'un diagramme de temps des signaux logiques sur une interface "ready-valid". 33
2.2 Exemple d'architecture matérielle reconfigurable à grain fin (gauche) et à gros grain (droite). Source : [Chattopadhyay 2013] 42

3.1 Architecture de détection qui intègre un MD reconfigurable dans le système cible. 49
3.2 Types de connexion des signaux supervisés pour le MDM à basse fréquence. 52
3.3 Intégration de l'architecture de détection dans le système cible. . . . 56
3.4 Démarche itérative pour la configuration de la partie non modifiable de l'architecture. 57
3.5 Utilisation du système pendant son cycle de vie. 59
3.6 Diagramme de séquence d'utilisation du système pour la détection dynamique des attaques (réinitialisation). 61
3.7 Diagramme de séquence d'utilisation du système pour la détection dynamique des attaques (exécution). Abréviation : IoC : indice de compromission. 62
3.8 Diagramme de séquence d'utilisation du système pour la détection dynamique des attaques (traitement d'une anomalie detectée par le MDL ou une interruption envoyée par le MDM). 63

4.1 Plateforme d'expérimentation composée d'une carte d'évaluation ML605 et d'une machine hôte. 66
4.2 Structure globale du prototype, incluant le système de base et les modifications apportées à chaque partie. 67
4.3 Architecture matérielle du prototype, basée sur un SoC généré par Chipyard. 69
4.4 Passage d'une commande depuis le terminal jusqu'au MDM, et retour de résultats. 76

5.1 Correspondance des blocs de la mémoire principale (blocs de 64 octets) dans un cache 2-associatif avec 64 *cache sets* et 64 octets par ligne de cache. Les blocs de la mémoire principale dont le numéro est multiple de 64 correspondent toujours au *cache set* d'index 0. 83

Liste des tableaux

2.1 Comparaison des méthodes de détection dynamique des attaques basées sur les traces architecturales. 32
2.2 Comparaison des méthodes de détection dynamique des attaques basées sur les compteurs matériels de performance. 40
2.3 Comparaison des méthodes de détection dynamique des attaques basées sur la supervision des signaux microarchitecturaux. 40
2.4 Comparaison des méthodes de détection dynamique des attaques avec l'utilisation du matériel reconfigurable pour la logique de détection. 44

5.1 Évaluation de la détection basée sur la détection de deux instructions de mesure de temps proche. 91
5.2 Évaluation de la détection basée sur la détection d'un long enchaînement des instructions de lecture mémoire de la taille d'une adresse mémoire. 93
5.3 Évaluation de la détection basée sur la détection des adresses mémoire accédées permettant de remplir un *cache set*. 95
5.4 Évaluation de la détection basée sur le nombre de défauts de cache mesuré par rapport au nombre d'accès au cache mesuré. 97
5.5 Surcoût de surface en fonction des différentes configurations pour la détection des attaques Prime+Probe. 97
5.6 Valeur maximale de *countgadget* observée en fonction des différentes valeurs de *gadgetSizeThresh* configurées dans la logique de détection. 106

Introduction

Aujourd'hui, les systèmes informatiques sont utilisés dans une grande variété de domaines tels que l'administration, la production automatisée, les communications et le divertissement. L'utilisation de ces systèmes informatiques a considérablement augmenté l'efficacité de certaines tâches et changé de nombreuses habitudes dans notre vie quotidienne. Les systèmes informatiques s'étant largement répandus, la technologie informatique a également évolué rapidement. Cette évolution s'est faite à différentes couches des systèmes informatiques, du logiciel (systèmes d'exploitation et logiciels utilisateur) au matériel (microarchitecture et technologie des puces). En général, ces développements ont conduit à une variété de systèmes plus petits et plus puissants avec plus de fonctionnalités et aussi plus complexes. Cette utilisation dans de larges domaines d'application, potentiellement critiques ou sensibles, rend également les problématiques de sécurité fondamentales dans tout le cycle de vie des systèmes. Cette problématique est malheureusement difficile à aborder, notamment à cause de la complexité des systèmes. De plus, les attaques récentes commencent à cibler des mécanismes très bas niveau, à une échelle où la modélisation même du système deviendrait trop complexe car de plus en plus proche du circuit et de ses propriétés physiques.

Une solution simple consisterait à sécuriser toutes les couches indépendamment, mais cela est malheureusement plus compliqué dans la pratique : les attaques bas niveau, dites matérielles, ont des impacts inter-couches difficiles à percevoir lorsque l'on observe une unique couche. En outre, certaines de ces vulnérabilités peuvent être introduites par des logiciels et peuvent donc se répandre dans de nombreux systèmes dès qu'ils installent ces logiciels, ce qui renforce la criticité de ces attaques. En 2018, les célèbres attaques Spectre et Meltdown ont été divulguées. Elles exploitent des vulnérabilités de la microarchitecture des systèmes liées à certaines optimisations des performances, telles que l'exécution spéculative et la mise en cache, et ont réussi à porter atteinte à la confidentialité des informations sur des processeurs modernes. Ces attaques sont difficiles à détecter de manière traditionnelle car, au niveau de la couche logicielle, elles correspondent à une utilisation correcte du système. La détection et la prévention de ces attaques posent donc de nombreux défis :

— le niveau élevé de complexité et de variabilité de la microarchitecture implique une grande difficulté à identifier toutes les sources de vulnérabilité dans tous les systèmes ;
— les contremesures impliquant une modification de la microarchitecture peuvent impacter significativement les performances globales du système complet, ce qui est généralement non désirable ;
— les contremesures doivent pouvoir s'adapter à l'évolution des attaques durant toute la durée de vie du système, dont le matériel peut parfois avoir une durée de vie excédant la dizaine d'années.

Dans cette thèse, nous proposons de donner quelques éléments de réponse à

ces défis. Nous avons en particulier cherché à identifier des méthodes de détection couplant une analyse d'informations bas niveau, proches du circuit, avec des informations de haut niveau, extraites de l'analyse des algorithmes d'attaque. Notre travail a abouti à la construction d'un framework apportant une preuve de concept de notre approche. Nos principales contributions portent sur :

- le couplage d'informations bas niveau (source d'information riche) et haut niveau (algorithmes d'attaque) permettant d'adresser efficacement certaines attaques matérielles, en particulier sans l'introduction de faux positifs ;
- la présentation d'une méthodologie pour aider les concepteurs à choisir les informations bas niveau pertinentes provenant de la microarchitecture ;
- la démonstration qu'un matériel reconfigurable, typiquement un FPGA, est à même d'adresser la détection d'attaques matérielles, permettant d'envisager une mise à jour matérielle des méthodes de détection.

En outre, notre approche est suffisamment générique pour s'adapter à différentes architectures de processeur sans nécessiter d'adaptation des logiciels existants. Nous estimons également que notre approche permettra de s'affranchir de certaines contremesures plus contraignantes comme on peut le trouver dans la littérature, en apportant une alternative flexible.

L'organisation de ce manuscrit est la suivante. Le chapitre 1 présente le contexte général, le modèle d'abstraction en couches d'un système informatique ainsi que les menaces associées, la terminologie de la sécurité informatique, une classification des attaques, et des grandes approches pour la prévention et la détection des attaques microarchitecturales qui ont spécialement retenu notre attention.

Le chapitre 2 présente un état de l'art plus approfondi des différentes techniques de détection dynamique des attaques logicielles et matérielles en lien avec nos travaux. Trois grandes catégories de méthodes sont présentées, celles qui sont non reconfigurables et utilisent les signaux architecturaux, celles qui sont non reconfigurables qui utilisent les signaux microarchitecturaux, et celles qui utilisent du matériel reconfigurable.

Le chapitre 3 présente le framework que nous proposons, en détaillant les changements requis au niveau de la microarchitecture et du système d'exploitation, la méthodologie pour sélectionner les informations microarchitecturales appropriées, l'intégration de ce framework dans un système informatique spécifique, ainsi que la description du fonctionnement du système final (intégrant nos mécanismes de détection) pendant son cycle de vie.

Le chapitre 4 présente un prototype d'ISA RISC-V que nous avons construit sur un FPGA, avec le générateur de systèmes sur puce Chipyard, le cœur de processeur Rocket, et le système d'exploitation Linux. Il est ouvert, facile à utiliser et personnalisable, permettant ainsi aux concepteurs de systèmes de l'adapter à leurs propres besoins et permettant à la communauté de se l'approprier et de lui apporter des améliorations et adaptations.

Pour finir, deux cas d'étude menés sur notre prototype sont présentés dans le chapitre 5. Ils montrent comment des logiques relativement simples, implémentées dans le module de détection, nous ont permis de détecter des attaques de classes

différentes (attaques visant les caches, et attaques de type ROP) sur un système complet exécutant un système d'exploitation, via l'exploitation d'informations provenant de la microarchitecture.

Chapitre 1

Contexte

Sommaire

1.1 Ordinateurs et exécution des logiciels 6
1.1.1 Présentation générale 6
1.1.2 Couches d'abstraction 7
1.1.3 Exécution des logiciels et menaces 11
1.2 Terminologie de la sécurité informatique 12
1.2.1 Sûreté de fonctionnement 12
1.2.2 Sécurité-immunité 14
1.3 Classification des attaques 15
1.3.1 Critères de classification 15
1.3.2 Exemples d'attaques 16
1.4 Sécurité couche basse 20
1.4.1 Empêcher la fuite d'information 21
1.4.2 Empêcher la descente d'information 22
1.4.3 Empêcher la remontée d'information 24
1.4.4 Détecter la présence de malveillance 25
1.5 Conclusion et objectif de la thèse 26

Ce premier chapitre présente tout d'abord une vision succinte d'un système informatique, à la fois d'un point de vue logiciel et matériel. Nous décrivons ainsi les différentes couches d'abstraction et leurs propriétés, ainsi que le mécanisme d'exécution des logiciels sur un ordinateur. Nous présentons ensuite une terminologie de la sécurité informatique, pour donner une idée générique des menaces et des moyens de protection existant dans ce domaine. Afin de mieux nous positionner par rapport aux attaques existantes et à la tendance d'évolution des attaques, nous proposons ensuite une classification des attaques basée sur deux critères choisis, en fournissant des exemples dans chaque catégorie. Nous terminons ce chapitre par un focus sur la sécurité de la microarchitecture des systèmes informatiques, des moyens de protection courants, pour enfin présenter l'objectif de la thèse et justifier nos contributions.

1.1 Ordinateurs et exécution des logiciels

1.1.1 Présentation générale

Un *ordinateur*, dans sa définition de base, est une machine qui peut être programmée pour exécuter automatiquement des séquences d'opérations arithmétiques ou logiques. Un ordinateur moderne se compose d'au moins un élément de traitement, généralement un processeur (en anglais *Central Processing Unit*, CPU), ainsi que d'un certain type de mémoire informatique, généralement des puces de mémoire à semi-conducteur. Le processeur peut également interagir avec les périphériques, comme des dispositifs d'entrée (clavier, soucis, capteur, etc.) pour la récupération des données à traiter, et de sortie (écran, imprimante, actionneur, etc.) pour échanger avec l'extérieur. Nous considérons par la suite qu'un ordinateur est l'ensemble de ces trois éléments : processeur, mémoire et périphériques.

Un *logiciel* est le regroupement de séquences d'opérations et d'un jeu de données nécessaires à ces opérations, généralement pour réaliser une ou plusieurs tâches. L'ensemble des opérations (appelées aussi *instructions*) supportées par un ordinateur, est appelé son *jeu d'instructions*.

Avec cette définition, nous pouvons constater que l'ordinateur n'est pas seulement l'ordinateur personnel que l'on emploie dans la vie quotidienne. D'autres systèmes peuvent être qualifiés d'"ordinateur". Cela va des appareils simples comme les objets connectés et les machines à café, des appareils à usage général comme les ordinateurs personnels et les téléphones portables, jusqu'aux systèmes complexes comme des robots.

Les ordinateurs occupent une place très importante dans notre société d'aujourd'hui et dans notre vie quotidienne. En 2020, malgré l'impact de la pandémie de Covid-19, un nombre total de 1,3 milliard de smartphones [IDC 2021] et 275 millions d'ordinateurs personnels [Gartner 2021] ont été vendus dans le monde entier. L'entreprise Arm déclare avoir vendu 25 milliards de puces basées sur leurs technologies, couramment utilisées dans les smartphones et les systèmes embarqués, pendant l'année 2020 [Arm 2021]. D'un autre côté, le nombre de logiciels distribués pour des appareils à usage général est aussi de plus en plus élevé. Google Play Store, un distributeur de logiciels pour des ordinateurs ayant un système d'exploitation Android, couramment utilisé dans les smartphones mais aussi dans les véhicules, dispose d'environ 3 millions de logiciels au total au début de l'année 2021, et plus de 3000 nouveaux logiciels chaque jour [Appinventiv 2021]. Le bon fonctionnement de l'exécution des logiciels sur les ordinateurs, et la protection contre des menaces sont donc des sujets fondamentaux.

En même temps, la complexité des ordinateurs augmente rapidement depuis un peu plus d'un demi-siècle, et va probablement continuer à augmenter dans les années à venir. La loi de Moore propose une estimation de l'augmentation de cette complexité en termes du nombre de transistors possible sur la même surface : ce nombre double tous les deux ans. Cela a pu être vérifié jusqu'à aujourd'hui, par exemple, la taille d'un processeur n'a pas beaucoup évolué, mais le nombre de tran-

sistors dans un processeur croît de manière exponentielle : quelques milliers en 1970, quelques millions en 1990, quelques milliards en 2010 et des dizaines de milliards en 2021 [Rupp 2015]. Ces transistors supportent des logiques supplémentaires, et ont été utilisés par des fabricants pour ajouter de nouvelles fonctionnalités et des optimisations : le parallélisme, les caches, l'exécution dans le désordre, l'exécution spéculative, le support de la virtualisation, la protection des exécutions comme ARM TrustZone et Intel SGX, etc. Avec un tel niveau de complexité, nous pouvons comprendre que la maîtrise complète d'ordinateur (incluant logiciel et matériel) par une seule personne est presque impossible, et la sécurisation de l'exécution du logiciel est loin d'être évidente.

1.1.2 Couches d'abstraction

Avec l'évolution des technologies et des outils, les ordinateurs d'aujourd'hui sont des systèmes réellement complexes. Afin de simplifier la compréhension du fonctionnement d'un ordinateur et de faciliter l'effort de conception, une vue hiérarchique de l'exécution d'un logiciel sur un ordinateur est souhaitable (voire indispensable). On définit ainsi des couches d'abstraction et on peut les présenter de manière incrémentale : la construction d'une nouvelle couche repose sur l'utilisation de fonctionnalités exposées par la couche directement inférieure, permettant ainsi d'obtenir des fonctionnalités de plus en plus évoluées. Les couches basses, les plus proches du matériel, donnent une vision précise et détaillée de l'ordinateur ; les couches hautes quant à elles, permettent d'avoir une vision plus globale avec des fonctionnalités avancées.

Pour illustrer ce principe, nous présentons ici un modèle de couches d'abstraction à quatre couches (allant des couches basses aux couches hautes), comme illustré dans la figure 1.1 : la couche physique, la couche microarchitecture, la couche architecture et la couche algorithmique. Parmi ces quatre couches, les couches physique et microarchitecture sont considérées comme des couches matérielles car leur description est très proche du circuit (description de portes logiques, de bascules, etc.). Les couches architecture et algorithmique sont considérées comme des couches logicielles car elles reposent sur une architecture du jeu d'instructions (en anglais *Instruction Set Architecture*, ISA). Cette architecture du jeu d'instructions fait en particulier office d'interface entre la couche matérielle et la couche logicielle. Notons que les couches logicielles sont volontairement simplifiées, la couche algorithmique pouvant notamment être elle-même subdivisée dans le cas d'un système d'exploitation en couche utilisateur et couche noyau par exemple. Les détails de chaque couche sont présentés par la suite.

Le résultat de l'exécution d'un logiciel n'est pas censé varier en fonction de sa représentation dans les différentes couches. Notons que cette hypothèse est à nuancer face à la présence d'un agent malveillant : l'ordinateur étant fondamentalement un système dont les ressources sont partagées, l'exploitation de vulnérabilités (notamment au niveau matériel) peut amener à l'observation ou la modification du comportement du logiciel. Dans cette section, nous faisons l'hypothèse d'une utilisation normale de l'ordinateur.

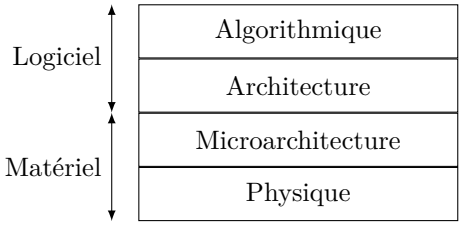


FIGURE 1.1 – Modèle de couches d'abstraction pour l'exécution des logiciels sur les ordinateurs.

1.1.2.1 Couche physique

La couche physique correspond à la vision physique des composants matériels. Elle décrit les différents composants logiques et séquentiels ainsi que leurs connexions pour former le circuit. Cette couche tient compte également des propriétés physiques des composants, typiquement le comportement thermique et électromagnétique. À titre d'exemple, la compatibilité électromagnétique qui vise à réduire l'interférence entre les composants et vis-à-vis de l'environnement, ou encore la gestion des différences de propagation des signaux dans le circuit, font partie de cette couche.

Aujourd'hui, les principaux ordinateurs sont des systèmes électroniques séquentiels. On utilise principalement les états 1 et 0, représentés par les tensions hautes et basses dans un circuit électronique. Les portes logiques permettant de manipuler ces états, par exemple ET, OU, NON, sont construites avec des transistors, résistances et fils électriques. Les états logiques sont stockés dans des structures comme des bascules (en anglais *Flip-flops*) construites à partir de portes logiques. Pour les ordinateurs non électroniques, par exemple les ordinateurs quantiques, les états à représenter et les portes logiques pour les manipuler sont différents.

La couche physique se charge de garantir que le fonctionnement des composants matériels correspond à leur description logique. Typiquement, elle doit concevoir de façon appropriée le circuit logique et placer des composants matériels, définir la fréquence maximale de fonctionnement du circuit, mettre en place des mécanismes permettant de réduire et corriger des erreurs dues aux propriétés physiques.

1.1.2.2 Couche microarchitecture

La couche microarchitecture est une description logique de la couche physique. Elle ne prend pas en compte les détails de conception des différents éléments logiques ou séquentiels, et notamment les phénomènes transitoires qui peuvent apparaître entre deux états stables. Elle est en générale décrite sous la forme de transfert de données entre des éléments de mémorisation (typiquement des registres) avec des opérations logiques entre ces données. Le *Register Transfer Level* (RTL) est un exemple de méthode de description courante. De même, des langages de programmation permettent de décrire et implémenter ces microarchitectures, notamment via le langage de description matérielle (en anglais *Hardware Description Language*,

HDL).

La description du fonctionnement logique des ordinateurs, que ce soit pour des processeurs, des mémoires ou des périphériques, se fait au niveau de cette couche. Elle définit toutes les connexions logiques, afin de réaliser entre autres les instructions conformément au jeu d'instructions. Pour la réalisation d'une même fonctionnalité, il peut y avoir de multiples possibilités en microarchitecture. Par exemple, pour réaliser une addition, il est possible d'utiliser un additionneur série, un additionneur parallèle à propagation de retenue, ou un additionneur parallèle à retenue anticipée, etc. Chaque type d'additionneur correspond à un circuit logique différent avec une capacité de calcul, une surface, et un délai différents. Pour accélérer l'exécution des séquences d'instructions dans le processeur, différentes possibilités peuvent être envisagées : la technique superscalaire qui consiste à envoyer simultanément plusieurs instructions à différentes unités d'exécution du processeur ; l'exécution dans le désordre qui réorganise l'ordre des instructions exécutées dans le processeur et donne la priorité aux instructions dont les ressources nécessaires sont déjà disponibles. De nombreuses autres optimisations comme le pipeline, le renommage des registres et la prédiction sont également utilisées. L'ensemble de ces optimisations est d'ailleurs fondamental pour chaque constructeur de processeur, puisque la performance est aujourd'hui un critère de choix fondamental dans l'adoption d'un processeur par les utilisateurs. Par conséquent, ces optimisations sont en général conservées secrètes par le constructeur et ne sont pas documentées car elles constituent en quelque-sorte la vraie valeur ajoutée par rapport à la concurrence.

La couche microarchitecture masque le détail d'implémentation du jeu d'instructions, y compris des optimisations. Ainsi, les couches supérieures peuvent simplement utiliser les instructions définies par le jeu d'instructions, et le résultat d'exécution est garanti par la couche microarchitecture.

1.1.2.3 Couche architecture

L'architecture, ou également appelée ISA, est un modèle abstrait d'ordinateur. L'ISA détermine l'ensemble des instructions supportées par un processeur, les types de données, les registres du processeur, le modèle de communication avec la mémoire principale de l'ordinateur ainsi qu'un ensemble d'entrées/sorties. Il existe deux grandes familles d'architectures, l'architecture RISC (*Reduced Instruction Set Computer*) qui est le premier jeu d'instructions formalisé par l'Université de Stanford et Berkeley en 1981, et qui possède un jeu d'instructions de taille fixe ; et l'architecture CISC (*Complex Instruction Set Computer*) définie comme tous les processeurs non RISC possédant un jeu d'instructions de taille variable, construit de manière incrémentale avec les générations de processeurs de cette famille. Pour la famille RISC, nous allons retrouver typiquement les différents microcontrôleurs et microprocesseurs d'Arm ainsi que le récent RISC-V proposé par l'université de Berkeley. Pour la famille CISC, nous allons retrouver typiquement les processeurs d'Intel (x86, x86-64) et un grand nombre de processeurs historiques.

Bien que les détails d'implémentation relèvent en général de la microarchitec-

ture, lorsque les contraintes d'implémentation sont imposées par le jeu d'instructions, elles relèvent de la couche architecture. Par exemple, le fait de représenter la mémoire comme un ensemble d'adresses entières relève de la couche architecture. Au niveau architecture, les logiciels se représentent comme une suite d'instructions réalisant des opérations arithmétiques et logiques sur des registres d'une part, et communiquant avec la mémoire principale et les entrées/sorties d'autre part. Les formes courantes des logiciels dans cette couche sont le code machine (utilisé directement par l'ordinateur) et le code assembleur (plus lisible par un humain). Nous considérons également que les éléments nécessaires pour le fonctionnement du code machine, tels que la convention d'appel des fonctions et la structure de la pile, décrites dans l'*Application Binary Interface* (ABI), relèvent de la couche architecture.

L'utilisation d'une même architecture de jeu d'instructions entre différents ordinateurs permet d'exécuter sur ces différents ordinateurs la même version d'un logiciel exprimé au niveau de la couche architecture, et de réduire l'effort de distribution des logiciels.

1.1.2.4 Couche algorithmique

La couche algorithmique est une description de haut niveau du comportement et du fonctionnement prévus du logiciel. Cette description est facile à comprendre par un humain, et indépendante de la machine utilisée. Une forme possible est le code source, généralement écrit dans un langage de programmation de haut niveau, comme du C et du JAVA. Il peut aussi contenir certaines parties écrites directement en code assembleur, dans ce cas, une adaptation manuelle à chaque différente architecture sur lequel s'exécute le logiciel est nécessaire. Un compilateur traduit le code source en code machine (de la couche architecture). L'utilisation de ce code source facilite le développement des logiciels, puisqu'il est plus accessible pour les développeurs.

Par exemple, le fonctionnement de l'authentification pour un logiciel bancaire simple est le suivant : le logiciel prend en entrée un identifiant et un mot de passe donnés par l'utilisateur via un clavier ou un écran tactile, les envoie à la banque, reçoit une réponse qui lui indique si la connexion est réussie, et affiche le résultat à l'utilisateur. La communication avec des périphériques est réalisée sous la forme d'appels de fonctions au niveau de la couche algorithmique, même si au niveau de la couche architecture, elle est traduite par des accès à des zones de mémoire correspondant aux périphériques.

1.1.2.5 Synthèse

Le modèle d'abstraction décrit dans cette section est compatible avec la chaîne de compilation classique sur un ordinateur : le programmeur décrit les fonctionnalités du logiciel dans un langage de haut niveau (couche algorithmique) ; une première couche d'abstraction est "supprimée" lors de la phase de compilation avec l'aide du compilateur (couche architecture) ; une seconde couche d'abstraction est

"supprimée" lors de l'exécution du logiciel sur le processeur qui peut être modélisée au niveau microarchitecture ou physique.

1.1.3 Exécution des logiciels et menaces

Si les logiciels qui s'exécutent sur nos ordinateurs sont de plus en plus nombreux et de plus en plus complexes, les attaques qui ciblent les systèmes informatiques sont malheureusement elles aussi de plus en plus nombreuses et de plus en plus subtiles. Comme nous le verrons dans la suite de ce chapitre, elles peuvent viser toutes les couches dont nous avons parlé dans la section précédente. Certaines attaques ciblent l'exécution du logiciel, pour en modifier le comportement, ou collecter des informations secrètes. Les attaques les plus simples à réaliser et aussi les plus nombreuses sont celles qui se situent dans les couches les plus élevées (la couche algorithmique notamment). Par exemple, un paramètre d'entrée mal contrôlé dans sa description algorithmique peut permettre à un attaquant de réaliser une injection SQL dans une base de données et de récupérer des données qu'il n'a pas le droit d'accéder. Les attaques visant les couches basses existent également, mais elles sont souvent plus difficiles à réaliser, car elles nécessitent une bonne compréhension de ces couches, qui sont particulièrement complexes.

Face à ce risque de plus en plus important d'attaques, il est fondamental de pouvoir avoir confiance en l'exécution du logiciel et de se poser la question de la façon de le rendre plus sûr. Pour construire un système sûr, il est important de s'assurer qu'à chaque couche d'abstraction, celle-ci n'introduit pas de vulnérabilités. Pour se faire, chaque couche a besoin de s'appuyer sur des propriétés de sécurité qui sont garanties par les couches inférieures : chaque couche fait confiance aux propriétés garanties par la couche de dessous, pour fournir à son tour un certain nombre de propriétés pour les couches supérieures. Ainsi, la confiance se construit de bas en haut, permettant au final d'avoir confiance dans l'intégralité de l'exécution du logiciel. Il est bien sûr possible de réaliser des tests sur chaque couche afin d'observer le système et d'identifier des problèmes potentiels. Cependant, les tests ne peuvent pas être exhaustifs et pour assurer l'absence de vulnérabilités dans une couche, il est nécessaire de produire une modélisation du système dans cette couche, et de vérifier formellement l'absence de vulnérabilités à l'aide d'outils automatiques. Force est de constater que, même aujourd'hui, vu la complexité des systèmes généralistes, cette approche est loin d'être facilement réalisable : la vérification formelle et l'implémentation d'un logiciel sans fautes, avec l'hypothèse que le matériel est de confiance, sont en réalité faisables seulement lorsque la complexité du logiciel est limitée. Par exemple, seL4 [Heiser 2019] est un micro-noyau formellement prouvé, de la spécification jusqu'à l'implémentation, mais uniquement pour certaines propriétés (certaines propriétés concernant le temps et le cache ne sont pas prouvées par exemple). Des systèmes d'exploitation commerciaux sont généralement évalués au niveau EAL4 (*Evaluation Assurance Level*) qui nécessite seulement de concevoir, tester et réviser méthodiquement. La vérification formelle du matériel est faisable à niveau d'abstraction élevé, avec des propriétés à vérifier bien définies, sur des com-

posants de complexité limitée dont le comportement est bien connu et modélisé. Par exemple, il est possible de spécifier formellement un ISA [Nikhil 2021], et vérifier si une partie de la microarchitecture du système (sans virgule flottante, exception, gestion de mémoire, etc.) respecte la spécification [Reid 2016]. Mais la modélisation et la vérification formelle d'un système complet, incluant des logiciels et le matériel, est d'une complexité nettement plus élevée. Ainsi, en pratique, ce sont en général les bancs de tests et l'observation du comportement du système qui sont utilisés pour évaluer les parties complexes du système et le système complet, même si au final, ils ne peuvent vérifer de façon exhaustive le système dans son ensemble.

Pour cette raison, les attaques visant les couches basses sont particulièrement difficiles à prévoir, à détecter et à éliminer. Elles sont plus dangereuses, car en exploitant des vulnérabilités des couches basses, elles brisent la chaîne de confiance qui permet au final d'avoir confiance dans le logiciel qui s'exécute dans les couches supérieures. Elles sont également plus difficiles à observer dans les couches supérieures à cause de l'abstraction et des hypothèses faites par les couches supérieures. Ainsi, pour assurer le bon fonctionnement du logiciel, il est important de protéger les couches logicielles et les couches matérielles.

Nous présentons la terminologie de la sécurité informatique dans la section suivante. Une description plus détaillée des attaques considérées et des exemples d'attaques venant de différentes couches d'abstraction sont fournis dans la section 1.3.

1.2 Terminologie de la sécurité informatique

Le terme de sécurité en français est ambigu. Il peut désigner aussi bien la sécurité-innocuité (en anglais *safety*) que la sécurité-immunité (en anglais *security*), qui sont deux concepts relevant de la sûreté de fonctionnement. Nos travaux s'inscrivant dans le cadre de la sécurité-immunité. Cette section a pour objectif de définir le vocabulaire dont nous aurons besoin par la suite. Dans un premier temps, nous décrivons les concepts généraux de la sûreté de fonctionnement, introduits dans [Laprie 1996] et mis à jour dans [Avizienis 2004], puis nous nous concentrons sur l'application de ces concepts au domaine de la sécurité-immunité.

1.2.1 Sûreté de fonctionnement

La *sûreté de fonctionnement* d'un système informatique est définie comme "la propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre". Le service délivré par un système est son comportement tel qu'il est perçu par ses utilisateurs. Un utilisateur étant un autre système (humain ou physique) qui interagit avec le système considéré. La non-sûreté de fonctionnement survient lorsque la confiance ne peut plus, ou ne pourra plus, être placée dans le service délivré. La sûreté de fonctionnement comporte trois axes principaux : les *attributs* qui la décrivent, les *entraves* qui empêchent sa réalisation et les *moyens* d'atteindre celle-ci (voir la figure 1.2).

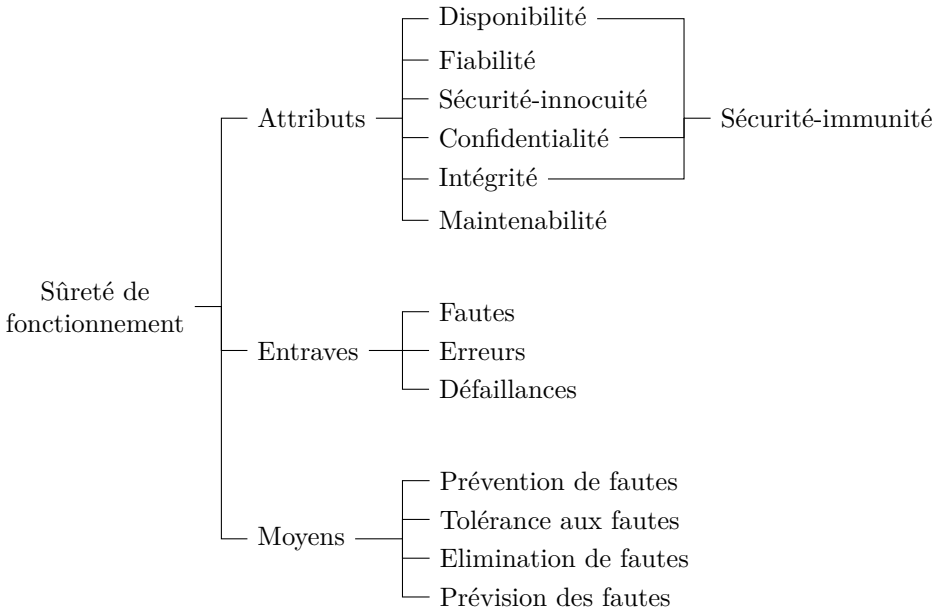


FIGURE 1.2 – Vocabulaire de la sûreté de fonctionnement.

La sûreté de fonctionnement peut être perçue selon les propriétés suivantes, appelées *attributs* de la sûreté de fonctionnement :

- *Disponibilité* : capacité du système à être prêt à l'utilisation ;
- *Fiabilité* : continuité du service ;
- *Sécurité-innocuité* : absence de conséquences catastrophiques pour l'environnement ;
- *Confidentialité* : absence de divulgations non autorisées de l'information ;
- *Intégrité* : absence d'altérations inappropriées de l'information ;
- *Maintenabilité* : aptitude aux réparations et aux évolutions.

Ainsi ces attributs permettent d'une part d'exprimer les propriétés devant être respectées par le système et d'autre part d'évaluer la qualité du service délivré vis-à-vis de ces propriétés. Les attributs à considérer dépendent des applications auxquelles le système est destiné.

Ces attributs peuvent être mis à mal par des *entraves*. Une entrave est une circonstance indésirable mais non-inattendue. Elle est la cause ou le résultat de la non-sûreté de fonctionnement. On distingue trois types d'entraves :

- *Défaillance* : survient lorsque le service délivré dévie de l'accomplissement de la fonction du système ;
- *Erreur* : la partie de l'état du système susceptible d'entraîner une défaillance ;
- *Faute* : la cause adjugée ou supposée d'une erreur.

Une faute est dite active lorsqu'elle produit une erreur. Par propagation, une erreur crée de nouvelles erreurs. Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système. Cette défaillance peut alors apparaître comme une faute du point de vue d'un autre composant. L'enchaînement

de ces entraves crée ainsi la chaîne fondamentale suivante :

$$\ldots \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow \text{erreur} \rightarrow \text{défaillance} \rightarrow \ldots$$

Enfin, les *moyens* de la sûreté de fonctionnement permettent de minimiser l'impact des entraves sur les attributs. Ce sont les méthodes et techniques permettant de conforter les utilisateurs quant au bon accomplissement de la fonction du système. On distingue quatre catégories en fonction de l'objectif visé :

- *Prévention des fautes* : empêcher l'occurrence ou l'introduction de fautes ;
- *Tolérance aux fautes* : fournir un service qui remplit la fonction du système en dépit des fautes ;
- *Elimination des fautes* : réduire le nombre et la sévérité des fautes ;
- *Prévision des fautes* : estimer la présence, la création et la conséquence des fautes.

Dans la suite de ce mémoire, nous nous intéressons uniquement à la sécurité-immunité, qui se définit comme une combinaison de la disponibilité, la confidentialité et l'intégrité.

1.2.2 Sécurité-immunité

La sécurité-immunité définit les propriétés selon lesquelles un système est dit "sécurisé" (protégé contre les fautes intentionnelles, aussi appelées *malveillances*). Contrairement à la sécurité-innocuité, qui définit les propriétés selon lesquelles un système est dit "sûr" (sans défaillance catastrophique pouvant conduire à des pertes de vies humaines ou des conséquences économiques importantes). Les concepts de sûreté de fonctionnement présentés précédemment sont génériques afin de pouvoir couvrir un grand nombre de concepts. Nous allons maintenant nous intéresser à leur application dans le contexte plus spécifique de la sécurité-immunité. Dans le reste de ce manuscrit, le terme *sécurité* ou de *sécurité informatique* sont à considérer au sens de sécurité-immunité, sauf si explicitement précisé.

Attributs Considérés du point de vue de la sécurité-immunité, les attributs de la sûreté de fonctionnement que l'on cherche à garantir peuvent être définis ainsi :

- *Confidentialité* : prévention de toute divulgation non autorisée d'information ;
- *Intégrité* : prévention de toute modification non autorisée d'information ;
- *Disponibilité* : prévention de toute rétention non autorisée d'information.

Malveillances Les définitions qui suivent proviennent de la terminologie des malveillances introduite lors du projet *Malicious- and Accidental- Fault Tolerance for Internet Applications* [MAFTIA 2003]. Il existe deux classes de fautes malveillantes : les logiques malignes et les intrusions. Une logique maligne est une partie du système conçue pour provoquer des dégâts (bombe logique) ou pour faciliter des intrusions futures (vulnérabilités créées volontairement). Elles peuvent être introduites dès la

conception du système (par un concepteur malveillant) ou en phase opérationnelle (par l'installation d'un logiciel contenant un cheval de Troie ou par une intrusion). Une intrusion est définie grâce aux concepts d'attaque et de vulnérabilité :

- *Attaque* : une faute d'interaction malveillante visant à violer une ou plusieurs propriétés de sécurité. C'est une faute externe créée avec l'intention de nuire, incluant les attaques lancées par des outils automatiques (vers, virus, etc.) ;
- *Vulnérabilité* : une faute, accidentelle ou intentionnelle, présente lors de la conception du système ou lors de son utilisation ;
- *Intrusion* : une faute malveillante correspondant au résultat d'une attaque qui a réussi à exploiter une vulnérabilité.

Les attaques, vulnérabilités et intrusions étant des fautes, les moyens de la sûreté de fonctionnement peuvent être appliqués pour améliorer la sécurité-immunité d'un système.

1.3 Classification des attaques

Dans cette section, nous proposons une classification des attaques menaçant la sécurité de l'exécution du logiciel. Pour chaque classe, nous donnons des exemples d'attaque avec une explication rapide de leur fonctionnement.

1.3.1 Critères de classification

Notre classification d'attaques est basée sur deux critères : l'origine de l'attaque et la vulnérabilité exploitée par l'attaque.

Origine de l'attaque Nous avons identifié trois origines d'attaque différentes :

- *Dispositif externe physique* : l'attaque provient de composants externes du système, qui interagissent avec le système indirectement via son environnement physique. Cela comprend principalement la modification et l'observation des propriétés physiques du système (par exemple la température, le champ électromagnétique, ou l'apparence visuelle) ;
- *Composant matériel* : l'attaque provient de composants matériels du système. Elle correspond à un composant matériel contenant des défauts volontairement introduits dans sa conception, fabrication ou diffusion, et qui a réussi à être intégré dans le système (pendant l'assemblage du système, ou être branché temporairement) ;
- *Composant logiciel* : l'attaque provient de composants logiciels du système. Cela comprend la manipulation des entrées d'un logiciel légitime (par exemple des entrées d'utilisateur, des fichiers ou des données du réseau), ou un logiciel malveillant installé dans le système.

Vulnérabilité visée Nous classifions les vulnérabilités visées par l'attaque à l'aide de notre modèle de couches d'abstraction. Nous obtenons ainsi quatre types de vulnérabilités différentes :

— *Vulnérabilité physique* : l'attaque vise une vulnérabilité liée aux propriétés physiques du système. Les propriétés physiques incluent par exemple l'émission d'énergie pendant l'utilisation du circuit, la variation des propriétés électroniques selon la température et la tension courante, ou les imperfections de fabrication ;
— *Vulnérabilité microarchitecturale* : l'attaque vise une vulnérabilité liée à la microarchitecture du système. Ces choix d'implémentation, principalement faits pour des raisons d'optimisation, incluent par exemple le partage des ressources, la persistance des états internes ;
— *Vulnérabilité architecturale* : l'attaque vise une vulnérabilité logicielle non algorithmique. Ces vulnérabilités incluent la possibilité d'accéder à des zones mémoire en dehors d'une table, de modifier des données de contrôle permettant la gestion de la pile ou l'appel de fonction, etc. Elles viennent principalement de l'utilisation des langages de programmation qui ne sont pas assez restrictifs, et des compilateurs qui n'ont pas ajouté suffisamment de vérifications ;
— *Vulnérabilité algorithmique* : l'attaque vise une vulnérabilité liée à la description haut niveau du comportement et des fonctionnalités du logiciel. Contrairement aux vulnérabilités architecturales, les vulnérabilités algorithmiques ne peuvent pas être résolues par la réécriture du logiciel dans un autre langage de programmation et l'utilisation d'un compilateur plus sécurisé. Le manque de vérification et de protection des données, le manque de prise en compte des différents chemins d'exécution, l'utilisation d'une clé de chiffrement cassable en force brute entrent dans cette catégorie.

1.3.2 Exemples d'attaques

Dans cette section, nous présentons des attaques typiques des différentes catégories (figure 1.3). Notons que dans cette figure, le nombre d'attaques dans chaque catégorie n'est pas proportionnel au nombre de carrés ou la taille du carré. Typiquement, la plupart des attaques existantes concerne l'interaction de type logiciel et exploite une vulnérabilité de la couche algorithmique.

1.3.2.1 Attaque par dispositif externe physique

Les attaques par dispositif externe physique nécessitent que l'attaquant ait un accès physique au circuit électronique de la machine pendant l'exécution du logiciel. La perte ou le vol de la machine, une personne interne malveillante permet de satisfaire cette condition, mais cette restriction forte rend l'attaque difficile à réaliser.

Avec un dispositif externe physique, une cible évidente est la modification ou l'exploitation des propriétés physiques de la cible. L'attaque par *injection de faute* [Hsueh 1997] par exemple, repose sur la modification de l'état d'un système afin d'altérer son flot de contrôle ou changer la valeur de certains de ses registres. Les

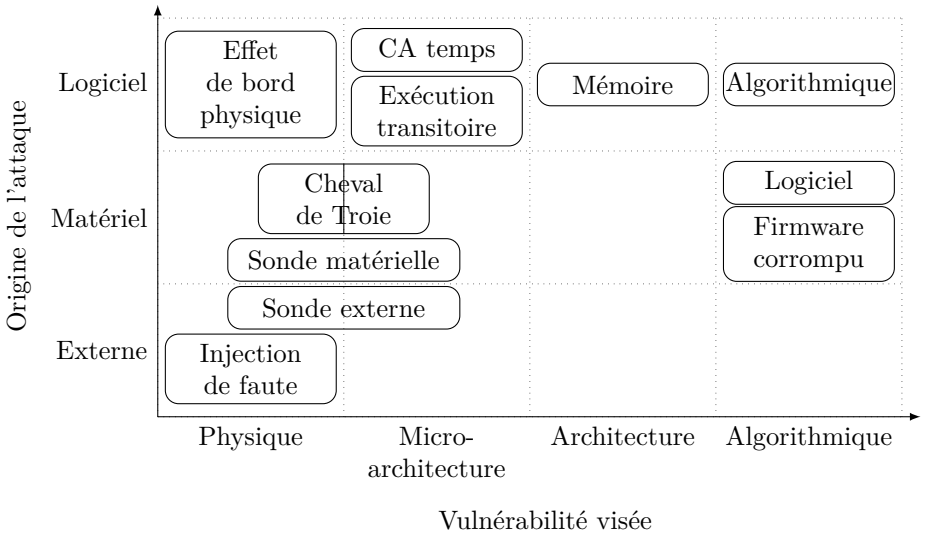


FIGURE 1.3 – Attaques représentatives de chaque catégorie pour l'exécution des logiciels. Abréviation : CA : canal auxiliaire.

méthodes usuelles d'injection de fautes reposent essentiellement sur des modifications de température ou de tension, la perturbation de l'alimentation ou de l'horloge du système, ou encore l'utilisation d'impulsions laser ou électromagnétiques sur le circuit.

Les attaques référencées sous le terme *sonde externe* dans la figure 1.3 correspondent à des attaques basées sur des sondes physiques pour observer le système. Elles visent des vulnérabilités physiques impactées par le changement des états logiques de la microarchitecture. Le *microprobing* [Skorobogatov 2017] consiste à placer des sondes sur des fils de cuivre et à observer l'évolution des niveaux de tension pendant l'exécution du système. Par exemple, lorsqu'un logiciel demande de réaliser un accès mémoire, cet accès est envoyé vers la mémoire principale à l'aide d'un bus constitué d'un ensemble de fils de cuivre contenant différents états binaires sous forme de niveaux de tension. En mesurant l'évolution des tensions, il est possible de déduire l'adresse mémoire utilisée par le logiciel. Les attaques par *canaux auxiliaires physiques* [Kocher 1996, Camurati 2018] utilisent des sondes à proximité du circuit, du millimètre à une dizaine de mètres en fonction du canal exploité. Les canaux typiques observables sont les émanations électromagnétiques, les ondes sonores, thermiques ou optiques, induites par les changements d'état des éléments logiques dans la microarchitecture. Ces mesures permettent de déduire des logiques du circuit activées dans le temps et certaines valeurs dans ces logiques. Dans le cas où l'utilisation des logiques est couplée avec l'exécution du logiciel, on peut en déduire des informations liées à l'exécution du logiciel.

1.3.2.2 Attaque par composant matériel

Les attaques par composant matériel demandent à l'attaquant d'installer un composant malveillant sur le système victime de manière permanente ou temporaire. L'installation du composant malveillant peut se faire soit par l'attaquant en étant proche de la machine, soit par la victime elle-même qui pense installer un matériel de confiance.

Les attaques référencées sous le terme *sonde matérielle* dans la figure 1.3 correspondent aux attaques qui utilisent un composant matériel malveillant comme sonde pour observer le comportement physique du système impacté par la microarchitecture. Un chargeur malveillant branché sur un téléphone peut être vu comme une sonde de la tension et de la consommation énergétique. En mesurant l'énergie transmise dans le temps, l'attaquant peut tracer l'impact de l'exécution du logiciel, et déduire par exemple le site Web visité par la machine cible [Clark 2013].

Des attaques du type *cheval de Troie matériel* [Bhasin 2013, Bhunia 2014] ciblent des vulnérabilités physiques ou microarchitecturales. Comme son nom indique, un cheval de Troie matériel est un composant matériel supposé légitime, qui intègre donc des fonctionnalités et des circuits identiques à ceux du composant légitime, mais qui contient également des modifications. Les modifications du type physique concernent par exemple l'épaisseur d'un fil cuivre, dont la réduction va diminuer la fiabilité du circuit pendant une utilisation fréquente. Les modifications du type microarchitecture concernant par exemple la modification de quelques connexions et portes logiques, généralement avec une condition d'activation, afin de modifier le résultat d'un calcul, de provoquer l'envoi des données secrètes vers l'extérieur, ou de créer une porte dérobée pour le logiciel.

Avec un composant matériel, il est également possible de viser des vulnérabilités algorithmiques. Similaire à un cheval de Troie mais sans modifier le matériel, l'attaque peut consister à remplacer le firmware légitime du composant matériel par un *firmware corrompu*, qui permet de modifier le comportement algorithmique du composant matériel. Par exemple, ce firmware peut faire croire qu'il a besoin d'un autre logiciel malveillant pour fonctionner, ou peut mentir sur le type du composant tel que vu par le noyau du système (au lieu de se déclarer comme une clé USB, le composant se déclare comme un adaptateur Ethernet ou un clavier, et peut ainsi contrôler le trafic réseau ou envoyer de fausses frappes clavier) [Nissim 2017]. Un *logiciel malveillant* peut être inclus dans un composant matériel contenant une mémoire, pour atteindre et infecter un ordinateur [Pham 2011]. Par exemple, une clé USB trouvée dans la rue (en anglais *USB drop attack*) peut contenir un tel logiciel, avec un script indiquant l'auto-exécution de ce logiciel après son branchement sur l'ordinateur. Dans ce cas, le composant matériel ne sert qu'à une première étape d'intrusion, la suite des attaques correspond à des attaques par composant logiciel.

1.3.2.3 Attaque par composant logiciel

Les attaques par composants logiciels sont relativement plus faciles à mener, car il n'y a pas besoin d'être physiquement proche de la machine cible, ni de disposer

des outils de modification du circuit. La plupart des attaques par logiciels peuvent être réalisées avec un ordinateur personnel, et un même binaire d'attaque peut être propagé sur le réseau et être exécuté sur plusieurs machines de même architecture.

Il est possible d'exploiter certaines vulnérabilités physiques avec un logiciel, par *effets de bord physiques*. L'attaque *rowhammer* [Kim 2014, Gruss 2018] consiste à exploiter un effet de bord physique de la DRAM (*Dynamic random access memory*). La DRAM est un type de composant mémoire souvent utilisée dans les mémoires principales des ordinateurs. Un accès fréquent dans une ligne de DRAM peut perturber la tension maintenue sur des lignes adjacentes. L'instabilité de la tension peut provoquer une mauvaise reconnaissance de la tension et donc une mauvaise interprétation de la valeur binaire associée, et ainsi changer la valeur d'un bit de la ligne adjacente sans directement manipuler la ligne. En complément d'autres stratégies qui positionnent avec précaution les données en mémoire, cette attaque permet de modifier une donnée spécifique d'un logiciel victime.

Parmi les attaques qui visent des vulnérabilités microarchitecturales, on peut citer les attaques par *canaux auxiliaires de temps* (en anglais *timing side-channel attacks*) et les attaques par *exécution transitoire* (en anglais *transient execution attacks*). Les attaques par canaux auxiliaires de temps [Osvik 2006] se basent sur des optimisations de performance et d'utilisation des ressources mises en place par la couche microarchitecture. En mesurant le temps utilisé pour certaines opérations du logiciel malveillant, ou en observant l'évolution des activités du système (la valeur des compteurs internes ou la disponibilité des ressources) dans le temps, l'attaquant peut inférer des informations sur l'exécution du logiciel victime partageant les mêmes ressources matérielles avec le logiciel malveillant. Les attaques par exécution transitoire [Kocher 2019] se basent sur des optimisations de performance qui cherchent à exécuter à l'avance certaines instructions, mais qui ne restaurent pas totalement les états de la microarchitecture si ces instructions ne doivent pas être exécutées au final. Un exemple de ce type d'optimisation est le franchissement d'une condition avant la validation de la condition. Une information secrète peut alors fuiter pendant cette exécution dans la microarchitecture, et être ensuite lue par un canal auxiliaire de temps.

Les attaques visant des vulnérabilités de la couche architecture sont généralement de type *corruption de mémoire* [Szekeres 2013] qui exploitent des vulnérabilités liées à l'organisation et l'utilisation de la mémoire dans un ordinateur. On peut citer les attaques de type *débordement de tampon* (en anglais *buffer overflow*) qui profitent de la trop grande permissivité des contrôles réalisées par certains langages de programmation et compilateurs. Ces accès en dehors de buffers en mémoire permettent de lire ou modifier des données de calcul ou de contrôle présentes dans ces zones mémoire, voire rediriger le flot d'exécution vers des instructions injectées dans ces zones. Les attaques du type *réutilisation de code* visent à rediriger le flot l'exécution vers des instructions déjà présentes dans la mémoire, comme les attaques *return-into-libc* [Nergal 2001] ou *return-oriented programming* (ROP) [Shacham 2007].

Et enfin, les attaques logicielles visant des vulnérabilités algorithmiques sont

incluses dans le label *algorithmique* de la figure 1.3. On peut citer par exemple l'*injection SQL*, qui bénéficie de la requête d'accès à une base de données non proprement vérifiée, pour arriver à accéder ou modifier certains contenus de la base. Contrairement au débordement de tampon, les critères de vérification ne peuvent pas être facilement ajoutés par un compilateur plus strict. Une porte dérobée (*backdoor*) laissée dans un logiciel par son développeur en est un autre exemple. On retrouve également des logiques malignes installées dans le système et menaçant la sécurité du système au sens logiciel, comme un cheval de Troie qui se fait passer pour un logiciel normal, mais qui exécute des tâches non voulues par l'utilisateur permettant d'exfiltrer des informations confidentielles. On peut aussi citer des logiciels comme les *rançongiciels* qui empêchent l'accès aux données de l'utilisateur.

1.4 Sécurité couche basse

Parmi les attaques présentées ci-dessus, nous nous sommes spécialement intéressés aux attaques par composant logiciel, visant des vulnérabilités microarchitecturales, ceci pour deux raisons principales : (1) ces attaques peuvent être particulièrement critiques, car potentiellement très difficiles à éliminer, et (2) des solutions innovantes sont encore à trouver pour ce type d'attaque. Dans cette catégorie, on retrouve notamment des attaques par canaux auxiliaires de temps et des attaques par exécution transitoire. Leur élimination peut même s'avérer très problématique due aux différents problèmes de performance, coût et faisabilité. Leur origine logicielle leur permet d'évoluer et de se propager plutôt rapidement, contaminant des systèmes ayant des optimisations vulnérables similaires. Les attaques par canaux auxiliaires de temps ont particulièrement attiré l'attention de la communauté scientifique depuis l'année 2014, où l'attaque Flush+Reload [Yarom 2014] a montré qu'il était possible de casser une implémentation de l'algorithme de chiffrement RSA en obtenant la clé secrète, sur un vrai système incluant de multiples niveaux de caches, sans la nécessité ni d'un accès physique, ni d'une collaboration avec la victime. Des attaques par exécution transitoire comme Spectre [Kocher 2019], annoncée publiquement au début de 2018, sont capables de faire fuiter des données sensibles dans la microarchitecture pour être lues par des canaux auxiliaires, impactant un nombre important de processeurs équipés des optimisations du type prédiction de branchement. Selon les données de Dimensions [Digital Science & Research Solutions, Inc. 2022], depuis 2014, plus de 100 articles de recherche consacrés aux canaux auxiliaires de temps et à l'exécution transitoire sont publiés chaque année.

Cette catégorie d'attaque consiste principalement, pour un logiciel malveillant, à collecter des traces de l'exécution d'un logiciel victime dans la microarchitecture. Le logiciel malveillant peut, à partir des états microarchitecturaux inférés, déduire des informations secrètes, telles qu'une clé de chiffrement, une séquence d'instructions ou de fonctions exécutées, etc. Le chemin de fuite de données est representé dans la figure 1.4. Dans cette section, nous discutons des grandes approches visant à

l'élimination des fuites d'informations secrètes et la tolérance aux attaques, et leurs contraintes respectives.

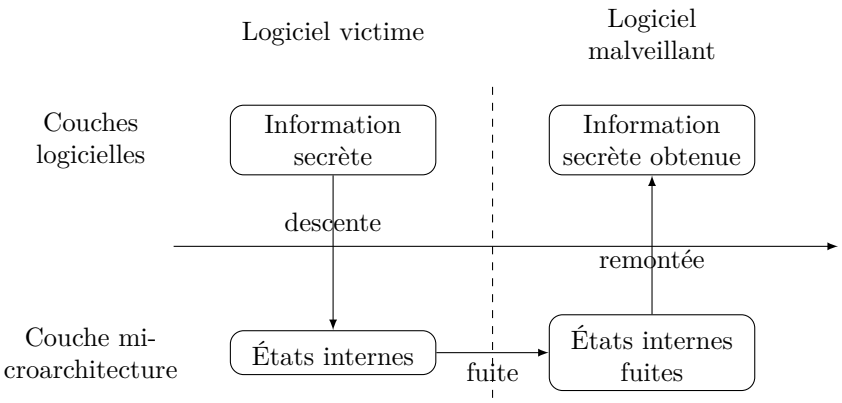


FIGURE 1.4 – Chemin de fuite de donnée entre deux logiciels à travers la microarchitecture.

1.4.1 Empêcher la fuite d'information

La fuite d'information dans la microarchitecture n'est possible que s'il y a au moins deux logiciels partageant une même ressource, et que les états internes de cette ressource, modifiés par le logiciel victime, persistent au moment de l'utilisation de cette ressource par un autre logiciel.

Étant donné que la suppression des états internes n'est pas réaliste pour des systèmes avec un minimum de complexité, on cherche plutôt à améliorer l'isolation des ressources. Deux types de partage de ressources existent : *le partage dans le temps*, c'est-à-dire que chaque logiciel utilise à son tour une ressource, comme un multiplicateur qui réalise un calcul d'abord pour un logiciel puis pour un autre ; *le partage dans l'espace*, lorsque plusieurs logiciels utilisent chacun une partie de cette ressource en même temps (par exemple, avec un processeur multithread, hyperthread, ou multicœur). Les protections pour les deux types de partage sont décrites ci-dessous, elles peuvent être utilisées conjointement si la ressource est partagée à la fois dans le temps et dans l'espace.

1. *Pour les ressources partagées dans le temps :* il est possible de réinitialiser des états internes après chaque utilisation de cette ressource. La réinitialisation empêche un logiciel d'obtenir des informations sur l'utilisation précédente de la ressource par un autre logiciel. Par exemple, Zhang et Reiter [Zhang 2013] proposent, au changement de machine virtuelle (en anglais *Virtual Machine*, VM), de nettoyer les états internes du cache L1 (partagé dans le temps) pour se protéger des attaques de temps sur le cache venant des autres VMs. L'inconvénient de la réinitialisation est qu'elle nécessite un temps de traitement et, dans certains cas, une adaptation de la structure matérielle pour supporter le nettoyage de certains états. L'impact sur la performance est élevé

si le nombre d'états internes à réinitialiser est grand, et si la fréquence de réinitialisation est élevée. L'exemple donné ci-dessus correspond à des réinitialisations peu fréquentes (au changement de VM et non au changement de logiciel) et peu profondes (elles concernent seulement le cache L1), la perte de performance est donc plus acceptable dans ce cas.

2. *Pour les ressources partagées dans l'espace :* il est possible de forcer une utilisation exclusive des ressources. Pour cela, on peut imaginer plusieurs solutions : (1) autoriser uniquement un logiciel à s'exécuter à un instant donné ; (2) dupliquer les ressources matérielles pour que chaque processus utilise sa propre copie de la ressource lorsqu'il s'exécute ; (3) réaliser un partitionnement de chaque ressource matérielle de façon à autoriser chaque processus à utiliser une partie seulement de chaque ressource ; (4) colorer les parties de ressources selon le logiciel qui les a utilisées, et ne pas autoriser l'utilisation des ressources colorées par un autre logiciel. Par exemple, Escouteloup et al. [Escouteloup 2021] proposent le cœur de processeur Salers pour se protéger des fuites d'information. Ce processeur gère l'exécution de deux tâches en parallèle, avec certaines ressources dupliquées, d'autres ressources partitionnées ou allouées spécifiquement à chaque tâche. Une réinitialisation des états est réalisée à chaque changement de contexte. L'inconvénient de ces méthodes est qu'elles souffrent soit d'un taux d'utilisation du matériel relativement faible, soit d'un besoin de modification importante pour chaque partie de ressource partagée pour inclure la couleur associée et la gestion de la coloration, soit d'une pénalisation dans la précision de la séparation. Plus ennuyeux dans beaucoup de cas, elles souffrent d'une perte de performance.

Dans tous les cas, les méthodes d'élimination de fuites ont un fort impact sur la performance et/ou le coût, ce qui n'est généralement pas acceptable.

1.4.2 Empêcher la descente d'information

Il s'agit ici de faire en sorte que les traces laissées dans la microarchitecture par le logiciel soient indépendantes de leurs données confidentielles (comme une clé de chiffrement). Il n'est pas possible de rendre tous les états microarchitecturaux identiques pour deux logiques différentes, mais des facteurs connus de fuites peuvent être considérés, notamment la différence de temps d'exécution, et l'utilisation de certaines ressources comme le cache.

1. *Rendre le temps d'exécution indépendant des secrets du logiciel.* L'approche *constant-time programming* cherche à assurer qu'un morceau de code critique est toujours exécuté dans le même temps sur la même machine. Ceci est réalisé avec l'utilisation des mêmes instructions et l'accès à une même séquence d'adresses mémoire, notamment en remplaçant des conditions par des calculs, indépendants des valeurs critiques. Par exemple, la librairie OpenSSL [OpenSSL 2022] a introduit une implémentation de l'exponentiation à temps constant dans la version 0.9.7h en 2005, puis a élargi aux autres

logiques dans les années suivantes. Brickell et al. [Brickell 2006] proposent d'accéder à l'avance et régulièrement aux lignes de caches potentiellement nécessaires pour laisser les mêmes états du cache dans le temps, afin de rendre les temps d'accès mémoire constants. Le *constant-time programming* est aujourd'hui couramment implémenté dans les librairies partagées de cryptographie. Cependant, l'effort humain nécessaire pour assurer la qualité de la réécriture du code est grand, et la perte de performance est élevée à cause de l'alignement du temps d'exécution au pire cas. Ces techniques sont ainsi beaucoup moins utilisées dans les autres logiciels et les parties moins critiques des librairies cryptographiques. En plus, son efficacité dépend de la microarchitecture : des choix d'implémentation non prévus existant dans la microarchitecture peuvent impacter le temps d'exécution prévu. Par exemple, certaines microarchitectures optimisent certaines instructions, et le temps d'exécution de ces instructions dépend des valeurs manipulées.

2. *Rendre le temps d'exécution indépendant des valeurs en entrée d'une instruction.* Par exemple, face aux attaques sur le chiffrement AES (*Advanced Encryption Standard*), Intel a proposé des instructions spécifiques AES, qui sont exécutées sur un composant spécifique, et qui permettent de réaliser les calculs dans un temps indépendant des valeurs en entrée [Gueron 2009]. Cependant, il est difficile d'assurer une durée d'exécution indépendant des données pour des fonctions arbitraires d'un logiciel. Avec la présence de nombreuses optimisations, une durée d'exécution constante signifie une durée d'exécution alignée avec le pire cas, et des modifications des matériels sont obligatoires. La perte de performance et le coût nécessaire sont potentiellement acceptables pour les applications critiques comme le cas de chiffrement, mais sont en général inacceptables pour les autres opérations non sensibles.
3. *Rendre les accès mémoire indépendants des secrets.* La technique *Oblivious RAM* cherche à rendre les adresses mémoire accédées indépendantes du logiciel exécuté [Goldreich 1996, Stefanov 2018]. En déplaçant fréquemment les données dans la mémoire, deux accès sur la même donnée ne seront pas réalisées à la même adresse ; le moment et l'ordre des accès peuvent être masqués en mettant en place une file interne dans un composant matériel dédié. L'*Oblivious RAM* est utilisée en cryptographie en complément du chiffrement des données. Ainsi, un logiciel malveillant ne peut pas déduire le comportement d'un logiciel légitime par la fréquence d'apparition et les valeurs des adresses ou des données manipulées. Cependant, le coût de cette technique est assez élevé à cause du déplacement des données et du stockage interne nécessaires. De plus, dans le cas d'une *Oblivious RAM* ou d'un cache partagé, des informations de temps peuvent toujours fuiter [Bao 2017].

Nous pouvons constater que le coût des techniques visant à prévenir la descente de l'information est toujours élevé. Cependant, comme il est possible d'appliquer ces techniques à une seule partie du logiciel, et qu'il n'y a également un coût supplémentaire que dans cette partie, elles restent intéressantes pour du code logiciel

critique comme une implémentation d'un algorithme cryptographique par exemple.

1.4.3 Empêcher la remontée d'information

Comme la fuite d'information est particulièrement difficile à éliminer totalement, et comme la descente d'information est impossible à empêcher systématiquement, certaines méthodes cherchent à éliminer la remontée de ces fuites d'information de la couche microarchitecture aux couches logicielles.

Les couches logicielles ne "voient" les états internes microarchitecturaux qu'à travers l'utilisation des instructions assembleur, aussi la récupération des états internes de la microarchitecture n'est pas triviale. Certains états internes sont volontairement et directement accessibles pour les logiciels, pour les aider à mieux optimiser l'utilisation des ressources matérielles, tels que le taux de mémoire occupée, le nombre d'instructions exécutées dans une période de temps donné. D'autres ressources peuvent être déduites avec ces informations accessibles. Une information couramment utilisée pour déduire des états internes est la *durée d'exécution d'instructions*. Comme les états internes servent beaucoup à l'amélioration de la performance d'exécution, la valeur de ces états peut impacter la durée d'exécution d'instructions. Plusieurs techniques de protection existent dans cette catégorie.

1. *Empêcher la récupération d'information.* La solution la plus évidente consiste à interdire l'accès aux informations de la microarchitecture par les logiciels, et spécialement celles qui permettent de déduire les états internes. Par exemple, Aviram et al. [Aviram 2010] proposent un système d'exploitation de cloud qui retourne des résultats de calculs uniquement liés aux entrées, et non aux horloges internes ou autres comportements non déterministes du système, pour se protéger contre la fuite d'information dans le cloud. Un moyen moins strict est de limiter l'accès à ces informations aux logiciels en lesquels on a pas confiance. Par exemple, dans la définition d'ISA RISC-V [Waterman 2021], nous pouvons constater que les accès aux compteurs matériels de performance sont soumis à l'obtention de privilèges plus élevés accordés par le noyau, qui peut potentiellement interdire aux logiciels utilisateurs de lire les valeurs de ces registres. Cependant, ces informations sont aussi utilisées par des logiciels normaux pour des raisons légitimes, et certaines informations comme les durées d'exécution précises pourraient être obtenues d'une autre manière, par exemple, à travers le réseau.
2. *Réduire la précision des informations.* Certaines solutions cherchent à réduire la précision des informations obtenues par l'attaquant, afin de minimiser la connaissance des états internes inférés. Par exemple, Martin et al. [Martin 2012] proposent de modifier les compteurs matériels de performance des processeurs et d'y ajouter des bruits aléatoires. Ils proposent également des mécanismes permettant de détecter l'utilisation d'un compteur logiciel partagé qui peut servir à réaliser un compteur de temps. Cependant, la réduction de la précision des informations ne permet pas d'empêcher la

réussite des attaques, à cause de la difficulté à générer un bruit parfait. Elle permet seulement d'augmenter l'effort nécessaire pour enlever les bruits.

Globalement, nous pouvons donc constater que l'élimination de la remontée d'information est soit limitée dans sa couverture des cas, soit limitée dans son efficacité, et qu'elle introduit une forte pénalité de performances et de coût.

1.4.4 Détecter la présence de malveillance

Les méthodes d'éliminations ou de limitations de fuites présentées ci-dessus ne sont pas parfaites et souffrent chacune de limitations importantes. Aussi, il est fondamental de penser à des méthodes complémentaires pour améliorer la sécurité des systèmes, en particulier vis-à-vis des attaques qui ciblent les couches basses des systèmes. La détection d'attaques fait partie de ces méthodes complémentaires. On peut distinguer deux grandes familles de détection :

1. *La détection statique* vise à identifier le caractère malveillant d'un logiciel par analyse statique de son code binaire ou de son code source. Cette détection se fait par exemple sur des plates-formes de diffusion d'applications ou par un logiciel antivirus. Le principe est d'analyser les fichiers du logiciel afin d'identifier des potentielles caractéristiques malveillantes. Par exemple, Irazoqui et al. [Irazoqui 2018] proposent de désassembler les fichiers binaires, et de trouver des logiques avec des propriétés similaires à certaines attaques visant des vulnérabilités microarchitecturales. Un des avantages de l'analyse statique est qu'elle n'est pas réalisée durant la phase d'exécution des logiciels et que par conséquent, elle n'impacte en aucune façon cette exécution en terme de performance. Cependant, l'analyse statique n'est pas efficace contre des logiciels qui ne contiennent pas tous les contenus au moment de l'analyse (téléchargement et modification du contenu pendant l'exécution), et des logiciels qui utilisent des techniques d'obfuscation pour empêcher le bon fonctionnement du désassemblage.
2. *La détection dynamique* vise à identifier le caractère malveillant d'un logiciel, par la détection des comportements anormaux lors de son exécution, ou par des interactions suspectes avec d'autres logiciels. L'intérêt de l'analyse dynamique est qu'elle ne souffre pas des problèmes de l'analyse statique vis-à-vis de l'obfuscation de code et du téléchargement de code en cours d'exécution. Par ailleurs, l'analyse dynamique permet d'activer des méthodes d'éliminations de fuites lourdes uniquement en présence d'anomalies, ce qui permet de réduire l'impact en terme de performances de ces méthodes. Les principales approches d'analyse dynamique proposées dans la littérature concernent des solutions logicielles, qui bénéficient des informations disponibles de la microarchitecture telles que les compteurs matériels de performance [Akram 2020] ; d'autres approches proposent une partie matérielle en complément afin de bénéficier d'informations microarchitecturales supplémentaires et ainsi améliorer la détection [Chen 2014]. Cependant, les

solutions purement logicielles sont limitées dans le type et la quantité d'informations disponibles qui ne sont pas toujours bien adaptées à la détection d'attaques, et sont également impactantes en terme de performances. Les solutions matérielles proposées, quant à elles, disposent d'informations plus précises pour réaliser de la détection, mais manquent souvent de flexibilité pour s'adapter à l'évolution des attaques.

En résumé de cette section, les méthodes de détection des attaques, en particulier la détection dynamique, nous semblent fondamentales pour compléter des méthodes d'élimination de ces attaques. Cependant, les solutions proposées actuellement ne permettent pas de satisfaire les besoins en faible impact sur la performance, et sont encore à améliorer concernant l'efficacité et l'adaptabilité aux nouvelles attaques.

1.5 Conclusion et objectif de la thèse

Dans ce premier chapitre, nous avons tout d'abord présenté un modèle d'abstraction en couches des systèmes informatiques, puis la terminologie de la sécurité informatique et en particulier la sécurité-immunité. Nous avons ensuite établi une classification des attaques menaçant les systèmes informatiques en fonction de leur origine et des vulnérabilités visées. Nous avons plus spécifiquement étudié les attaques d'origine logicielle qui exploitent des vulnérabilités microarchitecturales, qui concernent des attaques souvent subtiles, difficiles à détecter et à éliminer, et pour lesquelles des approches innovantes restent encore à proposer. Enfin, nous avons présenté des moyens de protections courants pour ces attaques et les limitations respectives de ces protections.

Une des leçons principales que l'on peut tirer de ce chapitre est que les méthodes pour détecter dynamiquement les attaques sont intéressantes pour compléter les méthodes visant à empêcher les fuites d'information. Cependant, les solutions existantes ne permettent pas de satisfaire en même temps les besoins de performance de la détection et de flexibilité face à la rapide évolution des attaques. Nous avons également constaté qu'il nous manque des connaissances sur l'impact de l'exécution des logiciels sur la microarchitecture. Ces connaissances pourront être utiles pour concevoir des composants matériels moins vulnérables et des mécanismes de détection des attaques plus efficaces.

Face à ce constat, l'objectif de cette thèse est de proposer un framework de détection dynamique des attaques basée sur l'analyse des signaux microarchitecturaux, qui a un impact relativement faible sur les performances et qui est suffisamment flexible au cours de son cycle de vie. Ce framework ne doit pas nécessiter de modifications du logiciel de l'utilisateur afin d'être compatible avec la grande quantité de logiciels existants dans les systèmes informatiques à usage général. Compte tenu du nombre et de l'évolution constante des microarchitectures d'ordinateurs, ce framework doit être ouvert, facile à appliquer et personnalisable, afin que les concepteurs de systèmes puissent l'adapter à leurs systèmes, et que la communauté puisse l'utiliser et l'améliorer.

Plus précisément, notre première et principale contribution est la proposition d'une architecture de détection dynamique des attaques basée sur l'observation et l'analyse des signaux microarchitecturaux, publiée dans [Mao 2020]. Le principe de cette architecture est de placer des sondes de collecte des signaux microarchitecturaux internes aux composants cibles, destinés à servir d'entrées à des algorithmes de détection d'attaques. L'objectif est donc de pouvoir détecter des attaques subtiles qui visent à exploiter la microarchitecture et qui donc laissent des traces au niveau de la microarchitecture. Le traitement et l'analyse des signaux collectés se fait à la fois dans du matériel reconfigurable et dans du logiciel. Le traitement matériel permet de garantir une très bonne performance, la reconfigurabilité permet d'offrir une flexibilité dans l'analyse, et le logiciel quant à lui peut réaliser une analyse plus globale puisqu'il a une bonne connaissance de l'environnement d'exécution. Cette architecture est conçue pour prendre en compte les problèmes de différence de fréquences entre une partie matérielle reconfigurable (pour la détection, typiquement un FPGA) et une partie matérielle non reconfigurable (typiquement le composant cible). Nous proposons également une méthodologie, publiée dans [Mao 2022], et des outils associés qui permettent (1) de sélectionner les signaux internes appropriés à observer pour une classe d'attaque spécifique et (2) de concevoir des algorithmes de détection adaptés pour cette classe à l'aide de ces signaux. Cette contribution fait l'objet du chapitre 3.

Notre seconde contribution est la réalisation d'un prototype basé sur un processeur RISC-V Rocket [Asanović 2016], équipé d'un noyau Linux, sur la plateforme FPGA Xilinx ML605. Cette implémentation fait l'objet du chapitre 4.

Enfin, notre dernière contribution est la mise en place de deux expérimentations dans lesquelles nous montrons qu'il est possible de concevoir avec succès des mécanismes de détection pour deux classes d'attaques qui exploitent des mécanismes différents : une classe d'attaques qui exploitent la microarchitecture sans altérer le flux de contrôle d'un programme (l'attaque par canal auxiliaire de temps passé par le cache) ; une classe qui altère le flux de contrôle (l'attaque ROP). Ces travaux, qui font l'objet du chapitre 5, permettent de montrer la généricité de notre approche.

Dans le chapitre suivant, nous présentons un état de l'art de la détection dynamique des attaques basées sur des analyses des signaux avant de présenter dans les chapitres suivants le détail des trois contributions.

Chapitre 2

Contexte technique et état de l'art

Sommaire

2.1 Détection au niveau architecture (non reconfigurable) . . . 30
2.1.1 Contexte technique . . . 30
2.1.2 Détection basée sur des systèmes commerciaux . . . 30
2.1.3 Détection basée sur des systèmes non commerciaux . . . 31
2.1.4 Conclusion . . . 32
2.2 Détection au niveau microarchitecture (non reconfigurable) 33
2.2.1 Contexte technique . . . 33
2.2.2 Compteurs matériels de performance . . . 34
2.2.3 Supervision de signaux . . . 37
2.2.4 Conclusion . . . 39
2.3 Matériel reconfigurable pour la détection . . . 40
2.3.1 Contexte technique . . . 41
2.3.2 État de l'art . . . 43
2.3.3 Conclusion . . . 43
2.4 Conclusion . . . 44

Comme notre travail porte sur la détection dynamique des attaques basée sur l'analyse des signaux, nous présentons dans ce chapitre l'état de l'art des différentes méthodes de détection dans ce même contexte. Nous présentons d'abord les méthodes qui utilisent du matériel peu ou pas reconfigurable (très peu d'ajustements possibles sur le circuit logique), que nous divisons en deux catégories : celles qui se concentrent uniquement sur les signaux reflétant le comportement des couches architecturales, et celles qui considèrent également les signaux reflétant le comportement de la microarchitecture. Nous discutons ensuite de la détection des attaques à l'aide de matériel reconfigurable, qui permet de modifier la logique de détection implémentée pendant le cycle de vie afin de modifier les différents détecteurs utilisés ou d'adapter la logique à l'évolution des attaques.

2.1 Détection au niveau architecture (non reconfigurable)

Dans cette section, nous présentons des méthodes de détection dynamique des attaques basées sur l'analyse des signaux architecturaux, à l'aide de matériel non reconfigurable pour implémenter les logiques de détection.

2.1.1 Contexte technique

Pendant l'exécution du logiciel, les informations architecturales, incluant notamment l'instruction exécutée et l'adresse mémoire accédée, sont traçables au niveau matériel. Comme elles sont uniquement collectées à chaque instruction, nous considérons la granularité d'une instruction.

Les traces architecturales collectées, si elles ne sont pas directement envoyées et traitées par une logique de détection, sont stockées temporairement dans une mémoire tampon. Pour augmenter la quantité d'informations présente dans la mémoire tampon de taille limité, et réduire l'utilisation de la bande passante pour la récupération des traces, la trace est souvent compressée et réduite au traçage des instructions clés et/ou des plages d'adresses mémoire accédées.

Dans les processeurs commerciaux, le traçage d'instructions existe principalement pour des raisons de débogage. Par exemple, l'Arm CoreSight [Arm 2017] est une technologie qui permet de surveiller le processeur et de collecter des informations à l'intérieur d'un SoC (*System-on-a-Chip*) Arm. Son composant Trace Macrocell [Arm 2011] peut être programmé pour collecter des traces d'instructions incluant les instructions clés, telles que les branchements, les exceptions et la synchronisation. Ces traces sont accessibles via un bus système (partagé avec d'autres composants) ou via le port de débogage. À l'aide des informations accessibles par le bus, l'analyse de la trace peut se faire dans un composant matériel indépendant du processeur pendant l'exécution. Cependant, limitée par la structure du bus et le fait qu'il soit partagé, la trace est compressée et réduite, il y a donc besoin de traitements supplémentaires pour reconstituer la trace complète. Même si le traçage existe sur des processeurs commerciaux, ces options de débogage ne sont pas forcément accessibles dans le produit final. Nous pouvons trouver, cependant, des SoCs avec des processeurs commerciaux, où le bus recevant les traces d'instructions est connecté à un composant de traitement, comme un FPGA dans le cas de SoC Xilinx Zynq par exemple [Xilinx 2022].

2.1.2 Détection basée sur des systèmes commerciaux

Quelques travaux utilisant le SoC Xilinx Zynq équipé d'un processeur Arm et d'un FPGA sont proposés dans la littérature.

Lee et al. [Lee 2015] proposent de détecter les attaques par réutilisation de code sur le SoC Xilinx Zynq. Le module de traçage est configuré pour récupérer les adresses des branchements rencontrés, l'exécution ou non de ces branchements, et

dans le cas des branchements indirects, l'adresse de destination du branchement. La trace est d'abord décompressée dans le FPGA, puis, l'adresse de destination des sauts ou de retour des fonctions est vérifiée vis-à-vis d'une base de données des destinations de sauts légitimes stockée dans la mémoire principale (générée au préalable par analyse statique du binaire à protéger) ou vis-à-vis des adresses de retours stockées dans une *shadow stack* (renseignée lors des appels de fonctions). Cette démarche permet de détecter une signature typique de réutilisation de code, dans laquelle les adresses de retour sont manipulées par l'attaquant, et sont ainsi différentes des adresses de retour stockées au moment de l'appel de fonction.

Abdul et al. [Wahab 2018] proposent une approche permettant de réaliser du traçage de flot d'informations (en anglais *Dynamic Information Flow Tracking*, DIFT) basé sur des étiquettes dans le SoC Xilinx Zynq. Le DIFT basé sur des étiquettes consiste à associer des étiquettes aux données, les propager et les vérifier pendant l'exécution, et permet ainsi de détecter plusieurs classes d'attaques du type corruption de mémoire ou algorithmiques, telles que les débordements de tampon, les fuites d'informations ou les injections SQL. La trace utilisée dans ce travail contient l'adresse des branchements et des sauts. La trace est d'abord décompressée à l'aide des informations connues du logiciel à protéger (générées au préalable par analyse statique et instrumentation du code), puis les étiquettes correspondantes sont calculées et vérifiées par un coprocesseur conçu par les auteurs.

Dans les deux cas, même si un FPGA (matériel reconfigurable) a été utilisé pour implémenter les logiques de détection, les auteurs l'utilisent seulement comme un outil de prototypage des logiques fixes, la modification de ces logiques dans le cycle de vie du matériel n'est pas discutée.

2.1.3 Détection basée sur des systèmes non commerciaux

En dehors des travaux utilisant les systèmes commerciaux, des recherches académiques proposent des architectures pour la détection d'attaques basées sur les traces architecturales.

Chen et al. [Chen 2006] proposent de collecter, au moment de l'exécution d'une instruction, son pointeur d'instruction, type, identifiant des opérandes, et l'adresse mémoire accédée. Ils compressent la trace et placent cette trace dans une mémoire tampon, puis ils effectuent l'analyse sur un autre cœur général à l'aide d'un programme de détection. La détection implémentée s'intéresse aux attaques de corruption de mémoire, telle que la détection des accès à une mémoire non allouée et la détection des entrées d'utilisateurs modifiant les données de contrôle.

L'architecture Raksha [Dalton 2007] propose une approche basée DIFT reposant sur l'intégration d'étiquettes dans le processeur. La taille des registres internes du pipeline est étendue pour inclure une partie accueillant une étiquette, modifiée en parallèle des données contenues dans le registre. Le pipeline est également modifié pour ajouter des mécanismes de propagation et de vérification des étiquettes. De même, le stockage des données en mémoire principale est aussi adapté pour y stocker les étiquettes, à proximité des données.

Harmoni [Deng 2012] propose l'implémentation d'une approche basée DIFT ainsi que d'autres vérifications basées sur les étiquettes dans un coprocesseur spécialement conçu. La trace collectée inclut des instructions exécutées, des registres source et destination, ainsi que l'étiquette associée aux données si elle existe. Le matériel utilisé est statique, car les différentes vérifications (détection des situations anormales) sont intégrées dans le même coprocesseur, en suivant différents chemins d'exécutions pour la manipulation des étiquettes.

PHMon [Delshadtehrani 2020] propose de collecter l'instruction exécutée, le pointeur d'instruction, la destination de branchement et l'adresse mémoire accédée. Il effectue l'analyse de trace sur un coprocesseur, composé d'un module de filtrage de trace et d'un module réalisant des calculs, pour interagir avec la mémoire ou notifier le processeur par interruption. Ce travail s'intéresse à la détection des attaques du type corruption de mémoire, et peut agir par exemple comme une *shadow stack* afin de détecter les attaques par réutilisation de code, ou détecter les accès imprévus à une plage d'adresses contenant une information critique.

2.1.4 Conclusion

Le tableau 2.1 compare les méthodes de détection basées sur les traces architecturales, où le matériel implémentant les logiques de détection n'est pas reconfigurable. Comme les traces représentent le comportement architectural des logiciels exécutés, les méthodes de détection associées utilisent ces traces pour détecter des attaques visant des vulnérabilités architecturales, typiquement les attaques par corruption de mémoire. À notre connaissance, il n'existe pas de détection d'attaques microarchitecturales (les attaques visant des vulnérabilités microarchitecturales) basée uniquement sur la trace d'instructions. Les algorithmes utilisés dans ces travaux se reposent directement sur des propriétés des attaques, qui, pour les attaques de corruption de mémoire, sont relativement simples, ne nécessitant pas l'utilisation de méthodes d'apprentissage automatique. Le traitement des traces architecturales nécessite toujours un support matériel, que ce soit des logiques matérielles intégrées, un composant indépendant, ou un cœur général dédié, afin de bénéficier de haute

Table 2.1 – Comparaison des méthodes de détection dynamique des attaques basées sur les traces architecturales.

	Attaques Visées	**Algorithme de Détection**	
		Type	**Position**
[Lee 2015]	Réutilisation de code	PM	Composant indépendant
[Wahab 2018]	Corruption de mémoire	PM	Composant indépendant
[Chen 2006]	Corruption de mémoire	PM	Cœur général dédié
[Dalton 2007]	Corruption de mémoire	PM	Intégré dans le pipeline
[Deng 2012]	Corruption de mémoire	PM	Composant indépendant
[Delshadtehrani 2020]	Corruption de mémoire	PM	Composant indépendant

Abréviation : PM : Pattern Matching.

performance du traitement du matériel pour atteindre la vitesse nécessaire pour la détection dynamique des attaques.

2.2 Détection au niveau microarchitecture (non reconfigurable)

Dans cette section, nous présentons des méthodes de détection dynamique des attaques basée sur l'analyse des signaux microarchitecturaux, pour lesquelles le matériel implémentant les logiques de détection est non reconfigurable. Nous clarifions tout d'abord la notion de signal microarchitectural. Puis, nous présentons les détections basées sur les compteurs matériels de performance (en anglais *Hardware Performance Counters*, HPCs), qui sont présents sur la plupart des processeurs commerciaux. Nous finissons par présenter les autres stratégies de détection, qui ne se basent pas sur les HPCs.

2.2.1 Contexte technique

Clarifions tout d'abord la notion de signal microarchitectural que nous utilisons dans ce manuscrit. Nous parlons de signal logique qui est une abstraction de l'effet physique dont les valeurs sont analogues et continues. Un signal logique dans notre contexte possède deux niveaux logiques : l'état 0, qui est une logique basse, correspond à un niveau de tension basse ; l'état 1, qui est une logique haute, correspond à un niveau de tension haute.

La figure 2.1 présente l'évolution des signaux logiques sur une interface de type "ready-valid". Quatre signaux sont présentés dans la figure, dont trois signaux à 1 bit (`Clock`, `Valid`, `Ready`) portant soit l'état 0 (signal bas) soit l'état 1 (signal haut), et un signal à 4 bits (`Data`) dont la valeur est décrite en hexadécimal sur la figure, pour représenter une donnée codée sur plusieurs bits. Du point de vue architectural — par exemple, si cette interface est utilisée pour échanger l'adresse mémoire d'une valeur à lire — seules sont visibles les trois adresses A, B et C, qui sont lues en séquence (les valeurs de `Data` lorsque `Ready` et `Valid` sont à 1). Les signaux de gestion (`Clock`, `Valid`, `Ready`) ainsi que l'évolution des signaux font partie quant à eux des informations microarchitecturales.

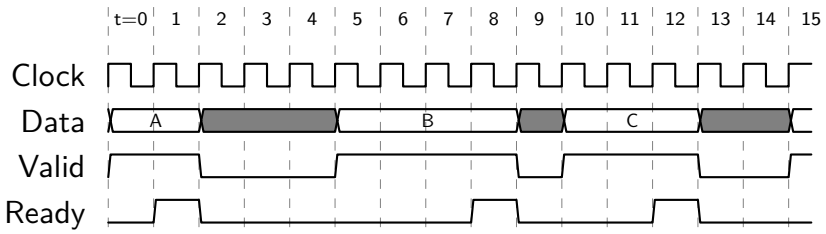


FIGURE 2.1 – Exemple d'un diagramme de temps des signaux logiques sur une interface "ready-valid".

Le signal d'horloge (`Clock`) est particulier, car les changements des autres signaux sont synchronisés sur le front montant de l'horloge. On appelle cycle d'horloge, ou période de l'horloge, le temps séparant un front montant du prochain front montant. Sur la figure, les instants `t=0` jusqu'à `15` correspondent aux 16 cycles d'horloge au total. Quand on parle d'une horloge à fréquence de 50 MHz (comme l'horloge principale de notre prototype présenté dans le chapitre 4), sa période est de 20 ns. Plus la fréquence est faible, plus la période est longue, et plus les signaux changent lentement.

L'interface "ready-valid" est un protocole de communication entre deux parties : un émetteur, qui contrôle les signaux `Valid` et `Data`, où le signal `Valid` haut indique que les données dans `Data` sont prêtes (comme dans les cycles 0 à 1, 5 à 8 et 10 à 12 sur la figure) ; et un récepteur, qui contrôle le signal `Ready`, indiquant que le récepteur est prêt (signal haut) ou non (signal bas) à recevoir des données. Les données (`Data`) transférées sur cette interface, peuvent être par exemple une adresse mémoire ou une instruction. Quand les signaux `Valid` et `Ready` sont tous les deux vrais, comme pendant les cycles 1, 8 et 12 sur la figure, un échange de données est effectué.

Ainsi, dans ce simple exemple, la supervision des signaux microarchitecturaux consiste à collecter la valeur à chaque cycle des quatre signaux microarchitecturaux. Les HPCs ne réalisent pas de la supervision des signaux à proprement parler mais permettant de stocker le nombre d'occurrences de certains évènements. Par exemple, compter le nombre d'occurrences à 1 du signal `Clock` permet de connaître le nombre de cycles d'horloge passés pendant l'exécution ; compter le nombre d'occurrences simultanées à 1 du signal `Valid` et du signal `Ready` permet de connaître le nombre de transactions des données réussies sur cette interface.

2.2.2 Compteurs matériels de performance

2.2.2.1 Contexte technique

Les HPCs sont des registres spéciaux situés à l'intérieur d'un processeur. Ces compteurs stockent le nombre d'occurrences d'évènements microarchitecturaux, tels que le nombre de défauts de cache et de succès de cache (en anglais *cache miss* et *cache hit*), le nombre d'instructions exécutées, le nombre d'accès au bus, le nombre d'erreurs de prédiction de branchement, etc. Le composant qui rassemble ces compteurs est généralement appelé *Performance Monitor Unit* (PMU).

Dans la plupart des processeurs modernes, le nombre d'évènements pouvant être collectés varie entre vingt et quelques centaines, mais seul un ensemble limité peut être sélectionné à la fois, environ 2 à 8 pour de nombreux processeurs. Par exemple, le processeur Arm Cortex-A53 implémentant le jeu d'instructions ARMv8-A, utilisé dans des systèmes informatiques tels qu'une Raspberry Pi 3, possède 6 registres de 32 bits que l'on peut configurer pour surveiller une sélection de 64 évènements [Arm 2018]. Pour les processeurs plus orientés vers les systèmes embarqués, le processeur ARM Cortex-M3 implémentant le jeu d'instructions

ARMv7-M, utilisé dans les produits comme Arduino Duo, ne dispose pas de compteurs [Arm 2010], il faut attendre son successeur Cortex-M55 implémentant le jeu d'instructions ARMv8.1, annoncé en février 2020, pour voir apparaître 8 compteurs de 16 bits sur 103 évènements différents [Arm 2020]. La machine hôte utilisée dans notre prototype est équipée d'un processeur Intel Core i5-8205U implémentant le jeu d'instruction x86-64 de microarchitecture Kaby Lake, qui compte 8 compteurs de 48 bits programmables par cœur, choisis parmi 208 évènements différents [Intel 2016].

D'autres composants matériels, en dehors du processeur, peuvent également posséder leurs HPCs. Tout comme pour le processeur, beaucoup d'évènements sont potentiellement collectables, mais peu d'évènements peuvent être collectés en même temps. Les processeurs graphiques de NVIDIA [Nvidia 2022] en sont un exemple. Ces compteurs reflètent l'empreinte microarchitecturale du logiciel sur un composant particulier.

Les HPCs qui existent aujourd'hui dans les systèmes sont principalement conçus pour des raisons d'optimisation, par exemple pour aider les développeurs à minimiser le nombre de défauts de cache et le nombre d'erreurs de prédiction de branchement afin de réduire le temps d'exécution total du logiciel. Le nombre de compteurs assez limité permet de réduire le nombre de logiques matérielles. Cette réduction du nombre de compteurs ne donne qu'une vision partielle de l'état interne du système, à partir de laquelle la détection dynamique est réalisée. La configuration de ces compteurs et l'accès aux valeurs peuvent se faire facilement par un logiciel, avec des instructions spécifiques de lecture des registres internes (`rdmsr` et `wrmsr` pour l'ISA x86, `mrc` et `mcr` pour l'ISA ARM, `csrr` et `csrw` pour l'ISA RISC-V).

2.2.2.2 Détection des attaques microarchitecturales

Bien que les HPCs sur les systèmes commerciaux n'aient pas été spécialement conçus pour la sécurité, plusieurs travaux de recherche les utilisent pour la détection dynamique des attaques visant la microarchitecture, et certains travaux proposent d'ajouter des HPCs orientés sécurité. Nous présentons ci-dessous quelques travaux représentatifs.

CC-Hunter [Chen 2014] propose d'introduire de nouveaux compteurs matériels afin de mieux détecter les attaques par canaux cachés temporels où deux processus contrôlés par l'attaquant s'échangent des messages non autorisés par le système, en mesurant le temps d'accès à certaines ressources partagées, comme le bus ou le cache. Pour cela, les auteurs ajoutent une logique de surveillance des évènements dans chaque composant matériel partagé susceptible de servir de support à ces canaux auxiliaires, afin de détecter l'utilisation suspecte par un processus de certaines ressources spécifiques déjà utilisées par un autre processus. Le nombre de compteurs matériels est limité à deux registres de 32 bits, et une mémoire tampon matérielle, en charge de mémoriser la valeur des compteurs durant 128 intervalles de temps, permet au logiciel d'accéder à ces valeurs et de réaliser les analyses. La détection est basée sur une signature d'utilisation de ressources pendant l'attaque pour le codage de chaque bit échangé sur le canal caché.

HexPADS [Payer 2016] propose de collecter des HPCs (nombre d'instructions exécutées, nombre de défauts de cache L3 et nombre d'accès au cache L3) et des métriques du noyau (nombre de fautes de page) pour la détection des attaques par canal auxiliaire de temps ainsi que l'attaque rowhammer. L'approche consiste à observer à la fois le nombre et le taux de défauts de cache [1] et le taux de fautes de page. Si le taux de défauts de cache est élevé alors que le taux de fautes de page est faible, alors, il est probable qu'une éviction de cache, caractéristique des attaques par cache, soit en train de s'exécuter. L'approche détermine ainsi des seuils concernant ces compteurs afin de détecter ces attaques.

Chiappetta et al. [Chiappetta 2016] proposent de collecter des HPCs (nombre d'instructions exécutées, nombre de cycles d'horloge du processeur, nombre d'accès au cache L2, nombre de défauts de cache L3 et nombre d'accès au cache L3) pour détecter des attaques temporelles sur les caches. Ils proposent trois approches différentes : la première approche calcule la corrélation entre le nombre d'accès au cache L3 de la victime et celui de l'attaquant, elle considère qu'une corrélation élevée révèle une attaque (la principale limitation étant que la corrélation doit être réalisée sur tous les processus deux à deux) ; la deuxième et la troisième approches utilisent de l'apprentissage automatique non supervisé (*Gaussian distribution*) et supervisé (*neural network*) respectivement, afin de séparer les logiciels légitimes et malveillants.

Li et Gaudiot [Li 2018a] proposent de collecter les HPCs (nombre de défauts de cache, nombre d'accès au cache, nombre d'instructions du type branchement exécutées et nombre d'erreurs de prédictions de branchement) pour la détection des attaques par exécution transitoire liée à la prédiction de branchement (Spectre). Ils utilisent différentes techniques d'apprentissage automatique supervisé (*logistic regression*, *support vector machine*, *atificial neural network*) pour construire des classifieurs, afin de distinguer les attaques et les logiciels légitimes.

SpyDetector [Kulah 2019] propose de collecter des HPCs (nombre de défauts de cache et nombre d'accès au cache L1 et L3) et de les associer aux quotas d'exécution attribués par l'ordonnanceur dans le noyau, afin de détecter l'utilisation concurrente du cache ou d'une autre ressource partagée qui est créée entre un processus malveillant et un processus victime pendant une attaque par canal auxiliaire de temps. Ce travail propose d'utiliser une approche comportementale basée sur l'identification des caractéristiques d'un logiciel légitime par apprentissage automatique non supervisé (*k-mean clustering*). Les clusters ainsi obtenus correspondent aux activités légitimes, et tout élément ne pouvant être intégré dans un des clusters est considéré comme une anomalie (et donc une potentielle attaque).

WHISPER [Mushtaq 2020] propose de collecter les HPCs (nombre de défauts de cache de données L1 et L3, nombre d'accès au cache L3 et nombre de cycles d'horloge du processeur) pour construire de manière supervisée différents classifieurs (*decision tree*, *random forest*, *support vector machine*) et les combiner à l'aide d'un vote majoritaire. Avec la composition de ces classifieurs relativement légers, les

1. qui représente le nombre de défauts de cache sur le nombre total d'accès au cache

auteurs ont pu obtenir un équilibre satisfaisant entre précision, performance et capacité à arrêter l'attaque avant sa finalisation. Le détecteur ainsi obtenu détecte des attaques par canal auxiliaire de cache, et avec l'ajout ou non des évènements supplémentaires (nombre d'instructions du type branchement exécutées, nombre d'erreurs de prédictions de branchement et nombre de fautes de page), il est capable de détecter des attaques par exécution transitoire basées sur les attaques par canal auxiliaire de cache.

2.2.2.3 Détection des attaques architecturales et algorithmiques

Les informations microachitecturales peuvent être pertinentes pour détecter différents types d'attaques, même si celles-ci ne ciblent pas particulièrement des vulnérabilités microarchitecturales. Nous en donnons ici quelques exemples.

Demme et al. [Demme 2013] proposent de collecter les HPCs (la liste complète n'est pas fournie, mais elle inclut le nombre d'accès au cache L1 et le nombre de micro-opérations arithmétiques exécutées) et de détecter certains logiciels malveillants sur Android et des *rootkits* Linux. Ils utilisent pour cela des classifieurs (*K-nearest neighbors*, *decision tree*, *random forest* et *artificial neural network*), afin de distinguer les attaques et les logiciels légitimes. Les auteurs mentionnent le besoin d'utilisation d'un composant matériel (que ce soit un cœur dédié, un FPGA ou un ASIC) pour implémenter les logiques de détection afin d'atteindre un délai de traitement adapté à la détection dynamique.

Sigdrop [Wang 2016] propose de collecter les HPCs (nombre d'instructions exécutées, nombre d'instructions de type retour de fonction et nombre d'instructions de type retour de fonction mal prédit) pour détecter des attaques ROP. La détection se fait par signature, basée sur le fait que les séquences d'instructions dans une attaque ROP sont courtes et que les retours de fonction ne reviennent pas aux adresses prévues. L'exécution de la détection s'effectue en logiciel, l'alerte est levée en comparant les HPCs collectés à des seuils.

PMUe [Li 2018b] propose également la détection d'attaques ROP basées sur des HPCs (nombre d'instructions exécutées et nombre de branchements indirects mal prédits), avec une logique de détection matérielle intégrée dans le PMU. Cette logique de détection permet de reconnaître les attaques ROP par signature similaire à celle utilisée par Sigdrop : une chaine d'instructions courtes séparées par des évènements de mauvaise prédiction de branchement.

2.2.3 Supervision de signaux

2.2.3.1 Contexte technique

Les travaux regroupés dans cette section concernent la détection d'attaque par analyse direct des signaux microarchitecturaux (en dehors des HPCs). Ces signaux reflètent l'état architectural et l'état microarchitectural du système, collectés à granularité du cycle d'horloge et filtrés si besoin.

La supervision des signaux arbitraires à granularité du cycle d'horloge est un outil courant pour le débogage dans la conception d'un composant matériel. Par exemple, Xilinx ChipScope [Xilinx 2012] ou Intel SignalTap [Intel 2020] peuvent être utilisés pendant la phase de prototypage d'un composant matériel sur FPGA. Ils consistent à placer des sondes directement sur les signaux microarchitecturaux choisis, échantillonner les valeurs des signaux à chaque cycle d'horloge, stocker les valeurs dans une mémoire tampon, puis envoyer le contenu de la mémoire tampon via le port de débogage lorsqu'une condition de déclenchement est atteinte. Le traitement des données est donc externalisé a posteriori. Ces outils de débogage ne sont pas adaptés pour être utilisés directement pour la détection dynamique des attaques sans aucun support matériel supplémentaire, à cause du délai de transfert, de la limitation de débit du port de débogage, et de la nécessité de rapidité de traitement d'une quantité importante de valeurs brutes collectées.

2.2.3.2 Détection basée sur des systèmes non commerciaux

Nous n'avons pas connaissance de méthodes de détection dynamique basées directement sur ces outils de débogage, nous n'avons pas connaissance non plus de propositions similaires de collecte de signaux arbitraires dans un SoC pour la détection. Cependant, certains travaux proposent de réaliser de la détection en se basant sur une petite sélection de signaux.

ROPecker [Cheng 2014] utilise les informations du *Last Branch Record* (LBR), collectées et mises à disposition du logiciel par certains processeurs, pour détecter des attaques ROP. Le LBR collecte les derniers branchements pris par le processeur dans une petite mémoire tampon (de 16 entrées par exemple). Une base de données des branchements et des séquences d'instructions utilisables par ROP est construite au préalable par l'analyse statique du logiciel à protéger. Pendant l'exécution, avant l'utilisation d'une page différente de la page courante ou avant un appel système critique, le contenu du LBR est vérifié vis-à-vis de la base de données préétablie afin de détecter que le logiciel protégé emprunte un branchement anormal et ainsi en déduire la présence des attaques ROP.

Backer et al. [Backer 2015] proposent de collecter les signaux exposés par les outils de débogage connectés aux composants matériels et au système (incluant l'instruction exécutée, l'adresse mémoire accédée, l'identifiant du master de bus qui initialise le transfert des données, le PID, le niveau de privilège actuel), pour détecter des attaques par corruption de mémoire. La principale raison qui pousse à utiliser les signaux de débogage est de limiter la modification sur les composants matériels existants. Les signaux collectés sont filtrés et comparés dans des composants matériels indépendants, vis-à-vis d'une politique définie au préalable, ou vis-à-vis de signatures des accès autorisés (générées avec l'analyse statique du logiciel et instrumentées dans le logiciel à protéger).

Ozsoy et al. [Ozsoy 2016] proposent de récupérer de multiples informations afin de détecter la présence de logiciels malveillants sur l'architecture x86 :

— des évènements matériels (fréquence de lecture/écriture mémoire, de bran-

chement choisi et d'accès mémoire non alignés) ;
— des informations plus précises sur des instructions exécutées (catégorie des instructions et des micro-opérations utilisées dans la microarchitecture, la fréquence des instructions de la même catégorie et la fréquence des micro-opérations) ;
— des adresses mémoire (distance entre les adresses accédées et la fréquence de différentes distances) ;
— des branchements (direction et type de branchement et leurs fréquences respectives).

Ces caractéristiques sont récupérées toutes les 10 000 instructions pour une base des logiciels légitimes et malveillants, afin d'entraîner des classifieurs différents (*logistic regression*, *neural network*). Ces classifieurs sont implémentés dans un composant indépendant à côté du processeur pour la détection dynamique des logiciels malveillants.

LiD-CAT [Reinbrecht 2020] collecte les signaux liés au cache (adresse demandée par le CPU, s'il s'agit d'un succès de cache ou d'un défaut de cache) afin de détecter les attaques par canal auxiliaire de cache. L'algorithme de détection est intégré au cache à protéger : en fonction de l'adresse demandée par le CPU, il reconnaît s'il s'agit d'une exécution de logiciel légitime ou de logiciel malveillant, puis, l'évolution des signaux est vérifiée à l'aide d'une machine à états basée sur une description formelle des attaques par canal auxiliaire de cache.

2.2.4 Conclusion

Le tableau 2.2 compare les méthodes de détection basées sur les HPCs présentées ci-dessus. Nous avons constaté que les HPCs sont fréquemment utilisés pour la détection des attaques microarchitecturales, et que les méthodes d'apprentissage automatique sont souvent utilisées afin d'augmenter la précision de la détection. En même temps, les attaques plus traditionnelles comme l'attaque ROP et les logiciels malveillants sont aussi détectables avec des HPCs. Certains détecteurs se basent sur les HPCs dans les processeurs existants avec une détection réalisée uniquement en logiciel, et peuvent donc être mis en place immédiatement dans des systèmes existants, même si la détection logicielle uniquement basée sur l'analyse des HPCs est forcément réduite en précision et en performance.

Le tableau 2.3 compare les méthodes de détection basée sur la supervision des signaux microarchitecturaux sans utilisation des HPCs. Nous n'avons pas connaissance de beaucoup de travaux dans cette catégorie. Cependant, les travaux existants montrent une grande variété d'attaques détectables et d'algorithmes utilisables. Le support matériel pour la logique de détection est généralement nécessaire pour le traitement de données, qui sont plus volumineuses que dans le cas des HPCs, sauf pour [Cheng 2014] qui exploite les informations présentes dans le LBR, qui est généralement de petite taille, et qui utilise un algorithme qui s'exécute à une fréquence faible. Ainsi, [Cheng 2014] peut s'exécuter sans accélérateur matériel.

Globalement, nous constatons que la détection dynamique des attaques basée sur

TABLE 2.2 – Comparaison des méthodes de détection dynamique des attaques basées sur les compteurs matériels de performance.

	Attaques Visées	**Algorithme de Détection**	
		Type	**Position**
[Chen 2014]	CTC	PM	Logiciel avec nouveaux compteurs
[Payer 2016]	SCA, Rowhammer	PM	Logiciel
[Chiappetta 2016]	SCA	PM ou ML	Logiciel
[Li 2018a]	SCA	ML	Logiciel
[Kulah 2019]	SCA	ML	Logiciel
[Mushtaq 2020]	SCA, TEA	ML	Logiciel
[Demme 2013]	Malware, SCA	ML	Composant indépendant
[Wang 2016]	ROP	PM	Logiciel
[Li 2018b]	ROP	PM	Intégrée dans le PMU

Abréviations : CTC : Covert Timing Channel ; SCA : Side Channel Attacks ; TEA : Transient Execution Attacks ; ROP : Return-Oriented Programming ; PM : Pattern Matching ; ML : Machine Learning ; PMU : Performance Monitoring Unit.

TABLE 2.3 – Comparaison des méthodes de détection dynamique des attaques basées sur la supervision des signaux microarchitecturaux.

	Attaques Visées	**Algorithme de Détection**	
		Type	**Position**
[Cheng 2014]	ROP	PM	Logiciel
[Backer 2015]	Corruption de Mémoire	PM	Composants indépendants
[Ozsoy 2016]	Malware	ML	Composant indépendant
[Reinbrecht 2020]	Cache SCA	FSM	Intégrée dans le cache

Abréviations : ROP : Return-Oriented Programming ; SCA : Side Channel Attacks ; PM : Pattern Matching ; ML : Machine Learning ; FSM : Finite State Machine.

des signaux microarchitecturaux couvre plus de classes d'attaques que la détection basée uniquement sur des signaux architecturaux, et elle est ainsi plus prometteuse pour construire une solution de détection dynamique des attaques dans les systèmes futurs.

2.3 Matériel reconfigurable pour la détection

Dans cette partie, nous nous intéressons à des mécanismes de détection matériels reconfigurables. Ces mécanismes ont pour double objectif de proposer des logiques de détection très bas niveau (proches de la microarchitecture), et modifiables dans le temps. Les systèmes actuels sont en général peu reconfigurables matériellement, principalement pour des raisons de performance, et sont plutôt reprogrammables logiciellement. Aussi, comme nous le montrons dans cette section, les travaux abordant des mécanismes de protection reconfigurables pour la détection d'attaques logicielles ou matérielles sont relativement peu nombreux à notre connaissance.

2.3.1 Contexte technique

La reconfigurabilité d'un matériel représente sa capacité à changer substantiellement la structure de son chemin de données. Un matériel non reconfigurable possède une architecture fixée au moment de la fabrication et dédiée à une fonctionnalité précise, contrairement au matériel reconfigurable qui contient un ensemble de blocs de base et un système de routage entre ces blocs pour permettre la construction de la fonctionnalité demandée. La reconfiguration est la modification des logiques dans les blocs et la connexion des chemins dans le circuit. Elle peut se faire régulièrement pendant le cycle de vie du matériel. En cas de reconfiguration partielle (une seule partie du matériel étant reconfigurée), elle peut même se faire pendant l'exécution de la partie qui n'est pas reconfigurée.

Selon la reconfigurabilité, c'est-à-dire la granularité (la taille des éléments non reconfigurables ainsi que la largeur des connexions) et la flexibilité dans le chemin de données, nous pouvons identifier deux grandes catégories de matériel reconfigurable :

1. *Reconfigurable à grain fin.* Fréquemment utilisé lors des reconfigurations, le matériel reconfigurable à grain fin a une granularité au niveau des bits, comme proposée par un FPGA. Ceci signifie que les opérations, unité de stockage et les connexions peuvent être ajustées par bit. Ce type de reconfiguration permet ainsi d'implémenter des logiques matérielles arbitraires. La partie gauche de la figure 2.2 présente un exemple d'un élément de base et un réseau de connexions dans un matériel reconfigurable à grain fin. Il faut noter que les éléments de base et leurs combinaisons, ainsi que la structure et la flexibilité des connexions diffèrent selon les fabricants et les besoins. Chaque élément de base est constitué d'une *Look-Up Table* (LUT) et d'un *Flip-Flop* (FF). La LUT peut être configurée pour générer une relation arbitraire entre les quatre signaux d'entrée, le FF permettant de stocker un bit de valeur. Nous pouvons aussi remarquer que beaucoup de ressources matérielles sont utilisées pour les connexions entre les blocs de logiques, garantissant une la flexibilité de connexion élevée.
2. *Reconfigurable à gros grain.* Un format moins flexible à plus grosse granularité (l'ordre des octets, comme 16 bits ou 32 bits [Podobas 2020]), comme dans un CGRA (*Coarse-Grained Reconfigurable Array*) est également utilisé. Une telle matrice reconfigurable est généralement conçue pour une application précise, comme une matrice spécifique permettant de réaliser différents calculs cryptographiques, les blocs de base et les connexions possibles sont également optimisés pour cette application. La partie droite de la figure 2.2 présente un exemple d'un élément de base et un réseau de connexions dans un matériel reconfigurable à gros grain. Nous pouvons remarquer que cet élément de base (appelé PE, *Processing Element*, dans la figure) contient un ALU (*Arithmetic Logic Unit*) pour l'exécution d'opérations (plus spécifiques que celles offertes par un LUT) et un registre (de plusieurs bits) pour le stockage. Le réseau de connexions est également moins dense, offrant moins de flexibilité.

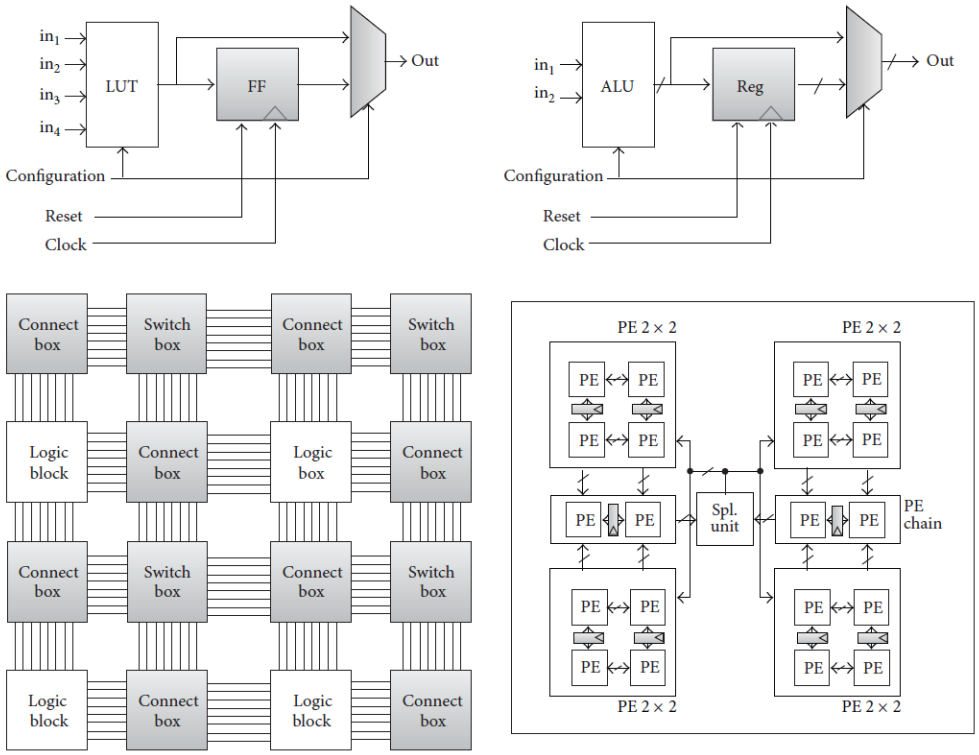


FIGURE 2.2 – Exemple d'architecture matérielle reconfigurable à grain fin (gauche) et à gros grain (droite). Source : [Chattopadhyay 2013]

Globalement, plus le niveau de reconfigurabilité est élevé (la granularité est fine et la flexibilité des chemins de données est grande) et moins le matériel est optimisé pour une application précise, plus la distance parcourue par les signaux est longue, plus la période d'horloge minimum pour la stabilisation des signaux physiques est longue. Ainsi, limité par la technologie actuelle, le matériel reconfigurable fonctionne à une fréquence moins élevée qu'un matériel non reconfigurable ou un processeur, et ceci est un point important à considérer pour la détection des attaques à l'aide de matériel reconfigurable.

Les matériels reconfigurables disponibles dans le marché sont généralement sous forme de FPGA grâce à leur grande reconfigurabilité, adaptée à une utilisation dans des domaines variés. Ils sont parfois utilisés conjointement avec des équipements radio [Ettus Research 2022], ou un processeur tel que le Xilinx Zynq SoC et sa nouvelle génération Versal SoC [Xilinx 2022], Intel SoC FPGAs [Intel 2022] et Intel Stellarton [EEJournal 2010]. Ils sont aussi utilisés par les fournisseurs de service cloud comme Amazon [Amazon Web Services 2022] et Alibaba [Alibaba 2018]. L'utilisation de matériel reconfigurable est courante pour l'accélération des calculs, dans les domaines tels que le multimédia, la radio définie par logiciel, la cryptographie et l'apprentissage automatique [Chattopadhyay 2013, Bossuet 2013, Colangelo 2017, Souza 2018].

2.3.2 État de l'art

Pour la détection des attaques, l'utilisation de matériel reconfigurable permet d'accélérer le traitement des données pour la détection et offre la possibilité de mise à jour du circuit. Cependant, parmi les travaux concernant la détection dynamique des attaques, nous avons principalement trouvé des usages d'accélération et très peu discutent de l'usage du matériel reconfigurable en dehors de son utilisation à des fins de prototypage.

Hutchings et al. [Hutchings 2002] présentent l'implémentation du string matching sous forme d'un automate fini non déterministe dans un FPGA pour la détection des intrusions réseaux. Les chaînes de caractères à comparer sont extraites à partir d'une base de données de règles de détection, et la comparaison permet d'identifier des contenus malveillants dans des paquets réseau. Les auteurs précisent que l'accélération de la vitesse de détection, à l'aide du matériel, permet d'appliquer de la recherche de motifs pour tous les paquets. Cette approche génère les logiques de détection à partir d'une base de données de règles, qui peuvent donc aisément être mises à jour en fonction de l'évolution de cette base de données.

FlexCore [Deng 2010] propose l'implémentation du DIFT ainsi que d'autres vérifications basées sur les étiquettes avec un composant matériel reconfigurable. Les attaques détectables et les traces collectées sont similaires à Harmoni [Deng 2012], cependant, tous les détecteurs ne sont pas présents en même temps dans le circuit de FlexCore : le matériel est reconfiguré pour changer de logique de vérification. FlexCore discute du problème de différence de fréquence avec le processeur lié à la structure reconfigurable. Les auteurs ont proposé la mise en place d'une mémoire tampon FIFO (*First In First Out*) pour gérer cette différence de domaine d'horloge, et cette mémoire tampon FIFO peut être programmée pour tracer uniquement les instructions clés selon le besoin du détecteur en cours, et ainsi éviter la saturation de messages dans la mémoire tampon.

Des détections dynamiques basées sur un apprentissage automatique accéléré par un matériel reconfigurable sont également proposées dans la littérature. Par exemple, Ioannou et Fahmy [Ioannou 2019] proposent de détecter les intrusions réseau avec un classifieur de type *neural network*, implémenté dans un FPGA, en collaboration avec un processeur, le tout intégré dans une passerelle de l'Internet des objets. Gordon et al. [Gordon 2021] proposent une architecture pour détecter des attaques réseaux par déni de service, dans laquelle l'extraction des caractéristiques est faite dans un processeur, et la classification (*K-nearest neighbors*) est implémentée dans un FPGA pour l'accélération. Dans ces travaux, les auteurs n'ont pas utilisé la reconfiguration pour la mise à jour des classifieurs, mais proposent de pouvoir programmer les paramètres des classifieurs.

2.3.3 Conclusion

Le tableau 2.4 compare les méthodes de détection avec l'utilisation du matériel reconfigurable présentées ci-dessus. En fait, seuls les deux premiers travaux utilisent

vraiment la reconfigurabilité du matériel pour le remplacement des logiques de détection, et ils mentionnent seulement très brièvement la capacité de l'évolution du matériel reconfigurable face aux nouvelles techniques de détection et aux nouvelles stratégies d'attaques. Nous avons mentionné les deux autres travaux principalement pour illustrer la capacité d'accélération d'algorithmes très différents pour la détection. Comme certains algorithmes de détection basés sur les HPCs utilisent également l'apprentissage automatique, l'implémentation de ces algorithmes dans une structure reconfigurable permet en plus de les faire évoluer (changer le nombre de nœud dans un *neural network* par exemple), de changer entre différents algorithmes ou de combiner ces algorithmes selon les besoins.

TABLE 2.4 – Comparaison des méthodes de détection dynamique des attaques avec l'utilisation du matériel reconfigurable pour la logique de détection.

	Attaques Visées	**Algorithme de Détection**	
		Type	**Position**
[Hutchings 2002]	Réseau	FSM	Composant reconfigurable
[Deng 2010]	Corruption de mémoire	PM	Composant reconfigurable
[Ioannou 2019]	Réseau	ML	Composant indépendant
[Gordon 2021]	DoS	ML	Composant indépendant

Abréviations : PM : Pattern Matching ; FSM : Finite-State Machine ; ML : Machine Learning ; DoS : Denial-of-Service.

2.4 Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art des différentes techniques de détection dynamique des attaques en lien avec notre étude.

Nous avons présenté tout d'abord les méthodes de détection basées sur les signaux architecturaux non reconfigurables. Une trace architecturale bien choisie et compressée est souvent utilisée pour la détection des attaques classiques en lien avec les données dans la mémoire. Ensuite, nous avons présenté les méthodes de détection basées sur les signaux microarchitecturaux, en particulier les méthodes utilisant les HPCs pour analyser le nombre d'occurrences de certains évènements. L'utilisation de signaux microarchitecturaux permet d'avoir accès à plus d'informations liées à l'implémentation du système et offre la possibilité de détecter des attaques de catégories plus variées. Nous avons ensuite présenté l'utilisation de matériel reconfigurable dans le contexte de la détection des attaques. Nous avons observé beaucoup d'usages de matériel reconfigurable pour l'accélération des logiques de détection, et quelques cas permettant de choisir entre différentes logiques selon le besoin du moment. Mais seulement très peu de travaux mentionnent l'utilisation de la reconfigurabilité pour la mise à jour du circuit suite à des nouveaux besoins de détection, pour faire face à de nouvelles attaques par exemple.

À la lumière des différentes travaux de l'état de l'art, nous avons pu constater

que l'observation des signaux microarchitecturaux a le potentiel de détecter différentes catégories d'attaques. Cependant, nous avons pu constater que les approches proposées sont en général adaptées à des attaques qui évoluent peu dans le temps, ce qui reste inadapté au niveau des attaques matérielles qui ont montré leur capacité à s'adapter aux contremesures mises en place au niveau logiciel notamment. Ainsi, s'intéresser à des mécanismes matériels et flexibles a du sens. De plus, l'utilisation du matériel permet d'envisager des méchanismes peu impactants sur le système car capables de fonctionner indépendamment du processeur. La reconfiguration du matériel offre une flexibilité dans le choix des détecteurs et la possibilité de mettre à jour le système au cours de son cycle de vie. Concernant le problème de fréquence du matériel reconfigurable, nous avons analysé les techniques de transmission existantes (bus, mémoire tampon FIFO, compteurs) et nous proposons des connexions dédiées pour les signaux afin de préserver le plus d'information possible. Dans le chapitre suivant, nous présentons le framework que nous avons conçu pour essayer d'apporter une solution efficace, générique et reconfigurable, à base d'analyse de signaux microarchitecturaux, à la détection dynamique d'attaques.

Chapitre 3

Détection d'attaques par caractérisation d'empreintes microarchitecturales

Sommaire

3.1 Modèle de menace et hypothèses 47
3.2 Architecture 48
3.2.1 Partie matérielle 49
3.2.2 Partie logicielle 53
3.2.3 De l'intégration à l'utilisation 54
3.3 Conception et intégration de l'architecture de détection . . 55
3.3.1 Processus d'intégration 55
3.3.2 Choix de la configuration 56
3.4 Utilisation du système 59
3.4.1 Vue globale 59
3.4.2 Détection dynamique 60
3.5 Conclusion 64

Ce chapitre présente le système de détection d'attaques que nous proposons, dont l'objectif est de détecter des attaques par la caractérisation de leur empreinte microarchitecturale. Il contient le modèle de menaces et les hypothèses, l'architecture de détection proposée, des méthodes d'intégration et de paramétrage de cette architecture pour un système cible, et des stratégies d'utilisation du système cible modifié. Une attention particulière est accordée aux défis et aux avantages de l'utilisation de matériel reconfigurable pour le Module de Détection (le composant central destiné à traiter des données microarchitecturales et détecter dynamiquement les attaques). Nous avons pris soin de gérer la différence de fréquence entre le matériel reconfigurable et les autres parties du système, tout en tirant avantage de sa reconfigurabilité après fabrication pour s'adapter à l'évolution rapide des attaques.

3.1 Modèle de menace et hypothèses

Nous considérons un attaquant possédant les privilèges d'un utilisateur normal, connecté à distance sur le système cible, qui cherche à attaquer un logiciel vic-

time s'exécutant sur le système. Ce modèle de menace est assez générique pour les attaques d'origine logicielle. Les hypothèses considérées sont les suivantes :

1. Le noyau du système cible est de confiance et l'attaquant n'a que des privilèges d'un utilisateur normal. Cette hypothèse est importante, car, comme nous le verrons dans la suite, la partie logicielle des algorithmes de détection est implémentée dans le noyau du système cible.
2. Le matériel du système cible est de confiance, l'attaquant n'a pas d'accès physique au système. Par conséquent, les attaques non intrusives par canaux auxiliaires basées sur l'accès physique (comme l'analyse de la mesure de puissance) ne sont pas considérées dans notre travail, car elles ne modifient pas l'état du système et leur comportement reste invisible au niveau microarchitectural. Les attaques par injection de fautes ne sont pas non plus considérées, car elles pourraient endommager les logiques de détection dans le matériel (comme des faisceaux laser qui pourraient créer des inversions de bits dans une logique).
3. L'attaquant peut exécuter n'importe quel logiciel d'attaque et n'importe quel logiciel légitime en espace utilisateur. Il a le contrôle total du processus d'attaque, et il peut contrôler les entrées du processus victime. Dans le cas des attaques microarchitecturales, il peut contourner les mécanismes d'isolation interprocessus en utilisant des informations microarchitecturales ; dans le cas des attaques par réutilisation de code, il peut exploiter une mémoire tampon non protégée pour diriger le flux de contrôle vers des codes existants.

3.2 Architecture

L'architecture que nous proposons pour la détection d'attaques est représentée dans la figure 3.1. Elle est construite à partir du système cible auquel on ajoute plusieurs sondes, un Module de Détection (appelé *MD* par la suite), composé d'une partie matérielle reconfigurable (appelée *MDM*, Module de Détection Matériel), d'une partie logicielle (appelée *MDL*, Module de Détection Logiciel), et des connexions appropriées.

Chaque composant de cette architecture a un rôle précis : (1) les sondes sont les éléments clés qui nous permettent d'extraire des signaux microarchitecturaux des composants cibles ; (2) le MDM reconfigurable se charge d'analyser les signaux microarchitecturaux, tout en bénéficiant de la capacité de parallélisme du matériel pour l'accélération des calculs et de la reconfigurabilité pour pouvoir être mise à jour ; (3) le MDL dans le noyau aide à la détection et la réaction contre les attaques avec un point de vue plus global du système, et permet de configurer et mettre à jour le MD ; et (4) les connexions sont conçues pour pouvoir échanger des informations importantes dans le respect des échéances temporelles tout en réduisant les modifications requises pour le système cible. Nous avons fait le choix de laisser le MD le plus indépendant possible du reste de système, car cela permet de (1) minimiser les modifications nécessaires sur le système cible, ce qui facilite

l'intégration de cette architecture de détection dans des systèmes cibles variés, et qui minimise l'adaptation nécessaire pour suivre l'évolution des systèmes ; (2) simplifier l'organisation du matériel par le regroupement des logiques reconfigurables, une seule zone de structure reconfigurable étant suffisante pour contenir le MDM ; et (3) faciliter la mise à jour du MD grâce à la centralisation des logiques à modifier.

Dans la suite, nous présentons l'architecture en discutant séparément de la partie matérielle et de la partie logicielle. Puis, comme certains éléments de cette architecture sont paramétrables lors de la conception et d'autres reconfigurables lors de l'utilisation, nous discutons de ces deux étapes du cycle de vie de l'architecture.

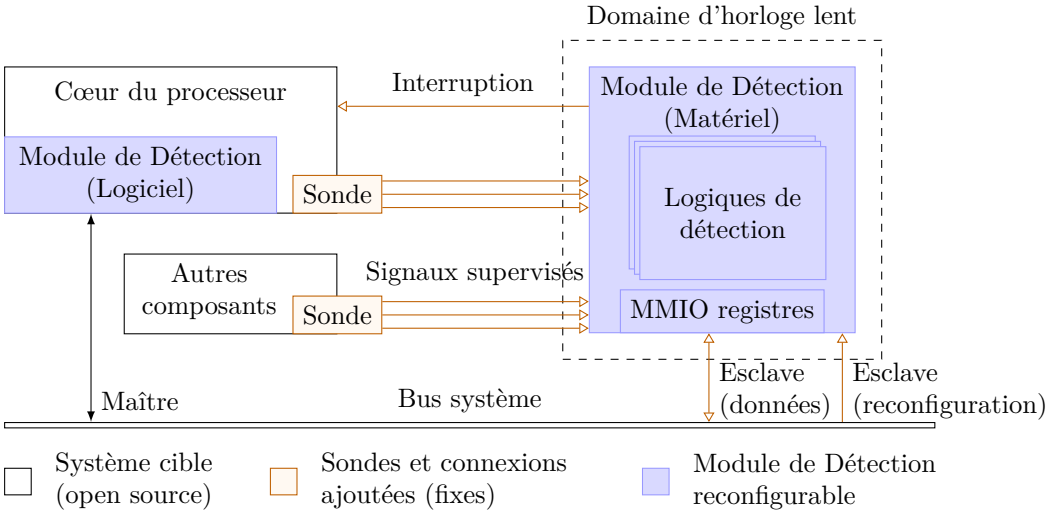


FIGURE 3.1 – Architecture de détection qui intègre un MD reconfigurable dans le système cible.

3.2.1 Partie matérielle

La partie matérielle de notre architecture de détection consiste principalement à mettre en place le MDM dans le système cible. Le MDM lui-même se charge d'analyser des signaux microarchitecturaux afin de détecter la présence d'attaques microarchitecturales dans le système. Il peut contenir de multiples logiques de détection afin de détecter différentes attaques simultanément, et il a besoin d'être reconfigurable pour pouvoir s'adapter à différentes situations et à l'évolution des attaques. Ces signaux microarchitecturaux proviennent des composants cibles, captés par des sondes intégrées dans ces derniers. Ils sont transmis au MDM à l'aide d'une connexion dédiée afin de préserver un débit optimal pour la détection, et de s'affranchir des perturbations dues aux autres activités du système. Trois autres connexions sont prévues entre le MDM et le système cible, incluant une connexion sur le bus pour échanger les informations pendant l'exécution, une interruption pour que le MDM puisse notifier le processeur, et une connexion pour reconfigurer les

logiques présentes dans le MDM. Les sondes, le MDM, les quatre connexions et la prise en compte du changement de domaine d'horloge pour la connexion dédiée aux signaux microarchitecturaux sont présentés de manière plus détaillée dans la suite.

3.2.1.1 Sondes

Les sondes sont intégrées aux composants cibles afin d'extraire les valeurs des signaux internes (appelés *signaux supervisés* dans la suite). Pour cette raison, les composants cibles doivent permettre l'installation de connexions supplémentaires vers l'extérieur du composant. Les composants cibles qui nous intéressent particulièrement dans cette thèse sont les composants matériels qui contiennent de nombreux états microarchitecturaux comme par exemple le cœur d'un processeur, ou des ressources critiques comme les différents niveaux de cache et les prédicteurs de branchement.

La présence et le placement des sondes sont cruciaux dans cette architecture, car la qualité de la détection dépend fortement de l'analyse des signaux supervisés par ces sondes. Si les différentes sondes sont placées de manière adéquate sur les composants cibles, elles peuvent fournir un grand nombre d'informations microarchitecturales à la granularité du cycle d'horloge, ce qui donne une vue précise de l'état du système et permet de construire une logique de détection appropriée. Le choix des signaux supervisés étant crucial, nous présentons une démarche itérative permettant d'assister ce choix dans la section 3.3.2.

3.2.1.2 Module de Détection

Le MD a été conçu comme un module autonome à la fois au niveau matériel et logiciel, pour une meilleure modularité du système. Ainsi, il est placé en dehors des composants existants.

Le MDM est mis en œuvre dans une structure entièrement reconfigurable pour effectuer différentes analyses spécifiques sur les signaux supervisés (représentées par les logiques de détection dans la figure 3.1), et bénéficier de la flexibilité des algorithmes de détection mis en œuvre pour faire face aux besoins de détection qui évoluent constamment. Il est équipé d'un ensemble de registres MMIO (*Memory-Mapped I/O*) et d'une zone mémoire, pour communiquer ses résultats intermédiaires au MDL situé dans le noyau du système, ou recevoir des configurations et des informations complémentaires venant du MDL. Il est également équipé d'un générateur d'interruptions, afin de pouvoir notifier le MDL en cas de besoin d'une intervention immédiate. Le type d'informations stockées dans les registres MMIO et le fonctionnement du générateur d'interruptions peuvent également être modifiés pendant l'exécution du système à l'aide de la reconfiguration.

Aucune hypothèse n'est faite quant au type d'algorithme implémenté dans les logiques de détection. Il est possible d'utiliser des algorithmes simples, tels que des heuristiques, ou plus sophistiqués, basés sur les machines à états ou l'apprentissage automatique. Comme le MD est indépendant du reste du système, il peut effectuer ces analyses sans affecter le fonctionnement normal du processeur.

3.2.1.3 Connexions

Nous avons prévu quatre canaux de communications entre le MDM et le reste du système (représentés par les flèches oranges dans la figure 3.1), ils sont présentés dans ce qui suit. Notons que, lorsque le MDM dispose d'une horloge plus lente que celle du processeur, par exemple à cause de la fréquence d'exécution limitée d'une structure reconfigurable, il est nécessaire de mettre en place des techniques de synchronisation adaptées sur chaque connexion.

1. *Signaux supervisés.* Les signaux supervisés provenant des sondes ont une connexion dédiée directe au MDM. Cette connexion dédiée garantit un transfert de données à haut débit et à faible latence qui n'est pas affectée par les autres activités du système. Il est fondamental de conserver les temps d'occurrences des évènements pour faire face aux attaques par canal auxiliaire de temps, mais la fréquence des informations reçues par le MDM peut être beaucoup plus élevée que sa fréquence de traitement. Pour ces raisons, une simple mémoire tampon FIFO ne suffit pas, nous proposons ainsi des connexions adaptées pour la synchronisation des signaux supervisés en cas de différence de fréquences d'horloge dans la section 3.2.1.4.

2. *Interruption.* Le MDM peut décider d'envoyer des interruptions au processeur, lorsqu'il estime qu'il y a besoin d'une intervention immédiate du MDL, par exemple lorsqu'une attaque est détectée. Le MDL décide s'il s'agit d'une réelle attaque et quelle action prendre, car il bénéficie d'une vue plus globale de l'environnement d'exécution. Nous n'avons pas ajouté d'autres signaux provenant du MDM vers les composants cibles en dehors de cette connexion sur la ligne d'interruption, car l'ajout des signaux nécessiterait des modifications significatives des composants cibles pour qu'ils puissent gérer ces signaux.

3. *Esclave du bus (données).* Le MDM est connecté au bus du système en tant qu'esclave, pour permettre au maître du bus (le MDL dans le processeur) la lecture et l'écriture dans les registres MMIO et la zone mémoire qu'il expose. Cette interface est utilisée lorsqu'une interruption est émise par le MDM, ou durant l'exécution périodique du MDL, pour échanger des informations supplémentaires. Dans les deux cas, c'est au MDL d'initier les demandes de lecture et d'écriture.

4. *Esclave du bus (reconfiguration).* Un canal de reconfiguration est prévu pour que le MDL puisse mettre à jour les logiques de détection mises en place dans la structure reconfigurable du MDM. Ce canal passe également par le bus du système, en tant qu'esclave, pour recevoir les logiques à mettre en place dans la structure reconfigurable. Comme la fréquence de reconfiguration d'une structure reconfigurable n'est pas nécessairement le même que sa fréquence d'exécution, cette connexion peut être indépendante de l'esclave du bus pour les données.

3.2.1.4 Connexion des signaux supervisés

Nous proposons plusieurs approches de synchronisation des signaux supervisés dans le cas d'un MDM à basse fréquence, qui visent à réduire la quantité de ressources matérielles utilisées tout en préservant suffisamment d'informations transmises pour la détection des attaques. Les types de connexion des signaux supervisés proposés sont illustrés sur la figure 3.2, sous l'hypothèse de la fréquence d'un composant cible *divN* fois plus élevée que la fréquence du MDM :

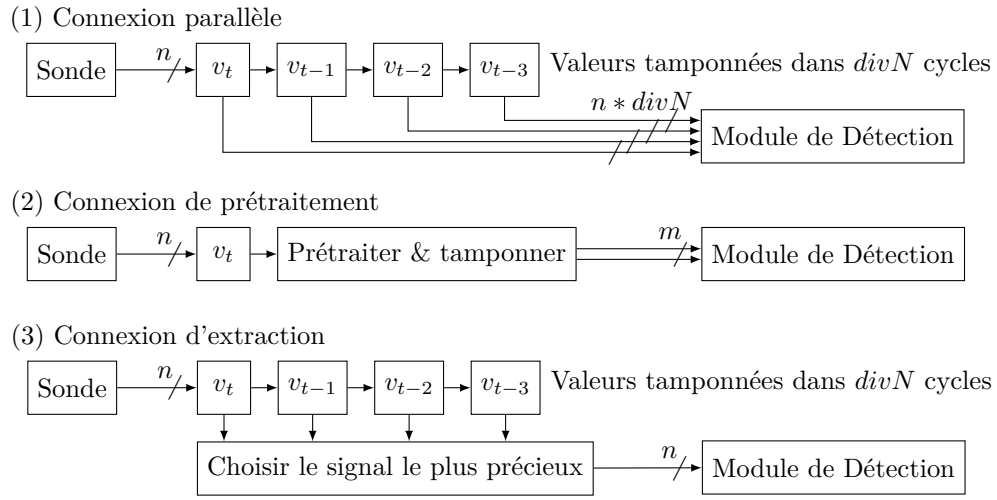


FIGURE 3.2 – Types de connexion des signaux supervisés pour le MDM à basse fréquence.

1. Une *connexion parallèle* collecte tous les signaux dans *divN* cycles adjacents et les transmet en parallèle au MDM. Ce type de connexion conserve au maximum les informations collectées et l'état des signaux à un moment précis, mais nécessite plus de logique de synchronisation et de logique interne pour les traiter.
2. Une *connexion de prétraitement* prétraite et met en mémoire tampon la valeur du signal supervisé, puis envoie les données traitées au MDM. Elle assure un compromis entre l'utilisation des ressources matérielles et la quantité de données utiles transmises, et peut réduire l'effort d'analyse dans le MDM. Par exemple, cette logique peut compter certaines occurrences d'évènements matériels ou extraire les traces d'instructions, et fournir la possibilité d'implémenter des solutions de détection existantes basées sur des compteurs matériels ou sur le traçage d'instructions.
3. Une *connexion d'extraction* est un cas particulier et simple de connexion de prétraitement qui sélectionne un signal intéressant dans une fenêtre de *divN* échantillons. Elle peut être utilisée lorsque la valeur du signal change rarement ou lorsqu'une certaine perte de données est acceptable au regard de la précision de la détection. Par rapport à une mémoire tampon FIFO, il

préserve le temps d'occurrence des évènements, ce qui est utile pour détecter les fuites d'informations temporelles.

Notons que différents types de connexion peuvent être mixtes, c'est-à-dire que des signaux différents provenant d'une même sonde peuvent utiliser des types de connexion différents.

3.2.2 Partie logicielle

Le MDL est un module situé dans le noyau du système. Il communique avec le MDM et participe à la détection, mais il est également en communication avec d'autres parties du noyau telles que le gestionnaire d'interruptions et le gestionnaire d'ordonnancement des tâches. Comme mentionné dans les hypothèses, le noyau et le MDL sont considérés comme fiables. Les principales fonctions du MDL consistent (mais ne sont pas limitées) à :

1. *Effectuer des analyses de données pendant l'exécution.* Les algorithmes de détection d'attaques sont implémentés en matériel et logiciel, car il existe des situations où le matériel seul ne suffit pas : (1) s'il existe une logique difficile ou trop coûteuse à mettre en œuvre dans le matériel ; (2) si le matériel ne dispose pas de suffisamment d'espace disponible pour mettre en place une logique spécifique (par exemple, lorsque plusieurs algorithmes de détection doivent être activés en même temps, ou lorsque l'algorithme lui-même est assez complexe).
2. *Fournir des informations supplémentaires pendant l'exécution.* Certaines informations importantes de haut niveau, représentant l'état global de l'environnement d'exécution ou l'état d'une tâche spécifique, ne sont disponibles qu'au niveau logiciel. Le MDL est chargé de collecter ces informations et de les envoyer par le bus au MDM, pour améliorer la précision de la détection. Ces informations supplémentaires doivent être fournies périodiquement ou sur la base d'un évènement (par exemple, un changement de contexte ou l'expiration d'un timer) afin que le MDM puisse suivre l'évolution du système. Par exemple, le MDL pourrait fournir l'identifiant du processus suivant (PID) lors d'un changement de contexte, afin que le MDM puisse distinguer chaque processus et déterminer plus précisément quel processus a un comportement malveillant.
3. *Gérer les interruptions.* Lorsqu'une interruption est levée par le MDM, le MDL est chargé de collecter et analyser des données disponibles et décider si effectivement une attaque a eu lieu, et quelle action entreprendre. Les actions possibles sont par exemple activer le partitionnement de ressources pour isoler le logiciel suspect des logiciels légitimes.
4. *Configurer le MDM.* Le MDL peut configurer dynamiquement le MDM en positionnant des valeurs dans les registres MMIO via le bus. Par exemple, les seuils de détection peuvent être ajustés et l'interruption peut être activée ou désactivée selon le niveau de sensibilité de détection requis.

5. *Reconfigurer le MDM.* Dans différents cas, il peut y avoir besoin de reconfigurer le MDM, pour changer les logiques de détection mises en place. Par exemple, le MDL peut souhaiter installer une logique de détection spécialement conçue pour une librairie de cryptographie pendant son exécution.
6. *Mettre à jour le MD.* Il est possible et utile de mettre à jour régulièrement le MD. Ces mises à jour, à la fois pour la partie logicielle et la partie matérielle, sont importantes, car elles permettent de couvrir les vulnérabilités non prévues au moment de la sortie de l'usine du système cible équipé du MD, et permettent de prendre en compte des nouvelles variantes des attaques. Plus de détails sur la mise à jour, la configuration et la reconfiguration du MDM sont fournies dans la section 3.4.

Notons que la partie logicielle ne nécessite aucune modification des logiciels utilisateurs, c'est-à-dire qu'il n'y a pas besoin d'y insérer des instructions spécifiques. Cela garantit que notre architecture est compatible avec la base des logiciels existants et adaptée aux systèmes non embarqués qui pourraient contenir des logiciels non contrôlables par l'opérateur du système.

En absence du MDL, le MDM peut être utilisé seul afin de collecter et d'analyser les signaux supervisés. Dans ce cas, il peut communiquer avec l'extérieur du système en utilisant le port de débogage, pour une analyse hors ligne plus approfondie. Cette méthode serait très similaire aux techniques existantes de sondage des signaux, mais légèrement meilleure, car l'analyse des différents signaux dans le MDM permet de produire des informations plus significatives. Cependant, de notre point de vue, l'analyse du seul matériel est trop limitée, la partie analyse logicielle du MD améliore significativement l'efficacité de la détection.

3.2.3 De l'intégration à l'utilisation

Deux phases d'utilisation de cette architecture peuvent être clairement identifiées : lors de l'intégration du MD au système cible, et lorsque le système cible avec le MD définitivement intégré est produit et utilisé. Le système cible avec le MD intégré et adapté spécifiquement à ce système est appelé dans la suite *système final*.

Pendant la phase d'intégration du MD au système cible, il est nécessaire de réaliser différents tests de façon à déterminer les signaux supervisés les plus pertinents et la taille prévue pour la structure reconfigurable. Ces paramètres sont laissés au choix des concepteurs du système parce qu'ils sont nécessairement liés aux besoins de sécurité. L'usage mixte des matériels non reconfigurables et reconfigurables permet l'intégration de sondes dans les composants cibles à haute fréquence tout en gardant le MD flexible, offrant ainsi un équilibre entre l'effort de conception, les performances du système et l'adaptabilité de détection pour les attaques futures. Une méthodologie pour intégrer notre MD dans un système cible et choisir les paramètres appropriés est présentée dans la section 3.3.

Afin de pouvoir déterminer les meilleurs signaux à superviser et les connexions à utiliser, nous avons besoin de faire fonctionner le système dans différentes configurations. Pour cela, dans cette phase, il est nécessaire que les composants cibles,

le MDM et ses connexions soient tous placés dans une structure reconfigurable telle qu'un FPGA. Ceci sous entend que l'on dispose au moins d'une description microarchitecturale (par exemple sous forme de HDL) de ces composants cibles. D'autres parties du système cible peuvent se trouver à l'extérieur de celui-ci, tant que la synchronisation est correctement configurée. Soulignons que cette architecture est indépendante de l'ISA, ce qui signifie que de nombreux systèmes sont compatibles tant que (1) les composants cibles peuvent fournir un accès à certains signaux internes et que (2) l'accès à un bus du système et à une ligne d'interruption sont fournis à la partie reconfigurable pendant la phase d'intégration.

La phase d'utilisation du système final intervient lorsque les choix définitifs des signaux à superviser et des types de connexions sont faits et intégrés dans le système cible (sous forme d'un ASIC par exemple). À ce stade, seul le MD reste reconfigurable afin de permettre une mise à jour des algorithmes de détection. En effet, dans cette phase, les signaux supervisés étant définitivement choisis, la détection se base forcément sur ces signaux, mais les algorithmes de détection restent quant à eux modifiables par le biais des mécanismes de reconfiguration matérielle et logicielle. Une description plus détaillée de l'utilisation du système final est fournie dans la section 3.4.

3.3 Conception et intégration de l'architecture de détection

Dans cette section, nous présentons d'abord les étapes et les modifications requises pour qu'un système cible intègre notre MD. Puis, nous présentons une démarche itérative qui aide au choix des paramètres de la partie non reconfigurable de l'architecture de détection.

3.3.1 Processus d'intégration

La figure 3.3 présente le processus d'intégration de notre architecture de détection dans le système cible. Cette intégration suit une approche de type boîte blanche, où une description précise de la microarchitecture des composants cibles est nécessaire ainsi qu'une description des attaques cibles qui doivent être détectées. Elle est divisée en quatre étapes principales, qui consistent en :

1. À partir du système cible et de la description des attaques à détecter, *concevoir et évaluer différents algorithmes de détection basés sur l'analyse des signaux microarchitecturaux*. Pendant cette étape, on peut identifier notamment une liste des signaux microarchitecturaux pertinents à superviser (généralement des signaux modifiés pendant l'occurrence de l'attaque), et différents algorithmes de détection d'attaque associés.
2. À partir des résultats de l'étape 1, *configurer l'architecture de détection*. La configuration concerne principalement la partie non modifiable de l'architecture une fois fabriquée. Certaines exigences comme la surface du matériel

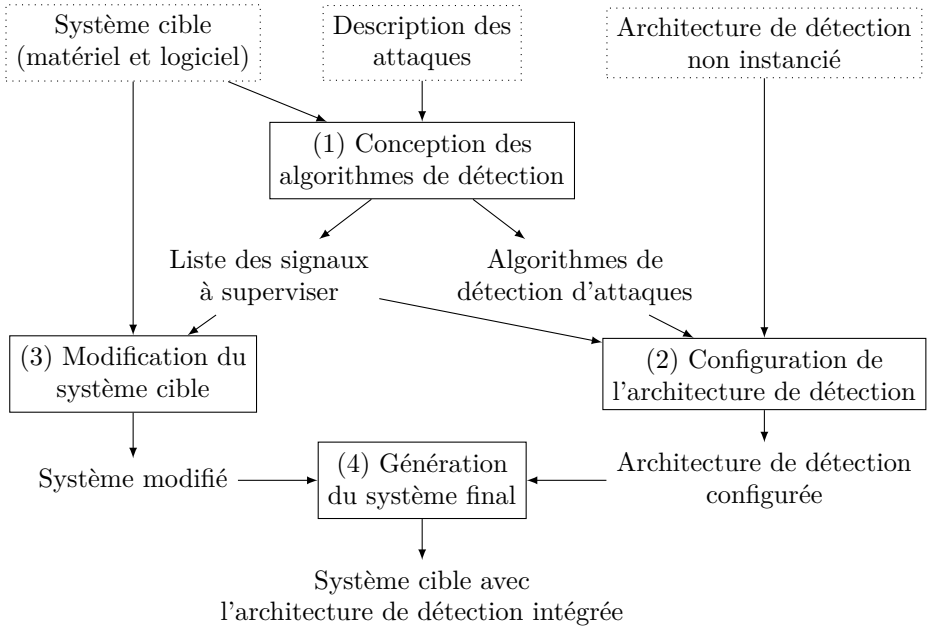


FIGURE 3.3 – Intégration de l'architecture de détection dans le système cible.

disponible, et des besoins comme la durée de vie de la détection, doivent être pris en compte dans cette étape. Une méthode itérative de la conception des algorithmes de détection à la configuration de l'architecture est présentée dans la section 3.3.2.

3. *Modifier le système cible (matériel et logiciel).* Pour la partie matérielle, il est nécessaire de modifier la microarchitecture du système cible pour extraire les signaux identifiés à l'étape 1 vers le MD. Le MDM doit également être connecté au bus du système et à la ligne d'interruption du processeur. Pour la partie logicielle, il faut modifier le noyau du système pour intégrer le MDL, permettant typiquement des communications de/vers le MDM et d'autres parties du noyau. Plus de détails de l'architecture finale ont été décrits dans la section 3.2.
4. Assembler le système modifié et l'architecture de détection configurée, afin de *générer le système final.*

3.3.2 Choix de la configuration

Notre architecture de détection contient des éléments non modifiables après la fabrication du système final : le placement des sondes, les signaux supervisés, le type de connexion pour chaque signal supervisé, et la taille prévue pour la structure reconfigurable du MDM. Nous n'avons volontairement pas fixé au préalable ces éléments, afin de laisser ce choix aux concepteurs du système. Cette configuration doit être faite précisément, car elle peut beaucoup impacter la qualité de détection dans la durée de vie du système.

Nous proposons ainsi une démarche itérative pour aider à la configuration de notre architecture, basée sur l'efficacité des algorithmes de détection conçus. Elle se compose de cinq étapes, comme illustré dans la figure 3.4 :

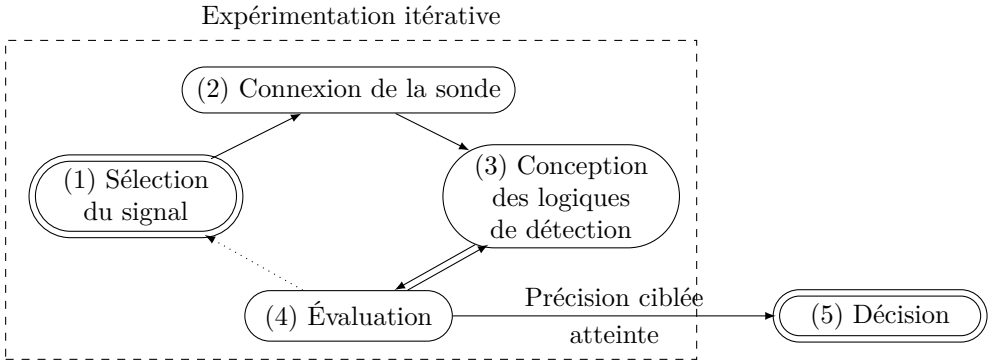


FIGURE 3.4 – Démarche itérative pour la configuration de la partie non modifiable de l'architecture.

1. *Sélection du signal.* Cette étape vise à sélectionner les signaux à superviser les plus appropriés pour la détection des attaques. Différentes approches itératives peuvent être utilisées ou combinées. Par exemple : (1) commencer par l'ensemble des signaux et états à l'intérieur d'un composant (comme le cache L1), évaluer et sélectionner par expérimentations les signaux qui discriminent au mieux les logiciels normaux et malveillants ; (2) commencer par une étape d'analyse des comportements d'attaque et la microarchitecture, afin de choisir les signaux qui semblent les plus susceptibles d'être influencés par les attaques (comme l'adresse envoyée dans le cache), puis identifier d'autres signaux complémentaires (comme le signal indiquant un défaut de cache, la demande d'accès mémoire depuis le cache à la mémoire), et terminer par comparer et choisir ceux qui offrent une meilleure performance de détection à l'aide d'expérimentations. En dehors des signaux d'un composant spécifique, nous avons constaté que la trace d'exécution comme le flux d'instructions est généralement intéressante (puisque les attaques microarchitecturales sont basées sur des séquences d'instructions relativement petites).
2. *Connexion de la sonde.* Cette étape consiste à choisir une technique appropriée de gestion des domaines d'horloge pour connecter chaque signal supervisé au MD. Cette étape n'est nécessaire que si le MD ne fonctionne pas à la même fréquence que le processeur. Le choix de connexion doit prendre en compte la quantité et la précision d'informations transmises, le surcoût de surface et de performance pour la transmission et la détection. Le choix des connexions appropriées est important car, dans le cas d'un MD reconfigurable, la fréquence peut être très inférieure à celle du cœur de processeur. Les différentes techniques de transmission que nous proposons ont été présentées dans la section 3.2.1.4.

3. *Conception des logiques de détection.* Cette étape vise à concevoir les algorithmes de détection destinés à être implémentés dans le MD, à la fois dans la partie matérielle et la partie logicielle. Les entrées de l'algorithme sont les signaux sélectionnés à l'étape 1 ainsi que des informations collectables au niveau noyau. La sortie de l'algorithme est une alerte lorsqu'une attaque est détectée. L'algorithme lui-même peut être conçu sur la base du comportement attendu des composants cibles ou des propriétés des attaques à détecter, ou être adapté à partir d'algorithmes de détection déjà connus. Étant donné que le MD est reconfigurable, donc aisément modifiable, une approche itérative est facilement utilisable pour étudier différents algorithmes de détection pour une attaque donnée.

4. *Évaluation.* Cette étape vise à réaliser des expérimentations pour évaluer la pertinence du détecteur d'attaques (avec les signaux, les connexions et les algorithmes définis dans les étapes précédentes) et sa surcharge. Afin de pouvoir réaliser des expérimentations dans un environnement plus réaliste et de pouvoir bénéficier d'un grand nombre et d'une grande variété de tests existants, notre prototype est équipé d'un système d'exploitation. Cela nous a permis d'évaluer la stabilité du détecteur au cours de différentes campagnes de test. Dans nos cas d'usage, nous avons choisi un ensemble de tests comprenant des attaques, et plusieurs autres ensembles de tests choisis dans l'état de l'art destinés à assurer une bonne couverture des applications légitimes. Si la logique de détection mise en œuvre présente trop de faux positifs ou de faux négatifs, il est nécessaire de revenir à l'étape 1 ou 3 et d'améliorer soit la sélection des signaux, soit la conception de la logique de détection elle-même.

5. *Décision.* Lorsque les logiques de détection présentent un bon taux de détection pour des attaques visées, et lorsque l'impact sur la performance et la surface requise sont acceptables, une dernière étape consiste à décider de la configuration de l'architecture finale. Le placement des sondes et les signaux supervisés sont définitivement choisis dans cette étape. Les signaux choisis sont ceux qui ont fourni les meilleurs taux de détection lors des phases précédentes, mais d'autres signaux qui sont considérés intéressants pour couvrir de futures attaques peuvent également être ajoutés. Des optimisations peuvent être réalisées, par exemple pour prioriser les algorithmes qui utilisent des signaux identiques, afin de réduire la modification des composants cibles. Le type de connexion pour chaque signal supervisé doit être choisi en fonction de l'impact sur la précision de la détection, sur la surface, et sur la performance. Finalement, la taille prévue pour la structure reconfigurable du MD doit être choisie selon la taille nécessaire pour implémenter les algorithmes : elle doit au moins pouvoir contenir chaque algorithme séparément, et idéalement, plusieurs algorithmes en même temps. Un espace supplémentaire doit être prévu pour permettre l'évolution des algorithmes de détection dans la durée de vie du système. Des optimisations des algorithmes peuvent

être faites, comme la réduction de l'utilisation des ressources matérielles et le partage des logiques entre différents algorithmes.

Notre architecture peut également servir de plateforme d'expérimentation pour évaluer l'efficacité de la protection contre des attaques mise en place par un composant matériel donné (idéalement un *soft intellectual property core*, *soft IP core*) dans un environnement d'exécution réel. Par exemple, il est possible d'installer un nouveau cache au sein d'un SoC avec notre architecture de détection intégrée, et d'analyser la corrélation entre les signaux microarchitecturaux de ce cache et certains secrets du logiciel victime en utilisant les étapes 1 à 4.

Nous illustrons deux cas d'usages pour la conception des algorithmes de détection en utilisant cette démarche itérative dans le chapitre 5.

3.4 Utilisation du système

Dans cette section, nous présentons l'utilisation du système final pour la détection des attaques.

3.4.1 Vue globale

Une spécificité intéressante de notre architecture de détection est que la logique matérielle peut être mise à jour pendant le cycle de vie du système. Nous envisageons donc qu'un système final équipé de notre architecture de détection ait des périodes de détection dynamique des attaques séparées par des moments de mise à jour occasionnels tout au long de son cycle de vie, ainsi qu'un changement de génération à la fin de son cycle de vie, comme illustré dans la figure 3.5 :

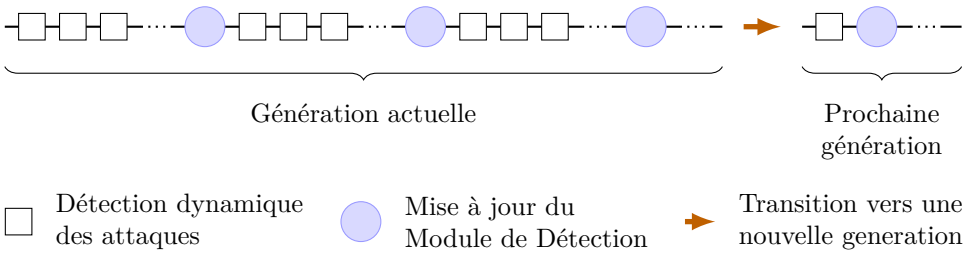


FIGURE 3.5 – Utilisation du système pendant son cycle de vie.

1. La *détection dynamique des attaques* (représentée par les carrés blancs) correspond aux périodes pendant lesquelles le MD fonctionne activement pour détecter des attaques. Cette période est la plus fréquente dans la durée de vie du système. Une présentation plus détaillée de cette période est fournie dans la section 3.4.2.
2. La *mise à jour du MD* (représentée par les cercles bleus) consiste à faire évoluer le MD pour s'adapter aux potentielles nouvelles variantes d'attaques. La mise à jour de la partie logicielle modifie le MDL situé dans le noyau ; la mise

à jour de la partie matérielle modifie ou ajoute des bitstreams contenant des logiques de détection à utiliser pour reconfigurer le MDM. Les mises à jour sont bien entendu plus rares que les périodes de détection dynamique d'attaques dans le cycle de vie. Il faut souligner que dans le cas où la taille de la structure reconfigurable ne peut plus contenir la totalité des logiques de détection devenues de plus en plus complexes, les algorithmes peuvent toujours être mis en œuvre en implémentant une partie dans le MDL. Notons également qu'il est possible d'ajouter des algorithmes de détection basés sur des signaux supervisés du système actuel pour détecter de nouvelles attaques. Les expériences réalisées semblent confirmer ce point, car nous montrons dans la section 5 que nous sommes capables de concevoir des détecteurs efficaces avec peu de signaux supervisés pour deux classes d'attaques complètement différentes.

3. Le *passage à la prochaine génération du système* (représentée par la flèche orange entre la génération actuelle et la prochaine génération) consiste à refaire la conception du système et produire un nouveau système, notamment en modifiant la configuration choisie dans la section 3.3, pour inclure de nouveaux signaux supervisés pertinents et augmenter la taille de la structure reconfigurable. Il se fait une seule fois à la fin du cycle de vie courant du système. Ce passage à la nouvelle génération permet d'améliorer plus efficacement la qualité de la détection par rapport à des mises à jour.

3.4.2 Détection dynamique

La période de détection dynamique est composée de trois phases principales : (1) la réinitialisation du MD qui est une phase de configuration des mécanismes de détection, (2) la phase d'exécution normale pour surveiller en continu des activités sur le système avec les mécanismes de détection activés, et (3) la phase de traitement des anomalies et des interruptions pour réagir aux activités suspectes détectées par le MD. Dans la suite, nous présentons plus en détail ces différentes phases dans la détection dynamique en les illustrant à l'aide de diagrammes de séquence représentatifs. Dans ces diagrammes, nous considérons qu'il y a un seul thread pour la partie logicielle (incluant le noyau, le programme d'utilisateur et le MDL), qui représente les exécutions sur un seul processeur ; la partie matérielle fonctionne en parallèle du processeur, et est illustrée comme un thread indépendant.

La figure 3.6 illustre la réinitialisation du MD. Tout d'abord, le MDL déclenche la reconfiguration du MDM en lui fournissant un fichier décrivant des logiques matérielles (`reconfigure()`) et le MDM se reconfigure suite à cette demande (`changeLogiques()`). Losque le MDM a terminé sa reconfiguration, le MDL initialise également ses valeurs internes pour se réinitialiser dans un état propre, apte à commencer la détection (`initialise()`). Ensuite, le MDL fournit les configurations nécessaires au MDM via le bus (`confParamsDetection()`), telles que les seuils de détection, les conditions d'interruptions et l'activation de la détection. La réinitialisation est déclenchée par le noyau au démarrage du système ou après une

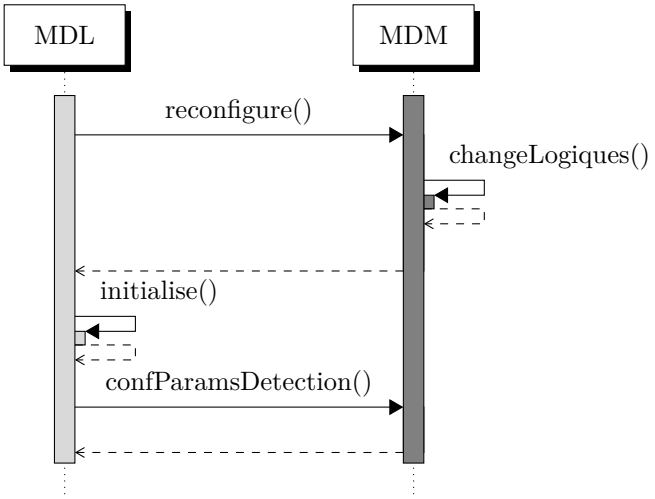


FIGURE 3.6 – Diagramme de séquence d'utilisation du système pour la détection dynamique des attaques (réinitialisation).

mise à jour, ou par le MDL lui-même en fonction du besoin des algorithmes de détection activés, autant de fois que nécessaire. La réinitialisation au démarrage du système est due à la nature volatile de la structure reconfigurable, les logiques reconfigurées étant effacées lorsque le système est réinitialisé. Pendant cette période de réinitialisation, la détection des attaques n'est pas assurée.

La figure 3.7 illustre le fonctionnement du MD pendant la phase d'exécution normale. Lorsque le MDM est activé par le MDL, il s'exécute en permanence, de façon indépendante du processeur. Le MDM reçoit à chaque cycle d'horloge des signaux supervisés venant des composants cibles par la connexion dédiée, et réalise des analyses à l'aide des logiques matérielles mises en place sur ces signaux (`analyseSignaux()`). Les résultats de ces analyses sont placés dans les registres MMIO ou dans la zone mémoire du MDM, prêts à être lus par le MDL. Ces analyses sont principalement effectuées pendant l'exécution des logiciels utilisateurs (`executeUtilisateur()`). Au moment du changement de contexte ou de l'exécution périodique du MDL (`executeMDL()`), le MDL peut optionnellement effectuer quelques actions selon le besoin. Plusieurs communications synchrones du MDL avec le MDM via le bus sont possibles, par exemple pour : (1) ajuster la configuration selon les besoins de sécurité du moment ; (2) mettre en pause la détection dynamique pendant l'exécution du MDL ; (3) être plus strict dans les critères de détection lorsque certains logiciels critiques sont exécutés (`confParamsDetection()`) ; (4) envoyer des valeurs pour aider à la détection, tel que le PID de la prochaine tâche à exécuter (`envoiInformation()`) ; ou (5) collecter les valeurs actuelles des résultats d'analyses de MDM (`demandeIoC(id)`). En fonction de ces résultats récupérés, le MDL peut effectuer quelques analyses plus poussées ou difficilement réalisables en matériel (`analyse()`). Cette boucle de traitements s'exécute jusqu'à l'arrivée d'évènements particuliers, comme la détection d'une tentative d'attaque

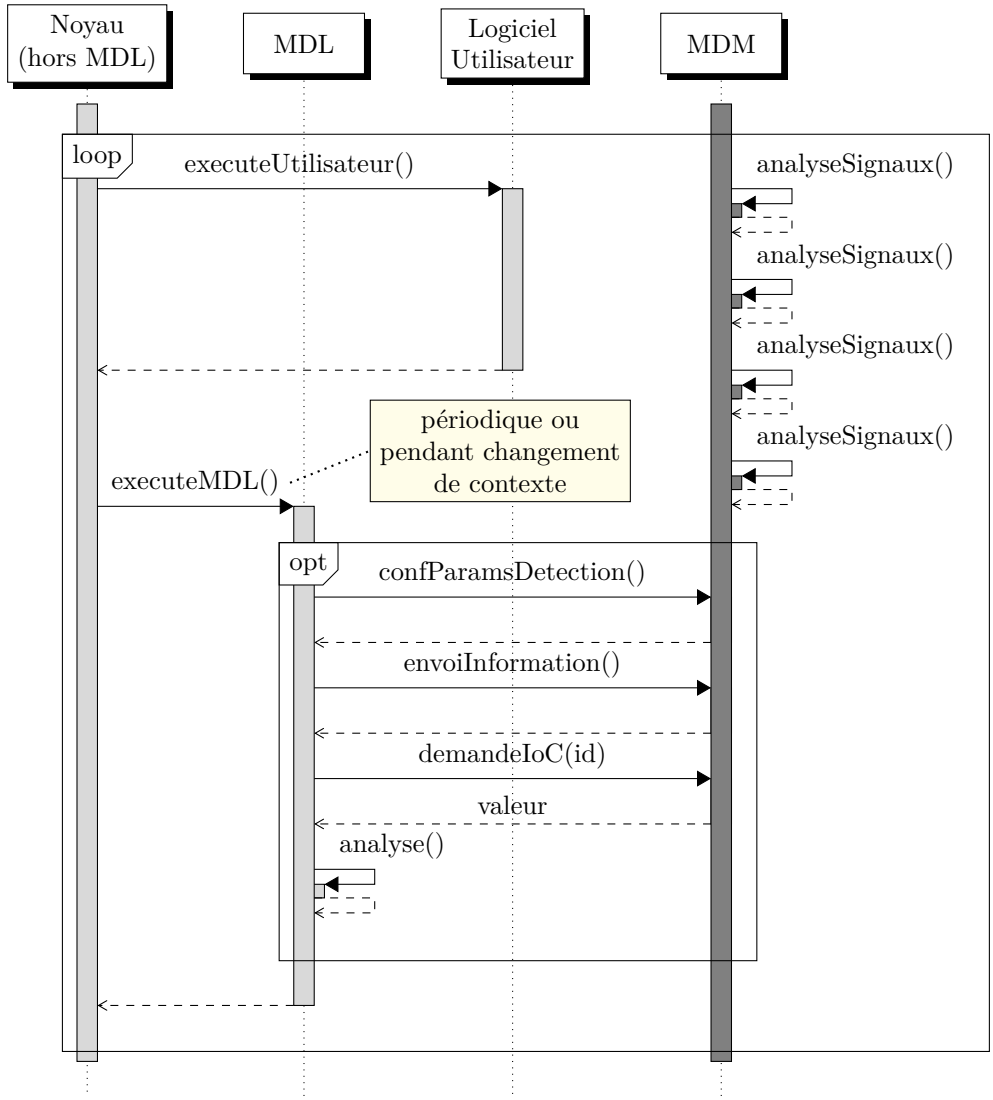


FIGURE 3.7 – Diagramme de séquence d'utilisation du système pour la détection dynamique des attaques (exécution). Abréviation : IoC : indice de compromission.

par le MDL, le déclenchement d'interruption, la réinitialisation, ou la mise à jour du MD.

La figure 3.8 illustre deux situations particulières qui peuvent se produire pendant la boucle normale de détection : (1) lorsqu'une anomalie est constatée et/ou qu'une action est requise pendant l'analyse du MDL (pendant `executeMDL()`), ou (2) lorsqu'une interruption est envoyée par le MDM pendant l'analyse continue des signaux (`interruption`). Dans le second cas, le logiciel utilisateur est interrompu, et le noyau appelle le MDL pour traiter l'interruption (`gereInterruption()`). Le MDL interroge ensuite le MDM pour connaître la cause de l'interruption (`demandeRaison()`), puis collecte des valeurs en lien avec cette interruption afin de mener une analyse (`demandeIoC(id)`). Dans les deux cas, le MDL doit dé-

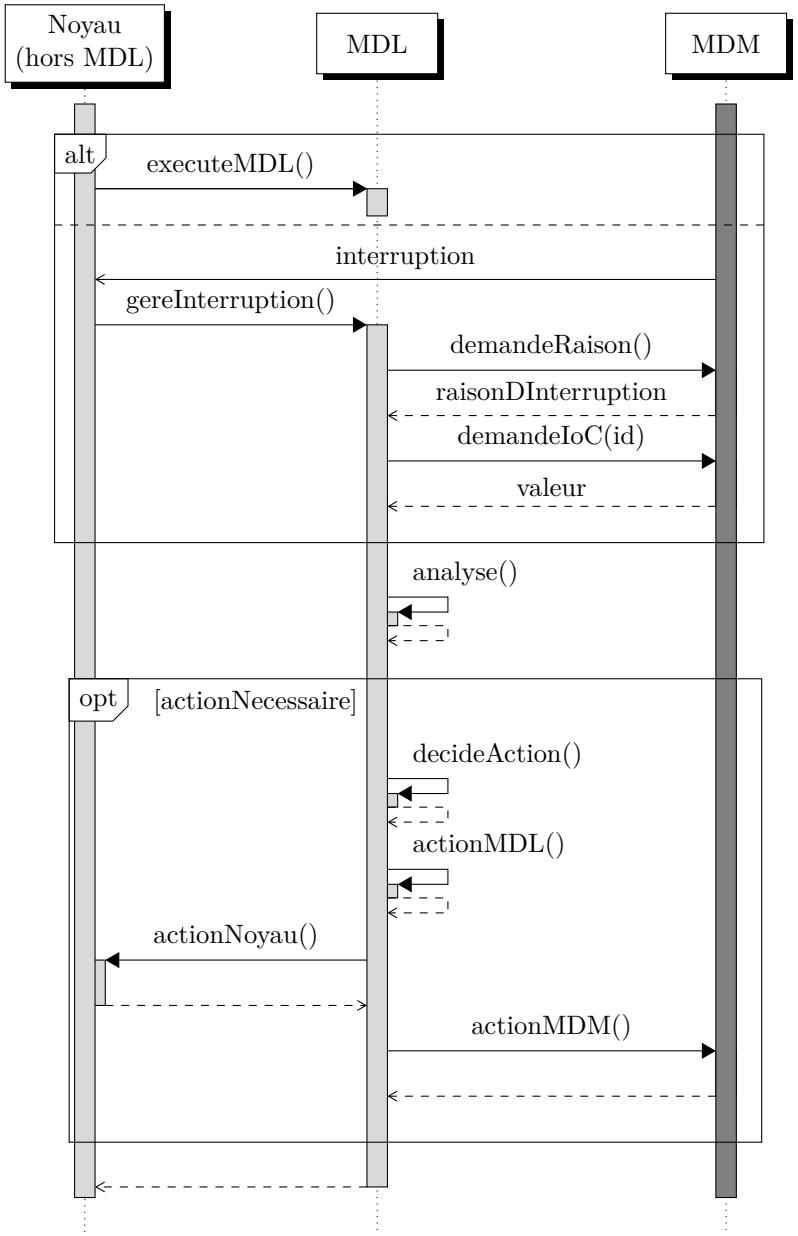


FIGURE 3.8 – Diagramme de séquence d'utilisation du système pour la détection dynamique des attaques (traitement d'une anomalie detectée par le MDL ou une interruption envoyée par le MDM).

cider s'il faut agir immédiatement (`analyse()`) et quelles actions entreprendre (`decideAction()`). Plusieurs types d'actions sont possibles : (1) des actions liées à la gestion interne du MDL, par exemple ajouter le logiciel suspect dans une liste noire pour mieux réagir à la prochaine exécution du logiciel (`actionMDL()`) ; (2) des actions à exécuter par le noyau, par exemple arrêter le logiciel suspect ou renforcer l'isolation entre ce logiciel et le reste du système (`actionNoyau()`) ; ou (3)

des actions à exécuter par le MDM, par exemple effectuer une reconfiguration pour changer les logiques de détection mises en place (`actionMDM()`).

3.5 Conclusion

Dans ce chapitre, nous avons présenté une architecture de détection dynamique des attaques, basée sur l'analyse des signaux issus de la microarchitecture, et des méthodes associées pour la conception et l'utilisation d'un système cible équipé de cette architecture. Cette architecture n'est pas limitée à une ISA spécifique de machine cible, et peut être adaptée à différents systèmes.

L'architecture proposée inclut un MD (matériel et logiciel) et des sondes dans le système cible. Le MDM est reconfigurable, afin d'offrir l'adaptabilité des logiques de détection durant toute la durée de vie du système. Les sondes permettent de superviser des signaux microarchitecturaux, et les valeurs de ces signaux sont envoyées au MDM via une connexion dédiée. Les connexions entre le MDM et le reste du système sont non reconfigurables, afin de fonctionner à une haute fréquence. Le MDL se situe dans le noyau, elle est le point central qui communique avec le MDM et qui prend des décisions finales.

Certains paramètres de l'architecture de détection sont laissés à la décision des concepteurs du système. Ces paramètres concernent principalement la partie non reconfigurable de l'architecture comme la position des sondes, les signaux supervisés, les connexions dédiées à chaque signal et la taille de la structure reconfigurable. Nous avons proposé une méthodologie pour aider à intégrer notre architecture dans le système cible et choisir intelligemment ces paramètres. Nous avons également présenté l'utilisation du système lors des différentes étapes de son cycle de vie.

Dans la suite, nous présentons l'implémentation d'un prototype de notre proposition, ainsi que des cas d'usage pour la configuration et l'utilisation de cette architecture pour la détection des attaques.

Chapitre 4

Implémentation

Sommaire

4.1 Présentation générale 65
4.1.1 Plateforme d'expérimentation 66
4.1.2 Vue d'ensemble 67
4.2 Matériel 68
4.2.1 Système de base 68
4.2.2 Signaux supervisés et connexions 70
4.2.3 Module de détection matériel 73
4.3 Logiciel 75
4.3.1 Configuration du noyau 75
4.3.2 Module de détection logiciel 75
4.4 Conclusion 78

Ce chapitre décrit le prototype que nous avons construit. Après une présentation générale, nous présentons plus précisément des éléments importants de la partie matérielle et de la partie logicielle. Les objectifs principaux de ce prototype sont : (1) démontrer la faisabilité de notre proposition et sa capacité à s'adapter à différentes attaques ; (2) permettre à la communauté d'utiliser ce prototype pour concevoir des logiques de détection sans être limitée par la collecte de signaux, pour mieux comprendre le comportement de la microarchitecture sous l'impact des logiciels dans un environnement réel. Le prototype est basé sur l'architecture RISC-V, dont la partie matérielle est un SoC généré principalement par Chipyard [Amid 2020], et dont la partie logicielle est un noyau Linux. Ce choix du système de base a permis de faciliter le paramétrage, l'ajout des sondes, le codage de la logique de détection, tout en représentant un environnement d'exécution réel (un vrai matériel avec un noyau couramment utilisé). MATANA, la version du prototype proposée dans l'article [Mao 2022], est disponible à l'adresse suivante : `https://gitlab.laas.fr/matana`.

4.1 Présentation générale

Dans cette section, nous présentons la plateforme d'expérimentation utilisée, ainsi qu'une vue d'ensemble du prototype construit et ses propriétés générales.

4.1.1 Plateforme d'expérimentation

Afin de disposer d'une certaine flexibilité, notre choix s'est porté sur une carte d'évaluation ML605 équipée d'un Virtex-6 XC6VLX240T-1FFG1156 FPGA. Ce FPGA dispose de 37,680 *slices* (chaque *slice* est composé de quatre LUTs et huit FFs) [Xilinx 2015], et le système de base RISC-V sans contrôleur de mémoire DDR occupe 25% des *slices*. La carte possède une mémoire DDR3 disposant de 512 Mo de RAM, qui peut être utilisée comme la mémoire principale du système. La plateforme d'expérimentation comprend également une machine hôte à la fois pour reconfigurer le FPGA via le port JTAG, et pour être utilisée comme un terminal via le port UART. Cette machine hôte sert principalement pendant la phase d'expérimentation et est destinée à être supprimée dans le système final. La figure 4.1 présente notre plateforme d'expérimentation.

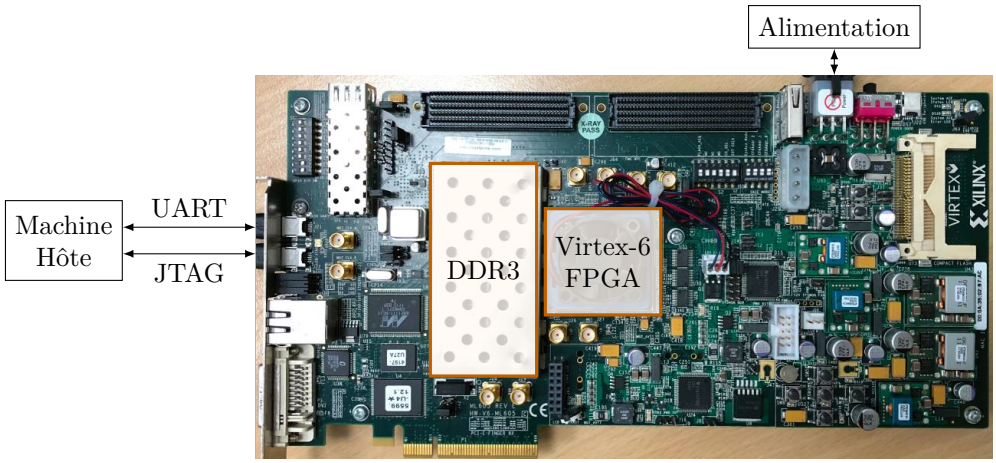


FIGURE 4.1 – Plateforme d'expérimentation composée d'une carte d'évaluation ML605 et d'une machine hôte.

Le FPGA est utilisé pour exécuter le SoC avec l'architecture de détection intégrée. Le SoC est divisé en deux parties : une partie non-modifiable (cœur de processeur, bus, caches, ...) et une partie modifiable (logique de détection). Les deux parties sont intégrées sur un même FPGA dans le prototype pour la simplicité de la mise en place, mais seule la partie modifiable est destinée à être intégrée sur un matériel reconfigurable tel qu'un FPGA dans le système final. Précisons que l'utilisation du FPGA est un choix délibéré pour le prototype, mais n'est pas une obligation pour le système final. On peut même imaginer un composant similaire à un FPGA, mais avec des blocs de base contenant des éléments logiques et séquentiels adaptés aux besoins de la détection.

La machine hôte de notre prototype est équipée d'un processeur Intel i6-8250U CPU avec un système x86-64 Ubuntu (18.04) sous forme de machine virtuelle. La génération du code et la compilation du code sont également réalisées sur cette machine, et la génération du bitstream (fichier contenant la configuration du FPGA) a été déportée sur un serveur plus puissant.

4.1.2 Vue d'ensemble

Le prototype est configuré pour fonctionner en 64 bits, il peut être porté sur une architecture 32 bits moyennant quelques adaptations.

La partie matérielle, comprenant un cœur de processeur RISC-V à exécution ordonnée (Rocket [Asanović 2016]) ainsi que les modifications apportées pour intégrer le système de détection, sont principalement générées par le générateur Chipyard depuis le langage Chisel [Bachrach 2012] sous forme de code Verilog. Une fois générés, des composants plus spécifiques à notre plateforme cible (la ML605) sont ajoutés (en général des IPs propriétaires de Xilinx). L'ensemble des logiques matérielles décrites en Verilog est ensuite synthétisé par l'outil Xilinx ISE pour générer le bitstream. En cas de réutilisation du prototype sur une autre plateforme, cette dernière étape ainsi que certaines IPs doivent faire l'objet d'une adaptation.

La partie logicielle qui contient le noyau Linux et le MDL, est compilée avec GCC sur la machine hôte. Le MDL est principalement codé en C, avec un script Python qui permet de générer des codes pour manipuler les registres MMIO selon leur déclaration dans le matériel. Nous avons mis en place une interface en espace utilisateur (appelée *interface MDL* dans la suite), pour communiquer avec le MDL dans le noyau (appelé *MDL noyau* dans la suite), afin de faciliter le contrôle et la récupération des données du MD. Le prototype, incluant le système de base et les modifications effectuées, est représenté dans la figure 4.2.

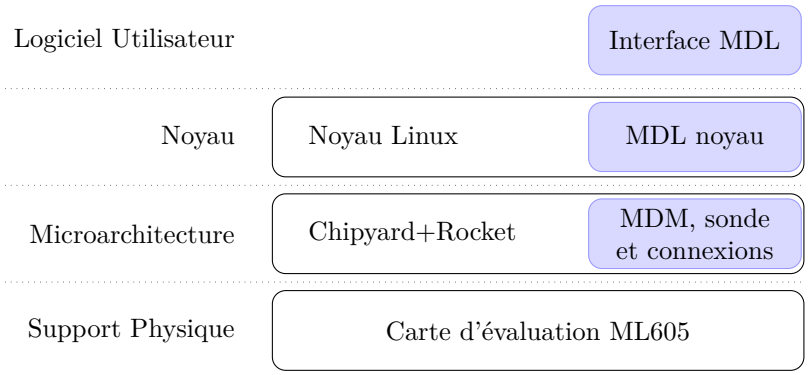


FIGURE 4.2 – Structure globale du prototype, incluant le système de base et les modifications apportées à chaque partie.

Pour faire fonctionner le système sur la plateforme d'expérimentation, il faut d'abord programmer le FPGA sur la carte ML605 avec le bitstream via l'interface JTAG depuis la machine hôte. Ensuite, le fichier binaire, incluant le noyau Linux instrumenté et des logiciels utilisateur, notamment l'interface MDL et les logiciels qui permettent de valider l'efficacité des algorithmes de détection, est copié dans la mémoire DDR de la carte à l'aide de l'interface UART. L'exécution du noyau Linux commence en mémoire DDR et l'utilisateur sur la machine hôte peut interagir avec le système grâce à un terminal depuis l'interface UART.

Le cœur de processeur ainsi que la plupart des composants sur le FPGA sont

cadencés à une fréquence de 50 MHz. Nous avons mis en place un paramètre $divN$ qui permet de générer une division d'horloge pour le MDM, afin de simuler l'utilisation du MDM dans un domaine d'horloge différent de celui du processeur. Dans nos expérimentations présentées au chapitre 5, nous avons étudié deux scénarios extrêmes : un premier sans division d'horloge ($divN = 1$) représentant le cas où la logique de détection est câblée dans le processeur ; et un deuxième avec une division d'horloge de 16 ($divN = 16$) correspondant à un scénario avec un processeur intégrant un module de détection totalement reconfigurable à base de FPGA.

Au total, la partie matérielle contient environ 40 lignes de modifications de code Chisel sur le générateur Chipyard et le cœur de processeur Rocket pour la déclaration des sondes et une première partie des connexions, 1300 nouvelles lignes de code Chisel pour mettre en place l'architecture de détection contenant cinq logiques de détection simples et assez différentes (chaque logique de détection contient plusieurs indicateurs d'attaque). Ces lignes de Chisel ajoutées permettent de générer plus de 20000 lignes de code Verilog dans le cas où deux logiques de détection sont activées avec une division d'horloge de 16. La partie logicielle contient 15 lignes de modification de code C et de Makefile sur le noyau Linux de base, 500 lignes de code C non générées et 100 lignes de code Python pour le MDL sans algorithme de détection.

Dans ce qui suit, nous décrivons plus en détail les parties matérielles et logicielles de notre prototype, à la fois le système original et les modifications que nous lui avons apportées.

4.2 Matériel

4.2.1 Système de base

Notre système matériel de base est principalement généré à partir de Chipyard v1.3. Ce choix nous permet d'avoir une grande souplesse concernant les différents processeurs cibles que l'on peut tester (multicœur, hiérarchie de cache, exécution dans le désordre, ...). De plus, Chipyard est open source, activement maintenu et facilement paramétrable.

Cette souplesse de configuration est possible à l'aide du langage Chisel. Basé sur Scala, il permet de concevoir et générer une logique matérielle à partir d'un langage orienté objet et orienté programmation fonctionnelle : il génère tout d'abord un format intermédiaire de description matérielle nommée FIRRTL [Izraelevitz 2017], et la description FIRRTL peut ensuite être traduite en code Verilog. Comparé à l'utilisation directe des langages de description matérielle classiques comme VHDL et Verilog, sa syntaxe est beaucoup plus compacte et facile à comprendre et à utiliser, comme on peut le constater dans les extraits de codes fournis dans la suite. Chisel est ainsi adapté à notre besoin de créer et d'ajuster rapidement des logiques de détection.

Différents cœurs de processeur peuvent être utilisés dans Chipyard. Nous avons choisi d'utiliser le cœur Rocket dans notre prototype afin de simplifier l'étude, mais

la démarche reste la même pour n'importe quel processeur. En particulier, des travaux préliminaires sont menés sur le cœur BOOM [Zhao 2020], qui est un cœur bien plus complexe avec exécution dans le désordre.

La structure simplifiée du système de base et les modifications faites sont présentées dans la figure 4.3. Le système de base est configuré en 64 bits, un seul cœur Rocket de taille moyenne, avec un contrôleur d'UART, un contrôleur d'interruption et un cache L2 de taille 32 Ko (64 sets, 8 lignes par set, 64 octets par ligne) connecté au contrôleur de DDR propriétaire de Xilinx. La configuration du cœur Rocket de taille moyenne est faite pour respecter un compromis entre la taille nécessaire du matériel pour l'implémenter et les fonctionnalités supportées. Il supporte de la mémoire virtuelle, des instructions de multiplication et division, des instructions atomiques et des instructions compressées. Il dispose d'un cache d'instruction L1 (L1I sur la figure) et d'un cache de données L1 (L1D sur la figure) de taille 4Ko chacun (64 sets, 1 ligne par set, 64 octets par ligne). Il ne possède pas d'unité à virgule flottante (FPU). En dehors de la mémoire ROM de démarrage (BootROM) et de la mémoire principale DDR, une mémoire de faible capacité sur le FPGA (BlockRAM sur la figure) est prévue pour réaliser des tests simples du système et traiter la réception du fichier binaire de noyau par UART et l'écriture de son

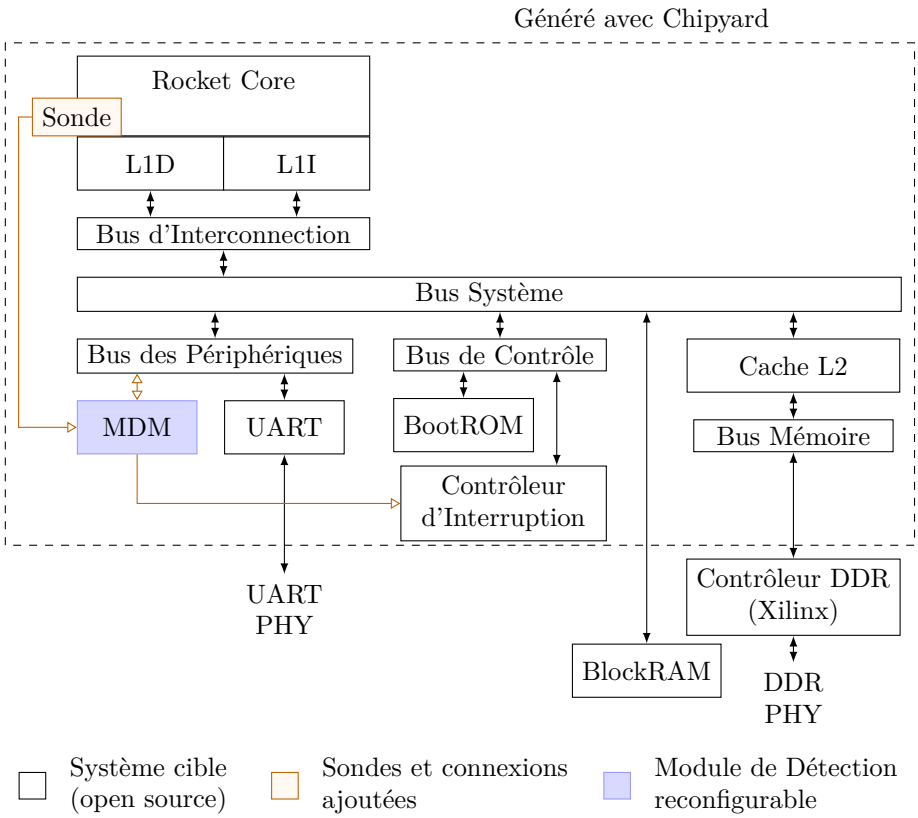


FIGURE 4.3 – Architecture matérielle du prototype, basée sur un SoC généré par Chipyard.

contenu dans le DDR. Ces paramètres du système peuvent être ajustés facilement si nécessaire, soit en Chisel pour la partie générée avec Chipyard, soit en macros Verilog pour le reste.

Nos modifications pour intégrer l'architecture de détection sont également représentées sur la figure 4.3 : une sonde placée dans le cœur Rocket, trois connexions (sauf l'esclave de reconfiguration) et un MDM. Elles sont toutes écrites en Chisel et intégrées dans le framework Chipyard pour la facilité de paramétrage et d'interconnexion. La sonde et les connexions sont compatibles avec les systèmes multicœurs, nous pouvons simplement dupliquer les algorithmes de détection pour supporter la détection dans chaque cœur séparément. Cependant, nous n'avons pas, dans nos travaux, conçu des algorithmes de détection adaptés aux multicœurs, qui prendraient en considération les relations entre les cœurs.

Les éléments les plus importants de la partie matérielle sont fournis dans les sections suivantes, le lecteur est invité à consulter le code en ligne pour plus de détail.

4.2.2 Signaux supervisés et connexions

Dans ce prototype, nous avons placé une seule sonde dans le cœur Rocket. Nous avons choisi cet emplacement de la sonde, car c'est l'endroit où beaucoup de signaux et d'informations sont accessibles : signaux internes de chaque étage du pipeline et de chaque unité d'exécution, les registres de contrôle et de statut (en anglais *Control Status Register*, CSR), les HPCs, les adresses et les valeurs échangées avec les caches L1 et leur adresse physique correspondante.

Dans la suite, nous présentons la démarche avec des extraits de code correspondants pour mettre en place cette sonde, surveiller des signaux, et les connecter jusqu'au MDM à basse fréquence. Cette démarche reste la même pour placer d'autres sondes et superviser d'autres signaux, ainsi, nous considérons que cette seule sonde placée dans le prototype n'impacte pas la généricité de notre approche. Elle est composée des principales étapes suivantes :

1. *Choisir des signaux supervisés.* À titre d'exemple, nous choisissons de superviser deux signaux de l'étage "Write Back" du pipeline de Rocket : l'instruction elle-même dans cet étage (32 bits) et la validité de cette instruction (1 bit). Nous avons en effet constaté que les instructions exécutées sont importantes, et nous avons observé qu'il y a moins d'aléas dans l'étage "Write Back" que dans l'étage "Execute", tout en permettant de collecter avec exactitude la liste des instructions exécutées. Pour des instructions compressées, nous avons vérifié qu'elles sont décodées en instructions non compressées au premier étage du pipeline.
2. *Déclarer l'interface que nous devons ajouter dans le composant cible*, comme illustré dans le listing 4.1. Nous avons besoin de faire sortir des signaux supervisés, ainsi les portes de l'interface sont du type "Output". Dans l'interface `MatanaCoreIO`, la porte indiquant l'instruction (`inst_data`) est un

entier non signé de 32 bits, et la porte indiquant la validité (`inst_valid`) est un booléen.

Listing 4.1 – Exemple de déclaration d'interface pour deux signaux supervisés, qui doivent être ajoutés dans l'interface du composant cible.

```
class MatanaCoreIO extends Bundle {
  val inst_data = Output(UInt(32.W))
  val inst_valid = Output(Bool())
}
```

3. *Inclure l'interface définie dans l'étape précédente dans la liste des interfaces du composant cible*, comme illustré dans le listing 4.2. Dans notre cas, le composant cible est le cœur de processeur, nous déclarons qu'il a une nouvelle interface du type `MatanaCoreIO`, nommée `matana`.

Listing 4.2 – Exemple de déclaration de nouvelle interface dans la liste des interfaces du cœur de processeur.

```
trait HasCoreIO extends HasTileParameters {
  implicit val p: Parameters
  val io = new CoreBundle()(p) with
    HasExternallyDrivenTileConstants {
    ... // Other core interface
    val matana = new MatanaCoreIO
  }
}
```

4. *Connecter l'interface avec les signaux à superviser à l'intérieur du composant cible*, comme illustré dans le listing 4.3. Nous distinguons deux lignes représentant des connexions matérielles de Chisel (symbole `:=`). À droite de ce symbole est indiqué le signal interne de l'étage "Write Back" que nous souhaitons superviser, et à gauche de ce symbole est indiquée la porte déclarée dans l'interface `matana`.

Listing 4.3 – Exemple de connexion de l'interface de la sonde avec les signaux supervisés dans le cœur Rocket.

```
class Rocket(tile: RocketTile)(implicit p: Parameters) extends
    CoreModule()(p)
    with HasRocketCoreParameters
    with HasCoreIO {
  ... // Other code of Rocket Core, pipeline, CSR, etc

  io.matana.inst_data := wb_reg_inst
  io.matana.inst_valid := wb_reg_valid
}
```

5. *Connecter l'interface au MDM.* À l'aide de deux lignes de code Chisel, ces signaux supervisés ont pu sortir de la tuile de processeur (contenant notamment le cœur de processeur et les caches L1). Puis six lignes de code Chisel permettent de relier ces signaux — venant de chaque cœur si de multiple instances de cœur Rocket sont déclarées — au MDM avant changement de domaine d'horloge.
6. *Choisir une technique de changement de domaine d'horloge pour chaque signal supervisé*, comme illustré dans le listing 4.4. Nous avons choisi la connexion parallèle, instrumentée dans la fonction Chisel `seq2par()` que nous avons définie (sa définition est illustrée dans le listing 4.5). Les signaux seront d'abord placés dans des mémoires tampon pendant *divN* cycles d'horloge, puis passés par deux étages de registres afin de stabiliser les signaux dans le domaine d'horloge lent. À ce moment, les signaux "lents" sont prêts à être traités par le MDM.

Listing 4.4 – Exemple de connexion des signaux supervisés avant d'entrer dans le MDM pour le changement du domaine d'horloge.

```
1 io.tiles.zipWithIndex.map { case (tile, i) =>
2   val fast_inst_data = MatanaSync.seq2par(divN, tile.inst_data)
3   val fast_inst_valid = MatanaSync.seq2par(divN, tile.inst_valid)
4   // Registered at slow side
5   withClock(slow.clock) {
6     detection.io.packs(i).vec_inst_data := RegNext(RegNext(
      fast_inst_data))
7     detection.io.packs(i).vec_inst_valid := RegNext(RegNext(
      fast_inst_valid))
8   }
9 }
```

Listing 4.5 – Fonction Chisel pour la connexion parallèle sur un signal du type "Data".

```
1 // Sequential to parallel logic, with most recent signal (input) at
      Vec(0)
2 def seq2par[T <: Data](clockDiv: Int, gen: T) : Vec[T] = {
3   val par = RegInit(VecInit(Seq.fill(clockDiv)(0.U.asTypeOf(
      chiselTypeOf(gen)))))
4   par.foldLeft(gen) { case (in, reg) => {
5     reg := in
6     reg
7   }}
8   par
9 }
```

Les deux autres connexions, c'est-à-dire la ligne d'interruption et le bus sont réalisées grâce aux systèmes de gestion d'interconnexions et des fonctions de changement de domaine d'horloge disponibles dans Chipyard. Elles nécessitent environ

dix lignes de code Chisel au total.

Dans le prototype actuel, nous n'avons pas encore implémenté la connexion de reconfiguration permettant de reconfigurer dynamiquement le MDM depuis le cœur Rocket. Cependant, ce travail ne constitue pas un défi scientifique en soi, il s'agit uniquement d'un travail d'ingénierie, qui a déjà été réalisé dans les travaux de recherche [Vipin 2018], et qui est déjà présent dans certains usages commerciaux, comme présenté dans la section 2.3. Nous avons cependant validé et testé la reconfiguration de la partie MDM de notre prototype à l'aide de l'outil Xilinx PlanAhead, tout en gardant le reste du système non modifié sur le FPGA. La reconfiguration a parfaitement fonctionné et le système s'est comporté comme prévu, avec une mise à jour des algorithmes de détection présents dans le MDM. Nous considérons ainsi que l'absence de la connexion de reconfiguration dans notre prototype ne remet pas en cause la validité de notre approche.

4.2.3 Module de détection matériel

Nous avons construit le MDM de manière modulaire, c'est-à-dire que chaque logique de détection est déclarée dans une classe Chisel séparée. Cela permet de gérer les paramètres, le fonctionnement interne et les registres MMIO de chaque algorithme indépendamment, ainsi que l'activation d'une ou plusieurs logiques en même temps.

Le listing 4.6 contient un extrait de la déclaration d'une logique de détection. Dans cet exemple, la logique de détection est très simple : "à chaque fois que le signal d'entrée `in.some_atk` est à 1, et que la détection est active par `in.isMonitoring`, alors on compte une tentative d'attaque, enregistrée dans le signal `atk_example`". Cette logique de détection expose trois registres MMIO liés à `atk_example`, générés par la fonction `countEventThreshAlarm()` : un registre à lecture seule qui compte le nombre de fois où `atk_example` est à 1, un registre de seuil de détection, et un registre d'alarme qui est positionné à 1 lorsque le compteur dépasse le seuil. Cette alarme peut être incluse dans les conditions de génération d'interruption. Les paramètres propres à la génération de cette logique de détection sont accessibles dans la variable `params`, et les paramètres globaux pour la génération du MDM (qui pourraient être déterminés en fonction de la configuration du système de base, par exemple *divN*) sont accessibles dans la variable `mp`.

Dans notre prototype, nous souhaitons illustrer l'utilisation des informations accessibles par le noyau pour améliorer la précision de la détection des attaques. Par exemple, on peut choisir d'ajouter l'information PID, le numéro de processus actuel exécuté dans le processeur. Cette information est envoyée par le MDL au changement de contexte. Le listing 4.7 présente le code correspondant à la gestion de PID dans le MDM. Dans ce code, nous pouvons remarquer la déclaration de trois registres internes : `nextPid`, `currentPid`, et `monitorPidMin`. Le registre `nextPid` est renseigné par l'écriture de la valeur dans l'adresse de l'offset `0x10` par rapport au plage d'adressage du MDM, puis, au cycle prochain, le registre `currentPid` prend la valeur de `nextPid`. Nous avons utilisé une condition `monitorPidMin` afin de filtrer

Listing 4.6 – Exemple d'une simple logique de détection dans le MDM.

```
class DetectExampleInternal(params: DetectExampleParams)(implicit
    mp: MatanaParams) extends DetectExample {
  // Logic that detect the attack
  val atk_example = Wire(Bool())
  atk_example := in.some_atk && in.isMonitoring

  // The regmap for bus
  override def regmap(offset: Int) =
    RegmapUtil.countEventThreshAlarm(atk_example, in.resetCounters,
     offset, "AtkExample") ++
    Nil
}
```

les activités dont le PID est inférieur à cette valeur (typiquement, les activités liées au noyau lui-même).

Listing 4.7 – Gestion de PID et la déclaration des registres MMIO associés dans le MDM.

```
val nextPid = RegInit(0.U(mp.pidWidth.W))
val currentPid = RegInit(0.U(mp.pidWidth.W))
val monitorPidMin = RegInit(60.U(mp.pidWidth.W))
currentPid := nextPid

// Monitor all pid >= monitorPidMin if detectEnable is true
isMonitoring := detectEnable && (currentPid >= monitorPidMin)

val regmap0 = Seq(
  0x10 -> Seq(
    RegField(mp.pidWidth, r=currentPid, w=nextPid,
      RegFieldDesc("pid",
                   "Write next PID, or read actual PID",
                   reset=Some(0)))),
  0x14 -> Seq(
    RegField(mp.pidWidth, monitorPidMin,
      RegFieldDesc("monitorPidMin",
                   "Monitor all process with PID >= monitorPidMin",
                   reset=Some(60))))
)
```

Nous pouvons constater que les logiques, que ce soit pour les paramètres du système de base, les signaux supervisés, les connexions ou le MDM, sont exprimées de façon simple à comprendre et à reproduire. Nous pouvons ainsi les faire varier rapidement et étudier différentes combinaisons de ces logiques. Nous présentons des signaux supervisés et des logiques pour la détection de deux classes d'attaque différentes dans le chapitre 5. Ils sont tous construits en suivant les mêmes démarches que les exemples simples fournis dans ce chapitre.

4.3 Logiciel

4.3.1 Configuration du noyau

Nous avons choisi d'utiliser un noyau Linux, car il est ouvert, utilisé largement à la fois par des systèmes à usage spécifique et à usage générique, et suffisamment léger pour être adapté à notre prototype. La version précise du noyau Linux choisi est 4.15, généré sur la base du projet freedom-u-sdk v1.0 [SiFive 2021]. Les versions 5.x ne sont pas utilisées à cause de la taille du binaire final qui est plus élevée. Le compilateur croisé utilisé, pour compiler le noyau et les autres logiciels en RISC-V qui s'exécutent sur notre plateforme d'évaluation, est la version 9.2.0 de la chaîne d'outils GCC sur notre machine hôte x86-64. Pour le système 64 bits, nous avons choisi d'utiliser l'option de compilation `rv64imac`, c'est-à-dire que notre cœur Rocket supporte les manipulations d'entiers, la multiplication et la division, les opérations atomiques et les instructions compressées. La manipulation directe des valeurs flottantes n'est pas utilisée, car notre configuration du cœur Rocket n'a pas de FPU. Grâce à l'arbre de périphériques (en anglais *device tree*) généré par Chipyard, le noyau Linux reconnaît donc avec succès le système matériel, notamment la position du contrôleur UART et la mémoire principale DDR, ainsi que la plage d'adresses occupée par le MDM.

4.3.2 Module de détection logiciel

Dans notre prototype, le MDL est divisé en deux parties, le MDL noyau, qui est le composant central du MDL situé dans le noyau Linux, et l'interface MDL, qui se charge d'analyser l'entrée du terminal et de communiquer avec le MDL noyau. La présence de l'interface MDL permet de configurer et d'extraire des données du MDL et du MDM, et d'effectuer des analyses post-exécution manuellement, sans avoir à régénérer l'image du noyau et à redémarrer le système. L'interface MDL facilite ainsi notre interaction avec le MD et la conception d'algorithmes de détection. Le MDL noyau est compilé avec le noyau Linux sous la forme d'un pilote du MDM. Il est déclaré comme un périphérique en mode caractère, il communique avec l'interface MDL à l'aide d'`ioctl`, et il est le gestionnaire des interruptions envoyées par le MDM. L'interface MDL est compilée séparément du noyau, et, pour des raisons de simplicité, elle est exécutée sous les mêmes privilèges que le reste du logiciel de notre prototype. Dans le système final, l'interface MDL doit être exécutée à un niveau de privilège élevé, accessible uniquement par l'administrateur système, afin de se protéger contre les attaques.

Le MDL que nous avons implémenté dans ce prototype permet de réaliser les actions suivantes :

1. *Effectuer des analyses des données pendant l'exécution.* Une fonction du MDL est appelée à chaque changement de contexte, et des traitements sur les données peuvent être effectués à cet instant. Dans notre étude de cas, nous n'avons pas effectué des analyses pendant l'exécution dans le logiciel,

car les simples analyses dans le MDM permettent déjà d'obtenir de bons résultats.

2. *Fournir des informations supplémentaires pendant l'exécution.* Dans la fonction MDL appelée lors du changement de contexte, nous récupérons la structure contenant les informations sur le processus suivant et fournissons le PID du processus suivant au MDM.
3. *Gérer les interruptions.* Nous déclarons le MDL noyau comme le gestionnaire d'interruptions émises par le MDM. Ainsi, à chaque interruption du MDM, une fonction du MDL noyau est appelée par le noyau Linux. Cette fonction est capable de communiquer avec le MDM, et d'agir selon les besoins. Dans notre phase d'analyse, nous avons collecté les données de l'exécution complète des logiciels légitimes et malveillants, de façon à pouvoir les comparer et obtenir par exemple, la relation entre le seuil de détection choisi et le taux de faux positifs associé à ce seuil. Ainsi, les interruptions n'ont pas été utilisées dans nos résultats. En revanche, elles sont intégrées dans le prototype afin de configurer le système pour qu'il puisse réagir durant une attaque.
4. *Configurer le MDM.* Nous configurons principalement le MDM manuellement à travers l'interface MDL dans notre prototype, avant et après l'exécution d'un logiciel sous test. Cela permet par exemple de tester l'impact de différentes configurations sur la détection, ou d'activer la détection uniquement lorsque des logiciels spécifiques sont exécutés.

Les registres MMIO du MDM peuvent être lus ou modifiés par l'utilisateur via l'interface MDL, comme illustré dans la figure 4.4. Dans le détail, la commande utilisateur est tout d'abord traduite en appel `ioctl` vers le MDL noyau ; le MDL noyau va ensuite effectuer l'opération vers le bon registre en convertissant la requête en adresse physique ; et dans le cas d'une opération de lecture, renvoyer le résultat à l'utilisateur.

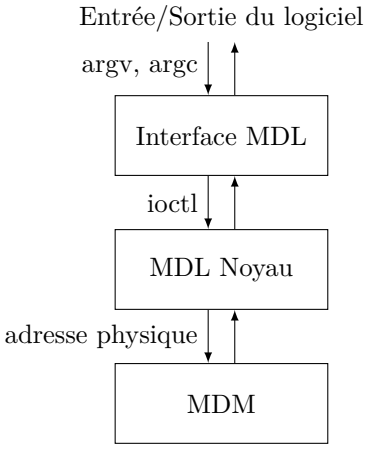


FIGURE 4.4 – Passage d'une commande depuis le terminal jusqu'au MDM, et retour de résultats.

Afin d'illustrer l'interaction entre le MDL noyau et le noyau Linux, poursuivons l'exemple de l'envoi du PID par le MDL lors du changement de contexte. Pour ce faire, nous avons intégré dans l'ordonnanceur du noyau Linux un appel à une fonction du MDL noyau à chaque changement de contexte. Cette fonction prend comme paramètre une structure faisant référence au processus suivant, et met à jour le registre correspondant au PID du MDM à l'adresse `reg_base_addr + MATANA_REG_PID`, comme illustré dans les listings 4.8 et 4.9. Durant nos expérimentations, nous avons observé que la valeur PID alterne fréquemment entre le PID 0 (l'ordonnanceur du noyau) et le PID du logiciel en cours d'exécution. Pour filtrer l'activité du noyau, il est donc important de mettre une valeur `monitorPidMin` > 0.

Listing 4.8 – Appel d'une fonction de MDL noyau au début de la fonction de changement de contexte de l'ordonnanceur du noyau Linux, avec comme paramètre le structure représentant le prochain processus à exécuter.

```
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
{
#if IS_ENABLED(CONFIG_MATANA_DRIVER)
    matana_context_switch(next);
#endif

    ... // Other code of context_switch
}
```

Listing 4.9 – Fonction du MDL noyau appelée dans le changement de contexte. Cette fonction extrait le PID du processus suivant et l'envoie au MDM si la détection par MD est active.

```
void matana_context_switch(struct task_struct *task) {
  if (matana_detect_enable) {
    // Write next PID to MMIO register
    uint32_t pid = task->pid;
    writel(pid, (reg_base_addr + MATANA_REG_PID));
  }
}
```

Dans la pratique, le nombre et la nature des registres MMIO varient fréquemment dans le MDM : l'ajout d'un nouveau paramètre, une nouvelle logique ou un indicateur de détection, etc. Il est donc important de pouvoir générer automatiquement les codes de gestion liés à ces registres : les logiques pour analyser les arguments à l'entrée de l'interface MDL ; la déclaration, l'utilisation des commandes `ioctl` correspondantes ; la relation entre les commandes `ioctl` et l'adresse physique des registres dans le MDM, etc. Heureusement, le générateur Chipyard génère aussi un fichier JSON par dispositif qui décrit tous les registres déclarés. Par exemple,

le listing 4.10 est un extrait du fichier JSON des deux registres déclarés dans le listing 4.7. Notre prototype inclut ainsi un script Python qui permet d'analyser ce fichier JSON et générer les codes nécessaires, avec la commande à envoyer à l'interface MDL et le nom des variables internes dérivées du nom des registres.

Listing 4.10 – Extrait d'une partie du fichier JSON qui décrit les informations des registres MMIO relatives à la gestion de PID.

```
{
  "peripheral" : {
    "displayName" : "deviceAt0x4000000",
    "baseAddress" : "0x4000000",
    "regfields" : [ {
      "pid" : {
        "byteOffset" : "0x10",
        "bitOffset" : 0,
        "bitWidth" : 32,
        "name" : "pid",
        "description" : "Write next PID, or read actual PID.",
        "resetValue" : 0,
        "accessType" : "RW"
      }
    }, {
      "monitorPidMin" : {
        "byteOffset" : "0x14",
        "bitOffset" : 0,
        "bitWidth" : 32,
        "name" : "monitorPidMin",
        "description" : "Monitor all process with PID >=
    monitorPidMin.",
        "resetValue" : 60,
        "accessType" : "RW"
      }
    } ]
  }
}
```

4.4 Conclusion

Dans ce chapitre, nous avons présenté le prototype de l'architecture de détection intégrée que nous avons réalisé durant cette thèse. Ce prototype répond avec succès à nos attentes : il est assez simple à comprendre, à mettre en place, à paramétrer et à modifier selon les besoins du concepteur du système final, tout en fonctionnant sur un vrai matériel non simulé.

Nous avons choisi de construire notre système cible avec : (1) une plateforme d'évaluation pour le support physique de notre prototype, composé d'une carte d'évaluation ML605 (avec un FPGA) et d'une machine d'hôte ; (2) la microarchitecture d'un SoC généré par le générateur du SoC Chipyard, avec un cœur Rocket, écrit en langage Chisel très compact et facilement paramétrable ; (3) la partie logi-

cielle basée sur un noyau Linux, qui est un noyau ouvert et très utilisé aujourd'hui.

Nous avons présenté des exemples basés sur des extraits de codes sources du prototype, afin de donner un aperçu de nos modifications sur le système cible et les éléments pour paramétrer l'architecture de détection. Pour la partie matérielle, nous avons présenté la supervision des signaux à l'intérieur d'un composant cible, leurs connexions jusqu'au MDM, l'utilisation d'une connexion appropriée, le modèle d'une simple logique de détection vis-à-vis d'une attaque donnée, et le traitement des informations venant du MDL pour le cas du PID. Pour la partie logicielle, nous avons présenté le traitement des commandes venant d'un utilisateur, l'envoi du PID vers le MDM, et la génération des codes de contrôle basée sur une description des registres déclarés dans le MDM.

Dans le chapitre suivant, nous présentons des algorithmes de détection d'attaques construits avec notre prototype. Le paramétrage et la modification aisée de notre prototype ont pu permettre d'expérimenter plusieurs algorithmes et leurs variantes avec peu d'efforts de programmation, et ont pu permettre de se concentrer sur la conception des algorithmes de détection.

Chapitre 5

Étude de cas

Sommaire

5.1 Étude de cas : CSCA 81
5.1.1 Contexte technique 82
5.1.2 Méthode d'évaluation 86
5.1.3 Conception du détecteur et évaluation 88
5.1.4 Conclusion 99
5.2 Étude de cas : Attaques ROP 100
5.2.1 Contexte technique 100
5.2.2 Méthode d'évaluation 102
5.2.3 Conception du détecteur et évaluation 103
5.2.4 Conclusion 107
5.3 Conclusion 107

Dans ce chapitre, nous présentons notre étude de cas réalisée sur le prototype. Cette étude de cas porte sur la détection en cours d'exécution des attaques de deux catégories différentes : (1) par canal auxiliaire de temps sur le cache (*Cache side-channel attack*, CSCA), plus précisément sur l'attaque Prime+Probe ; et (2) de type ROP, qui visent à détourner le flot de contrôle de façon à réutiliser des parties de code existantes. Pour chaque attaque, nous avons suivi les étapes décrites dans la section 3.3.2 pour le choix des signaux supervisés, la conception des algorithmes de détection, et l'évaluation. Pour la conception des algorithmes de détection, nous nous sommes principalement basés sur l'identification de la signature de l'attaque de manière heuristique, inspirée par les propriétés naturelles des attaques et des algorithmes de détection proposés dans la littérature. Nous démontrons qu'avec des heuristiques assez simples, nous sommes capables d'identifier un comportement malveillant, même dans le cas d'un système complexe exécutant un système d'exploitation et diverses applications.

5.1 Étude de cas : CSCA

Les CSCAs sont des attaques visant les vulnérabilités au niveau du cache qui ont été parmi les premières à être étudiées par la communauté scientifique. Elles sont particulièrement critiques car exploitables en tant que tel pour extraire des informations sur les logiciels en cours d'exécution (qui partagent un cache avec l'attaquant), mais servent également de support à d'autres attaques. Par exemple,

les attaques Spectre [Kocher 2019] et Meltown [Lipp 2018] chargent dans le cache des informations provenant de canaux auxiliaires microarchitecturaux (telles que la prédiction de branchement pour Spectre), puis utilisent les CSCAs pour lire les informations secrètes ; l'attaque rowhammer a besoin d'évincer fréquemment des lignes de mémoire du cache (ce qui correspond à la phase principale des CSCAs), afin d'effectuer un accès direct à la mémoire principale.

Dans cette section, nous nous sommes concentrés sur la détection d'une des CSCAs nommée Prime+Probe [Osvik 2006], car elle est relativement puissante (ne nécessitant aucune instruction spécifique ni de partage de zone mémoire) et totalement fonctionnelle dans notre prototype. Nous présentons le fonctionnement du cache et de l'attaque, les bancs de tests des logiciels légitimes ou malveillants que nous avons choisis ou construits, la conception des algorithmes de détection incluant les signaux choisis respectifs, ainsi que des résultats d'expérimentation. Nous discutons pour finir du potentiel de détection des autres variants de CSCAs.

5.1.1 Contexte technique

5.1.1.1 Fonctionnement du cache

Avant de présenter le fonctionnement des CSCAs, nous décrivons tout d'abord le fonctionnement du cache. La mémoire cache est un composant matériel conçu pour accélérer l'accès du processeur à la mémoire principale. Elle est construite sur le principe de localité temporelle et spatiale d'accès à la mémoire : lorsqu'un emplacement mémoire est accédé, il y a de fortes chances que ce même emplacement ou un emplacement proche soit accédé dans un futur proche. Quand le processeur demande l'accès à une donnée dans la mémoire principale et que celle-ci n'est pas dans le cache (on parle d'un défaut de cache), la donnée souhaitée et les données proches sont lues depuis la mémoire et sont stockées dans le cache, pour des accès futurs potentiels. Si cette donnée est déjà présente dans le cache (on parle d'un succès de cache), alors elle est lue directement dans le cache plutôt que dans la mémoire, l'accès est ainsi plus rapide. Le mécanisme de cache est une optimisation de performance couramment utilisée dans les processeurs modernes de toutes les tailles ; plusieurs niveaux de cache peuvent être utilisés pour d'avantage d'optimisation, comme dans le cas de notre prototype qui a deux niveaux de cache, L1 (proche du cœur de processeur) et L2 (partagé entre les cœurs).

Une structure de cache couramment utilisée est la suivante : le cache est divisé en plusieurs *cache sets*, chaque *cache set* contient un certain nombre de lignes, et une ligne de cache est l'unité élémentaire du cache, généralement de 64 octets. Dans le cas de notre prototype, le cache L2 est un cache de 64 *cache sets* de 8 lignes chacune (c'est un cache avec une associativité de 8, il est nommé 8-associatif), les deux caches L1 sont des caches de 64 *cache sets* d'une ligne (c'est un cache avec une associativité de 1, il est nommé cache direct). À chaque défaut de cache, les données proches de l'adresse accédée, dans la limite de 64 octets (la taille d'une ligne) sont lues et stockées dans une ligne de cache, cette ligne étant choisie en fonction d'une

politique de remplacement, par exemple, la moins récemment utilisée (en anglais *Least Recently Used*, LRU), sauf dans le cas du cache direct où une donnée est associée à une unique ligne de cache.

La correspondance entre les données de la mémoire centrale et l'index du *cache set* est faite en principe à partir de l'adresse physique. Comme illustré dans la figure 5.1, les bits de poids moyen de l'adresse indiquent dans quel *cache set* se placent les données ; les bits de poids faible de l'adresse correspondent au décalage dans la ligne de cache (en anglais *offset*, chaque adresse correspond à un octet de données) ; et les bits de poids fort de l'adresse constituent un *tag* associé à la ligne de cache qui stocke les données, pour déterminer si cette ligne contient des données provenant d'une adresse mémoire spécifique. Les pointillés représentent les destinations possibles d'un bloc de données mémoire dans le cache. On peut observer d'ailleurs qu'il existe une certaine compétition entre les blocs de données : des blocs avec des adresses différentes peuvent se retrouver dans le même *cache set* qu'un autre bloc. Cette propriété sera exploitée dans l'attaque Prime+Probe.

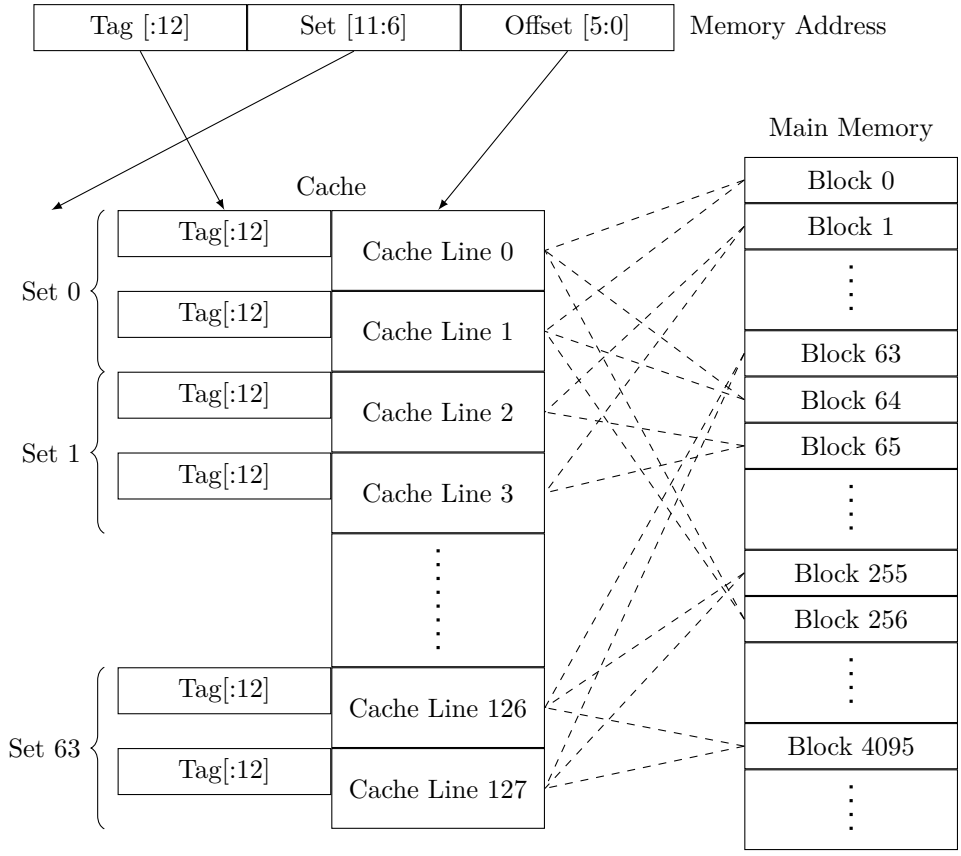


FIGURE 5.1 – Correspondance des blocs de la mémoire principale (blocs de 64 octets) dans un cache 2-associatif avec 64 *cache sets* et 64 octets par ligne de cache. Les blocs de la mémoire principale dont le numéro est multiple de 64 correspondent toujours au *cache set* d'index 0.

5.1.1.2 Fonctionnement des attaques Prime+Probe

Avec les propriétés du cache présentées ci-dessus, nous pouvons constater que si une ligne de mémoire est déjà présente dans le cache, les accès à cette ligne sont toujours accélérés. De plus, différentes adresses physiques (potentiellement occupées par différents logiciels) peuvent être amenées à utiliser de manière concurrente le même *cache set* et les mêmes lignes de cache. Les CSCAs consistent ainsi à espionner l'utilisation du cache par un logiciel victime, en manipulant le cache et en mesurant le temps d'accès aux données depuis un autre logiciel que l'attaquant contrôle.

Prime+Probe est une variante couvrant le plus d'architectures différentes parmi les CSCAs : elle ne nécessite pas de partage de zone mémoire entre le logiciel malveillant et le logiciel victime, ni la nécessité d'utiliser des instructions spécifiques. Les deux seules contraintes sont finalement la possibilité d'avoir un accès au même cache que la victime et un système de comptage du temps d'accès au cache. Par exemple, un processus s'exécutant sur le même processeur que le processus victime mais sur des cœurs différents partageant le cache de dernier niveau, peut réaliser une attaque Prime+Probe [Liu 2015].

L'attaque Prime+Probe se déroule en trois phases : la phase de préparation, la phase d'attaque proprement dite et la phase d'analyse.

Dans la phase de préparation, l'attaquant doit localiser un ou plusieurs *cache sets* qu'il souhaite surveiller. Ceux-ci peuvent être choisis en fonction des adresses des instructions clés ou des données utilisées par le logiciel victime (accessibles dans le binaire), ou expérimentalement. Ensuite, pour chaque *cache set*, l'attaquant doit trouver N adresses dont les *tags* sont différents et qui appartiennent à ce *cache set* (N étant le niveau d'associativité du cache, mais aussi le nombre de lignes dans un *cache set*) de son propre espace mémoire. Ceci n'est pas spécialement difficile, dans le cas du cache L2 de notre prototype par exemple, une page mémoire correspond à une ligne pour chaque *cache set* parmi les 64 *cache sets*, et huit pages mémoire permettent de fournir les huit lignes de *tags* différents nécessaires pour chaque *cache set*.

Notons que l'utilisation du principe de mémoire virtuelle n'a qu'un impact négligeable sur l'attaque. En effet, l'attaquant obtient de l'information sur les 12 bits de poids faible de l'adresse (ils sont communs entre adresse virtuelle et physique), or ces bits sont les plus porteurs d'information car ils couvrent en grande partie (voire la totalité) des bits relatifs à l'index du *cache set* où sont stockées les données. Dans le cas de notre prototype par exemple, la totalité de l'index du *cache set* peut être déduite à partir de ces 12 bits.

La phase d'attaque consiste à exécuter le logiciel malveillant dans le but de profiler l'utilisation de ces *cache sets* par le logiciel victime au cours du temps. Le listing 5.1 présente le code assembleur représentatif utilisé dans la phase d'attaque pour attaquer le cache L2 dans le cas de notre prototype. Ce code est répété de nombreuses fois pour collecter des données de temps d'accès, trois étapes peuvent être identifiées dans chaque répétition :

1. La première étape est appelée *Prime*, dans laquelle l'attaquant accède aux

Listing 5.1 – Un tour de la phase d'attaque de Prime+Probe en code assembleur d'ISA RISC-V, pour un cache de 8 lignes par *cache set*.

```
1     rdcycle a2
2     ld      a5,0(a5)      ; Prime: 8 accesses
3     ld      a5,0(a5)
4     ld      a5,0(a5)
5     ld      a5,0(a5)
6     ld      a5,0(a5)
7     ld      a5,0(a5)
8     ld      a5,0(a5)
9     ld      a5,0(a5)
10    rdcycle a4
11    ...                   ; Wait for victim to execute
12    rdcycle a2            ; Measure access time (begin)
13    ld      a5,0(a5)      ; Probe: 8 accesses
14    ld      a5,0(a5)      ; Can serve as Prime for next round
15    ld      a5,0(a5)
16    ld      a5,0(a5)
17    ld      a5,0(a5)
18    ld      a5,0(a5)
19    ld      a5,0(a5)
20    ld      a5,0(a5)
21    rdcycle a4            ; Measure access time (end)
22    ...                   ; Next round of Prime+Probe
```

N adresses précédemment identifiées appartenant à un *cache set*. L'accès est généralement organisé sous forme d'une liste chaînée, c'est-à-dire que l'adresse du prochain accès est stockée comme la valeur dans l'adresse de l'accès courant, afin d'empêcher la prédiction d'accès qui peut impacter l'occupation du cache. En raison de la politique de remplacement du cache, les 8 lignes de mémoire de l'attaquant vont occuper toutes les 8 lignes de cache dans ce *cache set* (des données précédemment présentes dans ce *cache set* sont évincées et ne sont plus dans le cache).

2. Ensuite, dans l'étape d'attente, le processus de l'attaquant attend que le processus victime s'exécute. Pendant ce laps de temps, si le processus victime accède à des adresses qui correspondent à ce *cache set*, certaines lignes de caches occupées par l'attaquant vont être remplacées par les lignes du processus victime.
3. La troisième étape est appelée *Probe*, dans laquelle l'attaquant accède aux N adresses précédemment identifiées et mesure le temps total requis. L'accès est aussi organisé sous forme d'une liste chaînée comme dans l'étape *Prime* mais dans l'ordre inverse. La liste chaînée permet non seulement de prévenir la prédiction d'accès, mais aussi la dépendance de données entre chaque accès (la prochaine lecture de mémoire se fait forcément après avoir fait la lecture actuelle) et permet ainsi d'empêcher l'exécution dans le désordre qui peut impacter la durée mesurée. L'ordre inverse est utilisé pour éviter l'effet de l'auto-remplacement dû au politique de remplacement de cache. Ces temps

d'accès sont collectés pour être traités dans la phase d'analyse. Notons que l'étape *Probe* peut être utilisée comme étape *Prime* du prochain tour de *Prime*, attente et *Probe*.

La phase d'analyse consiste à analyser les temps d'accès des N adresses collectées pendant la phase d'attaque et déduire le comportement du logiciel victime. Pendant un tour de *Prime*, attente et *Probe*, si le processus victime n'a pas accédé à une adresse qui correspond au *cache set* surveillé pendant l'étape d'attente, alors toutes les N lignes de l'attaquant sont toujours présentes dans le cache. L'accès à ces données génère donc N succès de cache, ce qui correspond à un temps d'accès court. Si, au contraire, le processus victime a accédé à une ou plusieurs adresses correspondant à ce *cache set*, alors certaines lignes de l'attaquant ont été évincées et l'accès à ces données sont des défauts de cache, ce qui correspond à un temps d'accès relativement plus long. L'attaquant apprend ainsi, selon le temps d'accès long ou court, une séquence d'accès du logiciel victime sur les *cache sets* surveillés pendant les étapes d'attente, ce qui lui permet de déduire certaines des informations secrètes de la victime.

5.1.2 Méthode d'évaluation

Afin de pouvoir évaluer les différents algorithmes de détection, nous avons sélectionné les logiciels de tests qui représentent les comportements malveillants et les comportements légitimes, et nous les avons adaptés à notre prototype.

5.1.2.1 Logiciels malveillants

Nous avons mis en place deux types d'attaques, une première simple qui vise à reproduire l'attaque Prime+Probe sur un cache, et une deuxième plus complexe qui vise à attaquer une implémentation d'AES. Dans les deux cas, nous utilisons une version adaptée de l'outil Mastik [Yarom 2016] pour réaliser la phase d'attaque de Prime+Probe. Il est configuré pour attaquer le cache L2 de notre prototype (64 *cache sets* et 8 lignes par *cache set*), d'ISA RISC-V, comme présenté dans le listing 5.1.

La première application vise à établir le profilage d'utilisation d'un ou plusieurs *cache sets*. Trois versions de ce logiciel ont été implémentées :

— une qui profile le *cache set* d'index 0 ;
— une qui profile le *cache set* d'index 1 ;
— une qui profile tous les 64 *cache sets* dans un tour (le temps d'accès est mesuré sur chaque *cache set* individuellement).

Dans chaque version, la boucle principale est constituée de la phase d'attaque (*Prime*, attente et *Probe*). Pendant l'étape d'attente de la phase d'attaque, un accès mémoire en dehors de la zone utilisée pour l'attaque est effectué pour simuler un accès fait par le victime, avec l'index du *cache set* à 0.

Pour la seconde application permettant l'extraction d'une clé AES, nous avons choisi une attaque sur le dernier tour qui couple à la fois les connaissances sur l'exé-

cution obtenues via un Prime+Probe et l'exploitation des informations du chiffré, comme décrit dans l'article [Osvik 2006]. Nous avons utilisé l'implémentation C de l'AES provenant de OpenSSL 1.1.1k, qui est basée sur la célèbre optimisation T-Table d'AES vulnérable aux attaques par cache à cause de sa grande taille (les données sont situées dans plusieurs *cache sets*) et les accès à ces tables sont dépendants du secret. L'attaque fonctionne comme suit : à chaque itération dans la phase d'attaque, l'attaquant effectue un *Prime* sur tous les *cache sets* pour enlever les données du logiciel victime, il génère un texte en clair aléatoire et demande au logiciel victime de chiffrer le texte avec la clé secrète ; l'attaquant attend jusqu'à la fin de l'exécution du logiciel victime lorsque le chiffré est renvoyé ; et il *Probe* tous les *cache sets* pour savoir quelles sont les lignes utilisées dans la T-Table pendant le chiffrement. Avec des études statistiques dans la phase d'analyse, en connaissant le chiffré et sachant que la distribution des clairs est aléatoire, la clé secrète peut être déduite au bout d'un certain nombre de tentatives. Deux versions de cette attaque ont été implémentées pour représenter l'attaque avec différent niveaux de perturbations externes :

— une version *SingleProcess* où le code de l'attaque et celui de la victime s'exécutent dans le même processus ;
— une version *SharedMemory* dans laquelle le code d'attaque et le code de la victime s'exécutent dans des processus distincts et échangent des clairs et des chiffrés via une mémoire partagée.

Dans notre prototype, la récupération complète de la clé AES est obtenue avec environ 3000 textes aléatoires différents pour la version *SingleProcess*, mais permet d'obtenir seulement 50% des bits de clé du dernier tour (déduites à partir de la clé AES initiale) pour la version *SharedMemory*. Les deux logiciels sont configurés pour lancer 3000 chiffrements dans notre évaluation.

Ces différents logiciels malveillants nous permettent d'évaluer nos algorithmes dans les cas où la boucle de la phase d'attaque est réalisée à différentes fréquences (pour un profilage de cache à différentes granularités), et d'évaluer la capacité de notre détecteur sur les attaques d'AES proches des cas réels.

5.1.2.2 Logiciels légitimes

Pour les comportements légitimes, nous avons rassemblé plusieurs bancs de tests existants et relativement petits, sélectionnés dans le but de représenter un large éventail d'algorithmes courants qui utilisent le CPU et la mémoire principale. Nous les avons adaptés et compilés avec la chaîne d'outils GCC décrite dans le chapitre 4 pour fonctionner sur notre système RISC-V. Ce travail a été facilité par la présence d'un noyau Linux dans notre prototype, et aussi par le fait que la plupart des codes sont écrits en C. Les bancs de tests que nous avons utilisés sont (dans l'ordre alphabétique) :

— Coremark [1], un banc de tests simples incluant des calculs sur des listes chaînées, une multiplication matricielle et des analyses avec une machine à états.

1. https ://github.com/eembc/coremark

— Coremark-PRO [2], un banc de tests contenant 9 algorithmes utilisés dans différents secteurs tels que l'automobile, les réseaux et la bureautique, y compris des manipulations d'entiers comme la compression image, le calcul de hash, et des manipulations de virgule flottante comme la transformation de Fourier et des réseaux de neurones.
— Dhrystone [3], un petit banc de tests centré sur les calculs d'entiers dans le cœur de processeur.
— Embench-IOT [4], un banc de tests contenant 19 algorithmes utilisés dans des systèmes embarqués, tels qu'une multiplication matricielle, une gestion de machine à états, un tri d'informations et du traitement d'images. La mesure de temps est faite explicitement avec l'appel de l'instruction `rdcycle`.
— STREAM [5], un petit banc de tests centré sur l'accès mémoire et le calcul en virgule flottante. Il est configuré avec `DSTREAM_ARRAY_SIZE=32768`.

Les algorithmes de détection conçus sont évalués avec ces logiciels malveillants et légitimes, afin d'identifier si chaque algorithme permet de bien détecter les logiciels malveillants avec le moins de faux positifs possible.

5.1.2.3 Étapes suivies pour l'évaluation

Les étapes suivantes ont été suivies pour l'évaluation des algorithmes de détection sur chaque logiciel de test (malveillant ou légitime) :

1. à l'aide de commandes envoyées sur l'interface MDL, réinitialiser l'état interne du MD et commencer la surveillance ;
2. exécuter le logiciel de test avec les bons paramètres ;
3. à l'aide de commandes envoyées sur l'interface MDL, arrêter la surveillance et collecter les résultats intermédiaires obtenus dans le MDM (stockées dans les registres MMIO).

Pour évaluer l'impact du dispositif expérimental, nous avons également suivi ces étapes mais avec aucun programme exécuté dans l'étape 2 ; pour évaluer l'impact potentiel du système d'exploitation, nous avons suivi ces étapes mais avec un appel système standard *sleep* exécuté dans l'étape 2. Pendant l'exécution, le MDL envoie le PID au MDM au moment du changement de contexte, conformément à ce qui a été décrit dans le chapitre 4, pour aider à filtrer les activités liées au noyau.

5.1.3 Conception du détecteur et évaluation

Trois catégories de signaux semblent pertinents pour détecter les CSCAs :
— les signaux liés au flot d'instructions, permettant d'identifier des motifs d'instructions utilisés par une attaque menée par logiciel ;

2. https ://github.com/eembc/coremark-pro
3. https ://www.netlib.org/benchmark/dhry-c
4. https ://github.com/embench/embench-iot
5. http ://www.cs.virginia.edu/stream

— les signaux associés aux accès mémoire, car l'attaquant doit effectuer beaucoup d'accès mémoire soigneusement choisis pour manipuler le cache ;
— les signaux représentant les évènements et l'état interne du cache, car c'est le composant vulnérable visé par l'attaquant.

Dans la suite, nous présentons les différents détecteurs que nous avons conçus, basés sur l'analyse de ces trois catégories de signaux, avec le MDM fonctionnant à 1/16 de la fréquence du cœur de processeur (le cycle d'horloge de MDM est appelé *MDMcycle* dans la suite).

5.1.3.1 Détection basée sur les instructions exécutées

Les CSCAs sont des attaques d'origine logicielle dont la stratégie d'attaque pour exploiter le cache est très similaire entre les différentes architectures, ce qui se traduit par un code assembleur relativement générique. Ainsi, une de nos réflexions consiste à trouver des signatures d'attaque par les instructions exécutées. Typiquement, l'enchaînement de code présenté dans le listing 5.1, qui a besoin d'être répété de nombreuses fois dans une attaque Prime+Probe, peut sembler une bonne source pour l'établissement d'une signature. Il faut pouvoir analyser finement la trace des instructions exécutées, ce qui n'est en général pas réalisé dans l'état de l'art par manque de connexion dédiée de collecte des signaux à haut débit, mais qui est possible avec notre architecture.

Les signaux supervisés choisis sont l'instruction en cours d'exécution (32 bits) et le signal de validité de l'instruction exécutée (1 bit), collectés dans le pipeline du processeur, comme dans l'exemple donné dans le chapitre 4. Nous avons choisi d'utiliser une connexion parallèle pour connecter ces signaux au MDM, afin de préserver au maximum les informations y compris les informations temporelles. Ce choix de connexion parallèle peut être remplacé par une autre connexion (comme la connexion de prétraitement) si des informations compressées peuvent suffire pour l'algorithme de détection, pour réduire la surface nécessaire dans le matériel.

En analysant le listing 5.1, deux signatures, fondamentales pour le bon fonctionnement de l'attaque, peuvent être identifiées :

— Deux mesures de temps sont nécessaires à chaque *Probe* pour pouvoir identifier si la victime a accédé à une des lignes dans le *cache set* supervisé. Une méthode courante consiste à utiliser des instructions spécifiques de mesure de temps, telles que `rdcycle`, `rdtime`, et `rdinstret` dans le cas du RISC-V (ces instructions sont appelées *instructions timer* dans la suite).
— Pour la lecture de la liste chaînée, il est nécessaire de lire des valeurs d'au moins la taille d'une adresse dans la mémoire, c'est-à-dire une lecture de 32 bits (`lw`) ou de 64 bits (`ld`). Ces instructions sont appelées *instructions loadword* dans la suite.

Instructions Timer Notre premier algorithme de détection a pour but de détecter l'exécution proche de deux *instructions timer*. Nous avons tout d'abord mesuré expérimentalement la durée de 8 accès, allant de 35 cycles à 300 cycles d'horloge

dans la plupart des cas. Cette durée est nettement plus courte que toutes les mesures de durée d'exécution rencontrées dans les bancs de tests (et le resterait même pour un système avec un *cache set* contenant plus de 8 lignes). Ainsi, elle ne génère pas de faux positif pour les bancs de tests. Cet intervalle entre deux instructions reste cependant relativement long par rapport à la valeur de *MDMcycle*, ainsi, il suffit de savoir s'il y a au moins une *instruction timer* dans un *MDMcycle* pour pouvoir identifier la présence des deux *instructions timer*. Ce constat permet de simplifier l'algorithme de détection et signifie que la connexion parallèle n'est donc pas forcément nécessaire pour cette détection et qu'elle pourrait être remplacée par une connexion de prétraitement. L'algorithme de détection obtenu est présenté dans le listing 5.2 : il détermine s'il y a au moins une *instruction timer* dans le *MDMcycle* courant (`in.pack_has_timer`), et si le dernier *MDMcycle* contenant une mesure de temps est situé dans une fenêtre de 32 *MDMcycles* (correspondant à 512 cycles du processeur, le résultat est stocké dans le registre `last_timer_in_window`). Si les deux conditions sont satisfaites, alors on considère qu'il y a une étape *Probe* qui a été exécutée (représenté par `atk_timer_timer32` évalué à vrai), et on compte son nombre d'occurrences. Il faut un nombre relativement grand de *Probe* au total pour réussir l'attaque, et il reste donc à déterminer le seuil permettant de lever l'alerte, sans toutefois générer de faux positifs. Ce seuil est donc évalué expérimentalement à l'aide de nos bancs de tests.

Listing 5.2 – Algorithme de détection des attaques Prime+Probe implémenté dans le MDM, basé sur les instructions de mesure de temps relativement proches.

```
1  // Detect two timer instructions in a window of 32 slow cycles
2  val windowSize: Int = 32
3  val last_timer_count = RegInit(0.U((log2Ceil(windowSize) + 1).W))
4  val last_timer_in_window = RegInit(false.B)
5  when (in.resetCounters) {
6    last_timer_count := 0.U
7    last_timer_in_window := false.B
8  } .elsewhen (in.isMonitoring && in.pack_has_valid) {
9    when (in.pack_has_timer) {
10     last_timer_count := 0.U
11     last_timer_in_window := true.B
12   }.otherwise {
13     last_timer_count := last_timer_count + 1.U
14     last_timer_in_window := Mux(last_timer_in_window && (
       last_timer_count >= windowSizeMax.U), false.B,
       last_timer_in_window)
15   }
16 }
17
18 val atk_timer_timer32 = Wire(Bool())
19 atk_timer_timer32 := last_timer_in_window && in.pack_has_timer &&
       in.isMonitoring
```

Le tableau 5.1 présente le nombre d'occurrences à vrai de la variable `atk_timer_timer32` sur l'ensemble des bancs de tests. Nous pouvons constater

qu'il n'y a pas de faux positif parmi les logiciels légitimes, et que le nombre de *Probe* détectés est élevé pour les logiciels ayant des comportements malveillants. Le seuil de détection peut ainsi être assez bas pour la détection au plus tôt des attaques. Nous avons également constaté qu'on détecte moins de tentatives de *Probe* sur l'attaque AES par *SharedMemory* par rapport à la version *SingleProcess*, même si elles exécutent toutes les deux le même nombre de *Probe* dans la réalité. Nous pensons que cela est en lien avec les activités du système et les deux processus ont évincé plus fréquemment les lignes de cache occupées par l'attaquant, résultant occasionnellement en une durée plus longue que la fenêtre prévue.

TABLE 5.1 – Évaluation de la détection basée sur la détection de deux instructions de mesure de temps proche.

Logiciel	**MDMcycles**	**Tentative de Probe**	
		Nombre	**Période moy.**
CoreMark *	53452817	0	N/A
CoreMark-PRO *	1041910280	0	N/A
Embench *	43957696	0	N/A
Dhrystone	11242585	0	N/A
STREAM	40513182	0	N/A
Attack SingleProcess	108337476	572429	189
Attack SharedMemory	724951250	553060	1311
Cache Profiling set 0	976975	39985	24
Cache Profiling set 1	816452	39985	20
Cache Profiling all set	15594885	1299297	12
No program	112200	0	N/A
Sleep 1	168206	0	N/A

La colonne *nombre* représente le nombre d'occurrences à vrai de la variable `atk_timer_timer32` à la fin de l'exécution, et *période moy.* représente le nombre moyen de cycles entre deux occurrences à vrai de la variable. Les bancs de tests incluant plusieurs algorithmes sont annotés avec * et seule la valeur moyenne est représentée.

Instructions Loadword Notre deuxième algorithme cherche à identifier un enchaînement de lecture des valeurs de la taille d'une adresse mémoire. Nous observons uniquement le fait qu'au moins une *instruction loadword* ait été exécutée pendant un *MDMcycle*, mais nous ne conservons pas le nombre précis d'instructions ayant été exécutées. Nous sommes conscients que cette simplification risque de détecter moins d'occurrences d'*instructions loadword* quand les accès sont principalement des succès de cache, car deux ou plus d'*instructions loadword* pourraient arriver dans le même *MDMcycle*, mais nous avons fait ce choix pour réduire la quantité d'informations à analyser. L'algorithme de détection obtenu est présenté dans le listing 5.3. Pour chaque *MDMcycle* ayant au moins une instruction valide, on détermine si

au moins une *instruction loadword* est exécutée. Le cas échéant, on incrémente le compteur `countlw`, sinon, on le décrémente de 2. Quand cette valeur atteint un seuil interne à 16 (`countLwThresh`), on considère qu'il y a un enchaînement de lectures suspectes (`atk_lw` positionné à vrai), et on compte le nombre d'occurrences de cet évènement. Ces valeurs 1, 2 et `countLwThresh` sont déterminées expérimentalement pour différencier au mieux les logiciels malveillants et légitimes tout en gardant une logique simple.

Listing 5.3 – Algorithme de détection des attaques Prime+Probe implémenté dans le MDM, basé sur la détection d'un long enchaînement d'instructions de lecture de la taille d'une adresse mémoire dans la mémoire.

```
// Detect chained load word : valid load c+1, valid not load c-2 (thresh 16)
val countLwThresh = 16
val countlw = RegInit(0.U(log2Ceil(countLwThresh + 1).W))
when (in.pack_has_valid && in.isMonitoring) {
  when (in.pack_has_loadword) {
    countlw := Mux(countlw <= countLwThresh.U - 1.U, countlw + 1.U, countLwThresh.U)
  }.otherwise {
    countlw := Mux(countlw >= 2.U, countlw - 2.U, 0.U)
  }
}

val atk_lw = RegInit(false.B)
atk_lw := (countlw === countLwThresh.U) && in.pack_has_valid && in.isMonitoring
```

Le tableau 5.2 présente le nombre d'occurrences à vrai de la variable `atk_lw` sur l'ensemble des bancs de tests. Nous avons globalement observé que cet algorithme souffre d'un nombre assez élevé de faux positifs, et qu'il ne parvient pas à différencier correctement les profilages de cache. Il permet cependant de distinguer les attaques complètes des attaques de profilage de cache uniquement. Différentes raisons pourraient contribuer à ce taux élevé de faux positifs : (1) nous avons extrait une *instruction loadword* par *MDMcycle*, au lieu d'en compter le nombre total ; (2) notre signature ne permet pas de représenter totalement la propriété des listes chaînées, il faut combiner avec d'autres logiques, déterminant par exemple si l'adresse d'un accès est identique à une valeur précédemment lue dans la mémoire.

5.1.3.2 Détection basée sur les adresses mémoire accédées

La manipulation de cache au niveau ISA consiste en la manipulation précise des adresses mémoire accédées pour remplir un *cache set*. Comme dans le cas précédent, ce type de détecteur n'a pas été réalisé dans l'état de l'art à notre connaissance.

Nous avons choisi de surveiller l'adresse virtuelle envoyée dans le pipeline du processeur au cache, et l'adresse physique correspondante obtenue dans le cache de données L1, avec leurs signaux de validité respectifs (incluant la lecture et l'écri-

TABLE 5.2 – Évaluation de la détection basée sur la détection d'un long enchaînement des instructions de lecture mémoire de la taille d'une adresse mémoire.

Logiciel	**MDMcycles**	**Enchaînement de load**	
		Nombre	**Période moy.**
CoreMark *	53452817	161861	330
CoreMark-PRO (1)*[1]	354207744	7269588	49
CoreMark-PRO (2)*[2]	566294708	7195	78707
Embench nettleaes	32402532	431116	75
Embench sglib	25955324	1368748	19
Embench others*	45688918	10	4363548
Dhrystone	11242585	13	864814
STREAM	40513182	46	880721
Attack SingleProcess	108337476	12037339	9
Attack SharedMemory	724951250	7505614	97
Cache Profiling set 0	976975	70	13957
Cache Profiling set 1	816452	72	11340
Cache Profiling all set	15594885	58	268877
No program	112200	53	2117
Sleep 1	168206	118	1425

La colonne *nombre* représente le nombre d'occurrences à vrai de la variable `atk_lw` à la fin de l'exécution, et *période moy.* représente le nombre moyen de cycles entre deux occurrences à vrai de la variable. Les bancs de tests incluant plusieurs algorithmes sont annotés avec * et seule la valeur moyenne est représentée.
[1] La ligne CoreMark-PRO (1) contient les algorithmes cjpeg, core, radix.
[2] La ligne CoreMark-PRO (2) contient les algorithmes linear, parser, sha, zip. Certains algorithmes de CoreMark-PRO ne sont pas évalués à cause de leur temps d'exécution trop long.

ture). Nous avons choisi d'utiliser une connexion d'extraction pour collecter ces informations, de telle sorte que nous ne conservons qu'une seule adresse valide par *MDMcycle* si elle est présente. Nous avons fait ce choix, car un accès à la mémoire prend généralement plusieurs cycles lors d'un succès de cache et encore plus lors d'un défaut de cache. Ce choix a été confirmé par notre expérimentation, montrant qu'une valeur par *MDMcycle* est suffisante pour préserver suffisamment d'informations pour la détection.

En analysant le fonctionnement des attaques Prime+Probe, une signature sur les adresses utilisées a été identifiée : pendant l'étape *Prime* ou *Probe* dans la phase d'attaque, pour remplir un *cache set* de 8 lignes, l'attaquant doit accéder à 8 adresses de même index de *cache set* et de *tags* différents.

Nous avons ainsi construit un algorithme pour la détection de ce comportement, et évalué les différents paramètres d'algorithme pour trouver expérimentalement une configuration simple permettant de différencier au mieux les logiciels malveillants et légitimes. Notre algorithme final est présenté dans le listing 5.4. Dans cet algorithme,

Listing 5.4 – Algorithme de détection des attaques Prime+Probe implémenté dans le MDM, basé sur les adresses mémoire correspondant au même *cache set*, mais portant des *tags* différents.

```
// diff tag same set c+2, diff tag diff set c+1, same tag same set c-4, same tag diff set c-2 (thresh 8)
val countMaThresh = 8
val countma = RegInit(0.U(log2Ceil(countMaThresh + 1).W))
val saved_dmemtag = RegInit(VecInit(Seq.fill(4)(0.U((mp.addrWidth - params.tagAddrLSB).W))))
val saved_dmemset = RegInit(VecInit(Seq.fill(4)(0.U((params.tagAddrLSB - params.setAddrLSB).W))))

val dmemtag = Wire(UInt((mp.addrWidth - params.tagAddrLSB).W))
dmemtag := in.dmemaddr_data(mp.addrWidth - 1, params.tagAddrLSB)
val dmemset = Wire(UInt((params.tagAddrLSB - params.setAddrLSB).W))
dmemset := in.dmemaddr_data(params.tagAddrLSB - 1, params.setAddrLSB)

// Update saved_dmem* array in FIFO way
when(in.dmemaddr_valid && in.isMonitoring){
  when(saved_dmemtag.map(_ =/= dmemtag).reduce(_&&_)){ // new tag
    when(saved_dmemset.map(_ =/= dmemset).reduce(_&&_)){ // new set
      saved_dmemset.foldLeft(dmemset) { (data, last) =>
        last := data
        last
      }
      countma := Mux(countma <= countMaThresh.U - 1.U, countma + 1.U, countMaThresh.U)
    }.otherwise{// old set
      countma := Mux(countma <= countMaThresh.U - 2.U, countma + 2.U, countMaThresh.U)
    }
    saved_dmemtag.foldLeft(dmemtag) { (data, last) =>
      last := data
      last
    }
  }.otherwise{ // old tag
    when(saved_dmemset.map(_ =/= dmemtag).reduce(_&&_)){ // new set
      saved_dmemset.foldLeft(dmemset) { (data, last) =>
        last := data
        last
      }
      countma := Mux(countma >= 2.U, countma - 2.U, 0.U)
    }.otherwise{ // old set
      countma := Mux(countma >= 4.U, countma - 4.U, 0.U)
    }
  }
}

val atk_memaccess = RegInit(false.B)
atk_memaccess := (countma === countMaThresh.U) && in.dmemaddr_valid && in.isMonitoring
```

seuls les signaux de l'adresse virtuelle (`in.dmemaddr_data` et `in.dmemaddr_valid`) sont utilisés, car ils donnent un meilleur résultat. Plus précisément, à partir de l'adresse mémoire, on extrait son *tag* (`dmemtag`) et son index de *cache set* (`dmemset`), et on conserve les 4 derniers *tags* et les 4 derniers index utilisés (`saved_dmemtag` et `saved_dmemset`). On incrémente un compteur interne (`countma`) de 2 si l'adresse mémoire du *MDMcycle* courante possède un index de *cache set* parmi les index récemment utilisés, mais que son *tag* ne fait pas partie des *tags* récemment utilisés. Ce compteur interne est aussi ajusté (incrémenté ou décrémenté), selon que l'index et le *tag* ont récemment été utilisés ou pas. Quand le compteur interne `countma` atteint un seuil interne de 8 (`countMaThresh`), on considère qu'une séquence d'accès suspecte a été détectée (représentée par `atk_memaccess` évalué à vrai). On compte ensuite le nombre d'occurrences à vrai de `atk_memaccess` dans l'expérimentation.

Pour cette heuristique, nous avons obtenu des résultats intéressants, comme présentés dans le tableau 5.3. Nous pouvons observer que pour les logiciels légitimes, une séquence d'accès suspecte est parfois détectée. Cependant, cette séquence apparaît beaucoup plus fréquemment en cas d'attaque, permettant de bien séparer les attaques par rapport aux logiciels légitimes. Nous avons également observé que l'impact de notre méthode de mesure et de l'activité du système d'exploitation reste relativement élevé, probablement à cause des accès mémoire lors de la création de processus, cela reste un aspect à prendre en compte dans l'algorithme final.

TABLE 5.3 – Évaluation de la détection basée sur la détection des adresses mémoire accédées permettant de remplir un *cache set*.

Logiciel	**MDMCycles**	**Séquence d'accès suspecte**	
		Nombre	**Période moy.**
CoreMark *	53452817	161	332005
CoreMark-PRO parser	67632009	29174	2318
CoreMark-PRO others *	1163695064	7854	148166
Embench *	43957696	162	271344
Dhrystone	11242585	175	64243
STREAM	40513182	140	289380
Attack SingleProcess	108337476	1339213	81
Attack SharedMemory	724951250	926154	783
Cache Profiling set 0	976975	118522	8
Cache Profiling set 1	816452	116022	7
Cache Profiling all set	15594885	5294473	3
No program	112200	77	1457
Sleep 1	168206	96	1752

La colonne *nombre* est le nombre d'occurrences à vrai de la variable `atk_memaccess` à la fin de l'exécution, et *période moy.* représente le nombre moyen de cycles entre deux occurrences à vrai de la variable. Les bancs de tests incluant plusieurs algorithmes sont annotés avec * et seule la valeur moyenne est représentée.

5.1.3.3 Détection basée sur les évènements du cache

En nous inspirant de certains algorithmes de détection basés sur les HPCs proposés dans la littérature [Chiappetta 2016, Payer 2016], nous avons testé la pertinence de plusieurs évènements pour la détection des attaques Prime+Probe, tels que l'accès mémoire initié par le processeur, le défaut de cache, la réception de la réponse de la mémoire dans le cache, etc.

Nous avons compté le nombre de *MDMcycles* où au moins un de ces évènements a eu lieu. Ce choix a été fait pour reproduire des compteurs dans le MDM similaire aux HPCs tout en réduisant la complexité de l'algorithme. La proportion entre le nombre de défauts de cache mesuré et le nombre d'accès mémoire mesuré reste la valeur la plus intéressante pour séparer les attaques et les logiciels légitimes, comme présenté dans le tableau 5.4. Ceci est cohérent avec les observations de la littérature qui soulignent que l'utilisation concurrente de cache indique une anomalie. Avec un seuil positionné à $4,5$ (nombre d'accès mémoire par nombre de défauts), seul un algorithme (CoreMark-PRO parser) est considéré comme faux positif selon les résultats à la fin d'exécution du logiciel. Cependant, la différence de proportion entre les attaques et les logiciels légitimes reste relativement faible par rapport à nos détecteurs basés sur les *instructions timer* et les adresses mémoire. Cette simple logique de détection n'est donc pas suffisante pour détecter rapidement l'attaque sans générer trop de faux positifs. Pour cette catégorie de signaux, nous n'avons malheureusement pas eu suffisamment de temps pour construire des algorithmes plus adaptés pour accentuer la différence entre les attaques et les logiciels légitimes. Mais nous considérons que c'est malgré tout une piste intéressante à poursuivre afin d'améliorer les algorithmes de détection existants basés sur les HPCs.

5.1.3.4 Surcoût de surface

Le tableau 5.5 présente l'empreinte en surface sur le FPGA du système dans 4 scénarios : le système de base sans MD, le système avec la détection de deux *instructions timer* proches uniquement (appelée *InstTimer*), le système avec la détection de la séquence d'accès mémoire permettant d'évincer un *cache set* uniquement (appelée *MemAccess*), et le système avec les deux algorithmes implémentés (*I+M*). L'implémentation complète avec les deux algorithmes augmente le nombre de FFs utilisés de 12% et le nombre de LUTs de 5% par rapport au système de base. Le principal surcoût de surface, selon nous, est dû à la structure de base nécessaire pour le MDM (les connexions et certains registres de configuration) et surtout à la synchronisation des signaux supervisés (pour gérer la différence de fréquence de 16 entre le MDM et le cœur de processeur). Ainsi, la combinaison des deux algorithmes n'utilise que 58% des FFs de la somme des deux, et même moins de LUTs utilisés par rapport au cas de *InstTimer* (probablement à cause de différentes optimisations faites par Xilinx ISE). Une certaine marge d'amélioration est possible en retravaillant la stratégie de synchronisation et les algorithmes de détection (l'utilisation en commun des signaux supervisés, l'optimisation des logiques).

Table 5.4 – Évaluation de la détection basée sur le nombre de défauts de cache mesuré par rapport au nombre d'accès au cache mesuré.

Logiciel	Nombre d'accès	Défaut de cache L1	
		Nombre	Accès moy.
CoreMark *	43757898	1612677	27,1
CoreMark-PRO cjpeg	12707020	2233294	5,7
CoreMark-PRO parser	47519666	12615229	3,8
CoreMark-PRO zip	1618863117	203864220	7,9
CoreMark-PRO others*	1478031099	79403106	18,6
Embench nettleaes	29781582	5134587	5,8
Embench qrduino	23683491	3124988	7,6
Embench others*	30954059	1476366	21,0
Dhrystone	9825086	459122	21,4
STREAM	29892676	4675529	6,4
Attack SingleProcess	284757	67360	4,2
Attack SharedMemory	506529	122096	4,1
Cache Profiling set 0	797902	224235	3,6
Cache Profiling set 1	675368	163457	4,1
Cache Profiling all set	13542173	5417242	2,5
No program	141632	32165	4,4
Sleep 1	411050	76996	5,3

La colonne *nombre d'accès* représente le nombre de *MDMcycles* où une instruction d'accès mémoire (lecture ou écriture) est exécutée à la fin de l'exécution. La colonne *nombre* représente le nombre de *MDMcycles* où un défaut de cache des données L1 se produit à la fin de l'exécution, et *accès moy.* représente le nombre moyen d'accès au cache mesuré entre deux défaut de cache mesuré (*nombre d'accès* divisé par *nombre*). Les bancs de tests incluant plusieurs algorithmes sont annotés avec * et seule la valeur moyenne est représentée.

Table 5.5 – Surcoût de surface en fonction des différentes configurations pour la détection des attaques Prime+Probe.

Ressources	Système de base	InstTimer		MemAccess		I+M	
		MD	Total	MD	Total	MD	Total
Flip-Flops	22167	2448	24615	2172	24339	2669	24836
LUTs	31668	1620	33288	946	32614	1354	33022
BlockRAM	26	0	26	0	26	0	26

La colonne *MD* est la quantité de ressources supplémentaires nécessaires pour implémenter l'algorithme correspondante dans le MDM, la colonne *total* est la quantité de ressources totales nécessaires pour le système finale avec l'algorithme de détection.

5.1.3.5 Généricité des détecteurs

Nous avons également analysé l'utilisation des détecteurs que nous avons conçus (principalement *InstTimer* et *MemAccess*) pour la détection des variantes des attaques et s'ils présentent un certain intérêt pour les logiciels légitimes.

Pour la variante Flush+Reload de CSCAs, où l'attaquant, pendant la phase d'attaque, utilise une instruction spécifique pour évincer une adresse mémoire en dehors du cache (appelée *flush*, non disponible pour les logiciels utilisateurs dans notre prototype) puis mesure le temps d'accès à cette adresse après la période d'attente : sans adaptation spéciale, l'algorithme *InstTimer* fonctionne toujours pour détecter les deux mesures de temps proches d'un défaut de cache, même si dans le cas d'un succès de cache, il est possible que les deux instructions fassent partie du même *MDMcycle* et soient donc identifiées comme une seule occurrence. Cependant, comme l'attaque nécessite toujours un grand nombre d'itérations, cet algorithme permet toujours d'identifier le comportement malveillant. Une adaptation de l'algorithme *InstTimer* spécialement pour la détection de Flush+Reload consiste à identifier l'utilisation d'une *instruction timer* proche d'un *flush*. Comme l'utilisation d'un *flush* est encore plus rare que l'utilisation d'*instruction timer* dans les logiciels légitimes, cet algorithme adapté devrait théoriquement présenter un faible taux de faux positifs.

Pour la variante de CSCAs où l'attaquant n'utilise pas les *instructions timer* pour la mesure de la durée des accès, mais utilise d'autres mécanismes moins précis tels qu'une bibliothèque, un compteur dans une boucle s'exécutant sur un cœur différent : l'attaque nécessite plus d'itérations pour réussir, l'algorithme *MemAccess* fonctionne toujours, mais bien évidemment l'algorithme *InstTimer* ne peut plus la détecter puisque les *instructions timer* ne sont plus utilisées. Pour détecter ces motifs de mesures de temps, il est possible de vérifier l'adresse physique de la fonction de la bibliothèque (fournie par le MDL) ou du compteur partagé (trouvée à l'aide des répétitions dans les instructions), et de rechercher l'accès à ces adresses physiques suspectes à la place des *instructions timer*.

Pour un système ayant un cache plus grand que celui de notre prototype (chaque *cache set* possédant beaucoup de lignes), le nombre d'accès pour un *Prime* ou un *Probe* est ainsi plus élevé, et la durée des accès est ainsi plus longue. Dans ce cas, il faut augmenter la taille de la fenêtre dans l'algorithme *InstTimer* pour couvrir ce changement de durée. Le seuil interne de *MemAccess* peut rester à 8, mais peut également être augmenté pour espérer générer encore moins de faux positifs avec les logiciels légitimes.

Pour l'attaque rowhammer, qui évince de façon répétitive le cache comme les CSCAs, mais ne nécessite aucune mesure de temps : l'algorithme *InstTimer* ne fonctionne plus, mais l'algorithme *MemAccess* est toujours efficace pour sa détection. De plus, comme dans l'attaque rowhammer, il n'y a pas besoin d'attente entre deux évincements de cache, le compteur interne `countma` a plus de chance d'être maintenu au seuil. Ainsi, le nombre de séquences d'accès détectées par *MemAccess* est plus élevé, ce qui facilite l'identification des comportements malveillants.

Pour les attaques temporelles qui ne visent pas le partage de cache, mais l'utilisation concurrente d'un autre composant matériel (comme l'ALU, les mémoires tampon et le bus) : l'algorithme *MemAccess* ne peut pas les couvrir, mais *InstTimer* reste efficace si la mesure de durée d'utilisation de ce composant est toujours faite avec les *instructions timer*.

En ce qui concerne l'utilisation des détecteurs pour des logiciels légitimes, l'algorithme *MemAccess* est indirectement une estimation de la mauvaise utilisation du cache. Fondamentalement, l'accès aux données d'un *cache set* spécifique et des *tags* différents brise très souvent les principes de localité temporelle et spatiale du cache. Par conséquent, l'algorithme *MemAccess* permet de mesurer la mauvaise utilisation du cache par les logiciels légitimes, et offre donc des informations intéressantes pour aider à optimiser ces logiciels.

5.1.4 Conclusion

Dans cette section, nous avons présenté trois catégories de signaux potentiellement pertinents à superviser pour la détection des attaques Prime+Probe. Une de ces catégories (les évènements du cache) a déjà été utilisée dans la littérature pour la détection des CSCAs tandis que les deux autres catégories de signaux (les instructions exécutées et les adresses mémoire accédées) n'ont pas été utilisées à notre connaissance. À partir de ces deux catégories de signaux non utilisés dans la littérature, nous avons pu concevoir trois différents algorithmes dont deux qui semblent prometteurs. Ces algorithmes ont nécessité moins de signaux par rapport aux détections basées sur les HPCs qui ont besoin de plusieurs évènements, car nous avons pu conserver plus d'informations sur un simple signal. Nous avons profité de la flexibilité et la modularité de notre prototype pour implémenter et évaluer rapidement différents variants de chaque algorithme, ce qui constitue la force de notre proposition.

Deux des quatre algorithmes de détection que nous avons conçus donnent des résultats spécialement intéressants, malgré le fait qu'ils s'exécutent dans un MDM dont la fréquence est 16 fois inférieure à celle du cœur de processeur : un algorithme qui cherche à détecter un nombre élevé d'occurrences de deux *instructions timer* proches dans le temps (*InstTimer*) ; un autre qui cherche à détecter un nombre élevé de séquences d'accès mémoire permettant d'occuper la totalité d'un *cache set* (*MemAccess*). L'algorithme *InstTimer* a une très bonne précision de détection, qui ne présente aucun faux positif de détection d'une tentative de *Prime* dans notre expérimentation, et permet ainsi de détecter les attaques au tout début de leur exécution. Dans les cas où la mesure de temps n'est pas faite avec les *instructions timer* ou lorsque la mesure de temps n'est pas utilisée, l'algorithme *MemAccess* permet de compléter la détection en considérant l'attaque sous un autre angle. Cette étude de cas nous donne un aperçu de l'intérêt de la reconfiguration du MD : quand un algorithme n'est plus adapté au variant, il est possible de mettre en place un autre algorithme assez différent pour renforcer la sécurité du système.

Bien sûr, ces détecteurs peuvent encore être améliorés. Pendant nos expérimen-

tations, nous nous sommes concentrés sur la détection des motifs qui représentent une tentative de *Prime* ou *Probe*, et nous nous sommes globalement basés sur le fait qu'un nombre élevé de tentatives dans une courte durée indique la présence d'une attaque Prime+Probe. Nous pouvons améliorer la relation entre les compteurs de tentatives et la détection d'attaque, par exemple, en considérant uniquement les tentatives relativement proches (pour se concentrer sur la phase d'attaque).

5.2 Étude de cas : Attaques ROP

Comme nous avons pu le voir dans le chapitre 2, différents types d'attaques (incluant des attaques microarchitecturales, des attaques de corruption de mémoire et de réutilisation de code, et des logiciels malveillants) pourront être détectées via l'analyse des signaux architecturaux et microarchitecturaux, que notre framework permet de collecter avec peu de perte d'information.

Pour démontrer la généricité de notre approche sur des attaques assez différentes, nous avons choisi d'utiliser notre framework pour détecter des attaques ROP. Les attaques ROP sont des attaques assez puissantes aujourd'hui qui font partie des attaques par réutilisation de code.

5.2.1 Contexte technique

5.2.1.1 Fonctionnement de ROP

L'attaque ROP [Shacham 2007] est un exemple d'attaque par réutilisation de code. Elle manipule les adresses de retour pour rediriger le flot de contrôle vers une série de petits morceaux de code assembleur (appelés *gadgets*) choisis par l'attaquant, dans le code exécutable du logiciel victime lui-même ou dans des bibliothèques partagées. Ces gadgets exécutent généralement des actions élémentaires, telles qu'un calcul sur un registre, et se terminent par une instruction de retour. Cette instruction de retour saute, dans le cas normal, vers la fonction appelante sur la base d'une adresse de retour préalablement fournie par la fonction appelante. Cependant, comme l'attaquant contrôle ces adresses de retour, l'instruction de retour saute en réalité vers une adresse spécifique choisie par l'attaquant pour continuer l'exécution. Les gadgets sont exécutés comme s'ils faisaient partie du programme de la victime, avec les mêmes privilèges et dans la même zone mémoire. Contrairement aux attaques traditionnelles par débordement de tampon, les attaques par réutilisation de code n'écrivent pas le code à exécuter dans la pile, contournant ainsi la protection de type "écriture XOR exécution", qui interdit l'exécution de code dans la pile. Il existe également d'autres variantes de ROP qui n'utilisent pas d'adresses de retour mais des appels de fonction ou des sauts indirects vers des gadgets.

Les attaques ROP peuvent être réalisées sur les plateformes d'ISA RISC-V [Jaloyan 2020]. Contrairement à x86, l'adresse de retour dans RISC-V n'est pas lue dans la pile, mais dans un registre spécifique, appelé `ra`. Lors de l'appel d'une fonction, l'appelant place l'adresse de retour dans `ra` ; lorsque la fonction retourne,

elle exécute l'instruction `ret` (alias de `jalr zero, 0(ra)`), qui saute à l'adresse pointée par `ra` et continue l'exécution dans la fonction appelante. En cas d'appels de fonctions imbriqués, le contenu du registre `ra` est sauvegardé dans la pile et restauré si nécessaire. Cela permet à un attaquant qui contrôle la pile (par débordement de tampon par exemple) de contrôler également l'adresse de retour dans les fonctions. Un exemple de gadget sur RISC-V est présenté dans le listing 5.5. Nous pouvons constater que pour réaliser une action élémentaire d'une seule instruction utile, un gadget doit contenir au moins 3 instructions supplémentaires (contrairement à une seule instruction supplémentaire sur une architecture x86). Ces trois instructions supplémentaires sont utilisées pour : lire l'adresse de retour stockée dans la pile dans le registre `ra`, incrémenter le pointeur de pile et faire un saut mémoire à l'adresse présente dans `ra`. Selon nos observations, la combinaison de ces 3 instructions peut être trouvée à la fin d'une fonction qui a besoin d'appeler une autre fonction, parfois aussi de façon mélangée parmi d'autres instructions qui se chargent de restaurer des valeurs de registres. Les gadgets RISC-V sont ainsi globalement plus difficiles à trouver que les gadgets x86.

Listing 5.5 – Exemple d'un gadget sur RISC-V construit avec notre générateur de ROP : ce gadget simple ajoute 1 au registre a7 et peut être utilisé, par exemple, pour préparer la valeur de a7 pendant l'exécution d'une chaîne de ROP.

```
1 gadget_a7plus1:
2   addi a7, a7, 1; Elemental action
3   ld ra, 0(sp)  ; Read the next gadget address from the stack to ra
4   addi sp, sp, 8; Increment the stack pointer
5   ret           ; Jump to the next gadget address stored in ra
```

5.2.1.2 Protections contre les attaques ROP

Comme les attaques ROP exécutent uniquement des instructions dans le code existant, il est difficile d'empêcher l'existence de telles attaques. En dehors des méthodes classiques visant à empêcher le contrôle de la pile par l'attaquant, les principales approches de protection vis-à-vis des attaques ROP consistent à les détecter dynamiquement :

— La vérification de l'intégrité du flot de contrôle (en anglais *Control Flow Integrity*, CFI) consiste à analyser statiquement le code du logiciel à protéger, et construire un graphe de contrôle qui contient toutes les sources et les destinations des branchements ainsi que leurs correspondances. Puis, le logiciel à protéger est instrumenté pour vérifier ce graphe de contrôle au moment des différents sauts, par des moyens purement logiciels [Abadi 2009] ou avec l'assistance du matériel [Danger 2018].

— L'utilisation d'une *shadow stack* à côté de la pile permet de vérifier si les adresses de retours de fonctions correspondent aux valeurs utilisées au moment de l'appel de la fonction. La *shadow stack* peut être implémentée dans

la mémoire principale directement et être vérifiée avec des instructions ajoutées dans le logiciel à protéger [Dang 2015], ou dans une mémoire isolée dédiée et être vérifiée par un matériel au moment des appels et des retours de fonctions [Lee 2015, Delshadtehrani 2020].
— La détection de la signature comportementale des attaques basée sur les informations microarchitecturales, comme dans [Wang 2016, Cheng 2014]. Cette catégorie de travaux a été présentée dans le chapitre 2.

5.2.2 Méthode d'évaluation

Comme l'environnement d'exécution est identique à l'étude de cas précédente, nous utilisons les logiciels légitimes et les étapes d'évaluation qui ont été présentés dans la section 5.1.2. Dans cette partie, nous nous concentrons sur la description des logiciels ayant un comportement malveillant du type ROP utilisés dans notre expérimentation.

Malheureusement, aucun banc de test ou base de code des attaques ROP n'a été publié pour RISC-V jusqu'à présent. Nous avons ainsi suivi les principes décrits par Jaloyan et al. [Jaloyan 2020] pour construire un générateur d'attaques ROP. Ce générateur contient un script Python permettant de générer des gadgets et des chaînes de ROP variées en code C et en code assembleur, pour ensuite être compilés en exécutables RISC-V. Pour des raisons de simplicité, les gadgets sont construits par le générateur, et non pas identifiés dans les binaires de librairies existantes. Chaque exécutable généré contient :

— une vulnérabilité très basique de débordement de tampon en C par la fonction `memcpy()`, donnant l'accès à l'attaquant au contrôle d'une zone dans la pile y compris d'une adresse de retour ;
— un grand nombre de gadgets ROP (de nombre paramétrable) répartis aléatoirement dans l'espace code de l'exécutable, pour imiter le comportement des attaques réelles qui utilisent différents gadgets dans différents emplacements de la mémoire ;
— une séquence d'exécution des gadgets (de longueur et de type d'instructions paramétrables), représentée par une liste d'adresses des gadgets qui seront placés dans la pile avec la vulnérabilité `memcpy()`.

L'objectif final de la séquence de gadgets est d'ouvrir un *shell* dans le terminal, c'est-à-dire exécuter l'équivalent du code C `execve("/bin/sh", {"/bin/sh", NULL}, {NULL})`. Comme chaque gadget contient principalement des instructions élémentaires, il est nécessaire de connaître les valeurs à positionner dans chaque registre pour cet appel système : `a7` prend la valeur 221 (qui identifie l'appel système `execve`), `a0` contient l'adresse de la chaîne de caractère `"/bin/sh"`, `a1` contient l'adresse contenant la valeur de `a0`, puis `a2` prend la valeur 0. Lorsque ces 4 registres sont correctement affectés, l'exécution de l'instruction `ecall` permet de déclencher l'appel système `execve`. Pour l'évaluation, nous générons des exécutables dont la longueur de la séquence des gadgets exécutée varie de 5 à 30 gadgets (chaque gadget contient une ou zéro action élémentaire), les gadgets étant mélangés avec 1000

autres gadgets dans la mémoire.

Il faut garder à l'esprit que les attaques ROP générées par notre générateur, même si elles ne représentent pas forcément les gadgets ROP et les attaques ROP dans la réalité, nous permettent de tester de multiples combinaison de chaînes et de gadgets et de vérifier la pertinence de notre détecteur dans des multiples cas.

5.2.3 Conception du détecteur et évaluation

Trois catégories de signaux sont potentiellement pertinentes pour la détection des attaques ROP :

— les signaux liés au flot d'instructions ;
— les signaux liés au pointeur d'instructions, aux adresses de sauts réalisés et leurs destinations, tels que les branchements et les retours de fonction ;
— les signaux des évènements de branchement, tels que le succès ou non de la prédiction de branchement, la prise ou non de branchement, etc.

Nous avons choisi d'analyser les signaux liés au flot d'instruction pour la détection, afin d'exploiter la possibilité de réutiliser les mêmes signaux pour la détection d'attaques différentes (CSCA et ROP). Pour l'utilisation des autres signaux, il est possible d'implémenter des algorithmes de détection tels que *shadow stack* [Lee 2015] ou basés sur les évènements [Wang 2016]. Nous avons configuré le MDM à la même fréquence que le cœur de processeur, avec un simple changement de paramètre grâce à la flexibilité de notre prototype. Ainsi, le *MDMcycle* est égal à un cycle d'horloge du processeur, et aucune connexion spéciale n'est nécessaire, ce qui signifie que les signaux supervisés sont connectés directement au MDM. Cette configuration nous permet de simplifier la conception de l'algorithme de détection, c'est-à-dire de réduire l'effort de parallélisation des logiques de traitements pour différents cas. En principe, le même algorithme dans le MDM pourrait être traduit en version parallèle et supporte donc un MDM plus lent que le processeur.

5.2.3.1 Détection basée sur instructions exécutées

Comme nous étions confiants dans le fait que les attaques ROP puissent être détectées en analysant le flot d'instructions, nous avons choisi les mêmes signaux supervisés que dans la section 5.1.3.1 : 32 bits d'instruction en cours d'exécution et 1 bit de validité de l'instruction exécutée, collectés dans le pipeline du processeur.

Nous avons ensuite essayé de construire une signature d'attaque ROP à partir de son fonctionnement et des travaux existants dans la littérature : une attaque ROP est caractérisée par un certain nombre de courtes séquences d'instructions consécutives (les gadgets) qui se terminent toujours par une instruction de saut indirect. Différentes variantes de ROP utilisent des sauts indirects générés de différentes façons, comme le retour de l'appel de fonction, l'appel de fonction indirecte, ou le saut indirect à l'adresse contenue dans un registre tout simplement. En RISC-V, toutes ces techniques utilisent l'instruction `jalr` [Waterman 2019].

Pour détecter ce comportement, nous avons construit l'algorithme présenté dans

le listing 5.6. Le nombre d'instructions entre deux sauts indirects est tracé à l'exécution dans un registre à décalage (`npack_jalr`) pour suivre la taille du potentiel gadget. Lorsqu'un saut indirect se produit (`in.pack_has_jalr` évalué à vrai), nous regardons si la distance en nombre d'instructions exécutées entre l'instruction courante et le dernier saut indirect est inférieure à un seuil (`gadgetSizeThresh`). Le cas échéant, on incrémente un compteur interne qui compte le nombre de gadgets courts consécutifs (`countgadget`) ; si la distance est supérieure au seuil, alors on décrémente le compteur de 2 (pour prendre en compte certaines chaînes de gadgets qui incluent parfois des gadgets longs afin d'esquiver la détection [Carlini 2014]). Nous considérons qu'une attaque est survenue lorsque le compteur interne `countgadget` atteint un certain seuil (`ropChainThresh`), qui représente globalement le nombre maximal de courtes séquences d'instructions autorisées dans une exécution légitime.

Listing 5.6 – Algorithme de détection des attaques ROP implémenté dans le MDM, basé sur les instructions de saut indirect relativement proches.

```
1  val npackSizeMax = 16
2  val gadgetSizeThresh = 8
3  val npack_jalr = RegInit(0.U(npackSizeMax.W)) // if last N valid
       cycles has jalr
4
5  val npack_jalr_orR1 = RegInit(false.B)
6  when (in.isMonitoring && in.pack_has_valid) {
7    npack_jalr := Cat(npack_jalr(npackSizeMax-2,0), in.pack_has_jalr)
8    npack_jalr_orR1 := npack_jalr(gadgetSizeThresh-2, 0).orR || in.
       pack_has_jalr
9  }
10
11 val countgadget = RegInit(0.U(mp.counterWidth.W))
12 when (in.resetCounters) {
13   countgadget := 0.U
14 }.otherwise {
15   when (in.isMonitoring && in.pack_has_jalr) { // evaluate counter
       when jalr
16     when (npack_jalr_orR1) { // has jalr adjacent
17         countgadget := countgadget + 1.U
18       }.otherwise { // no jalr adjacent
19         countgadget := Mux(countgadget >= 2.U, countgadget - 2.U,
       0.U)
20       }
21   }
22 }
```

Les seuils `gadgetSizeThresh` et `ropChainThresh` ont besoin d'être déterminés avec précaution afin de détecter au mieux les attaques ROP, avec le moins de faux positifs possibles. La taille en nombre d'instructions de chaque gadget dépend fortement des ISAs, et la longueur d'une chaîne de gadgets dépend fortement des attaques ROP existantes sur l'ISA donnée. À notre connaissance, il n'y a pas eu une telle étude exhaustive pour l'ISA RISC-V, mais nous pouvons nous appuyer sur les études sur l'ISA x86 [Cheng 2014, Chen 2009]. Ces études montrent que les

attaques ROP nécessitent en pratique une chaîne longue de 15 gadgets au minimum, avec au maximum 5 instructions dans chaque gadget en général (équivalent à 7 instructions en RISC-V en raison des 2 instructions supplémentaires pour la gestion du registre d'adresse de retour). En théorie, la chaîne ROP la plus courte est de longueur 5 : quatre actions élémentaires pour mettre la bonne valeur dans le registre correspondant, une dernière pour réaliser l'appel système. Il est donc pertinent de pouvoir détecter les chaînes possédant au minimum 5 gadgets. Cependant, en pratique, il est quasiment impossible de rencontrer une situation idéale dans laquelle les gadgets contiennent exactement les actions élémentaires nécessaires. Par exemple, pour affecter la valeur 221 au registre `a7`, il est très peu probable que l'on puisse disposer d'une instruction qui réalise exactement cette opération, et il est beaucoup plus probable que l'attaquant utilise 221 fois un gadget qui incrémente le registre `a7` de 1. Dans ce cas, la longueur de la chaîne ROP devient très élevée et donc largement au-dessus du seuil de 5, ce qui est cohérent avec la longueur de chaîne d'au moins 15 gadgets observée par les études sur l'ISA x86.

Pour cette raison, nous avons étudié la valeur maximum que peut atteindre `countgadget` pendant l'exécution des logiciels légitimes avec `gadgetSizeThresh` allant de 8 à 16 (pour couvrir la taille courante des gadgets, sans que trop de logiciels légitimes soient détectés comme une attaque). Nous avons également réalisé la même étude pour des logiciels malveillants, de longueur de chaîne et de taille de gadget différentes, générés par notre générateur de ROP. Nous avons ainsi pu vérifier que l'algorithme fonctionne comme prévu et agit correctement lorsque l'on fait varier ces deux caractéristiques de l'attaque. Les résultats sont fournis dans le tableau 5.6. Pendant l'évaluation des logiciels malveillants, nous avons constaté que les informations PID du noyau sont particulièrement importantes, pour filtrer les activités du système et empêcher les attaques ROP d'échapper à notre détection en raison des interruptions dans le comptage de longueur de chaîne de gadgets au moment du changement de contexte (`countgadget` décrémente pendant l'activité du système).

À la lumière des résultats du tableau 5.6, le choix du seuil `ropChainThresh` approprié selon `gadgetSizeThresh` est assez simple : il suffit de le définir comme la valeur maximale du `countgadget` mesurée sur la plupart des logiciels légitimes plus un. Globalement, nous pouvons observer que `ropChainThresh` doit augmenter avec l'augmentation de `gadgetSizeThresh`, pour éviter les faux positifs sur les logiciels légitimes. Comme indiqué précédemment, on peut s'attendre à ce que les attaques ROP pratiques sur RISC-V nécessitent au moins 15 gadgets d'au plus 7 instructions. Deux combinaisons de seuils nous semblent spécialement intéressantes pour couvrir les attaques ROP pratiques avec une marge de sécurité :

— Positionner `gadgetSizeThresh` à 8 et `ropChainThresh` à 5 permet de détecter assez tôt dans l'exécution toutes les chaînes de ROP correspondant à nos analyses basées sur les attaques x86 (seuls les premiers 5 gadgets sont exécutés avant la détection), et de couvrir les attaques ROP théoriquement les plus petites sur une architecture RISC-V.

— Pour anticiper l'usage des gadgets plus grands, par exemple pour les gadgets

TABLE 5.6 – Valeur maximale de *countgadget* observée en fonction des différentes valeurs de *gadgetSizeThresh* configurées dans la logique de détection.

Logiciel	**gadgetSizeThresh**								
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
CoreMark *	4	7	10	11	11	11	11	13	17
CoreMark-PRO *	4	7	10	11	11	11	11	13	54
Embench wikisort	4	758	788	800	801	801	801	801	801
Embench others *	4	7	10	11	11	11	11	13	17
Dhrystone	4	7	10	11	11	11	11	11	17
STREAM	4	7	7	11	11	11	11	11	17
ROP len 5	5	10	10	10	10	11	11	12	17
ROP len 8	8	10	10	10	11	11	11	12	17
ROP len 11	11	11	11	11	11	11	11	12	17
ROP len 15	15	15	15	15	15	15	15	15	17
ROP len 20	20	20	20	20	20	20	20	20	20

Seule une sélection des attaques ROP générées est présentée dans le tableau par simplicité, ce sont des attaques ROP de longueur de chaîne différentes, et dont la taille de chaque gadget est inférieur à 8 instructions. Les bancs de tests incluant plusieurs algorithmes sont annotés avec * et seule la valeur moyenne est représentée.

construits avec plus de 3 instructions en dehors des actions élémentaires, positionner `gadgetSizeThresh` à 14 et `ropChainThresh` à 12 semble une bonne option. Ceci permet de détecter les gadgets de taille jusqu'à 14 instructions, tout en détectant une exécution de ROP avant la fin (d'une chaîne de 15 gadgets), avec très peu de faux positifs sauf pour un algorithme présent dans le banc de test Embench-IOT.

Positionner `gadgetSizeThresh` à une valeur plus grande que 15 n'est pas intéressant, car il ne permet pas de couvrir les attaques ROP de 15 gadgets avec peu de faux positifs.

Nous avons conduit une analyse plus détaillée de l'algorithme wikisort dans le banc de test Embench-IOT afin de comprendre pourquoi il introduit beaucoup de faux positifs. En inspectant le code, nous avons découvert qu'il utilise plusieurs petites fonctions C non mise à plat dans la fonction appelante (appelé *inlining* en anglais) malgré leur faible taille, telles que `TestCompare()`. Ceci nous semble expliquer la longue "chaîne de petits gadgets" trouvée par notre algorithme de détection.

5.2.3.2 Surcoût de surface

Le surcoût de surface pour la détection de ROP avec l'algorithme présenté ci-dessus (appelée *InstJalr*) est de 1,90% de FFs et de 1,95% de LUTs par rapport au système de base. Cette valeur est beaucoup plus faible que dans le cas de la détection des CSCAs, principalement parce que le MDM est synchronisé avec le

composant cible et que l'effort de synchronisation des connexions est réduit.

5.2.3.3 Généricité des détecteurs

Nous avons analysé l'utilisation de l'algorithme *InstJalr* pour la détection des variantes des attaques et étudié s'il présente un certain intérêt pour les logiciels légitimes.

InstJalr permet non seulement de détecter les attaques ROP qui utilisent les instructions de retour de fonction, mais aussi celles qui utilisent les sauts indirects (*Jump-Oriented Programming*) [Bletsch 2011] et celles qui utilisent les appels de fonctions (*Pure Call-Oriented Programming*) [Sadeghi 2017]. En effet, comme nous l'avons considéré dès le début, une attaque ROP n'est pas limitée à l'utilisation des instructions de retour de fonction, elle peut utiliser les sauts indirects en général, notre algorithme est donc indépendant de l'utilisation de ces techniques.

En ce qui concerne les logiciels légitimes, *InstJalr* mesure l'utilisation intense de codes de petite taille, ce qui peut être impactant pour la performance des codes en général. Par conséquent, *InstJalr* permet de détecter un tel comportement sur les logiciels légitimes, et peut constituer un outil intéressant pour permettre d'optimiser le code de ces logiciels (ce qui est typiquement le cas pour l'algorithme wikisort de Embench-IOT).

5.2.4 Conclusion

Dans cette section, nous avons présenté l'utilisation de notre framework pour la conception et la détection d'un algorithme de détection pour les attaques ROP basé sur des signaux liés au flot d'instructions (*InstJalr*). Les résultats de détection sont très bons, avec très peu de faux positifs et des bons taux de détection, pour différentes longueurs de chaîne et tailles de gadgets.

Pour continuer à améliorer l'algorithme *InstJalr*, nous avons identifié deux pistes intéressantes. Premièrement, les signaux correspondant au défaut de prédiction de l'adresse de retour ou au défaut de prédiction de branchement peuvent contribuer à l'amélioration de la détection, car les gadgets vont généralement sauter à une adresse non prévisible. Deuxièmement, comme l'adresse du gadget n'est pas prévisible et est généralement non utilisée avant le début de chaîne de ROP (code non mis en cache), le temps de lecture des instructions à exécuter va être plus long. Cela pourrait contribuer à la détection des attaques ROP, et à la simplification de la logique de détection quand le MDM fonctionne plus lentement que le processeur.

5.3 Conclusion

Dans ce chapitre, nous avons présenté l'utilisation de notre framework pour concevoir et évaluer des algorithmes de détection pour deux catégories d'attaques de nature assez différente. Nous avons pu profiter de la flexibilité de notre approche, qui nous a permis de tester différents algorithmes sur une plateforme réaliste avec

relativement peu d'effort. Nous avons constaté que pour ajouter une attaque à détecter ou un algorithme de détection, certaines étapes se répètent (comme l'ajout des sondes et la définition de la méthode d'évaluation), l'effort de conception est donc réduit. Nous avons réussi à obtenir de bons taux de détection pour ces deux catégories d'attaques avec des algorithmes que nous avons conçus avec des heuristiques relativement simples. Ces algorithmes de détection sont facilement implémentables dans notre architecture de détection, qui résout notamment le problème de bande passante d'informations dont souffrent actuellement les propositions existantes. Différents algorithmes peuvent être activés en même temps dans le système final tant que les ressources disponibles dans le MDM sont suffisantes.

Même si nous n'avons pas eu le temps de conduire plus de recherches sur la détection d'autres types d'attaques spécifiques, nous sommes confiants sur le potentiel de la détection des différentes catégories d'attaque à la lecture des travaux de l'état de l'art et compte tenu de la généricité de notre approche. Ces travaux nous donnent une indication pour le choix des signaux intéressants à superviser, et nous sommes convaincus que les logiques de détection présentées dans ces travaux peuvent être adaptées et intégrées dans notre framework :

- Pour les attaques microarchitecturales (telles que les CSCAs et Spectre) en général, nous pouvons observer qu'elles présentent des motifs d'instructions courts et répétés, utilisant des composants matériels d'une manière particulière. En conséquence, il y a une forte probabilité qu'elles puissent être distinguées des logiciels légitimes en fonction des instructions exécutées et des évènements matériels, même à basse fréquence.
- Des mécanismes classiques de protection avec nécessité d'une zone mémoire matérielle isolée, tels que le DIFT basé sur les étiquettes, la vérification de l'intégrité du flot de contrôle et *shadow stack*, peuvent être implémentés dans le MDM grâce à la mémoire disponible sur la structure reconfigurable.
- Pour les algorithmes ayant besoin d'utiliser des machines à états ou de l'apprentissage automatique, ils pourront être implémentés dans le MDM reconfigurable et être accélérés, toujours en fonction de la surface disponible dans la structure reconfigurable.

Si la surface disponible n'est pas suffisante ou si certaines logiques sont trop complexes à mettre en place dans le MDM, le MDL peut compléter le MDM pour implémenter ces algorithmes. Il est également possible d'utiliser la reconfiguration dynamique et partielle pour mettre en œuvre différents algorithmes en fonction du contexte.

Notre framework possède néanmoins quelques limitations, que nous avons pu identifier notamment lors de nos différentes études de cas :

1. Au cours de la phase de conception, une compréhension approfondie des attaques cibles et des composants concernés est nécessaire, avec éventuellement un grand nombre d'expériences requises pour identifier les signaux les plus pertinents et concevoir la logique de détection. Cette limitation n'est pas vraiment spécifique à notre framework, mais est plutôt due à la complexité

des attaques et du système au niveau de la microarchitecture.

2. L'équilibre entre la complexité du circuit et la quantité d'information collectée n'est pas facile à trouver. La connexion parallèle qui conserve le plus d'informations nécessite beaucoup de ressources matérielles, mais la simplification de certaines connexions pourrait avoir un impact négatif sur la détection d'attaques non considérées ou actuellement inconnues.
3. Nous avons fait l'hypothèse que le noyau est de confiance, ou, du moins, que le MDL est de confiance. Ce qui n'est pas une hypothèse qui est forcément assurée. Plusieurs solutions peuvent être considérées pour assurer la sécurité du MDL en fonction de l'architecture du SoC considéré. Si le SoC embarque un *Trusted Execution Environment* (TEE) — ce qui est tout à fait réaliste pour les systèmes embarqués critiques qui sont les plus susceptibles d'adopter des mécanismes de protection comme notre framework — le MDL peut être implémenté dans le TEE. Sinon, si le SoC embarque un système d'exploitation "grand public", tel que le système d'exploitation Linux utilisé dans nos expériences, certaines mesures de protection bien connues du noyau peuvent être mises en œuvre. Nous pouvons également envisager qu'une partie du MDM se charge de vérifier l'intégralité du MDL au démarrage et pendant l'exécution.

Chapitre 6

Conclusion

Sommaire

6.1 Conclusion 111
6.2 Perspectives 112

6.1 Conclusion

Les systèmes informatiques se développent rapidement et sont aujourd'hui très répandus dans notre société et notre vie quotidienne. Malheureusement, ils sont aussi victimes d'attaques informatiques de plus en plus nombreuses et de plus en plus sophistiquées. La sécurisation de ces systèmes est donc devenue un objectif fondamental, mais cette mission n'est pas forcément évidente à accomplir face à la complexité des systèmes modernes. Les couches basses des systèmes informatiques (en particulier la microarchitecture) sont difficiles à protéger, car les attaques exploitant ces vulnérabilités sont relativement récentes, difficiles à détecter et bloquer, et peuvent provoquer de sérieux dégâts.

Dans cette thèse, nous avons cherché à proposer une solution flexible et performante pour la détection de différentes classes d'attaques logicielles et matérielles qui ont une empreinte spécifique sur la microarchitecture. Nous proposons un framework permettant l'intégration d'un détecteur couplant l'analyse matérielle de signaux microarchitecturaux avec des informations de plus haut niveau provenant du logiciel. La flexibilité de la détection est assurée par un module de détection reconfigurable, facilement intégrable sur différents systèmes cibles. Cette approche permet notamment d'affiner et de compléter les logiques de détection dans le temps, même après fabrication du composant. Nous avons proposé également une méthodologie de construction de détecteurs matériels et logiciels de manière itérative, reposant sur notre architecture, allant du choix des signaux matériels à observer à la logique de détection à déployer.

Un prototype open source utilisant le framework a été implémenté sur une carte d'évaluation ML605 (avec un FPGA), avec un système cible RISC-V fonctionnant sous Linux. Une étude de cas a été menée, dans laquelle nous décrivons comment nous avons pu construire, en suivant notre méthodologie itérative, des logiques de détection adaptées aux attaques par canal auxiliaire de cache et aux attaques ROP. Nous avons évalué et comparé différentes stratégies de détections d'attaques et nous

avons montré qu'il était possible de construire des détecteurs simples pour ces deux classes d'attaques sans faux positifs.

Notre étude a montré que du matériel reconfigurable et du logiciel, équipés de logiques de détection des attaques basées sur l'analyse de signaux microarchitecturaux, est une voie prometteuse pour détecter différentes classes d'attaques logicielles et matérielles sans pour autant impacter les performances, tout en offrant de la flexibilité dans la définition des logiques de détection. Le matériel pour la sécurité ne doit pas être limité à des solutions statiques qui ne peuvent pas être modifiées après la fabrication ou à des solutions qui ne comprennent que quelques bits de configuration programmables. La flexibilité pour s'adapter à de nouvelles menaces est particulièrement utile aujourd'hui, car la durée de vie du matériel est relativement longue par rapport à la vitesse d'évolution des attaques logicielles, et parce que les attaques logicielles sont capables de cibler les vulnérabilités du matériel. Les signaux microarchitecturaux fournissent une quantité importante d'informations sur le système, et il est intéressant de les considérer — non seulement ceux déjà disponibles dans les processeurs commerciaux — dans la conception des algorithmes de détection.

Suite à notre étude, les concepteurs de composants matériels ou de systèmes peuvent envisager d'inclure cette sécurité adaptable dans leur conception ; les chercheurs peuvent utiliser les outils, la méthodologie et le prototype proposés pour concevoir de nouveaux algorithmes de détection et mieux comprendre le comportement de la microarchitecture complexe des systèmes.

6.2 Perspectives

Plusieurs perspectives à ces travaux sont envisageables. Premièrement, enrichir l'étude de cas en se penchant sur la détection d'autres attaques, que ce soit par l'adaptation d'algorithmes existants, ou la conception de nouveaux. Il serait notamment intéressant d'apporter des preuves formelles sur la couverture des différentes classes d'attaques pour un jeu de signaux donnés.

Deuxièmement, étudier des logiques de détection plus complexes. En effet, même si les expériences que nous avons menées dans cette thèse tendent à montrer que des logiques de détection assez simples permettent déjà d'identifier sans faux positifs des logiciels malveillants pour les attaques temporelles sur les caches ou les attaques ROP, il serait intéressant d'étudier des algorithmes plus complexes (tels que des algorithmes d'apprentissage automatique et des machines à états) afin d'augmenter la couverture et la précision de la détection pour les attaques et leurs variantes découvertes pendant la durée de vie du système. Les algorithmes d'apprentissage pouvant parfois être coûteux, il serait également intéressant d'étudier jusqu'à quel point ces algorithmes peuvent être complètement embarqués dans la partie matérielle du module de détection, et à défaut déterminer les parties pouvant être implémentées matériellement et celles qui nécessitent une implémentation logicielle.

Troisièmement, étudier l'architecture et la méthodologie à mettre en place dans

les cas d'un système multithread et/ou multicœur. Par exemple, dans un système multithread, on pourrait imaginer utiliser une mémoire interne pour stocker les valeurs des registres pour chaque processus, et réaliser une détection par processus ainsi qu'une détection globale pour suivre le comportement global du système. Dans le cas d'un système multicœur, on pourrait évaluer les différentes possibilités, allant d'un seul MDM central jusqu'à un MDM par cœur, leur impact sur le système, leur performance, ainsi que les moyens de communications internes du MDM ou entre les MDMs à adopter.

Quatrièmement, étudier des moyens d'automatisation de sélection de signaux supervisés. Pour le moment, la sélection des signaux est faite manuellement selon notre connaissance sur l'attaque ou le composant vulnérable. Cela reste ainsi difficilement utilisable pour des utilisateurs qui sont peu ou pas formés dans le domaine de la sécurité. On pourrait envisager par exemple une sélection de signaux reposant sur une analyse statique à partir d'un modèle d'attaquant et d'un modèle, au moins partiel, du système.

Cinquièmement, étudier la conception de nouveaux composants reconfigurables adaptés aux besoins des logiques de détection. Les FPGAs sont en effets conçus de manière très généraliste pour supporter un très grand nombre d'algorithmes de nature très variés. Cependant, si nous souhaitons rendre viable l'intégration de composants reconfigurables dans les processeurs à des fins de sécurité, une spécialisation des FPGAs pour la sécurité permettrait de mieux préparer les composants élémentaires et les ressources de routage pour la détection, et d'augmenter la fréquence de fonctionnement.

Et sixièmement, étudier les possibilités d'extension du périmètre de la sonde, et ainsi étendre le périmètre du framework. Par exemple, les propriétés physiques peuvent être intéressantes à superviser, nous pouvons imaginer l'ajout des capteurs physiques en tant que sonde. Ils permettent potentiellement de détecter des attaques qui sont d'origine matérielle ou externe, et de détecter des signes d'anomalies pour assurer la sécurité-innocuité des systèmes.

Bibliographie

[Abadi 2009] Abadi, M., Budiu, M., Erlingsson, U. et Ligatti, J. *Control-Flow Integrity Principles, Implementations, and Applications.* ACM Transactions on Information and System Security (TISSEC), vol. 13, no. 1, pages 4 :1–4 :40, 2009. (Cité en page 101.)

[Akram 2020] Akram, A., Mushtaq, M., Bhatti, M. K., Lapotre, V. et Gogniat, G. *Meet the Sherlock Holmes' of Side Channel Leakage : A Survey of Cache SCA Detection Techniques.* IEEE Access, vol. 8, pages 70836–70860, 2020. (Cité en page 25.)

[Alibaba 2018] Alibaba. *Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances.* 2018. [En ligne]. Disponible : `https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057`. (Cité en page 42.)

[Amazon Web Services 2022] Amazon Web Services, I. *Amazon EC2 F1 Instances.* 2022. [En ligne]. Disponible : `https://aws.amazon.com/ec2/instance-types/f1/`. (Cité en page 42.)

[Amid 2020] Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A. J., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y. S., Asanovic, K. et Nikolic, B. *Chipyard : Integrated Design, Simulation, and Implementation Framework for Custom SoCs.* IEEE Micro, vol. 40, no. 4, pages 10–21, 2020. (Cité en page 65.)

[Appinventiv 2021] Appinventiv. *Google Play Store Stats and Facts You Should Know in 2021.* 2021. [En ligne]. Disponible : `https://appinventiv.com/blog/google-play-store-statistics/`. (Cité en page 6.)

[Arm 2010] Arm. *Arm Cortex-M3 Technical Reference Manual.* 2010. [En ligne]. Disponible : `https://developer.arm.com/documentation/ddi0337/h`. (Cité en page 35.)

[Arm 2011] Arm. *CoreSight Program Flow Trace Architecture Specification PFTv1.0 and PFTv1.1.* 2011. [En ligne]. Disponible : `https://developer.arm.com/documentation/ihi0035/b/`. (Cité en page 30.)

[Arm 2017] Arm. *ARM CoreSight Architecture Specification v3.0.* 2017. [En ligne]. Disponible : `https://developer.arm.com/documentation/ihi0029/e`. (Cité en page 30.)

[Arm 2018] Arm. *Arm Cortex-A53 MPCore Processor Technical Reference Manual r0p4.* 2018. [En ligne]. Disponible : `https://developer.arm.com/documentation/ddi0500/j/`. (Cité en page 34.)

[Arm 2020] Arm. *Arm Cortex-M55 Processor Technical Reference Manual.* 2020. [En ligne]. Disponible : `https://developer.arm.com/documentation/101051/0002`. (Cité en page 35.)

[Arm 2021] Arm. *Arm Partners are Shipping More than 900 Arm-based Chips per Second Based on Latest Results.* 2021. [En ligne]. Disponible : `https://www.arm.com/company/news/2021/05/arm-partners-are-shipping-more-than-900-arm-based-chips-per-second`. (Cité en page 6.)

[Asanović 2016] Asanović, K., Avižienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, P., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D., Richards, B., Schmidt, C., Twigg, S., Vo, H. et Waterman, A. *The Rocket Chip Generator.* Rapport technique UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016. (Cité en pages 27 et 67.)

[Aviram 2010] Aviram, A., Hu, S., Ford, B. et Gummadi, R. *Determinating Timing Channels in Compute Clouds.* Dans Proceedings of the 2010 ACM workshop on Cloud computing security workshop, page 103. ACM Press, 2010. (Cité en page 24.)

[Avizienis 2004] Avizienis, A., Laprie, J., Randell, B. et Landwehr, C. E. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pages 11–33, 2004. (Cité en page 12.)

[Bachrach 2012] Bachrach, J., Vo, H., Richards, B. C., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J. et Asanovic, K. *Chisel : Constructing Hardware in a Scala Embedded Language.* Dans The 49th Annual Design Automation Conference 2012, pages 1216–1225. ACM, 2012. (Cité en page 67.)

[Backer 2015] Backer, J., Hely, D. et Karri, R. *On Enhancing the Debug Architecture of a System-on-Chip (SoC) to Detect Software Attacks.* Dans 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), pages 29–34. IEEE, 2015. (Cité en pages 38 et 40.)

[Bao 2017] Bao, C. et Srivastava, A. *Exploring Timing Side-Channel Attacks on Path-ORAMs.* Dans 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 68–73. IEEE, 2017. (Cité en page 23.)

[Bhasin 2013] Bhasin, S., Danger, J.-L., Guilley, S., Ngo, X. T. et Sauvage, L. *Hardware Trojan Horses in Cryptographic IP Cores.* Dans 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 15–29. IEEE, 2013. (Cité en page 18.)

[Bhunia 2014] Bhunia, S., Hsiao, M. S., Banga, M. et Narasimhan, S. *Hardware Trojan Attacks : Threat Analysis and Countermeasures.* Proceedings of the IEEE, vol. 102, no. 8, pages 1229–1247, 2014. (Cité en page 18.)

[Bletsch 2011] Bletsch, T., Jiang, X., Freeh, V. W. et Liang, Z. *Jump-Oriented Programming : A New Class of Code-Reuse Attack.* Dans Proceedings of

the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11, page 30. ACM Press, 2011. (Cité en page 107.)

[Bossuet 2013] Bossuet, L., Grand, M., Gaspar, L., Fischer, V. et Gogniat, G. *Architectures of Flexible Symmetric Key Crypto Engines—a Survey : From Hardware Coprocessor to Multi-crypto-processor System on Chip.* ACM Computing Surveys, vol. 45, no. 4, pages 41 :1–41 :32, 2013. (Cité en page 42.)

[Brickell 2006] Brickell, E., Graunke, G., Neve, M. et Seifert, J.-P. *Software Mitigations to Hedge AES Against Cache-based Software Side Shannel Vulnerabilities.* Cryptology ePrint Archive, 2006. (Cité en page 23.)

[Camurati 2018] Camurati, G., Poeplau, S., Muench, M., Hayes, T. et Francillon, A. *Screaming Channels : When Electromagnetic Side Channels Meet Radio Transceivers.* Dans Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 163–177. ACM, 2018. (Cité en page 17.)

[Carlini 2014] Carlini, N. et Wagner, D. *ROP is Still Dangerous : Breaking Modern Defenses.* Dans Proceedings of the 23rd USENIX Security Symposium, pages 385–399. USENIX Association, 2014. (Cité en page 104.)

[Chattopadhyay 2013] Chattopadhyay, A. *Ingredients of Adaptability : A Survey of Reconfigurable Processors.* VLSI Design, vol. 2013, pages 1–18, 2013. (Cité en pages vii et 42.)

[Chen 2006] Chen, S., Falsafi, B., Gibbons, P. B., Kozuch, M., Mowry, T. C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G. R., Lin, B. et Schlosser, S. W. *Log-based Architectures for General-Purpose Monitoring of Deployed Code.* Dans Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, pages 63–65. ACM, 2006. (Cité en pages 31 et 32.)

[Chen 2009] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B. et Xie, L. *DROP : Detecting Return-Oriented Programming Malicious Code.* Dans Information Systems Security, 5th International Conference, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2009. (Cité en page 104.)

[Chen 2014] Chen, J. et Venkataramani, G. *CC-Hunter : Uncovering Covert Timing Channels on Shared Processor Hardware.* Dans 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pages 216–228. IEEE, 2014. (Cité en pages 25, 35 et 40.)

[Cheng 2014] Cheng, Y., Zhou, Z., Yu, M., Ding, X. et Deng, R. H. *ROPecker : A Generic and Practical Approach For Defending Against ROP Attacks.* Dans 21st Annual Network and Distributed System Security Symposium. The Internet Society, 2014. (Cité en pages 38, 39, 40, 102 et 104.)

[Chiappetta 2016] Chiappetta, M., Savas, E. et Yilmaz, C. *Real Time Detection of Cache-based Side-Channel Attacks Using Hardware Performance Counters.* Applied Soft Computing, vol. 49, pages 1162–1174, 2016. (Cité en pages 36, 40 et 96.)

[Clark 2013] Clark, S. S., Mustafa, H. A., Ransford, B., Sorber, J., Fu, K. et Xu, W. *Current Events : Identifying Webpages by Tapping the Electrical Outlet.* Dans 18th European Symposium on Research in Computer Security, volume 8134 of *Lecture Notes in Computer Science*, pages 700–717. Springer, 2013. (Cité en page 18.)

[Colangelo 2017] Colangelo, P., Luebbers, E., Huang, R., Margala, M. et Nealis, K. *Application of convolutional neural networks on Intel® Xeon® processor with integrated FPGA.* Dans 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7. IEEE, 2017. (Cité en page 42.)

[Dalton 2007] Dalton, M., Kannan, H. et Kozyrakis, C. *Raksha : A Flexible Information Flow Architecture for Software Security.* Dans 34th International Symposium on Computer Architecture, pages 482–493. ACM, 2007. (Cité en pages 31 et 32.)

[Dang 2015] Dang, T. H., Maniatis, P. et Wagner, D. *The Performance Cost of Shadow Stacks and Stack Canaries.* Dans Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, pages 555–566. ACM, 2015. (Cité en page 102.)

[Danger 2018] Danger, J.-L., Facon, A., Guilley, S., Heydemann, K., Kuhne, U., Si Merabet, A. et Timbert, M. *CCFI-Cache : A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity.* Dans 2018 21st Euromicro Conference on Digital System Design (DSD), pages 529–536. IEEE, 2018. (Cité en page 101.)

[Delshadtehrani 2020] Delshadtehrani, L., Canakci, S., Zhou, B., Eldridge, S., Joshi, A. et Egele, M. *PHMon : A Programmable Hardware Monitor and Its Security Use Cases.* Dans 29th USENIX Security Symposium, pages 807–824. USENIX Association, 2020. (Cité en pages 32 et 102.)

[Demme 2013] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S. et Stolfo, S. J. *On the Feasibility of Online Malware Detection with Performance Counters.* Dans The 40th Annual International Symposium on Computer Architecture, pages 559–570. ACM, 2013. (Cité en pages 37 et 40.)

[Deng 2010] Deng, D. Y., Lo, D., Malysa, G., Schneider, S. et Suh, G. E. *Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric.* Dans 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pages 137–148. IEEE Computer Society, 2010. (Cité en pages 43 et 44.)

[Deng 2012] Deng, D. Y. et Suh, G. E. *High-Performance Parallel Accelerator for Flexible and Efficient Run-Time Monitoring.* Dans IEEE/IFIP International Conference on Dependable Systems and Networks, pages 1–12. IEEE Computer Society, 2012. (Cité en pages 32 et 43.)

[Digital Science & Research Solutions, Inc. 2022] Digital Science & Research Solutions, Inc. *Dimensions*. 2022. [En ligne]. Disponible : `https://app.dimensions.ai/discover/publication`. (Cité en page 20.)

[EEJournal 2010] EEJournal. *Intel's Stellarton Mixes CPU and FPGA*. 2010. [En ligne]. Disponible : `https://www.eejournal.com/article/20101123-stellarton/`. (Cité en page 42.)

[Escouteloup 2021] Escouteloup, M., Lashermes, R., Fournier, J. et Lanet, J.-L. *Under the Dome : Preventing Hardware Timing Information Leakage*. Dans CARDIS 2021-20th Smart Card Research and Advanced Application Conference, page 21, 2021. (Cité en page 22.)

[Ettus Research 2022] Ettus Research. *USRP B200mini Data Sheet*. 2022. [En ligne]. Disponible : `https://www.ettus.com/wp-content/uploads/2019/01/USRP_B200mini_Data_Sheet.pdf`. (Cité en page 42.)

[Gartner 2021] Gartner. *Gartner Says Worldwide PC Shipments Grew 10.7% in Fourth Quarter of 2020 and 4.8% for the Year*. 2021. [En ligne]. Disponible : `https://www.gartner.com/en/newsroom/press-releases/2021-01-11-gartner-says-worldwide-pc-shipments-grew-10-point-7-percent-in-the-fourth-quarter-of-2020-and-4-point-8-percent-for-the-year`. (Cité en page 6.)

[Goldreich 1996] Goldreich, O. et Ostrovsky, R. *Software Protection and Simulation on Oblivious RAMs*. Journal of the ACM, vol. 43, no. 3, pages 431–473, 1996. (Cité en page 23.)

[Gordon 2021] Gordon, H., Park, C., Tushir, B., Liu, Y. et Dezfouli, B. *An Efficient SDN Architecture for Smart Home Security Accelerated by FPGA*. arXiv :2106.11390 [cs], 2021. [En ligne]. Disponible : `http://arxiv.org/abs/2106.11390`. (Cité en pages 43 et 44.)

[Gruss 2018] Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W. et Yarom, Y. *Another Flip in the Wall of Rowhammer Defenses*. Dans 2018 IEEE Symposium on Security and Privacy, pages 245–261. IEEE Computer Society, 2018. (Cité en page 19.)

[Gueron 2009] Gueron, S. *Intel's New AES Instructions for Enhanced Performance and Security*. Dans Fast Software Encryption, volume 5665, pages 51–66. Springer, 2009. (Cité en page 23.)

[Heiser 2019] Heiser, G. *Formal Verification and seL4*. 2019. [En ligne]. Disponible : `https://www.cse.unsw.edu.au/~cs9242/19/lectures/09a-sel4.pdf`. (Cité en page 11.)

[Hsueh 1997] Hsueh, M., Tsai, T. K. et Iyer, R. K. *Fault Injection Techniques and Tools*. Computer, vol. 30, no. 4, pages 75–82, 1997. (Cité en page 16.)

[Hutchings 2002] Hutchings, B., Franklin, R. et Carver, D. *Assisting Network Intrusion Detection with Reconfigurable Hardware*. Dans Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pages 111–120. IEEE Comput. Soc, 2002. (Cité en pages 43 et 44.)

[IDC 2021] IDC. *Smartphone Shipments Return to Positive Growth in the Fourth Quarter Driven by Record Performance by Apple, According to IDC*. 2021. [En ligne]. Disponible : `https://www.idc.com/getdoc.jsp?containerId=prUS47410621`. (Cité en page 6.)

[Intel 2016] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C & 3D) : System Programming Guide*. 2016. [En ligne]. Disponible : `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html`. (Cité en page 35.)

[Intel 2020] Intel. *Intel Quartus Prime Pro Edition User Guide : Debug Tools*. 2020. [En ligne]. Disponible : `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-debug.pdf`. (Cité en page 38.)

[Intel 2022] Intel. *Intel® FPGA Products - FPGA and SoC FPGA Devices and Solutions*. 2022. [En ligne]. Disponible : `https://www.intel.com/content/www/us/en/products/details/fpga.html`. (Cité en page 42.)

[Ioannou 2019] Ioannou, L. et Fahmy, S. A. *Network Intrusion Detection Using Neural Networks on FPGA SoCs*. Dans 2019 29th International Conference on Field Programmable Logic and Applications (FPL), pages 232–238, 2019. (Cité en pages 43 et 44.)

[Irazoqui 2018] Irazoqui, G., Eisenbarth, T. et Sunar, B. *MASCAT : Preventing Microarchitectural Attacks Before Distribution*. Dans Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, pages 377–388, 2018. (Cité en page 25.)

[Izraelevitz 2017] Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J. et Bachrach, J. *Reusability is FIRRTL Ground : Hardware Construction Languages, Compiler Frameworks, and Transformations*. Dans 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 209–216, 2017. (Cité en page 68.)

[Jaloyan 2020] Jaloyan, G., Markantonakis, K., Akram, R. N., Robin, D., Mayes, K. et Naccache, D. *Return-Oriented Programming on RISC-V*. Dans The 15th ACM Asia Conference on Computer and Communications Security, pages 471–480. ACM, 2020. (Cité en pages 100 et 102.)

[Kim 2014] Kim, Y., Daly, R., Kim, J. S., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K. et Mutlu, O. *Flipping Bits in Memory Without Accessing Them : An Experimental Study of DRAM Disturbance Errors*. Dans ACM/IEEE 41st International Symposium on Computer Architecture, pages 361–372. IEEE Computer Society, 2014. (Cité en page 19.)

[Kocher 1996] Kocher, P. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. Dans Advances in Cryptology — CRYPTO

'96, Lecture Notes in Computer Science, pages 104–113. Springer Berlin Heidelberg, 1996. (Cité en page 17.)

[Kocher 2019] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. et Yarom, Y. *Spectre Attacks : Exploiting Speculative Execution.* Dans 2019 IEEE Symposium on Security and Privacy, pages 1–19. IEEE, 2019. (Cité en pages 19, 20 et 82.)

[Kulah 2019] Kulah, Y., Dincer, B., Yilmaz, C. et Savas, E. *SpyDetector : An Approach for Detecting Side-Channel Attacks at Runtime.* International Journal of Information Security, vol. 18, no. 4, pages 393–422, 2019. (Cité en pages 36 et 40.)

[Laprie 1996] Laprie, J.-C., Arlat, J., Deswarte, Y., Powell, D., Kanoun, K., Kaâniche, M. et Crouzet, Y. Guide de la sûreté de fonctionnement. Cépaduès, 1996. (Cité en page 12.)

[Lee 2015] Lee, Y., Heo, I., Hwang, D., Kim, K. et Paek, Y. *Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices.* Dans Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, pages 3 :1–3 :8. ACM, 2015. (Cité en pages 30, 32, 102 et 103.)

[Li 2018a] Li, C. et Gaudiot, J. *Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters.* Dans 30th International Symposium on Computer Architecture and High Performance Computing, pages 25–28. IEEE, 2018. (Cité en pages 36 et 40.)

[Li 2018b] Li, W., Li, M., Ma, Y. et Yang, Q. *PMU-extended Hardware ROP Attack Detection.* Dans 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification, pages 183–187. IEEE, 2018. (Cité en pages 37 et 40.)

[Lipp 2018] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. et Hamburg, M. *Meltdown : Reading Kernel Memory from User Space.* Dans 27th USENIX Security Symposium, pages 973–990. USENIX Association, 2018. (Cité en page 82.)

[Liu 2015] Liu, F., Yarom, Y., Ge, Q., Heiser, G. et Lee, R. B. *Last-Level Cache Side-Channel Attacks are Practical.* Dans 2015 IEEE Symposium on Security and Privacy, pages 605–622. IEEE Computer Society, 2015. (Cité en page 84.)

[MAFTIA 2003] MAFTIA. *Malicious- and Accidental-Fault Tolerance for Internet Applications - Conceptual Model and Architecture.* MAFTIA Deliverable D21 Project IST-1999-11583, 2003. (Cité en page 14.)

[Mao 2020] Mao, Y., Migliore, V. et Nicomette, V. *REHAD : Using Low-Frequency Reconfigurable Hardware for Cache Side-Channel Attacks Detection.* Dans

IEEE European Symposium on Security and Privacy Workshops, pages 704–709. IEEE, 2020. (Cité en page 27.)

[Mao 2022] Mao, Y., Migliore, V. et Nicomette, V. *MATANA : A Reconfigurable Framework for Runtime Attack Detection Based on the Analysis of Microarchitectural Signals*. Applied Sciences, vol. 12, no. 3, page 1452, 2022. (Cité en pages 27 et 65.)

[Martin 2012] Martin, R., Demme, J. et Sethumadhavan, S. *TimeWarp : Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks*. Dans Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, pages 118–129. IEEE Computer Society, 2012. (Cité en page 24.)

[Mushtaq 2020] Mushtaq, M., Bricq, J., Bhatti, M. K., Akram, A., Lapotre, V., Gogniat, G. et Benoit, P. *WHISPER : A Tool for Run-Time Detection of Side-Channel Attacks*. IEEE Access, vol. 8, pages 83871–83900, 2020. (Cité en pages 36 et 40.)

[Nergal 2001] Nergal. *Advanced Return-into-lib(c) Exploits (PaX Case Study)*. Phrack Magazine, vol. 58, no. 4, 2001. [En ligne]. Disponible : `http://phrack.org/issues/58/4.html#article`. (Cité en page 19.)

[Nikhil 2021] Nikhil, R. S. *An Introduction to the Official Formal Specification of the RISC-V Instruction Set Architecture*. 2021. [En ligne]. Disponible : `https://open-src-soc.org/2021-03/media/slides/3rd-RISC-V-Meeting-2021-03-30-14h00-Rishiyur-Nikhil.pdf`. (Cité en page 12.)

[Nissim 2017] Nissim, N., Yahalom, R. et Elovici, Y. *USB-based Attacks*. Computers & Security, vol. 70, pages 675–688, 2017. (Cité en page 18.)

[Nvidia 2022] Nvidia. *Profiler User's Guide*. 2022. [En ligne]. Disponible : `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`. (Cité en page 35.)

[OpenSSL 2022] OpenSSL. *OpenSSL Changelog*. 2022. [En ligne]. Disponible : `https://www.openssl.org/news/changelog.html`. (Cité en page 22.)

[Osvik 2006] Osvik, D. A., Shamir, A. et Tromer, E. *Cache Attacks and Countermeasures : The Case of AES*. Dans The Cryptographers' Track at the RSA Conference 2006, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. (Cité en pages 19, 82 et 87.)

[Ozsoy 2016] Ozsoy, M., Khasawneh, K. N., Donovick, C., Gorelik, I., Abu-Ghazaleh, N. et Ponomarev, D. *Hardware-based Malware Detection Using Low-Level Architectural Features*. IEEE Transactions on Computers, vol. 65, no. 11, pages 3332–3344, 2016. (Cité en pages 38 et 40.)

[Payer 2016] Payer, M. *HexPADS : A Platform to Detect "Stealth" Attacks*. Dans Engineering Secure Software and Systems - 8th International Symposium, pages 138–154. Springer, 2016. (Cité en pages 36, 40 et 96.)

[Pham 2011] Pham, D. V., Syed, A. et Halgamuge, M. N. *Universal Serial Bus Based Software Attacks and Protection Solutions*. Digital Investigation, vol. 7, no. 3-4, pages 172–184, 2011. (Cité en page 18.)

[Podobas 2020] Podobas, A., Sano, K. et Matsuoka, S. *A Survey on Coarse-Grained Reconfigurable Architectures from a Performance Perspective*. IEEE Access, vol. 8, pages 146719–146743, 2020. (Cité en page 41.)

[Reid 2016] Reid, A., Chen, R., Deligiannis, A., Gilday, D., Hoyes, D., Keen, W., Pathirane, A., Shepherd, O., Vrabel, P. et Zaidi, A. *End-to-End Verification of ARM Processors with ISA-Formal*. Dans Computer Aided Verification, pages 42–58. Springer International Publishing, 2016. (Cité en page 12.)

[Reinbrecht 2020] Reinbrecht, C., Hamdioui, S., Taouil, M., Niazmand, B., Ghasempouri, T., Raik, J. et Sepúlveda, J. *LiD-CAT : A Lightweight Detector for Cache ATtacks*. Dans IEEE European Test Symposium, pages 1–6. IEEE, 2020. (Cité en pages 39 et 40.)

[Rupp 2015] Rupp, K. *40 Years of Microprocessor Trend Data*. 2015. [En ligne]. Disponible : `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`. (Cité en page 7.)

[Sadeghi 2017] Sadeghi, A., Niksefat, S. et Rostamipour, M. *Pure-Call Oriented Programming (PCOP) : Chaining the Gadgets using Call Instructions*. Journal of Computer Virology and Hacking Techniques, vol. 14, no. 2, pages 139–156, 2017. (Cité en page 107.)

[Shacham 2007] Shacham, H. *The Geometry of Innocent Flesh on the Bone : Return-into-libc Without Function Calls (on the x86)*. Dans Proceedings of the 2007 ACM Conference on Computer and Communications Security, pages 552–561. ACM, 2007. (Cité en pages 19 et 100.)

[SiFive 2021] SiFive. *SiFive Freedom Unleashed SDK*. 2021. [En ligne]. Disponible : `https://github.com/sifive/freedom-u-sdk/tree/v1_0`. (Cité en page 75.)

[Skorobogatov 2017] Skorobogatov, S. *How Microprobing Can Attack Encrypted Memory*. Dans Euromicro Conference on Digital System Design, pages 244–251. IEEE Computer Society, 2017. (Cité en page 17.)

[Souza 2018] Souza, M. A., Maciel, L. A., Penna, P. H. et Freitas, H. C. *Energy Efficient Parallel K-Means Clustering for an Intel® Hybrid Multi-Chip Package*. Dans 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pages 372–379. IEEE, 2018. (Cité en page 42.)

[Stefanov 2018] Stefanov, E., Dijk, M. V., Shi, E., Chan, T.-H. H., Fletcher, C., Ren, L., Yu, X. et Devadas, S. *Path ORAM : An Extremely Simple Oblivious RAM Protocol*. Journal of the ACM, vol. 65, no. 4, pages 18 :1–18 :26, 2018. (Cité en page 23.)

[Szekeres 2013] Szekeres, L., Payer, M., Tao Wei et Song, D. *SoK : Eternal War in Memory.* Dans 2013 IEEE Symposium on Security and Privacy (SP) Conference, pages 48–62. IEEE, 2013. (Cité en page 19.)

[Vipin 2018] Vipin, K. et Fahmy, S. A. *FPGA Dynamic and Partial Reconfiguration : A Survey of Architectures, Methods, and Applications.* ACM Computing Surveys (CSUR), vol. 51, no. 4, pages 72 :1–72 :39, 2018. (Cité en page 73.)

[Wahab 2018] Wahab, M. A., Cotret, P., Allah, M. N., Hiet, G., Biswas, A. K., Lapotre, V. et Gogniat, G. *A Small and Adaptive Coprocessor for Information Flow Tracking in ARM SoCs.* Dans 2018 International Conference on ReConFigurable Computing and FPGAs, pages 1–8. IEEE, 2018. (Cité en pages 31 et 32.)

[Wang 2016] Wang, X. et Backer, J. *SIGDROP : Signature-based ROP Detection using Hardware Performance Counters.* arXiv :1609.02667 [cs], 2016. [En ligne]. Disponible : `http://arxiv.org/abs/1609.02667`. (Cité en pages 37, 40, 102 et 103.)

[Waterman 2019] Waterman, A. et Asanovic, K. *The RISC-V Instruction Set Manual, Volume I : User-Level ISA.* 2019. [En ligne]. Disponible : `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf`. (Cité en page 103.)

[Waterman 2021] Waterman, A., Asanovic, K., Hauser, J. et Division, C. *The RISC-V Instruction Set Manual, Volume II : Privileged Architecture.* 2021. [En ligne]. Disponible : `https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf`. (Cité en page 24.)

[Xilinx 2012] Xilinx. *UG029 - ChipScope Pro Software and Cores.* 2012. [En ligne]. Disponible : `https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/chipscope_pro_sw_cores_ug029.pdf`. (Cité en page 38.)

[Xilinx 2015] Xilinx. *Virtex-6 Family Overview.* 2015. [En ligne]. Disponible : `https://docs.xilinx.com/v/u/en-US/ds150`. (Cité en page 66.)

[Xilinx 2022] Xilinx. *Xilinx Adaptive SoCs.* 2022. [En ligne]. Disponible : `https://www.xilinx.com/products/silicon-devices/soc.html`. (Cité en pages 30 et 42.)

[Yarom 2014] Yarom, Y. et Falkner, K. *FLUSH+RELOAD : A High Resolution, Low Noise, L3 Cache Side-Channel Attack.* Dans Proceedings of the 23rd USENIX Security Symposium, pages 719–732. USENIX Association, 2014. (Cité en page 20.)

[Yarom 2016] Yarom, Y. *Mastik : A Micro-Architectural Side-Channel Toolkit.* 2016. [En ligne]. Disponible : `https://cs.adelaide.edu.au/~yval/Mastik/`. (Cité en page 86.)

[Zhang 2013] Zhang, Y. et Reiter, M. K. *Düppel : Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud*. Dans 2013 ACM SIGSAC Conference on Computer and Communications Security, pages 827–838. ACM, 2013. (Cité en page 21.)

[Zhao 2020] Zhao, J., Korpan, B., Gonzalez, A. et Asanovic, K. *SonicBOOM : The 3rd Generation Berkeley Out-of-Order Machine*. Fourth Workshop on Computer Architecture Research with RISC-V, page 7, 2020. (Cité en page 69.)

Titre : Détection dynamique d'attaques logicielles et matérielles basée sur l'analyse de signaux microarchitecturaux

Résumé : Les systèmes informatiques ont évolué rapidement ces dernières années, ces évolutions touchant toutes les couches des systèmes informatiques, du logiciel (systèmes d'exploitation et logiciels utilisateur) au matériel (microarchitecture et technologie des puces). Si ce développement a permis d'accroître les fonctionnalités et les performances, il a également augmenté la complexité des systèmes (rendant plus difficile la compréhension globale du système), et par là même augmenté la surface d'attaque pour les pirates. Si les attaques ont toujours ciblé les vulnérabilités logicielles, au cours des deux dernières décennies, les attaques exploitant les vulnérabilités matérielles des systèmes informatiques sont devenues suffisamment graves pour ne plus être ignorées. En 2018, par exemple, la divulgation des attaques Spectre et Meltdown a mis sur le devant de la scène les problèmes que peuvent poser certaines optimisations faites dans la microarchitecture des systèmes. Malheureusement, la détection et la protection contre ces attaques se révèlent particulièrement complexes, et posent donc aujourd'hui de nombreux défis : (1) le niveau élevé de complexité et de variabilité de la microarchitecture implique une grande difficulté à identifier les sources de vulnérabilité ; (2) les contremesures impliquant une modification de la microarchitecture peuvent impacter significativement les performances globales du système complet ; et (3) les contremesures doivent pouvoir s'adapter à l'évolution des attaques. Pour donner des éléments de réponse, cette thèse s'est intéressée à l'utilisation des informations qui sont disponibles au niveau de la microarchitecture pour construire des méthodes de détection efficaces.

Ces travaux ont en particulier abouti à la construction d'un framework permettant la détection d'attaques qui laissent des empreintes au niveau de la couche microarchitecturale. Ce framework propose : (1) d'utiliser les informations microarchitecturales pour la détection des attaques, couvrant efficacement les attaques visant les vulnérabilités microarchitecturales ; (2) de proposer une méthodologie pour aider les concepteurs dans le choix des informations pertinentes à extraire de la microarchitecture ; (3) d'utiliser des connexions dédiées pour la transmission de ces informations microarchitecturales afin de garantir une haute bande passante ; et (4) d'utiliser du matériel reconfigurable en conjonction avec du logiciel pour implémenter la logique de détection des attaques. Cette combinaison de logiciel et matériel reconfigurable (constituant le module de détection) permet à la fois de réduire l'impact sur les performances grâce à l'accélération matérielle, et de mettre à jour la logique de détection afin de s'adapter à l'évolution des menaces par la reconfiguration au cours du cycle de vie du système. Nous présentons en détails les changements requis au niveau de la microarchitecture et du système d'exploitation, la méthodologie pour sélectionner les informations microarchitecturales appropriées, l'intégration de ce framework dans un système informatique spécifique, ainsi que la description du fonctionnement du système final pendant son cycle de vie. Cette thèse décrit pour finir deux cas d'étude menés sur un prototype (basé sur un cœur RISC-V) sur un FPGA, et montre comment des logiques relativement simples implémentées dans le module de détection nous ont permis de détecter des attaques de classes différentes (attaques visant les caches et attaques de type ROP) sur un système complet exécutant un système d'exploitation, via l'exploitation d'informations provenant de la microarchitecture.

Mots clés : Sécurité logicielle et matérielle ; Système de détection d'intrusion ; Co-conception matériel/logiciel ; Matériel reconfigurable

Title : Dynamic software and hardware attack detection based on microarchitectural signals analysis

Abstract : In recent years, computer systems have evolved quickly. This evolution concerns different layers of the system, both software (operating systems and user programs) and hardware (microarchitecture design and chip technology). While this evolution allows to enrich the functionalities and improve the performance, it has also increased the complexity of the systems. It is difficult, if not impossible, to fully understand a particular modern computer system, and a greater complexity also stands for a larger attack surface for hackers. While most of the attacks target software vulnerabilities, over the past two decades, attacks exploiting hardware vulnerabilities have emerged and demonstrated their serious impact. For example, in 2018, the Spectre and Meltdown attacks have been disclosed, that exploited vulnerabilities in the microarchitecture layer to allow powerful arbitrary reads, and highlighted the security issues that can arise from certain optimizations of system microarchitecture. Detecting and preventing such attacks is not intuitive and there are many challenges to deal with : (1) the great difficulty in identifying sources of vulnerability implied by the high level of complexity and variability of different microarchitectures ; (2) the significant impact of countermeasures on overall performance and on modifications to the system's hardware microarchitecture is generally not desired ; and (3) the necessity to design countermeasures able to adapt to the evolution of the attack after deployment of the system. To face these challenges, this thesis focuses on the use of information available at the microarchitecture level to build efficient attack detection methods.

In particular, we describe a framework allowing the dynamic detection of attacks that leave fingerprints at the system's microarchitecture layer. This framework proposes : (1) the use of microarchitectural information for attack detection, which can effectively cover attacks targeting microarchitectural vulnerabilities ; (2) a methodology that assists designers in selecting relevant microarchitectural information to extract ; (3) the use of dedicated connections for the transmission of information extracted, in order to ensure high transmission bandwidth and prevent data loss ; and (4) the use of reconfigurable hardware in conjunction with software to implement attack detection logic. This combination (composing the so-called detection module) reduces the performance overhead through hardware acceleration, and allows updating detection logic during the system lifetime with reconfiguration in order to adapt to the evolution of attacks. We present in detail the proposed architecture and modification needed on the system, the methodology for selecting appropriate microarchitectural information and for integrating this framework into a specific computer system, and we describe how the final system integrating our detection module is able to detect attacks and adapt to attack evolution. This thesis also provides two use case studies implemented on a prototype (based on a RISC-V core with a Linux operating system) on an FPGA. It shows that, thanks to the analysis of microarchitectural information, relatively simple logic implemented in the detection module is sufficient to detect different classes of attacks (cache side-channel attack and ROP attack).

Keywords : Software and hardware security ; Intrusion detection system ; Hardware/software co-design ; Reconfigurable hardware