Garantir l'isolation microarchitecturale des processeurs

Mathieu Escouteloup

▶ To cite this version:

Mathieu Escouteloup. Garantir l'isolation microarchitecturale des processeurs. Cryptographie et sécurité [cs.CR]. Université de Rennes, 2021. Français. NNT : 2021REN1S109 . tel-03684907

HAL Id: tel-03684907
https://theses.hal.science/tel-03684907v1

Submitted on 1 Jun 2022

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

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

UNIVERSITÉ DE RENNES 1

THESE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique

Par
Mathieu ESCOUTELOUP

Garantir l'isolation microarchitecturale des processeurs

Thèse présentée et soutenue à Rennes, le jeudi 16 décembre 2021
Unité de recherche : IRISA
Thèse N° :

Rapporteurs avant soutenance :

Guy Gogniat — Professeur à l'Université Bretagne Sud, France
David Monniaux — Directeur de recherche à VERIMAG, France

Composition du Jury :

Examinateurs :	Régis Leveugle	Professeur à Grenoble INP, France
	Karine Heydemann	Maître de conférence à Sorbonne Université, France
	Clémentine Maurice	Chargée de recherche au CNRS, France
Dir. de thèse :	Christophe Bidan	Professeur à CentraleSupelec, France
Co-dir. de thèse :	Jacques Fournier	Ingénieur de recherche au CEA Leti, France
Encadrant :	Ronan Lashermes	Ingénieur de recherche à l'Inria Rennes, France

Remerciements

Avant toute discussion technique, je souhaiterais remercier l'ensemble des personnes qui m'ont soutenu durant ces années. Cette thèse est aussi la vôtre, car rien n'aurait été possible seul.

Tout d'abord, je tiens à remercier Ronan Lashermes qui fut bien plus qu'un simple encadrant. Je ne compte plus les heures à discuter et à échanger sur des thèmes divers et variés, tes nombreux encouragements, tes conseils . . . Sans oublier ton inestimable support technique ! Merci à Jacques Fournier pour ta confiance et pour tes nombreux conseils qui, même à distance, n'en étaient pas moins essentiels. Je tiens également à remercier Jean-Louis Lanet et Christophe Bidan qui ont accepté de faire un bout de ce chemin avec moi. Enfin, je voudrais remercier Guy Gogniat et David Monniaux qui ont accepté de relire cette thèse, ainsi que Clémentine Maurice, Karine Heydemann et Régis Leveugle pour avoir accepté d'être membres de mon jury.

Si cette thèse n'est pas la finalité de mon parcours, elle n'en est pas pour autant la première étape. C'est aussi la suite de plusieurs années de rencontres. Évidemment, il y a les professeurs que j'ai pu rencontrer, d'Arthez à Grenoble en passant par Orthez et Toulouse. Mais aussi, je tiens à remercier Grégoire avec qui j'ai tant appris durant mes trois années d'apprentissage. C'est avec toi que j'ai découvert et mis en pratique tout un pan de la conception de circuits numériques dont aujourd'hui encore je ne cesse de me servir.

Je remercie et également mes collègues et amis. Benoît, Cédric, Kévin, Léopold, Ludovic, Pierre-Yves avec qui j'ai eu l'occasion lors de nombreuses pauses-café / envahissements de bureau / discussions continues en distanciel de défaire et refaire le monde.Bastou, Tochou, Lutente, Pénou, Lamy et Gachette avec qui j'ai passé tant de temps à discuter de problèmes logiques et bien moins logiques alors que finalement, c'était d'aller aux Copains qui nous intéressait. Baptiste, Chloé, Victor et Cécile avec qui les sujets de discussions et activités réconfortantes ne manquent pas. Et enfin Claire, Léa, Marie et Quentin avec qui j'ai grandi depuis tout ce temps jusqu'à en arriver là.

Finalement, je tiens à remercier mes proches. Merci Mathilde pour tes encouragements, pour ton soutien en toute circonstance, mais aussi pour avoir osé relire ce document si obscur. Merci Rebloch et Oïana qui se sont avérés être deux co-bureaux certes inattendus, mais très efficaces. Merci à ma famille et mes grands-parents pour toutes ces petites attentions et ces fameux repas qui remontent le moral. Merci à Marion d'avoir choisi une direction proche : enfin quelqu'un avec qui on peut parler de son boulot ! Et enfin merci à mes parents pour leur exemple au quotidien, pour leur soutien sans faille et pour avoir toujours fait en sorte que j'ai les meilleures chances de réussir.

Sommaire

Remerciements i

Sommaire iii

Liste des figures viii

Liste des tableaux ix

Liste des codes x

Liste des acronymes xi

1. Introduction 1
- 1.1. Contexte . . . 1
- 1.2. Sécurité dans les systèmes . . . 2
- 1.3. Objectifs et contributions . . . 3
- 1.4. Organisation du manuscrit . . . 4

Comprendre les microarchitectures modernes 5

2. Composants microarchitecturaux 7
- 2.1. Principes généraux . . . 7
  - Circuits logiques et mémoires . . . 7
  - Organisation du processeur . . . 9
  - Caractéristiques des jeux d'instructions . . . 11
- 2.2. L'enjeu des performances . . . 12
  - Limites du modèle de base . . . 13
  - Utilisation d'un pipeline . . . 13
  - Mémoires caches . . . 15
- 2.3. Spéculation . . . 19
  - Pipeline et exécution dans le désordre . . . 19
  - Prédiction dynamique de branchement . . . 20
  - Prefetcher . . . 22
- 2.4. Parallélisme des exécutions . . . 22
- 2.5. Mémoire virtuelle . . . 23
- 2.6. Mécanismes de sécurité . . . 24
  - Privilèges . . . 24
  - Exceptions et interruptions . . . 25
- 2.7. Conclusion . . . 26

3. Exploitation de la microarchitecture par le logiciel 27
- 3.1. Objectifs de sécurité . . . 27
- 3.2. Principes microarchitecturaux exploités . . . 28
- 3.3. Attaques par analyse des variations temporelles . . . 29
  - Principes . . . 29
  - Attaques sur les mémoires caches . . . 30
  - Étude généralisée du système . . . 33

3.4. Attaques par contention dynamique de ressources 36
Principes 37
Impact sur les timings 37
3.5. Attaques par exécution transitoire 38
Principes 38
Spectre 39
Meltdown 41
Échantillonnage des données microarchitecturales 42
3.6. Attaques par canal contrôlé 44
3.7. Synthèse et conclusion 45

4. Solutions existantes contre l'exploitation logicielle. 49
4.1. Solutions pour la gestion des ressources partagées 49
Suppression des mécanismes 49
Effacement des traces 50
Partitionnement des ressources 52
4.2. Solutions pour la gestion de la spéculation 55
Modifications du logiciel 55
Renforcement du matériel 57
4.3. Solutions de gestion du temps et des évènements 59
Modification des évènements microarchitecturaux 60
Indéterminisme du fonctionnement 60
Altération des mesures de temps 61
Exécution en temps constant 62
4.4. Stratégie globale 63
Prise en compte ciblée des failles. 63
Rôles des couches d'abstraction 64
4.5. Conclusion et suite 67

REPENSER LE JEU D'INSTRUCTIONS 69

5. Modifier l'ISA pour la sécurité 71
5.1. Principes de base 71
5.2. Organisation des domaines de sécurité 72
Hiérarchie statique 72
Hiérarchie hybride 74
Hiérarchie dynamique 77
5.3. Politique d'isolation microarchitecturale dans l'ISA 78
Forme des modifications de l'ISA 78
Stratégies d'abstraction du matériel 79
5.4. Conclusion 80

6. Domes 83
6.1. Stratégie de modification de l'ISA 83
ISA de base : RISC-V 32 bits 83
Objectifs d'implémentation 83
6.2. Proposition d'implémentation 84
Modèles et représentation 84
Opérations et instructions 86
Utilisation des capacités : les exceptions 90
6.3. Support d'une politique d'isolation 92

6.4. Impact sur le logiciel . . . 93
Conception de la hiérarchie . . . 93
Compilation . . . 96
Scénario de `dome.switch` fréquents . . . 96
6.5. Extension du modèle . . . 97
Accès aux compteurs de performances . . . 97
Intégrité du flot d'exécution . . . 97
Chiffrement et authentification de la mémoire . . . 98
Augmentation du nombre de domes . . . 98
6.6. Version rétrocompatible avec les privilèges statiques . . . 100
6.7. Conclusion . . . 101

ADAPTER LA MICROARCHITECTURE **103**

7. Conception des ressources partagées **105**
7.1. Définitions . . . 105
Ressource partagée . . . 105
Problématique et objectifs . . . 106
7.2. Allocation statique des ressources . . . 108
Principe . . . 108
Mécanismes pour l'implémentation . . . 108
7.3. Séparation des ressources . . . 109
Principes . . . 109
Mécanismes pour l'implémentation . . . 109
7.4. Suppression des traces . . . 111
Principe . . . 112
Mécanismes pour l'implémentation . . . 112
7.5. Homogénéité . . . 113
7.6. Modèle mémoire et renforcement des frontières . . . 114
Objectif . . . 114
Accès contrôlé . . . 114
Duplication . . . 115
Cohérence . . . 115
7.7. Conclusion . . . 116

8. Implémentation d'une politique d'isolation **119**
8.1. Contexte . . . 119
8.2. Processeur Aubrac . . . 120
Description de l'architecture de base . . . 120
Support des domes . . . 121
8.3. Processeur Salers . . . 122
Description de l'architecture de base . . . 122
Support des domes . . . 123
8.4. Mémoire cache . . . 125
Fonctionnement général des caches . . . 125
Partie "mémoire" des caches . . . 126
Partie "contrôleur" des caches . . . 127
Partie "bus" des caches . . . 127
8.5. Généricité de l'approche . . . 129
8.6. Autres mécanismes . . . 130

8.7. Conclusion 130

9. Évaluation **133**
9.1. Timesecbench : une suite de tests pour la sécurité 133
Objectif et contraintes de développement 133
Scénario d'attaquant 134
Vulnérabilités couvertes par Timesecbench 134
9.2. Mesure d'efficacité des contremesures 136
Comparaison des implémentations vulnérables et sécurisées 136
Isolation temporelle 137
Isolation spatiale 139
9.3. Analyse des performances et du coût 141
Performances 141
Coût en ressources matérielles 144
9.4. Bilan et conclusion 145

10. Conclusion **147**
10.1. Résumé des travaux 147
10.2. Bilan synthétique 149
10.3. Travaux futurs et perspectives 149

Publications et communications **153**

Bibliographie **155**

ANNEXES 165

A. Paramètres des processeurs **167**
A.1. Processeur Aubrac 167
A.2. Processeur Salers 168

B. Liste des instructions pour les domes **169**

Liste des figures

1.1. Nombre de transistors par circuit électronique. 1

2.1. Représentation d'un signal d'horloge. 8
2.2. Représentation d'un circuit logique et son chemin critique. 8
2.3. Modèle d'architecture de von Neumann. 9
2.4. Modèle d'architecture d'Harvard. 10
2.5. Représentation d'un système informatique. 10
2.6. Représentation d'un pipeline 5 étages. 14
2.7. Avancée des instructions dans un pipeline. 14
2.8. Découpage d'une adresse dans un cache. 16
2.9. Cache avec une organisation associative. 16
2.10. Cache avec une organisation directe. 16
2.11. Cache avec une organisation associative par ensemble. 16
2.12. Politique LRU utilisée dans un cache. 18
2.13. Exemple de hiérarchie mémoire. 18
2.14. Impact d'une mauvaise prédiction de branchement dans un pipeline. 20
2.15. Principe de fonctionnement du BTB. 21
2.16. Principe de fonctionnement du BHT. 21
2.17. Machine d'état du compteur du BHT. 21
2.18. Principe de fonctionnement du RSB. 22
2.19. Illustration du multicœur. 22
2.20. Illustration du multithreading. 23
2.21. Fonctionnement d'une machine virtuelle. 23

3.1. Différents niveaux de partage. 36
3.2. Contention d'une ressource 37
3.3. Evolution d'une attaque par exécution transitoire. 39
3.4. Déroulement d'une attaque par échantillonnage microarchitectural. 43
3.5. Classification des attaques logicielles exploitant la microarchitecture. 47

4.1. Opération de flush. 51
4.2. Partitionnement spatial des ressources. 52
4.3. Partitionnement temporel d'une ressource. 54

5.1. Hiérarchie statique des domaines de sécurité. 72
5.2. Hiérarchie hybride des domaines de sécurité. 74
5.3. Hiérarchie dynamique des domaines de sécurité. 77
5.4. Opération de séparation sur la hiérarchie. 77
5.5. Opération de fusion sur la hiérarchie. 78

6.1. Description de la table d'un dome. 85
6.2. Opérations sur les CSRs lors d'un `dome.switch` (modèle hybride). 86
6.3. Modifications de l'état d'une configuration avec `dome.check`. 89
6.4. Déroulement de l'exécution d'un `dome.switch`. 90
6.5. Cycle de gestion d'une exception. 91
6.6. Mécanisme d'exception dans l'ISA RISC-V. 91
6.7. Mécanisme d'exception avec les domes. 92

6.8. Division d'un dome en logiciel. 94
6.9. Fusion d'un dome en logiciel. 94
6.10. Scénario d'application des domes : démarrage sécurisé. 95
6.11. Opérations sur les CSRs lors d'un `dome.switch` (modèle hybride). 100

7.1. Partage d'une ressource entre plusieurs entités. 105
7.2. Duplication d'une ressource pour chaque entité. 105
7.3. Partitionnement spatial d'une mémoire. 109
7.4. Implémentation du partitionnement spatial d'un cache. 109
7.5. Disponibilité d'une ressource avec un partitionnement temporel. 110
7.6. Partitionnement d'une ressource avec latence. 110
7.7. Partitionnement temporel de deux étages d'un pipeline. 111
7.8. Cycle de vie d'une ressource partagée. 112
7.9. Architecture hétérogène *vs.* homogène. 113

8.1. Vue globale de la microarchitecture Aubrac. 120
8.2. Vue globale de la microarchitecture Salers. 122
8.3. Vue globale de l'unité de gestion des domes. 123
8.4. Scénario d'utilisation d'une unité de répartition. 124
8.5. Vue globale de la mémoire cache. 125
8.6. Bus mémoire version 1 : lecture et écriture. 128
8.7. Bus mémoire version 1 : contention 128
8.8. Bus mémoire version 2 : support des domes 129

9.1. Scénario utilisé pour Timesecbench. 134
9.2. Scénario utilisé pour les caches L1 partagés temporellement. 134
9.3. Scénario utilisé pour la contention d'unité d'exécution. 136
9.4. Évaluation de l'isolation du L1D avec partage temporel. 137
9.5. Évaluation de l'isolation du L1I avec partage temporel. 137
9.6. Évaluation de l'isolation du BTB avec partage temporel. 138
9.7. Évaluation de l'isolation du BHT avec partage temporel. 138
9.8. Évaluation de l'isolation du L1D avec partage spatial. 139
9.9. Évaluation de l'isolation de la contention dynamique d'une unité d'exécution. 140

Liste des tableaux

2.1. Table de vérité d'un NON. 7
2.2. Table de vérité d'un OU. 7
2.3. Représentation d'une instruction à différents niveaux du système. 9

4.1. Comparaisons des approches par niveau d'abstraction. 67

6.1. Description des capacités existantes. 85
6.2. Détail des champs composant une configuration de dome. 85
6.3. Description des informations de statut. 86
6.4. Différents états d'une configuration de dome. 87
6.5. Liste des instructions arithmétiques et logiques sur les domes. 87
6.6. Liste des instructions mémoires sur les domes. 88
6.7. Liste des instructions de statut des domes. 89
6.8. Liste des instructions de switch des domes. 90
6.9. Description des informations d'instance. 93
6.10. Ajout d'une clé 128 bits à chaque configuration. 98
6.11. Exemples illustrant l'impact de l'index. 99

7.1. Récapitulatif des principes de conception et mécanismes. 117

8.1. Latence pour des accès au cache L1. 127
8.2. Récapitulatif des ressources partagées modifiées. 131

9.1. Moyennes géométriques normalisées obtenues pour Embench 141
9.2. Moyennes géométriques normalisées obtenues pour Embench (version SMT) 142
9.3. Nombre de cycles par test de Timesecbench. 142
9.4. Période et fréquence d'horloge pour Aubrac. 143
9.5. Période et fréquence d'horloge pour Salers. 143
9.6. Ressources du FPGA utilisées pour Aubrac. 144
9.7. Ressources du FPGA utilisées pour Salers. 144

Liste des codes

2.1. Exemple d'un calcul de PGCD en C. 11
2.2. Exemple d'un calcul de PGCD en assembleur RISC-V. 11
2.3. Exemple d'une boucle en assembleur RISC-V. 16
2.4. Exemple d'addition de trois valeurs de la pile mémoire en assembleur RISC-V. 16

3.1. Exemple d'une trace de cache. 30
3.2. Exemple d'une attaque Spectre-PHT. 39
3.3. Exemple d'une attaque Meltdown-US. 41

4.1. Exemple d'implémentation de retpoline en x86. 55
4.2. Exemple d'implémentation d'un masque sur un index. 56

6.1. Chargement et copie d’une configuration champ par champ. 89

9.1. Isolation temporelle entre un cheval de Troie et un espion. 137

Liste des acronymes

AES Advanced Encryption Standard.
ALU unité arithmétique et logique.

BHT branch history table.
BRU unité de branchement.
BTB branch target buffer.

CISC Complex Instruction Set Computer.
CSR registre de contrôle et de statut.

DES Data Encryption Standard.

ECDSA Elliptic Curve Digital Signature Algorithm.

FF registre.
FIFO first-in first-out.
FPGA circuit logique programmable.
FPU unité de traitement des nombres flottants.
FSM machine à états finis.

GPR registre général.

IM information mutuelle.
Intel SGX Intel Software Guard Extensions.
Intel TSX Intel Transactional Synchronization Extensions.
ISA architecture de jeu d'instructions.

LLC last-level cache.
LRU least-recently used.
LUT lookup table.

MMU memory management unit.
MULDIV unité de multiplication et division.

NIST National Institute of Standards and Technology.
NLP prédicteur de la prochaine ligne.

PGCD plus grand commun diviseur.
PHT pattern history table.
pLRU pseudo least-recently used.
PTW page table walker.

RISC Reduced Instruction Set Computer.
RNG générateur de nombres aléatoires.
RSB return stack buffer.

SMT simultaneous multithreading.

TCB trusted code base.
TEE environnement d'exécution de confiance.
TLB translation lookaside buffer.

VLIW Very Long Instruction Word.
VM machine virtuelle.

Introduction 1.

1.1. Contexte

1.1 Contexte 1
1.2 Sécurité dans les systèmes . 2
1.3 Objectifs et contributions . 3
1.4 Organisation du manuscrit 4

Depuis leur développement dans la première moitié du XX^{ieme} siècle, les ordinateurs ont pris une place prépondérante. Aussi bien dans notre vie personnelle que professionnelle, nous sommes entourés de ces objets électroniques nous aidant dans notre vie quotidienne. Cela concerne nos ordinateurs personnels, mais aussi nos téléphones, nos objets connectés ou même les multiples serveurs distants que nous utilisons en naviguant sur internet.

Au cœur de ces systèmes, nous retrouvons un même composant matériel : le processeur. Responsable de l'exécution des calculs, il dirige le fonctionnement des ordinateurs. Ainsi, de par son rôle stratégique, le processeur est le sujet de nombreux travaux d'optimisation. Sa capacité à exécuter rapidement un grand nombre d'opérations influe directement sur les performances du système complet. Depuis de nombreuses années, une question est donc au centre des recherches : comment améliorer les performances des processeurs ? Les réponses successives ont mené aux processeurs modernes.

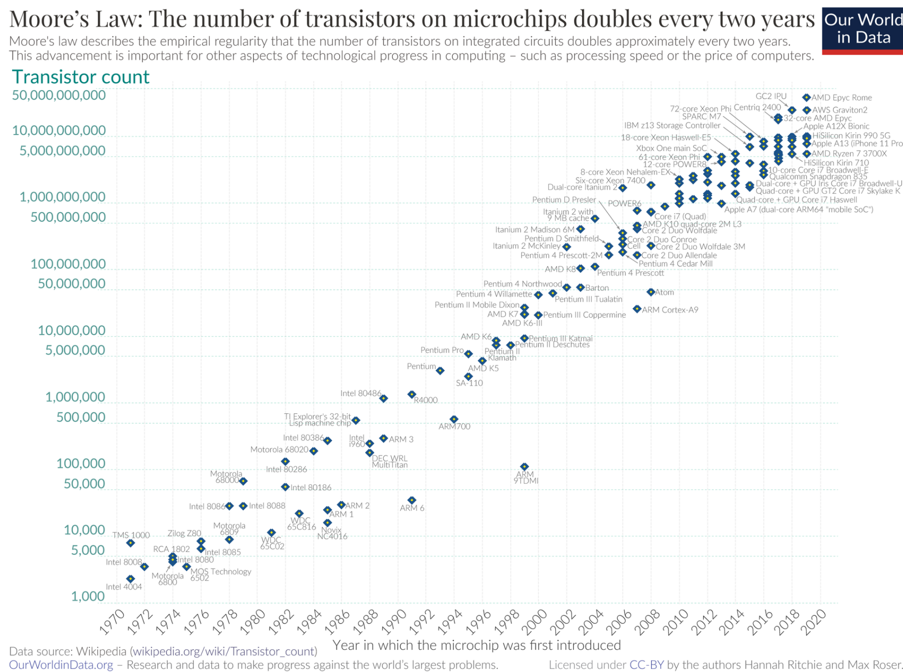


Figure 1.1. – Nombre de transistors par circuit électronique. Cette figure est extraite de la page Wikipédia dédiée à la loi de Moore [1].

Les systèmes n'ont pour cela pas cessé de se complexifier. L'évolution du nombre de transistors utilisés l'illustre parfaitement. Un transistor est un composant électronique élémentaire pour la conception des circuits. La loi de Moore prédisait un doublement du nombre de transistors tous les dix-huit mois. L'évolution réelle est présentée sur la Figure 1.1. Au début des années 1970, pour la fabrication de l'Intel 4004, 2300 transistors étaient nécessaires. Ce nombre n'a cessé d'augmenter au fil des années

pour finalement dépasser le milliard moins de quarante années plus tard.

Avec la diversification d'utilisations des processeurs, leurs contraintes ont également évolué. Le coût de fabrication est bien souvent un élément clé pour la viabilité d'un projet. L'utilisation d'un processeur doit pouvoir répondre à des besoins de rentabilité. Mais la consommation est aussi devenue critique. En parallèle de l'augmentation du nombre de transistors, la miniaturisation des systèmes a mené à des problèmes de dissipation thermique. La densité de mécanismes fonctionnant simultanément est si importante que l'ensemble peut produire une chaleur importante. Les contraintes de consommation sont également devenues essentielles avec l'avènement des objets connectés. Ces systèmes aux ressources limitées doivent être capables de fonctionner sans une source d'alimentation permanente. Enfin, une autre contrainte s'impose avec le temps à cause des menaces pesant sur ces systèmes : la sécurité.

1.2. Sécurité dans les systèmes

Longtemps, la protection des systèmes est restée la préoccupation de marchés de niche, comme les secteurs de la défense ou des cartes bancaires. Il est essentiel que les équipements électroniques d'une centrale nucléaire soient robustes, que des équipements médicaux fonctionnent toujours comme attendu, que nos données bancaires soient protégées ... Dans ces milieux, des problématiques comme la protection des données, leur intégrité ou l'extrême fiabilité des systèmes sont donc essentielles.

Le déploiement du numérique dans notre quotidien a généralisé ces menaces. Chaque téléphone ou ordinateur est susceptible de renfermer des données sensibles de ses utilisateurs. Cela concerne aussi bien les informations personnelles pour des opérations de paiements en ligne ou des papiers administratifs, que des informations professionnelles sur des projets confidentiels. Ainsi, ils représentent une porte d'entrée intéressante pour des personnes aux intentions malveillantes. Finalement, on ne compte plus chaque année le nombre de cyberattaques visant le vol de données d'utilisateurs.

En réponse à ces menaces, les logiciels ont largement évolué afin d'offrir de nouvelles protections. Cette approche a l'avantage d'être facilement applicable à tous les systèmes. En cas de faille détectée dans un programme, le fournisseur peut développer une mise à jour et l'envoyer aux utilisateurs. C'est ce que font la plupart des possesseurs de téléphones ou d'ordinateurs, consciemment ou non. Mais pour fonctionner, les logiciels ont besoin de composants matériels, et plus particulièrement des processeurs. Bien que connues, les failles sur ces derniers ont longtemps été négligées. Pire encore, dues aux nombreux développements notamment pour améliorer les performances, ces faiblesses se sont répandues avec le temps. Ce constat s'est vérifié en 2018 avec la découverte de Spectre et Meltdown, des failles majeures concernant la grande majorité des processeurs de cette dernière décennie. Finalement, le matériel est devenu le point faible des systèmes. Parce qu'ils ont accès aux différentes données du système, les processeurs sont des cibles de choix. Ils représentent une base friable sur laquelle on tente désespérément de bâtir un édifice robuste.

À la différence du logiciel, les composants matériels sont fixés à la fabrication. Il n'est pas (ou peu) possible de les mettre à jour. Les

corrections sont généralement appliquées entre deux générations de processeurs différentes. Cependant, les processeurs modernes sont le résultat de plusieurs décennies d'optimisations, principalement pour les performances. Ils rassemblent de nombreux mécanismes qui, parce que la sécurité a jusqu'ici été ignorée, représentent tous de potentielles sources d'informations pour un attaquant. Cette complexité et l'ampleur du problème compromettent la possibilité de simplement ajuster les systèmes existants. Leur conception doit donc être repensée afin de les rendre entièrement robustes dès les fondements. Cela passe par la fabrication de composants matériels, et notamment de processeurs, adaptés. Ces derniers doivent par exemple être capables de protéger les données sensibles qu'ils manipulent.

De plus, le contexte scientifique autour de la conception de nouveaux processeurs est propice à ce genre d'initiatives. Historiquement, les industriels sont des acteurs majeurs dans ce domaine et les principaux fabricants. Pour des raisons concurrentielles, l'organisation de leurs processeurs et leur fonctionnement interne sont des informations privées. D'un point de vue extérieur, chaque processeur peut être vu comme une boîte noire. Depuis plusieurs années maintenant, une nouvelle architecture de jeu d'instructions (*instruction set architecture* ou ISA) appelée RISC-V reçoit un engouement particulier. Étant libre et ouverte, de nombreux projets sont développés dans son sillon relançant les possibilités de travaux et d'échanges.

1.3. Objectifs et contributions

Dans cette thèse, nous nous intéressons à la problématique de conception de processeurs avec des contraintes de sécurité. Plus particulièrement, notre objectif est de démontrer que l'ISA a pour cela un rôle essentiel à jouer. L'ISA peut être vue comme une spécification du langage commun ou l'interface entre le matériel et le logiciel. Elle est le premier élément qui définit l'ensemble du système. Nous verrons que reprendre le fonctionnement d'un processeur depuis ses fondements passe nécessairement par une adaptation de l'ISA.

Les contributions de cette thèse sont les suivantes :

- Un état de l'art sur les attaques logicielles exploitant la microarchitecture et les solutions existantes est proposé. Les différents principes fondamentaux utilisés en sont notamment déduits pour permettre une approche globale.
- Le rôle de l'ISA pour la sécurité est défini.À partir d'une étude des différents niveaux du système, de leurs rôles actuels et de leurs limites, nous définissons un cahier des charges pour une nouvelle ISA.
- Une nouvelle ISA respectant ce cahier des charges est détaillée. Basée sur l'ISA RISC-V existante, cette dernière est étoffée afin de fournir les mécanismes nécessaires. Notamment, une nouvelle notion de domaine de sécurité appelée *dome* est ajoutée. Celle-ci permet au logiciel d'indiquer des contraintes de sécurité au matériel.
- Des principes génériques de conception pour la sécurité sont définis à partir d'une étude des attaques connues et des solutions proposées. Les mécanismes permettant leur mise en place dans les processeurs sont décrits. Ils sont directement compatibles avec la

nouvelle ISA.
- Deux processeurs partant de cette nouvelle ISA et respectant les principes de conception sont implémentés. Les différents mécanismes remis en question pour la sécurité sont adaptés à ces nouvelles garanties de sécurité. L'impact de leur application sur la microarchitecture, l'organisation interne d'un processeur, peut ainsi être étudié.
- Un ensemble de tests d'évaluation est proposé. Appelé *Timesecbench*, il est utilisé pour éprouver et évaluer nos deux implémentations.

1.4. Organisation du manuscrit

Ce manuscrit se décompose en trois parties. Dans la première, nous étudions le fonctionnement des processeurs et certaines des attaques connues les ciblant. Après une revue des solutions existantes et de leurs limites, l'objectif est de détailler notre stratégie et les contraintes à respecter. Dans le premier chapitre de cette partie (chapitre 2), nous étudions le fonctionnement des processeurs. Les processeurs modernes se composent de nombreux mécanismes visant à optimiser les performances et nous verrons comment chacun d'eux joue un rôle sur l'exécution des opérations. Dans le deuxième chapitre (chapitre 3), un état de l'art sur les attaques logicielles exploitant la microarchitecture est détaillé. Ces attaques cherchent à utiliser le fonctionnement des processeurs pour retrouver des informations normalement secrètes. Nous verrons qu'elles ne s'appuient finalement que sur quelques principes microarchitecturaux. Enfin, dans le dernier chapitre de cette partie (chapitre 4), l'objectif est de définir une stratégie globale contre ces attaques. À partir d'un état de l'art des solutions déjà existantes, nous tenterons de généraliser une approche. Nous verrons à ce moment-là le rôle de chacun des niveaux du système : le matériel, le logiciel et l'ISA.

La deuxième partie est consacrée à l'étude du rôle de l'ISA et à sa modification. L'objectif du premier chapitre (chapitre 5) est de définir un cahier des charges de ce que doit permettre l'ISA. Son rôle d'interface entre le matériel et le logiciel est essentiel pour envisager une approche globale. Nous nous appuierons pour cela sur les ISA existantes. Dans le second chapitre (chapitre 6), nous introduisons notre proposition de nouvelle ISA. Les différentes modifications qu'elle implique sont détaillées. L'impact sur le logiciel ainsi que la possibilité de réutilisation pour d'autres garanties de sécurité sont également abordés.

Enfin, la troisième partie s'intéresse à la modification du matériel. Dans le premier chapitre (chapitre 7), nous définissons des principes de conception génériques. Leur objectif est de guider un concepteur matériel cherchant à implémenter des garanties de sécurité. Des mécanismes permettant l'application de ces principes sont également présentés. Dans le chapitre suivant (chapitre 8), deux microarchitectures de processeur sont présentées. Les modifications pour l'application des principes sont détaillées. Ces implémentations sont finalement évaluées dans le chapitre 9. Un nouvel ensemble de tests pour la sécurité est présenté. Il est appliqué sur nos processeurs afin d'évaluer les garanties de sécurité offertes. Une évaluation de l'impact sur les performances ainsi que le coût est aussi proposée.

Une conclusion générale et de futures perspectives sont présentées dans le dernier chapitre (chapitre 10).

Comprendre les microarchitectures modernes

Composants microarchitecturaux 2.

Dans ce chapitre, nous étudions les différents éléments qui composent les processeurs. L'objectif est de rappeler les principes génériques ainsi que les principaux mécanismes matériels utilisés pour optimiser l'exécution des programmes. Ils seront nécessaires pour la suite de ce manuscrit. Ces sujets sont évoqués bien plus en détail dans certains ouvrages de référence de Hennessy et Patterson [2, 3].

[2]: Hennessy et al. (2011), *Computer Architecture*
[3]: Patterson et al. (2017), *Computer Organization and Design RISC-V Edition*

2.1 Principes généraux 7
Circuits logiques et mémoires 7
Organisation du processeur 9
Caractéristiques des jeux d'instructions 11
2.2 L'enjeu des performances . 12
Limites du modèle de base 13
Utilisation d'un pipeline . . 13
Mémoires caches 15
2.3 Spéculation 19
Pipeline et exécution dans le désordre 19
Prédiction dynamique de branchement 20
Prefetcher 22
2.4 Parallélisme des exécutions 22
2.5 Mémoire virtuelle 23
2.6 Mécanismes de sécurité . . 24
Privilèges 24
Exceptions et interruptions 25
2.7 Conclusion 26

2.1. Principes généraux

Les processeurs, que nous allons étudier tout au long de ce manuscrit, sont des composants matériels. Leur fonctionnement est basé sur des propriétés physiques. Après un rappel sur les circuits logiques et les mémoires, nous étudierons les principes de bases pour la conception de processeurs.

Circuits logiques et mémoires

Fonctions logiques On appelle un circuit logique une catégorie spécifique de circuits électroniques. Ils sont basés sur des blocs élémentaires appelés portes logiques. Ils représentent des fonctions appliquées à des valeurs binaires. On peut par exemple citer les trois portes basiques que sont NON, ET, et OU. Les tables de vérité de deux portes sont présentées sur les Tables 2.1 et 2.2. Elles illustrent le fonctionnement très simple de ces éléments électroniques fondamentaux.

Dans les circuits électroniques modernes, ces fonctions logiques sont implémentées à l'aide de transistors. L'état des signaux électriques représente alors les valeurs binaires. En connectant des portes logiques, il est ainsi possible de créer des fonctions plus complexes. On distingue généralement deux types de circuits logiques :

1. les circuits combinatoires dont le résultat ne dépend que des entrées,
2. les circuits séquentiels dont le résultat dépend des entrées, mais également des valeurs internes enregistrées. Ces dernières reflètent l'état du système suite à des opérations antérieures.

En plus des portes logiques de base réalisant des fonctions booléennes, les circuits séquentiels s'appuient sur un autre bloc élémentaire appelé *registre* ou *bascule*. Celui-ci a pour but de mémoriser une entrée et de la copier sur sa sortie tant qu'une nouvelle valeur n'est pas mémorisée. L'action de mémorisation, c'est-à-dire de modification de la valeur stockée, est déclenchée par un signal particulier appelé *signal d'horloge*. À l'échelle d'un circuit, un même signal d'horloge permet généralement d'activer un grand nombre de registres simultanément.

Table 2.1. – Table de vérité d'un NON. IN est l'entrée et OUT la sortie.

IN	OUT
0	1
1	0

Table 2.2. – Table de vérité d'un OU. IN0 et IN1 sont les entrées et OUT la sortie.

IN1	IN0	OUT
0	0	0
0	1	1
1	0	1
1	1	1

Signal d'horloge Le signal d'horloge est un signal électrique cyclique : sa forme se répète au cours du temps. Généralement, il prend la forme d'un signal carré décrite sur la Figure 2.1. La fréquence d'une horloge (en Hz) correspond au nombre de fois où un motif se répète chaque seconde. Ainsi, les différentes bascules sont régulièrement activées pour

mémoriser de nouvelles données selon l'évolution des entrées et calculs en cours. Généralement, le front montant (la transition de l'état bas vers l'état haut de l'horloge) est utilisé pour déclencher les registres. Selon la fréquence à laquelle se répète ce motif, les données se propagent plus ou moins rapidement dans le circuit, de registre en registre. Une horloge plus rapide permet donc d'accélérer la propagation des calculs.

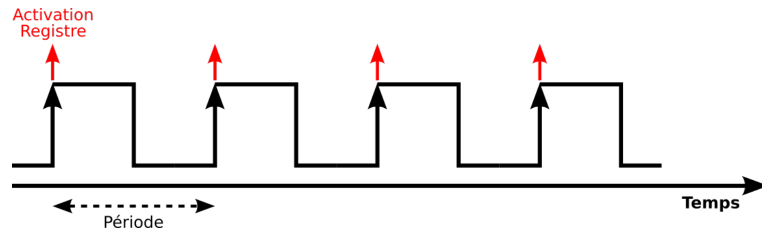


Figure 2.1. – Représentation d'un signal d'horloge. L'horloge est un signal qui se répète indéfiniment. Un évènement sur cette horloge vient cycliquement enclencher la mémorisation dans les registres. Ici, cet évènement est un front montant : le passage de l'état bas vers l'état haut du signal. La période correspond à la durée d'un motif, soit la forme répétée cycliquement.

La fréquence maximale de l'horloge dépend du circuit lui-même. Pour se propager, un signal électrique met un certain temps. Pour qu'une bascule puisse correctement mémoriser son entrée, il est nécessaire que les différents signaux se soient correctement propagés. Ainsi, on appelle chemin critique le chemin du circuit où le signal met le plus de temps à se propager d'une entrée/registre jusqu'à une sortie/registre. Autrement dit, il correspond au temps de propagation le plus long possible. La période d'une horloge, qui est la durée d'un motif, ne doit pas être supérieure à ce temps. Finalement, à partir de la valeur du chemin critique et du temps de mémorisation, on peut calculer :

Chemin critique T_{min} + temps de mémorisation $T_{mem} < P$ période de l'horloge et la fréquence $f = \frac{1}{P}$.

La Figure 2.2 illustre le chemin critique d'un circuit simple. Chaque porte logique ajoutée sur un chemin demande un délai de propagation supplémentaire. Pour réduire un chemin critique, l'idée de base est de diminuer le nombre de portes logiques qu'il traverse. La technologie de fabrication des transistors influe elle aussi sur ce temps de propagation. Les circuits devant être extrêmement rapides doivent donc jouer sur ces deux facteurs pour augmenter la fréquence d'horloge.

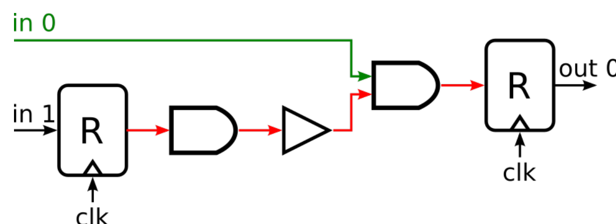


Figure 2.2. – Représentation d'un circuit logique et son chemin critique. Le circuit contient ici deux entrées *in*, deux registres *R*, une sortie *out*, une horloge *clk* et des portes logiques. Le chemin critique est ici représenté en rouge : il correspond au chemin reliant les deux bascules. C'est donc son temps de propagation qui détermine la fréquence d'horloge de la totalité du circuit.

Contraintes technologiques et mémoires Les circuits logiques décrits précédemment sont conçus en utilisant des transistors. Cette technologie est particulièrement efficace pour la conception de circuits rapides. Cependant, elle est également très coûteuse. De par le volume constamment croissant d'informations à stocker, envisager de toutes les mémoriser dans des bascules n'est pas réaliste. De plus, dès que l'alimentation d'un circuit est coupée, ces registres perdent les données qu'ils contiennent.

D'autres technologies sont donc utilisées pour constituer des mémoires, des composants capables de conserver un grand volume de

données grâce à leur faible coût. La contrepartie évidemment est au niveau des performances, où les temps d'accès pour récupérer une donnée sont plus ou moins longs (plusieurs cycles d'horloge). En comparaison, la valeur d'un registre est accessible instantanément.

On considère généralement trois types de mémoires :

1. Les registres pour contenir de petits volumes de données, mais accessibles rapidement (une latence de l'ordre de la nanoseconde). Ils perdent leur contenu à chaque coupure de l'alimentation.
2. La mémoire dynamique permet de contenir de plus gros volumes de données à un prix raisonnable. Elle est plus lente qu'un registre (une latence de plusieurs dizaines de nanosecondes). Elle perd également son contenu sans alimentation : elles sont appelées *mémoires volatiles*.
3. La mémoire de masse permet de contenir de très gros volumes de données. Le contenu est lui conservé même sans alimentation : elles sont appelées *mémoires non volatiles*. Elle est cependant bien plus lente (une latence de plusieurs millisecondes).

Organisation du processeur

On définit un ordinateur comme un système capable de traiter des opérations de manière séquentielle. C'est un système composé de trois parties : la mémoire, le processeur et les entrées-sorties. Le système complet est représenté sur la Figure 2.3.

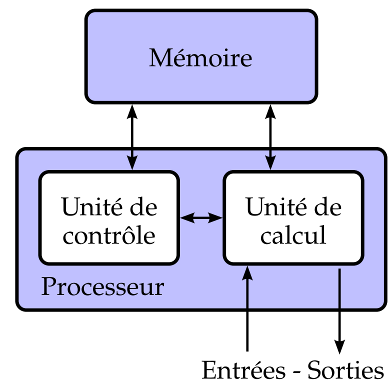


FIGURE 2.3. – Modèle d'architecture de von Neumann. Une même mémoire contient les données et les instructions.

La *mémoire* contient les différentes informations nécessaires pour les opérations. Il y a les *données*, qui correspondent aux valeurs traitées, et les *instructions*, qui indiquent les opérations à effectuer. Toutes sont représentées et manipulées par le système dans le format binaire. Dans le cas des instructions, l'*architecture de jeu d'instructions (instruction set architecture ou ISA)* est responsable d'associer théoriquement une valeur binaire à chaque opération réalisable. Elle peut être vue comme une spécification du système : "la valeur binaire B correspond à l'opération matérielle X". Un exemple est présenté sur le Table 2.3.

TABLE 2.3. – Représentation d'une instruction à différents niveaux du système. La valeur en binaire est celle vue et interprétée par le matériel. La valeur en assembleur est l'équivalent textuel du binaire. C'est la représentation d'une instruction perçue par le programmeur. Enfin, l'opération décrit le calcul que doit réaliser le processeur quand il reçoit cette instruction.

Binaire	Assembleur	Opération
0000 0000 00010100 00000101 0001 0011	**addi** x10, x8, 1	x10 = x8 + 1

Les *entrées-sorties* représentent l'interface avec le monde extérieur. Elles permettent de faire rentrer des données dans le système durant l'exécution (*e.g.* les touches utilisées sur un clavier) ou d'en faire sortir (*e.g.* l'affichage sur un écran).

Enfin, le *processeur* est la partie responsable de l'exécution des opérations. Comme illustré sur la Figure 2.3, il se décompose lui-même en deux parties :

1. Une unité de contrôle responsable de la récupération des instructions en mémoire une par une. Selon les instructions reçues, elle pilote l'unité de calcul selon les opérations à réaliser.
2. Une unité de calcul responsable d'effectuer les opérations correspondantes aux instructions récupérées. Elle accède également à la mémoire pour récupérer ou écrire les données nécessaires.

Le processeur est donc conçu pour exécuter un jeu d'instructions précis. Plus généralement, on dit d'un processeur qu'il est une implémentation de l'ISA. Nous verrons par la suite que pour une ISA donnée, il est possible de concevoir des processeurs bien différents, mais qui, du point de vue du logiciel, ne font qu'exécuter les mêmes opérations.

Le processeur représente la partie centrale d'un ordinateur. Son fonctionnement détermine par exemple les performances du système, c'est-à-dire la capacité à réaliser rapidement toutes les opérations. Les processeurs sont basés sur des circuits logiques séquentiels. Ainsi, la fréquence de l'horloge est un facteur déterminant sur les performances du processeur, sa capacité à exécuter rapidement le plus d'opérations. L'organisation interne du processeur a aussi une influence sur son fonctionnement et donc ses performances. Deux grands principes d'architectures de processeurs datant des premiers ordinateurs ont toujours des principes appliqués aujourd'hui.

Architecture de von Neumann Cette architecture décrit un système où la mémoire contient aussi bien les instructions que les données. Ainsi, le processeur n'a besoin que d'un seul port d'accès à la mémoire comme présenté sur la Figure 2.3. Cette spécificité impacte directement l'organisation du processeur. Si le circuit est ainsi globalement simplifié, le processeur doit lui nécessairement exécuter chaque opération en deux étapes :

1. utiliser l'accès pour récupérer l'instruction,
2. réaliser ensuite un accès pour récupérer les données.

Stocker les instructions et les données dans une même mémoire est également un atout pour la programmation du système. Cela permet le développement de codes qui se modifient eux-mêmes au cours de l'exécution. Il est ainsi possible de développer du code plus flexible et qui évolue selon le résultat d'opérations antérieures.

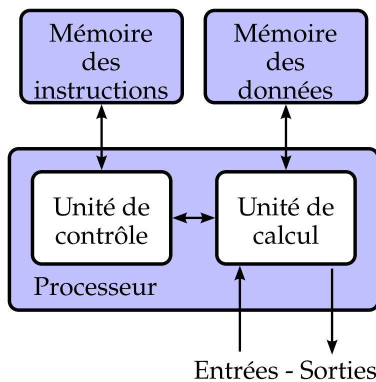


Figure 2.4. – Modèle d'architecture d'Harvard. Deux mémoires distinctes contiennent les données et les instructions.

Architecture d'Harvard L'architecture d'Harvard adopte une organisation différente en distinguant deux mémoires : une pour les instructions et une autre pour les données. Comme illustrés sur la Figure 2.4, deux ports d'accès sont donc nécessaires au processeur. Le circuit est globalement complexifié. Cependant, cela représente un avantage du point de vue des performances. Il devient possible d'effectuer deux accès à la mémoire simultanément. Le temps total pour exercer une opération est donc réduit.

Nous verrons par la suite avec la Figure 2.13 que les processeurs modernes ne sont finalement qu'une version hybride de ces deux architectures.

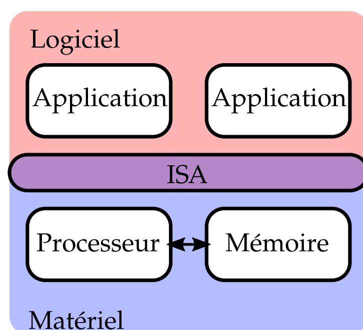


Figure 2.5. – Représentation d'un système informatique.

Abstractions d'un système Un ordinateur (ou tout système informatique) peut-être vu selon plusieurs niveaux d'abstraction. Une représentation globale est faite sur la Figure 2.5. L'*ISA* fixe les instructions utilisables sur le système. Ces instructions représentent un moyen d'indiquer aux systèmes les opérations que l'on souhaite réaliser.

Le *matériel* est lui responsable de l'exécution des opérations et du stockage des informations. Il est pour cela composé de la mémoire, du processeur et autres entrées-sorties. C'est la partie fixe du système : il est déterminé à la fabrication. Dans la littérature, un processeur peut avoir plusieurs appellations comme *microprocesseur* ou *cœur*. Lors de

la conception d'un processeur, différents choix d'implémentations sont possibles : l'ISA ne fixe que la fonctionnalité d'une opération, pas la manière dont elle est réalisée. Lorsque l'on parle de *microarchitecture*, on parle de l'organisation interne d'un processeur permettant la réalisation des opérations.

Le *logiciel* correspond lui à une suite d'instructions pour réaliser les applications voulues. Cette suite est stockée en mémoire et est parcourue instruction par instruction par le processeur durant l'exécution. C'est la partie programmable du système : en modifiant les instructions et données stockées en mémoire, il est possible de changer les applications réalisées. Concevoir des applications uniquement en assembleur (la transcription littérale des instructions en binaire) serait trop complexe et peu pratique. Ainsi, un développeur logiciel a généralement plusieurs niveaux d'abstractions. Il existe notamment des langages qui, à l'aide d'un *compilateur*, peuvent être traduits en binaire. Les Codes 2.1 et 2.2 sont par exemple équivalents, mais en langage C et en assembleur. On constate que le premier simplifie la lecture du code.

Code 2.1: **Exemple d'un calcul de PGCD en C.** Ce code réalise un calcul du plus grand commun diviseur (PGCD) entre deux entiers selon l'algorithme d'Euclide.

```
1  int pgcd(int a, int b) {
2    if (a == b) {
3      return a;
4    } else if (a > b) {
5      return pgcd(a-b, b);
6    } else {
7      return pgcd(a, b-a);
8    }
9  }
10
```

Code 2.2: **Exemple d'un calcul de PGCD en assembleur RISC-V.** Ce code réalise un calcul du PGCD entre deux entiers selon l'algorithme d'Euclide.

```
1  start:
2    addi sp, sp, -8
3    lw a4, 0(sp)
4    lw a5, 4(sp)
5
6  ## si a4 = a5
7  ## alors PGCD = a4 = a5
8  egal:
9    bne a4, a5, sup
10   sw a4, 0(sp)
11   addi sp, sp, 4
12   jalr zero, ra, 0
13
14 ## si a5 < a4
15 ## alors on fait a6 = a4 - a5
16 sup:
17   blt a5, a4, inf
18   sub a6, a5, a4
19   sw a4, 0(sp)
20   sw a6, 4(sp)
21   addi sp, sp, 8
22   j start
23
24 ## sinon
25 ## on fait a6 = a5 -a4
26 inf:
27   sub a6, a4, a5
28   sw a5, 0(sp)
29   sw a6, 4(sp)
30   addi sp, sp, 8
31   j start
32
```

Caractéristiques des jeux d'instructions

Le jeu d'instructions ou ISA influe directement sur l'implémentation du processeur. Selon les opérations définies, ce dernier devra être conçu pour les implémenter. Ainsi, il existe plusieurs types de jeu d'instructions, qui peuvent varier dans leur manière de représenter les données ou les instructions.

Représentation des données Une instruction spécifie également les données avec lesquelles ont lieu les opérations. Ces données, selon les architectures, peuvent prendre différentes formes. Voici quelques modèles :

- Le modèle à accumulateur possède un registre interne utilisé par défaut pour stocker le résultat des instructions. Des instructions de `load` et de `store` sont nécessaires pour respectivement recevoir ou envoyer des données en mémoire.
- Le modèle mémoire-mémoire n'effectue des opérations que sur des données en mémoire. Le résultat est ainsi directement stocké en mémoire. Plusieurs accès sont donc nécessaires pour chaque instruction.
- Le modèle registre-registre effectue des opérations uniquement sur des données contenues dans des registres. De même, le résultat est stocké dans un registre. Ce modèle nécessite lui aussi des instructions dédiées de `load` et de `store` pour accéder à la mémoire.

Les ISA utilisées actuellement suivent quasiment toute le modèle registre-registre. Les mémoires étant trop lentes, chaque accès ralentit le système. Le modèle registre-registre offre une flexibilité intéressante au programmeur qui dispose de plusieurs emplacements rapides d'accès pour stocker des données. Les accès mémoires pour récupérer des données ne sont pas systématiques. Pour la suite de ce manuscrit, sauf indication contraire, nous ne considèrerons donc que ce modèle là.

Familles de jeu d'instructions Les différents jeux d'instructions peuvent également varier dans leur choix de représentation des opérations. En

voici les trois principales familles :

1. Les ISA de type Reduced Instruction Set Computer (RISC) visent à associer des opérations simples à chaque instruction. Chacune ne nécessite donc que peu de cycles d'horloge pour être exécutée. Un autre avantage est la simplicité d'implémentation : un nombre réduit d'opérations élémentaires est suffisant. Le logiciel est ensuite responsable de les combiner pour réaliser des opérations plus complexes.
2. Les ISA de type Complex Instruction Set Computer (CISC) visent à associer des opérations complexes à chaque instruction. Contrairement au RISC, un plus grand nombre de cycles d'horloge est nécessaire pour exécuter chaque instruction. De plus, l'implémentation est complexifiée : chaque instruction mène à plusieurs sous-opérations au niveau du matériel. En revanche, ce jeu d'instructions est intéressant du point de vue logiciel de par la variété d'opérations qu'il offre. Une seule instruction CISC peut-être l'équivalent de plusieurs instructions RISC. Cela impacte également la taille des programmes et des mémoires : moins d'instructions sont nécessaires pour des opérations complexes.
3. Les ISA de type Very Long Instruction Word (VLIW) visent à regrouper plusieurs sous-opérations dans des instructions très longues. L'objectif de ce genre d'architecture est de permettre au processeur de réaliser un grand nombre d'opérations en même temps. Pour cela, le compilateur joue un rôle essentiel : il s'assure que les différentes parties d'une même instruction utilisent des mécanismes différents du processeur. Cela permet d'éviter des conflits internes d'accès aux ressources lors de l'exécution.

Historiquement, les caractéristiques des processeurs RISC et CISC étaient opposées. Ce contraste est maintenant obsolète et se résume à un compromis entre coût de décodage (traduction de l'instruction en opérations matérielles) et coût mémoire (place pour stocker les instructions).

Architectures d'ISA existantes Plusieurs ISA ont été développées. Chacune d'entre elles a des caractéristiques qui lui sont propres. Par la suite, trois d'entre elles seront particulièrement évoquées :

1. L'ISA propriétaire x86 d'Intel. C'est un jeu d'instructions CISC que l'on retrouve dans les processeurs d'Intel et d'AMD, notamment à destination des serveurs ou ordinateurs personnels.
2. L'ISA proriétaire ARM d'Arm. C'est un jeu d'instructions RISC que l'on retrouve majoritairement dans les processeurs d'Arm principalement à destination des systèmes embarqués et téléphones.
3. L'ISA libre et ouverte RISC-V maintenue par la fondation du même nom. C'est la cinquième version d'ISA RISC développée par l'Université de Berkeley. Au départ développée pour des travaux de recherche, elle est ces dernières années au cœur de nombreux projets industriels.

2.2. L'enjeu des performances

La conception d'un processeur est guidée par les besoins de performances. Elles correspondent à sa capacité à exécuter rapidement des opérations. Dans cette section, nous allons dans un premier temps définir

les limites d'un processeur de base du point de vue des performances. Ensuite, nous introduirons deux mécanismes essentiels pour améliorer les performances : le pipeline et les mémoires caches.

Limites du modèle de base

Un processeur basé uniquement sur une des architectures précédentes serait fonctionnel. Il serait capable d'exécuter une à une les instructions reçues, ce qui est la fonctionnalité minimale requise. Cependant, les processeurs sont généralement utilisés dans des systèmes demandant à exécuter de plus en plus d'opérations rapidement.

Latence d'exécution Un processeur a un fonctionnement séquentiel : il exécute les instructions les unes après les autres. Ses performances sont donc évaluées sur sa capacité à exécuter un maximum d'opérations sur une période de temps donnée.

Chaque instruction nécessite plusieurs actions de la part du processeur :

1. récupérer l'instruction elle-même en mémoire,
2. la traduire au niveau matériel pour savoir à quelle(s) opération(s) elle correspond,
3. récupérer les opérandes nécessaires,
4. effectuer des calculs,
5. effectuer des accès mémoires (si nécessaires),
6. écrire le résultat.

Ainsi, le processeur doit attendre d'avoir réalisé l'ensemble de ces actions pour une instruction avant de passer à la prochaine.

L'architecture d'Harvard pourrait être une alternative intéressante pour accélérer l'exécution d'une action. Les accès mémoires pour l'instruction et les données peuvent théoriquement être effectués simultanément. En pratique cependant, cela n'est pas possible de par le format des instructions. Généralement, c'est l'instruction qui indique où aller chercher la donnée en mémoire. Il faut donc nécessairement la récupérer en amont.

Latence mémoire À la latence d'exécution vient s'ajouter la latence mémoire. Les mémoires étant basées sur des technologies plus lentes que le processeur, chaque accès ralentit le processeur. Un temps d'attente est nécessaire avant de recevoir la réponse de la mémoire pour poursuivre l'exécution. Cela contribue donc à allonger le temps d'exécution nécessaire à chaque instruction, ce qui diminue le débit d'exécution.

Si les technologies pour les processeurs et les mémoires ont bien évolué au fil des années, cette disparité n'a fait que s'accentuer. La mémoire est ainsi devenue un facteur majeur de ralentissement du système. On appelle ce phénomène le *mur mémoire*.

En réponse à ces limites, les microarchitectures des processeurs se sont complexifiées. Les concepteurs les ont fait évoluer afin de répondre aux contraintes mentionnées.

Utilisation d'un pipeline

Les différentes actions nécessaires sont difficilement parallélisables pour une même instruction. Une stratégie d'accélération est de réaliser

ces différentes sous-opérations matérielles simultanément, mais pour différentes instructions. Ce fonctionnement est possible, car chacune de ces sous-opérations fait appel à des ressources matérielles différentes du processeur. On appelle *pipeline* ce type de mécanisme matériel.

Une organisation classique de pipeline est un découpage en 5 étages distincts. Chaque étage est donc responsable d'exécuter une partie de l'exécution d'une instruction différente. On retrouve généralement les étages décrits sur la Figure 2.6.

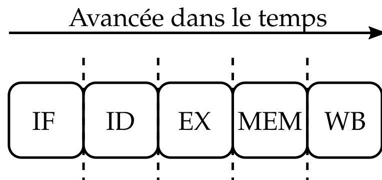


Figure 2.6. – Représentation d'un pipeline 5 étages.

Fetch La liste des instructions composant le logiciel est stockée dans une mémoire. Pour pouvoir les exécuter, le processeur doit donc aller les récupérer une par une en respectant l'ordre du programme. On appelle cette étape le chargement de l'instruction ou plus communément l'*instruction fetch (IF)*.

Decode Une fois une instruction récupérée en mémoire, il est nécessaire d'identifier à quelle(s) opération(s) elle correspond et quelle(s) donnée(s) elle utilise. On appelle cette étape le décodage ou l'*instruction decode (ID)*. Son rôle est de fixer les signaux internes du processeur qui contrôleront les opérations dans les étages suivants. C'est également à ce moment-là que l'on accède aux registres nécessaires pour lire les valeurs utilisées. Ils sont appelés registres généraux (GPR) et sont des emplacements définis par l'ISA.

Execute Une fois que le processeur a identifié quel type d'opération il doit effectuer et avec quelles données, il doit alors la réaliser. On appelle cela l'exécution ou *execute (EX)*. À la fin de cette étape, on a donc le résultat de l'opération disponible.

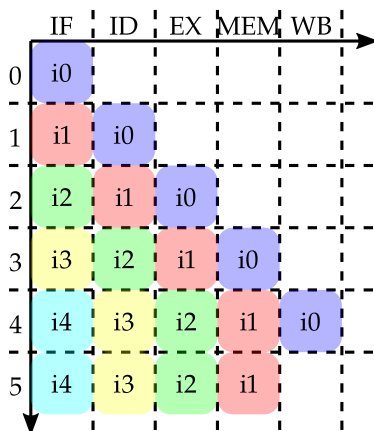


Figure 2.7. – Avancée des instructions dans un pipeline.

Memory access Tout comme les instructions, les données sont stockées en mémoire. Il est donc parfois nécessaire d'aller les récupérer pour les stocker dans des registres, ou à l'inverse d'aller écrire en mémoire pour actualiser les données qui s'y trouvent. On appelle cette étape l'accès mémoire ou *memory access (MEM)*.

Write-back Enfin, le résultat d'une opération est généralement stocké dans un GPR pour pouvoir être réutilisé ultérieurement. On appelle cela l'étape de rangement du résultat ou *write-back (WB)*.

L'exécution d'une instruction est complète uniquement après avoir traversé chaque étage. Du point de vue des performances, plusieurs effets sont à noter.

Le pipeline accentue la latence pour chaque instruction. Généralement, il faut un cycle d'horloge par étage et donc ici 5 cycles pour chaque instruction. Auparavant, un ou deux cycles pouvaient suffire.

Cependant, cet effet peut-être compensé après un certain temps d'exécution. Un exemple est illustré sur la Figure 2.7. À chaque cycle, une instruction passe d'un étage à un autre. Elle est considérée comme exécutée quand elle a traversé l'ensemble du pipeline avec succès. Le pipeline se remplit à chaque cycle pour finalement être complet au cycle 4 : chaque étage contient alors une instruction différente (le cas idéal).

Finalement, après 5 cycles d'horloge, le processeur termine l'exécution d'une nouvelle instruction à chaque cycle.

On notera qu'à cause de l'organisation du pipeline, il existe une dépendance structurelle entre les instructions qui se succèdent. Si une instruction est ralentie dans un étage (*e.g.* un accès mémoire dure plus d'un cycle), la suivante doit également attendre. Cela permet de conserver l'ordre des instructions dans le programme. On appelle ce phénomène une *bulle*. Le pipeline n'est alors plus entièrement rempli, ce qui impacte le débit d'instructions exécutées. Nous verrons par la suite que de nouveaux mécanismes visent essentiellement à réduire l'impact de ces bulles.

Le pipeline a aussi pour effet de diminuer le chemin critique. Chaque étage dispose de ses registres : le chemin est donc coupé en autant de morceaux. De ce fait, la fréquence de fonctionnement peut être augmentée. Ainsi, les processeurs avec de grandes contraintes de performances auront généralement des pipelines plus profonds. L'exécution d'une instruction est alors découpée en sous-opérations encore plus simples. Finalement, le pipeline contribue à augmenter le débit global au détriment de la latence.

Mémoires caches

Nous avons vu que la latence mémoire est un facteur limitant dans les performances des processeurs. Celui-ci est ainsi capable d'exécuter bien plus vite les instructions (une chaque cycle) que la mémoire n'est capable de lui en envoyer (plusieurs dizaines de cycles). Un mécanisme pour tenter de contrer ce problème est l'utilisation de *mémoires caches*.

Une mémoire cache est une mémoire plus rapide que la mémoire principale (généralement de la mémoire dynamique) : moins de cycles d'horloge sont nécessaires pour accéder à ses données. En contrepartie, les mémoires caches ont une contenance réduite. Augmenter la taille d'une mémoire cache augmente également sa complexité. Chaque donnée doit être accessible à tout moment : certains mécanismes matériels sont pour cela nécessaires. Un plus grand nombre de données implique donc plus d'opérations matérielles à réaliser et une latence accrue. Ainsi, ces mémoires sont généralement conçues pour contenir une quantité limitée de données, en plus du fait qu'elles nécessitent des technologies de fabrications onéreuses.

Le but des mémoires caches est de stocker localement une copie de certaines données ou instructions situées dans la mémoire principale. Ces copies deviennent plus rapidement accessibles. Cependant, seule une petite partie de la mémoire principale peut être placée dans un cache. Lors d'un accès du processeur à une donnée, on appelle un *cache hit* le fait que la donnée demandée se trouve dans le cache. Du point de vue du pipeline, cela revient à effectuer un accès mémoire accéléré : moins de cycles d'horloge sont nécessaires. À l'inverse, si la donnée demandée n'est pas dans le cache, on appelle cela un *cache miss*. Cela revient donc à effectuer un accès mémoire ralenti : plus de cycles d'horloge sont nécessaires pour aller accéder à la mémoire principale.

Finalement, le choix des données ou instructions placées dans un cache est essentiel. Il détermine directement quels accès effectués par le pipeline seront accélérés (*hit*) ou ralentis (*miss*). Pour maximiser l'efficacité, ce choix se base sur le principe de localité.

Code 2.3: Exemple d'une boucle en assembleur RISC-V.

```
1   li x1, 10
2 loop:
3   addi x2, x2, 1
4   ...
5   bne x1, x2, end_loop
6 end_loop:
7   ...
8
```

Code 2.4: Exemple d'addition de trois valeurs de la pile mémoire en assembleur RISC-V.

```
1 add-3:
2   lw x10, 0(x2)
3   lw x11, 4(x2)
4   lw x12, 8(x2)
5   add x13, x10, x11
6   add x13, x13, x12
7
```

Principe de localité Le principe de localité considère qu'un programme n'accède qu'à une petite partie de la mémoire pendant une courte période de temps. Il est basé sur deux observations communes à la plupart des programmes :

1. La *localité temporelle* qui considère que si une adresse est utilisée une fois, il y a de grandes chances qu'elle le soit à nouveau dans un futur proche. C'est par exemple le cas des instructions dans des boucles ou fonctions comme sur le Code 2.3. Les lignes 2 à 5 sont le corps d'une boucle répétée tant que $x1 \neq x2$. Chaque instruction de ces lignes est réutilisée à chaque tour de boucle.

2. La *localité spatiale* qui considère que lorsqu'une adresse est utilisée une fois, il y a de grandes chances que les adresses voisines le soient aussi dans un futur proche. C'est par exemple le cas de la plupart des instructions d'un programme, qui se suivent linéairement, ou alors d'un tableau de données manipulé case par case. Dans le Code 2.4, on voit que le programme accède à une plage mémoire (dont l'adresse est stockée dans `x2`) linéairement. De même, les instructions sont placées les unes à la suite des autres en mémoire.

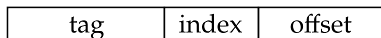


Figure 2.8. – Découpage d'une adresse dans un cache.

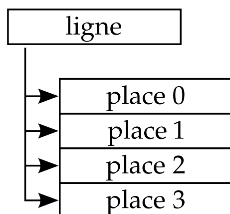


Figure 2.9. – Cache avec une organisation associative.

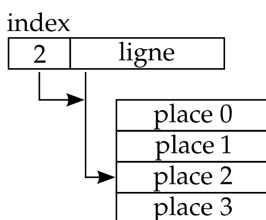


Figure 2.10. – Cache avec une organisation directe.

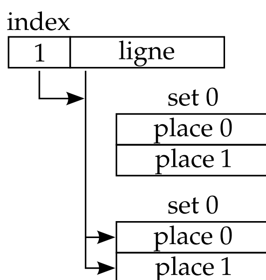


Figure 2.11. – Cache avec une organisation associative par ensemble.

Pour exploiter au mieux la localité temporelle, les caches gardent donc temporairement des données récemment utilisées. Pour la localité spatiale, la stratégie couramment employée est de décomposer la mémoire en petits blocs appelés *ligne de cache*. Chaque ligne regroupe des données placées consécutivement en mémoire. Quand le processeur veut accéder à une adresse, l'ensemble de la ligne est temporairement stockée dans le cache. Le système suppose alors qu'il y a de grandes chances que plusieurs données qui la composent soient utilisées.

Organisation d'un cache Un cache est généralement constitué de plusieurs emplacements pour stocker des lignes différentes. Quand une nouvelle ligne est utilisée, il est alors nécessaire de déterminer dans quel emplacement elle sera conservée. Il existe trois types d'organisation pour cette sélection :

1. *Associative* où une ligne peut être mise dans n'importe quel emplacement. Le cache est libre de choisir lequel choisir, généralement à partir d'une politique de remplacement détaillée plus loin. C'est l'organisation représentée sur la Figure 2.9.
2. *À correspondance directe* où une ligne ne peut être mise que dans un seul emplacement. Celui-ci est déterminé en fonction d'une partie de l'adresse de la ligne appelée *index* comme détaillé sur la Figure 2.8. C'est l'organisation représentée sur la Figure 2.10.
3. *Associative par ensemble* qui est un hybride des deux premières. Un groupe d'emplacements appelé *set* est sélectionné selon l'index, au sein duquel l'emplacement final est choisi librement par le cache. C'est l'organisation représentée sur la Figure 2.11.

L'associativité a l'avantage d'offrir une certaine flexibilité au système. Plusieurs emplacements sont susceptibles d'accueillir chaque ligne utilisée. Cependant, elle a aussi un coût supérieur à la correspondance directe, proportionnel au nombre d'emplacements à gérer. Ainsi, les structures pleinement associatives sont généralement de petites tailles. Dans le cas des mémoires caches, la stratégie associative par ensemble

est la plus utilisée, représentant un bon compromis entre flexibilité et coût.

Le cache associe également des informations à chaque ligne de cache stockée. Parmi elles, on retrouve notamment la validité de la ligne ou son *tag*. Ce dernier correspond à une autre partie de l'adresse et est essentiel pour distinguer plusieurs lignes. Ainsi, quand une donnée est demandée au cache, le tag de chaque ligne contenue est comparé avec celui de l'adresse reçue pour savoir si la ligne correspondante est présente (*hit*) ou non (*miss*).

Principe de remplacement dans un cache Une mémoire cache a une taille limitée et ne peut contenir qu'un certain nombre de lignes. Au bout d'un certain nombre d'accès, il y a donc de grandes chances que le cache soit rempli. Dans le cas d'un cache ou d'un set associatif, il est nécessaire de choisir quelle ligne doit être remplacée. Un algorithme dédié, appelé *politique de remplacement*, est alors implémenté. Voici les politiques de remplacement les plus courantes :

- *least-recently used (LRU)* pour *le moins récemment utilisé* évince la ligne utilisée le moins récemment. Elle consiste à tenir à jour un historique en associant une valeur de compteur à chaque ligne. Pour cela, à chaque accès, le compteur de la ligne correspondante est mis à 0 quand les autres sont incrémentés. Quand un remplacement est nécessaire, la ligne avec la valeur de compteur la plus haute est choisie et évincée.
- *pseudo least-recently used (pLRU)* vise à émuler le LRU mais en réduisant son coût. Dans le cas du LRU, le coût des compteurs augmente de manière importante avec le nombre de lignes (un plus grand nombre de registres est nécessaire). De plus, la complexité du système est également croissante avec un plus grand nombre de valeurs à gérer. Une variante appelée *bit-pLRU* n'associe donc qu'un bit à chaque ligne. Celui-ci est mis à 1 quand la ligne a été utilisée récemment. Seules les lignes n'ayant pas été sollicitées récemment (bit = 0) peuvent être évincées. Dans le cas où toutes les lignes ont été utilisées récemment, alors l'ensemble des bits est remis à 0.
- *first-in first-out (FIFO)* remplace la ligne stockée la plus ancienne. Elle peut être implémentée en utilisant une valeur pointant toujours vers la valeur la plus ancienne. Cette valeur se met à jour à chaque remplacement effectué.
- *Aléatoire* vise à évincer une des lignes aléatoirement. Des algorithmes dits *pseudo-aléatoires* permettent d'imiter simplement ce fonctionnement. Pour de réelles propriétés aléatoires, des algorithmes cryptographiques sont également utilisables, mais à un coût plus important.
- *etc.*

Notes : À l'exception de la politique aléatoire, toutes les autres mènent à un fonctionnement déterministe du cache, directement dépendant des accès précédents. À partir d'une séquence d'accès connue, il est possible de savoir exactement quelles lignes seront stockées à la fin ou non. Il est également important de les distinguer de l'organisation directe, qui elle est fixe et ne dépend aucunement des lignes déjà stockées.

Le choix de la politique de remplacement est essentiel, car il détermine l'état du cache et donc son efficacité. La figure Figure 2.12 représente l'évolution d'un cache en utilisant une politique LRU. Si aucune place n'est disponible, alors la donnée la plus anciennement utilisée est supprimée comme dans les Figure 2.12b et Figure 2.12d.

Hiérarchie mémoire Une mémoire cache idéale serait assez grande pour contenir en même temps toutes les adresses utilisées. Cependant, pour des raisons de coûts et de performances, cette mémoire n'est

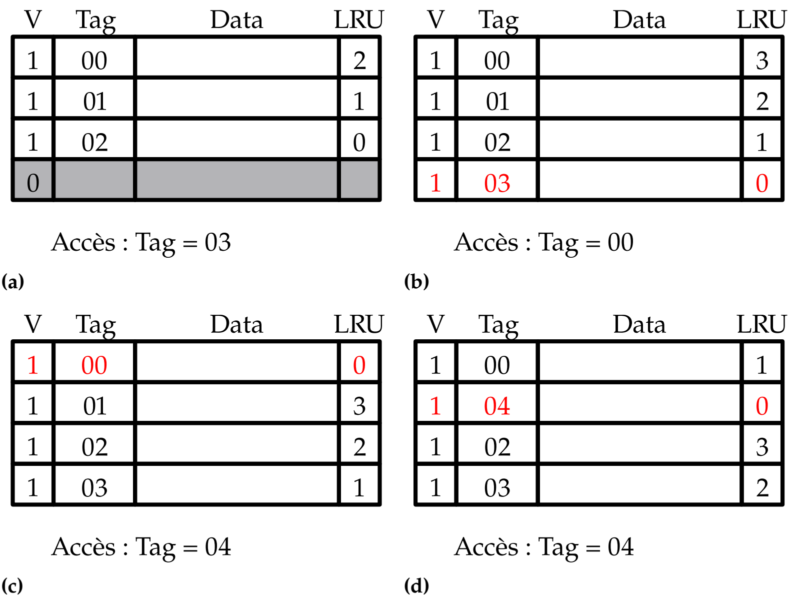


Figure 2.12. – Politique LRU utilisée dans un cache. Les informations présentes pour chaque ligne du cache sont le bit V indiquant la validité d'une ligne, le tag pour identifier la ligne stockée et la donnée de la ligne. En plus de ces informations, pour implémenter une politique LRU, un compteur est également associé à chaque ligne : une valeur faible indique une ligne utilisée récemment. Chaque compteur se met à jour à chaque accès. Dans la Figure 2.12a, il reste un emplacement libre : il est utilisé en priorité. Après cela, le cache est plein et la politique de remplacement s'applique. Dans la Figure 2.12b, c'est la première ligne qui est utilisée et son compteur est donc réinitialisé dans la Figure 2.12c. Finalement, au moment d'accéder à une nouvelle ligne non présente, c'est la ligne 2 avec le compteur le plus élevé qui est sélectionnée dans la Figure 2.12d.

pas réaliste. La vitesse d'une mémoire décroît en fonction de sa taille, notamment du fait de la complexité. Une mémoire suffisamment grande deviendrait donc obligatoirement trop lente. Pour contrer ce problème, on retrouve généralement plusieurs niveaux de mémoires qui constituent la hiérarchie mémoire, comme présenté sur la Figure 2.13. Chacun de ces niveaux a des objectifs différents et le but de l'ensemble est d'émuler cette mémoire idéale. Pour cela, quand le niveau le plus bas n'a pas la donnée demandée, il transmet la requête au niveau supérieur et ainsi de suite jusqu'à la mémoire principale.

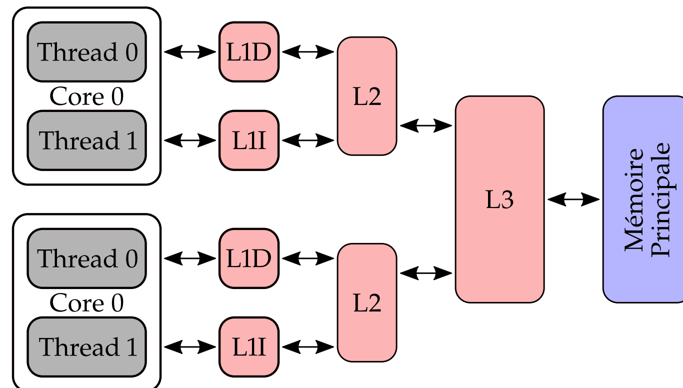


Figure 2.13. – Exemple de hiérarchie mémoire.

Le premier niveau de cache (abrégé L1) fait en sorte que chaque cache *hit* ait la latence la plus faible possible. Généralement, les mémoires de ce niveau sont donc de petites tailles pour favoriser les performances. De plus, on retrouve la plupart du temps deux caches distincts pour les accès de données et les accès d'instructions.

Le second niveau de cache (abrégé L2) a lui pour but de réduire l'impact des cache *misses*. Les mémoires de ce niveau sont plus grosses, mais également plus lentes : un accès demande plus de cycles. À l'inverse du L1, le niveau L2 (et ceux au-dessus) est unifié : les instructions et données sont stockées ensemble.

On retrouve ensuite selon les implémentations jusqu'à deux niveaux supérieurs (L3 et L4). Ces caches sont de tailles croissantes et partagés

entre plusieurs cœurs du système. Ils contribuent à diminuer l'impact d'un cache *miss* : s'ils sont plus lents que les niveaux L1 et L2, ils restent bien plus rapides que la mémoire principale.

2.3. Spéculation

La spéculation est un principe utilisé pour accélérer l'exécution des instructions. L'idée générale est de prévoir les futurs évènements (redirection du flot d'exécution, accès mémoires, *etc.*) pour les anticiper et réduire leur impact sur les performances.

Pipeline et exécution dans le désordre

Le mécanisme le plus simple usant de spéculation dans un processeur est le pipeline lui-même. Le fait de récupérer en avance les prochaines instructions en mémoire avant d'avoir complètement exécuté les précédentes est une forme d'anticipation. Cependant, le fonctionnement basique d'un pipeline impose rapidement des limites à la spéculation.

Dépendances de données Il existe au sein des programmes des dépendances entre les instructions. Par exemple, l'opérande d'une instruction peut être le résultat d'une instruction précédente. Le Code 2.2 représente le cas classique d'un calcul de PGCD, un algorithme mathématique simpliste. On y retrouve alors plusieurs cas de dépendances (*e.g.* les lignes 3 et 5, 4 et 5, 18 et 20 *etc.*). Ces dépendances peuvent créer des bulles au sein du pipeline : il faut attendre que le résultat soit disponible avant de faire avancer la prochaine instruction. Ces mêmes bulles peuvent aussi être créées dans le cas où un accès mémoire est ralenti à cause d'un cache *miss* par exemple (*e.g.* si la ligne 3 est un *miss*, la ligne 4 devra attendre également). Des mécanismes de renvoi de registre (*register forwarding*) existent pour limiter l'impact des dépendances de données. Ils servent à rendre utilisable un résultat dès qu'il est calculé. À l'échelle d'un pipeline, un résultat est donc disponible en interne avant qu'il ne soit stocké dans un GPR par le dernier étage.

Les dépendances deviennent un facteur limitant d'autant plus important dans le cas de *processeurs superscalaires*. Ces derniers cherchent à augmenter le débit d'exécution en traitant plusieurs instructions à la fois dans chaque étage. Pour cela, la plupart des mécanismes sont dupliqués (*e.g.* plusieurs décodeurs et unités d'exécutions). Ainsi, en cas de bulle, le nombre d'instructions ralenties est d'autant plus important. Or, le nombre d'instructions en cours d'exécution étant plus grand, le nombre de dépendances à gérer l'est également.

Réorganisation dynamique des instructions Un mécanisme utilisé dans les processeurs complexes pour diminuer les blocages dus aux dépendances est l'exécution dans le désordre. Son rôle est d'optimiser les cas où aucune dépendance n'est bloquante. L'objectif devient alors d'exécuter une instruction dès que toutes ses dépendances sont résolues. Cela s'oppose donc au format linéaire d'un pipeline basique, où une instruction n'est forcément exécutée qu'après toutes les précédentes. Des mécanismes matériels sont alors responsables de préserver l'ordre du programme. Ils s'assurent que si les instructions sont exécutées dans un ordre différent de celui prévu par le logiciel, l'état architectural

(celui visible par le logiciel) reste celui attendu. Même dans le cas d'un algorithme simple comme celui du Code 2.2, on retrouve plusieurs cas où cette optimisation peut être bénéfique. Par exemple, en cas de blocage de la ligne 10, les lignes 11 et 12 peuvent tout de même être exécutées.

Si les opérandes se situent généralement dans des registres, il existe également des opérations réalisant des accès mémoires. De la même manière, des dépendances se créent si des instructions accèdent à une même adresse. Il existe donc des processeurs qui étendent l'exécution dans le désordre à ces accès mémoire. Ils tentent alors de prédire les adresses utilisées ainsi que les valeurs lues/écrites.

Pour être implémentée au niveau matériel, l'exécution dans le désordre demande un grand nombre de ressources coûteuses. Des tables sont par exemple nécessaires pour mémoriser l'ordre original du programme, réaliser des sauvegardes temporaires des états en cas d'erreur *etc.* Ce mécanisme est donc généralement réservé aux processeurs avec d'importantes contraintes sur les performances. C'est par exemple le cas de la plupart des processeurs grand public (*e.g.* les processeurs Intel ou AMD) fabriqués depuis de nombreuses années.

Prédiction dynamique de branchement

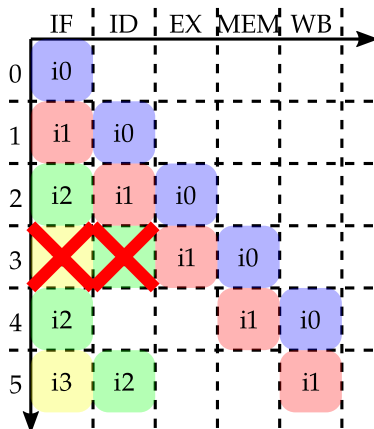


Figure 2.14. – Impact d'une mauvaise prédiction de branchement dans un pipeline. Dans cet exemple, `i1` représente une instruction de branchement dont l'adresse cible ne sera connue que dans l'étage EX. Le temps que cette instruction avance jusque-là, d'autres ont été préchargées par anticipation. Ici, il s'avère que la cible n'a pas été correctement anticipée : cela est corrigé en effaçant les mauvaises instructions. Dans notre cas, cela représente 2 instructions sur 5 que le pipeline peut contenir, donc 40%.

Il existe dans toutes les ISA des instructions pour rediriger le flot d'exécution vers d'autres adresses. On appelle un saut (direct si l'adresse est connue directement en décodant l'instruction, indirect si elle est à calculer) une instruction redirigeant toujours l'exécution vers une autre adresse. On appelle un branchement conditionnel une instruction dont la redirection ne s'effectue que si une condition est vérifiée (*e.g.*comparaison de valeurs). Ces instructions cassent la linéarité du flot d'exécution, en sautant vers une nouvelle adresse. Ce genre d'évènement peut avoir un impact considérable sur l'efficacité du pipeline s'il n'est pas correctement prévu.

De base, un pipeline récupère en mémoire les instructions à des adresses consécutives. Or, cette situation n'est plus correcte en présence d'instructions de redirection. La figure Figure 2.14 reprend le fonctionnement d'un pipeline 5 étages. Les sauts/branchements conditionnels ne sont généralement exécutés que dans le troisième étage. L'adresse de la prochaine instruction (et la vérification de la condition) n'est donc connue qu'à ce moment-là. Ainsi en cas de redirection, les instructions déjà préchargées entre temps sont erronées et doivent être effacées : 40% du pipeline s'exécute donc inutilement. Ce constat est d'autant plus vrai dans des processeurs pour hautes performances. Les étages sont encore plus nombreux et le nombre d'instructions traitées simultanément est très important. En réponse, des stratégies ont été mises en place pour réduire ces pertes. Nous allons ici nous intéresser à la prédiction de branchement dynamique.

La prédiction de branchement dynamique cherche à prédire les futures redirections du flot d'exécution pour les anticiper, en se basant sur celles ayant déjà eu lieu dans le passé. Au moment de récupérer l'instruction suivante, le processeur peut savoir où la chercher. Un exemple simple est celui d'une boucle : une fois une première itération exécutée, il est possible de déterminer (ou en partie) les instructions exécutées pour les itérations suivantes. La prédiction de branchement cherche pour cela à répondre à deux questions :

1. À quelle adresse une instruction est susceptible de rediriger le flot

d'exécution ?
2. À quelle adresse de destination le flot d'exécution peut être redirigé ?

Nous distinguerons trois mécanismes matériels couramment utilisés.

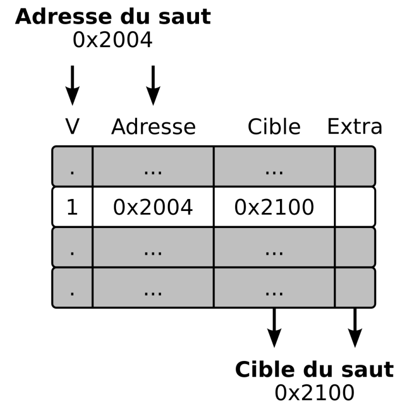


Figure 2.15. – Principe de fonctionnement du BTB. À chaque saut ou branchement rencontré, le processeur enregistre les informations dans le BTB. Plus tard, si la même adresse d'instruction est utilisée, la table donnera les informations nécessaires pour anticiper la redirection.

BTB Le branch target buffer (BTB) est une table utilisée pour stocker des informations sur chaque instruction de redirection rencontrée. Ces informations stockées peuvent être :

- l'adresse de l'instruction de redirection pour la détecter plus rapidement dans le futur,
- l'adresse de destination de la redirection pour pouvoir anticiper,
- des informations sur le type de l'instruction (Est-ce un saut ? Un branchement conditionnel ? Un retour de fonction ?) afin de faire le lien avec d'autres mécanismes.

Au cours de l'exécution, le processeur consulte le BTB pour chaque adresse d'instruction qu'il utilise. Si une adresse correspond à un saut déjà rencontré, alors la table retourne les informations correspondantes au processeur. Celui-ci peut alors anticiper la redirection associée.

Le fonctionnement du BTB est finalement similaire à un cache pour stocker des informations de branchement. Une politique de remplacement est utilisée afin de conserver les informations les plus pertinentes lorsque la table est pleine.

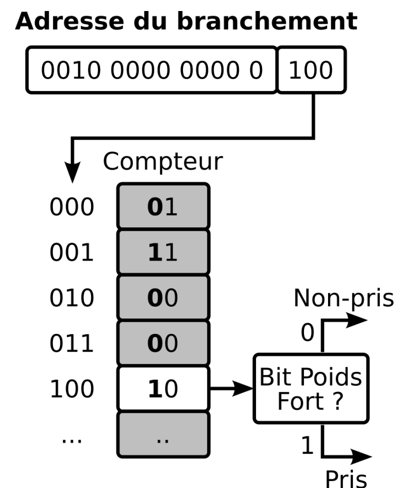


Figure 2.16. – Principe de fonctionnement du BHT.

BHT et PHT Les branch history table (BHT) et pattern history table (BHT) servent à prédire le fonctionnement des branchements conditionnels. Ces opérations visent à ne rediriger le flot d'exécution que si une condition est vérifiée. Ainsi, si la cible potentielle reste la même (le BTB peut la mémoriser), elle n'est utilisée que lorsque la condition est respectée. L'idée derrière le BHT ou le PHT est donc de constituer un historique du fonctionnement des précédents branchements rencontrés. Concrètement, elle part du constat que si la condition d'un branchement a récemment été vérifiée (ou non), il y a de fortes chances qu'elle le soit toujours.

L'implémentation la plus simple est celle du BHT qui associe un compteur à différents offsets d'adresses (*e.g.* si on a 8 compteurs, toutes les adresses `xxx010` partagent le même compteur). Elle est représentée sur la Figure 2.16. La valeur de chaque compteur donne une indication sur le respect ou non de la condition pour chaque branchement. Selon le modèle de la Figure 2.17, le compteur est mis à jour à chaque branchement conditionnel rencontré avec l'offset correspondant. Il est incrémenté en cas de branchement pris, ou décrémenté en cas de branchement non pris. Une valeur haute indique donc qu'un futur branchement à cet offset a de grandes chances d'être pris. Contrairement au BHT, le PHT est plus complexe et ne considère pas chaque branchement individuellement, mais par séquence. Il est ainsi capable de reconnaître des enchaînements de branchements et de les anticiper.

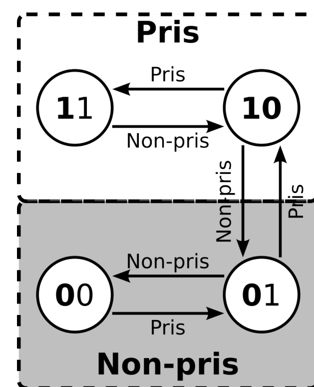


Figure 2.17. – Machine d'état du compteur du BHT. Le bit de poids fort du compteur indique si un saut a de grandes chances d'être pris ou non.

RSB Les BTB et BHT/PHT sont particulièrement efficaces pour prédire les sauts directs et les branchements conditionnels. Cependant, le cas des branchements indirects est autrement plus complexe : leur cible change en fonction des valeurs de registres. Ils sont donc susceptibles de

rediriger l'exécution vers n'importe quelle adresse. Mais il existe certains cas d'utilisation des sauts indirects qui sont plus facilement prédictibles : les retours de fonction.

Une fonction est un bout de code effectuant une tâche bien précise. Quand un programme a besoin de réaliser cette tâche, il redirige l'exécution vers l'entrée de la fonction correspondante. Une fois terminée, celle-ci retourne l'exécution vers le programme qui l'a appelée.

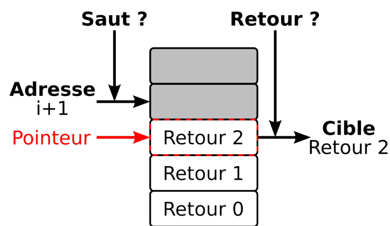


Figure 2.18. – Principe de fonctionnement du RSB.

Le mécanisme dédié à la prédiction des retours de fonctions est le return stack buffer (RSB). Chaque fois qu'un saut vers une fonction est détecté, l'adresse de retour est enregistrée. Quand un retour est détecté, alors la dernière adresse stockée dans la pile est utilisée. Ce fonctionnement est illustré sur la Figure 2.18.

C'est donc en combinant ces différents mécanismes qu'il est possible de réduire l'impact des redirections sur l'utilisation du pipeline. Ainsi, dans l'étage de fetch, il est possible de récupérer avec de grandes chances la bonne instruction à chaque cycle et ainsi limiter le nombre de bulles. D'autres prédicteurs de branchement bien plus complexes existent également et permettent d'augmenter la fiabilité des prédictions.

Prefetcher

Un autre phénomène impactant les performances d'un pipeline est l'apparition d'un cache *miss* lors d'un accès mémoire pour une donnée ou une instruction. Le système doit alors attendre que l'adresse visée soit ramenée dans le cache pour pouvoir être utilisée. Cette opération peut nécessiter l'accès à un ou plusieurs niveaux supérieurs de la hiérarchie mémoire et donc de nombreux cycles. Ainsi, certains mécanismes cherchent à anticiper les futurs accès mémoires : les prefetchers. Leur rôle est de ramener localement les données et instructions ayant de grandes chances d'être utilisées. On retrouve des prefetchers tout le long de la hiérarchie mémoire, avec des algorithmes différents pour être capable d'anticiper au mieux les futurs besoins. Comme pour la spéculation dynamique de branchement, la plupart de ces mécanismes conservent un historique des accès précédents pour essayer d'en déduire les suivants.

2.4. Parallélisme des exécutions

L'accélération d'un seul et unique programme, même en utilisant tous les mécanismes décrits jusqu'ici (ou d'autres), a tout de même des limites. Il n'est pas possible de prédire indéfiniment le comportement futur du flot d'exécution. De plus, il existe des dépendances entre les instructions qui font que le système est parfois obligé d'attendre. Ainsi, au lieu d'exclusivement miser sur le parallélisme entre les instructions, une autre voie vise à exploiter le parallélisme entre les programmes. L'idée est de considérer que plusieurs programmes ou bouts de programmes indépendants peuvent être exécutés simultanément : à l'échelle du système, l'ensemble des programmes sont donc exécutés plus rapidement. Cela est particulièrement intéressant si l'on considère que cela s'applique pour de nombreux cas d'utilisation.

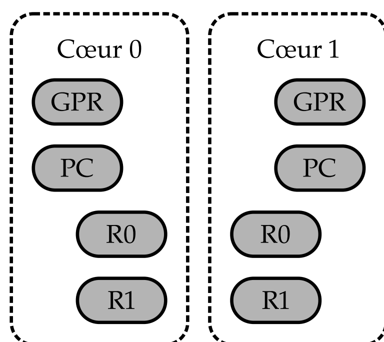


Figure 2.19. – Illustration du multi-cœur. Chaque ressource est dupliquée au sein des différents cœurs physiques afin que chacun d'eux puisse exécuter des programmes indépendamment et parallèlement.

Pour exploiter ce parallélisme, l'approche la plus simple est de dupliquer l'ensemble du processeur plusieurs fois comme illustré sur la Figure 2.19. On appelle couramment chaque instance un cœur physique. Le logiciel devient alors responsable de choisir quel programme s'exécute

sur chaque cœur. Théoriquement, cette méthode permet d'accélérer l'exécution globale N fois pour N processeurs en payant le prix fort.

Une autre approche appelée multithreading vise également à exploiter le parallélisme, mais en limitant le coût des duplications. Un exemple est présenté sur la Figure 2.20. L'idée est de créer au sein d'un même cœur physique différents emplacements appelés *threads* (ou cœurs logiques ou même harts) susceptibles d'exécuter chacun un programme différent en simultané. Pour cela, seuls les éléments essentiels sont dupliqués dans chaque thread (ceux qui définissent l'état d'un programme comme l'adresse en cours d'exécution, les registres *etc.*) et les autres sont partagés entre eux. Du point de vue logiciel, plusieurs cœurs sont donc disponibles pour exécuter des programmes, mais à un coût moindre du point de vue matériel. L'objectif est de s'approcher du cas idéal où toutes les unités d'exécutions sont utilisées pleinement. La contrepartie se situe bien sûr au niveau des ressources mises en commun : chaque thread est susceptible de les utiliser et donc de ralentir les autres s'ils en avaient également besoin. On appelle généralement simultaneous multithreading (SMT) la technique consistant à exécuter plusieurs threads en même temps.

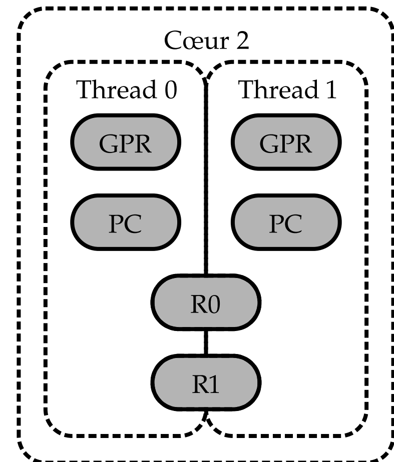


Figure 2.20. – Illustration du multithreading. Seules les ressources essentielles sont dupliquées au sein d'un même cœur. Les autres sont partagées entre les différents threads qui les utilisent ou non selon les instructions à exécuter.

2.5. Mémoire virtuelle

Un autre mécanisme couramment utilisé dans les processeurs est la mémoire virtuelle. Il permet la traduction d'adresses dites *virtuelles* (manipulées par le logiciel) vers les adresses réellement utilisées par le système au niveau matériel, les adresses dites *physiques*. Contrairement aux mécanismes précédents, la mémoire virtuelle n'est pas une optimisation pour les performances. Initialement, l'objectif de la mémoire virtuelle était de permettre d'étendre l'espace mémoire utilisable. Pour cela, certaines adresses virtuelles sont renvoyées vers la mémoire de masse (*e.g.* le disque dur). Ainsi, la taille limitée de la mémoire principale pouvait être contournée. De nos jours, elle est particulièrement utilisée pour la création de *processus* : des programmes ayant leur propre ensemble d'adresses physiques accessibles par des adresses virtuelles. Cet ensemble est lui appelé *espace d'adressage*. Définir un espace d'adressage pour chaque programme par le biais de la mémoire virtuelle permet de contrôler leurs accès mémoires. Chacun ne peut accéder qu'aux adresses physiques qui leur sont assignées par le logiciel responsable de la gestion du processeur(*e.g.* généralement le *système d'exploitation*). Il est ainsi possible :

- de choisir explicitement les plages d'adresses mémoires partagées entre des programmes,
- d'isoler certaines plages mémoires en ne les attribuant qu'à certains programmes exécutés,
- de mettre en place des protections spécifiques à certaines adresses. Lors de la traduction virtuelle vers physique, il est généralement associé à chaque adresse des bits de statut pour restreindre l'utilisation de la page ou non (seulement en écriture, en lecture *etc.*).

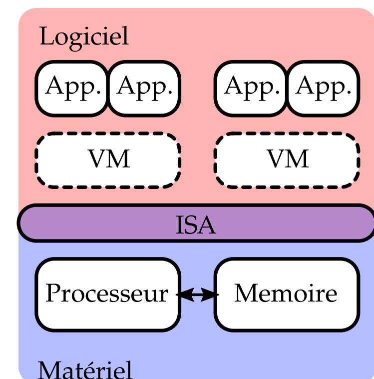


Figure 2.21. – Fonctionnement d'une machine virtuelle. Chaque machine virtuelle crée une nouvelle abstraction du matériel. Les applications exécutées au sein d'une VM ont donc l'illusion d'être placées sur un système dont les caractéristiques sont définies par la VM. Cela permet par exemple d'avoir des applications portables exécutables sur n'importe système. Seule la VM elle-même doit être adaptée.

La mémoire virtuelle est également utilisée pour la création de *machines virtuelles (VM)*. Une VM est le nom donné à l'émulation d'une machine par un programme. Une représentation d'un système avec des VM est réalisée sur la Figure 2.21. Ainsi, les programmes exécutés et contrôlés par cette VM ont l'illusion d'être placés sur une machine avec

des caractéristiques précises (certains espaces mémoire, périphériques, *etc.*). Dans ce cas, la mémoire virtuelle est donc utilisée pour mettre en place l'abstraction au niveau de la mémoire.

Pagination Associer directement à chaque adresse virtuelle possible une adresse physique serait trop coûteux. Des informations pour chaque traduction possible devraient alors être conservées. Au lieu de cela, l'association se fait généralement à l'échelle d'une plage mémoire appelée *page*. Les tailles de pages peuvent varier selon les architectures (*e.g.* 4kB pour l'ISA RISC-V). Ainsi, on associe généralement une page virtuelle à une page physique : seuls les bits de poids forts des adresses sont changés, les bits de poids faibles permettant de choisir une adresse précise de la page. Les informations de traduction sont ainsi regroupées pour plusieurs adresses différentes.

Implémentation matérielle de la mémoire virtuelle La traduction est réalisée par un composant matériel dédié appelé memory management unit (MMU). Il s'appuie sur plusieurs informations :

- l'adresse virtuelle utilisée,
- l'adresse physique correspondante,
- les bits de protection,
- l'adresse où ces informations sont stockées (dans le cas d'une organisation en plusieurs niveaux des pages).

On appelle ce groupe d'information une *entrée de page*. Quand le processeur cherche à accéder à une adresse virtuelle, un autre composant matériel appelé page table walker (PTW) (qui appartient à la MMU) récupère l'entrée nécessaire à la traduction. Son fonctionnement se résume à une machine d'états parcourant la mémoire jusqu'à l'obtention de l'information voulue. Cette récupération pouvant durer de nombreux cycles, elle ralentirait chacun des accès mémoires. Pour cela, un mécanisme semblable à un cache est utilisé : un translation lookaside buffer (TLB) (lui aussi situé dans la MMU). Le rôle du TLB est de conserver les informations nécessaires aux traductions effectuées récemment afin de pouvoir les utiliser rapidement dans le futur. Lors d'un accès mémoire, la MMU vérifie donc en premier le TLB pour voir si les informations sont disponibles avant de faire une demande au PTW si nécessaire.

2.6. Mécanismes de sécurité

Au sein d'un même système, les programmes d'utilisateurs différents doivent généralement cohabiter. Il est souvent nécessaire d'établir une hiérarchie, une organisation de ces utilisateurs. Il existe pour cela des mécanismes de sécurité définis par l'ISA et intégrés au processeur.

Privilèges

Le mécanisme de distinction des rôles le plus basique et répandu est celui des privilèges. On appelle un *privilège* un niveau d'utilisation avec des droits définis sur le système. Ces droits comprennent généralement la possibilité d'accéder à certains registres et de modifier certaines valeurs pour la configuration du processeur. On dit d'un utilisateur qu'il est privilégié à partir du moment où il dispose de plus de droits que le minimum possible.

Organisation Les différents niveaux de privilèges sont fixés pour un système donné. Ils varient en nombre (généralement entre un et quatre) selon les implémentations. Même si les droits peuvent changer/ se mêler, on nomme généralement :

1. Le niveau machine, racine, noyau ou *root* comme le niveau de privilège disposant de tous les droits. Il est donc le plus haut niveau et est présent sur tous les systèmes. Par défaut, un système sans niveaux de privilège n'a que des utilisateurs de niveau racine (vu qu'aucune distinction n'est faite).
2. Le niveau hyperviseur comme le niveau de privilège responsable de la gestion des machines virtuelles.
3. Le niveau super-utilisateur comme le niveau responsable de la bonne exécution des utilisateurs classiques. Il gère les mécanismes d'exception, la mémoire virtuelle *etc.*
4. Le niveau utilisateur qui correspond aux droits les plus limités.

Il n'est pas rare que les niveaux 1 à 3 soient fusionnés. Un utilisateur peut alors soit être privilégié avec tous les droits, soit sans privilège. Généralement, le niveau privilégié correspond alors au *système d'exploitation*, la partie logiciel responsable de la gestion du système. Les applications sont, elles, exécutées au niveau de privilège le plus bas.

Implémentation Au niveau du matériel, un niveau de privilège indique le droit d'accéder à certains registres spéciaux. Appelés registres de contrôle et de statut (CSR), ils servent notamment à la configuration du système. Par exemple, ce sont certains de ces registres qui sont utilisés pour paramétrer le mécanisme de mémoire virtuelle ou pour la gestion des exceptions (que nous verrons plus loin). D'autres permettent aussi de définir l'état actuel du système : le privilège en cours d'exécution est indiqué dans l'un d'eux.

Finalement, le matériel est responsable de s'assurer que les CSR ne sont utilisés qu'avec le niveau de privilège requis. Pour cela, il vérifie pour chaque accès à ces registres que le privilège en cours d'exécution a les droits nécessaires. Un privilège peut notamment avoir le droit de lire un registre, mais pas de le modifier. Si un privilège n'a pas les droits nécessaires, le jeu d'instructions spécifie des mécanismes pour effectuer des demandes à un niveau "plus privilégié". Concrètement, cela correspond à lancer l'exécution du nouveau privilège en utilisant le mécanisme dédié (*e.g.* une instruction). Celui-ci vérifiera alors que la demande du niveau inférieur est légitime et si c'est le cas, il effectuera l'opération lui-même avant de rendre la main sur l'exécution.

Exceptions et interruptions

Lors de l'exécution d'un programme, des erreurs ou évènements inattendus peuvent apparaître. Cela peut être le cas si une instruction inconnue est détectée. Pour ces situations, les processeurs disposent généralement d'un mécanisme de levée d'*exception*. Les informations propres au problème détecté sont mémorisées en plus de l'adresse de l'instruction et le flot d'exécution est ensuite redirigé vers une nouvelle adresse. À cette adresse, spécifiée par le biais d'un registre dédié, se trouve un code responsable de gérer le problème détecté.

En général, les mécanismes d'exception sont couplés avec les niveaux de privilèges. Lors de la détection, le système change généralement le

niveau en cours d'exécution avant la redirection. En amont, seul ce même niveau peut donc choisir l'adresse de redirection. Les exceptions sont également utilisées pour changer explicitement le privilège en cours d'exécution. Cela sert par exemple à un niveau peu (ou pas) privilégié à rendre le contrôle à un autre niveau plus privilégié pour effectuer une tâche spécifique.

Même si elles peuvent être traitées de la même manière, les exceptions doivent être distinguées des *interruptions*. Une interruption est déclenchée par un évènement externe et indépendant des instructions en cours d'exécution. C'est par exemple le cas si un périphérique a terminé une opération. Elle sert donc à indiquer au système qu'une situation doit être traitée. De la même manière, le processeur est capable de rediriger l'exécution en cas d'interruption détectée.

2.7. Conclusion

Les microarchitectures des processeurs modernes sont le résultat de plusieurs dizaines d'années de travaux. Ce sont des systèmes complexes faisant appel à de nombreux mécanismes. Ainsi, l'exécution d'une seule instruction mène à plusieurs opérations matérielles.

Le but principal de la plupart de ces mécanismes est d'améliorer les performances. L'idée principale pour cela est d'améliorer le débit d'instructions exécutées. L'utilisation d'un pipeline permet d'augmenter la fréquence d'horloge tout en parallélisant l'exécution de plusieurs instructions. Les mémoires caches réduisent considérablement les temps des accès mémoires. Les mécanismes de spéculation sont également essentiels pour réduire l'impact de certains évènements sur l'exécution. Si ces mécanismes se concentrent sur l'exécution d'un seul programme, d'autres comme le multithreading ou les organisations multicœurs permettent d'optimiser les cas où de multiples programmes indépendants sont à exécuter. La microarchitecture est alors conçue pour permettre l'exécution de différents programmes simultanément.

Contrairement aux précédents mécanismes, la mémoire virtuelle est utilisée dans un intérêt purement fonctionnel. Elle permet notamment au logiciel une meilleure gestion de la mémoire.

Dans la suite de cette thèse, nous allons voir que ces différents mécanismes ont été conçus sans tenir compte des besoins de sécurité. Nous verrons que si certaines problématiques sont connues depuis longtemps, elles ont été ignorées et se sont amplifiées avec le temps. Ainsi, des mécanismes de sécurité de base comme les privilèges ne sont plus suffisants. À partir du prochain chapitre, nous allons particulièrement nous intéresser aux attaques logicielles exploitant la microarchitecture. Ces attaques détournent directement le fonctionnement des mécanismes microarchitecturaux présentés afin d'en exfiltrer des informations.

Exploitation de la microarchitecture par le logiciel 3.

Résumé du chapitre : Dans ce chapitre, nous nous intéressons aux attaques logicielles exploitant les mécanismes de la microarchitecture. De nombreux travaux ont montré que des programmes normalement isolés pouvaient interagir grâce au matériel : un principe microarchitectural comme le partage de ressources est notamment utilisé. Après quelques rappels sur des éléments et concepts nécessaires, nous étudions les différentes attaques exploitant la microarchitecture en quatre parties : les attaques par analyses des variations temporelles, les attaques par contention dynamique de ressources, les attaques par exécution transitoire et enfin les attaques par canaux contrôlés.

3.1 Objectifs de sécurité 27
3.2 Principes microarchitecturaux exploités 28
3.3 Attaques par analyse des variations temporelles . . . 29
Principes 29
Attaques sur les mémoires caches 30
Étude généralisée du système 33
3.4 Attaques par contention dynamique de ressources . 36
Principes 37
Impact sur les timings 37
3.5 Attaques par exécution transitoire 38
Principes 38
Spectre 39
Meltdown 41
Échantillonnage des données microarchitecturales . 42
3.6 Attaques par canal contrôlé 44
3.7 Synthèse et conclusion . . . 45

3.1. Objectifs de sécurité

Contexte Nous avons vu dans le chapitre précédent que les processeurs sont des composants électroniques essentiels. Ils sont responsables de l'exécution des calculs au sein de systèmes variés, que ce soit des ordinateurs, des serveurs, de téléphones ou d'autres systèmes embarqués. Selon les cas d'application, ils peuvent être utilisés pour le traitement de données sensibles et confidentielles. Ils doivent alors être capables d'assurer la protection de ces données.

Ce dernier aspect a longtemps été négligé au profit des performances. L'existence de potentielles failles exploitables est connue depuis de nombreuses années. L'ajout de nouveaux mécanismes, comme vu précédemment, n'a fait qu'accroître les faiblesses de ces systèmes. Cela mène aux processeurs modernes de ces dernières années, cibles d'attaques certes plus complexes, mais surtout plus puissantes.

Modèle d'attaquant Un modèle d'attaquant définit les capacités d'un attaquant. Il détermine la manière dont ce dernier peut influer sur le système et quelles actions il peut mener. De la même manière, ce modèle délimite les menaces que doivent considérer les solutions mises en place. Dans ce manuscrit, nous nous intéressons aux attaques logicielles exploitant la microarchitecture. Elles peuvent être définies comme des programmes qui, au cours de leur exécution, utilisent les différents mécanismes matériels à disposition à des fins malveillantes (*e.g.* retrouver des informations secrètes).

Dans notre cas, nous considérons donc un attaquant capable d'exécuter des programmes sur un processeur et d'effectuer des mesures de temps. Généralement, les ISA incluent des instructions permettant de connaître le nombre de cycles exécutés. Cette fonctionnalité est commune et essentielle pour de nombreux programmes. En exécutant deux mesures de cycles, un attaquant est capable de déduire le nombre de cycles qui s'est écoulé entre ces deux instructions. Il peut ainsi connaître le temps qu'a utilisé le processeur pour exécuter les instructions entre ces deux mesures. Dans notre modèle, l'attaquant dispose du minimum de droits sur le système. Il n'est donc pas privilégié et n'a accès qu'à ses propres données.

Scénarios d'attaques Une fois les capacités d'un attaquant définies, il est nécessaire de détailler le but qu'il cherche à atteindre. Les attaques décrites dans ce chapitre sont utilisées dans deux scénarios différents.

Le premier scénario d'attaquant est celui des **attaques par canaux cachés**. Leur but est la transmission d'information entre un émetteur et un récepteur par le biais d'un canal initialement non prévu à cet effet [4, 5]. C'est un scénario intéressant pour évaluer un système de par le pouvoir qu'il donne à l'attaquant : celui-ci maîtrise ici le canal de bout en bout. Il est donc possible pour cet attaquant de mettre en place un protocole très complexe pour optimiser les performances de la transmission. Généralement, on appelle l'émetteur "cheval de Troie" (ou *trojan* en anglais) et le récepteur "espion"(ou *spy* en anglais). Dans notre cas, nous verrons que des canaux cachés peuvent être créés en manipulant les variations temporelles. Ce genre de technique est connu depuis déjà plus de deux décennies [6], et représente une première approche intéressante avant la mise en place d'attaques par canal auxiliaire [7].

Le second scénario généralement utilisé est celui des **attaques par canaux auxiliaires**. Leur but est ici la récupération d'informations par l'attaquant, toujours appelé "espion", sur une victime. C'est ici un réel scénario d'observation où l'attaquant a une influence plus limitée. Il doit prendre en compte davantage d'éléments qu'il ne maîtrise pas, comme la forme des informations reçues. Là encore, il a été montré depuis de nombreuses années que les variations temporelles peuvent pour cela être exploitées [8]. Nous verrons même qu'il est parfois possible pour un attaquant d'inverser ce scénario, en envoyant lui-même des informations afin d'influencer la victime.

3.2. Principes microarchitecturaux exploités

Variation temporelle Lors de son exécution, un programme met un certain temps à être entièrement exécuté par le processeur. Le rôle des différents mécanismes d'optimisation vus dans le chapitre précédent est de réduire autant que possible ce temps d'exécution. Le *autant que possible* a ici une importance particulière : ces accélérations n'étant pas toujours effectives, des disparités peuvent être observées entre plusieurs exécutions d'un même programme ou d'une même instruction. Ce sont ces variations que l'on appelle variations temporelles. En les détectant et en les analysant, nous verrons qu'il est possible pour un attaquant de les exploiter et d'en déduire des informations.

Partage de ressources Les ressources matérielles étant disponibles en nombre limité, elles sont partagées entre les différents programmes qui s'exécutent sur le système. L'exemple le plus parlant est le cas du "multithreading" présenté dans le chapitre précédent. Mais cela est également vrai pour des programmes qui s'exécutent à tour de rôle sur un même cœur physique : ils utilisent l'un après l'autre les mêmes ressources. Dans la suite de cette thèse, quand l'on évoquera une ressource partagée, on considèrera donc le cas général d'une ressource matérielle accessible par deux entités logicielles différentes.

Pour décrire une ressource partagée, on utilise généralement deux états différents. Le premier est l'**état persistant** qui correspond à un état qui se maintient dans le temps. C'est notamment le cas des ressources utilisant des registres ou des mémoires qui influencent son fonctionnement

(*e.g.* les mémoires caches). À l'inverse, l'**état transitoire** correspond à un état temporaire d'une ressource à un moment précis. L'état transitoire qui nous intéresse particulièrement ici est la contention dynamique, c'est-à-dire la détection de l'utilisation ou non d'une ressource à un moment donné.

Nous verrons par la suite que ces états peuvent représenter des sources d'information pour un attaquant.

3.3. Attaques par analyse des variations temporelles

Dans cette section, nous étudions une première catégorie d'attaques utilisant l'analyse des variations temporelles au niveau de la microarchitecture. Nous nous concentrons dans un premier temps sur les attaques ciblant les mémoires caches. De par leur fonctionnement et leur nombre, elles sont une parfaite représentation du problème. Nous verrons ensuite que l'ensemble des mécanismes de la microarchitecture peuvent être une source de variation temporelle.

Nous distinguons les attaques exploitant l'état persistant des ressources de celles dues à la contention dynamique de ressources (et donc exploitant un état transitoire de celles-ci). Ces dernières ayant un fonctionnement différent, elles sont étudiées dans la section 3.4.

Principes

Comme vu précédemment, les microarchitectures des processeurs modernes sont le résultat de nombreuses années d'optimisations de leurs performances. L'un des principes particulièrement implémentés est la réutilisation d'informations liées aux exécutions passées pour la spéculation. Cela concerne aussi bien :

- les caches qui mémorisent les adresses déjà utilisées pour accélérer les futurs accès mémoires,
- les prédicteurs de branchements qui conservent un historique des précédents branchements pour en déduire les suivants,
- ou les prefetchers qui conservent un historique des précédents accès mémoires pour anticiper les prochains.

Ces optimisations s'appuient sur des données ou métadonnées stockées en interne pour accélérer l'exécution d'un programme. Il y a donc une dépendance qui se crée entre l'état persistant de ces ressources et les variations temporelles qui en découlent. À partir de ce constat, de nombreux travaux ont montré la possibilité de les exploiter pour en déduire des informations.

L'exemple le plus représentatif de ce type d'attaques est celui des mémoires caches. Par définition, une donnée à l'intérieur d'un programme est plus rapidement disponible (un cache *hit*) qu'une donnée qui n'y est pas (un cache *miss*). De par le fonctionnement très souvent déterministe des caches, il existe un lien direct entre le temps d'un accès et les accès réalisés auparavant. Un temps d'accès peut donc être représentatif d'une information particulière. Ainsi, si un attaquant est capable de le mesurer, il peut ensuite en déduire des informations (*e.g.* que la donnée a été utilisée auparavant ou non).

Attaques sur les mémoires caches

Depuis les années 1990, de nombreux exemples d'attaques [6, 8] exploitant les caches ont été présentés. Nous utilisons ici deux classifications différentes couramment mentionnées et qui illustrent la variété et le potentiel de ces attaques.

Cryptanalyse

La cryptanalyse est un ensemble de techniques visant à retrouver les informations secrètes liées aux algorithmes cryptographiques. Ces derniers sont les piliers de nombreux mécanismes de sécurité. On retrouve un grand nombre de travaux s'intéressant aux fuites d'informations dues à l'exécution de certains de ces algorithmes sur des processeurs : Diffie-Hellman [8], RSA [7, 9], Advanced Encryption Standard (AES) [10-13] ... Historiquement, une première classification [14] a donc été utilisée pour décrire les capacités d'un attaquant à partir des mémoires caches du point de vue de la cryptanalyse.

Attaques basées sur les traces La première technique repose sur la récupération et l'analyse des traces d'accès à un cache semblables à celle présentée dans le Code 3.1. Plusieurs travaux ont montré que des informations sont déductibles à partir de la liste des *hits* et *misses* d'un programme.

Code 3.1: **Exemple d'une trace de cache.** Cette dernière donne le résultat de 16 accès différents. M représente un cache *miss* et H un cache *hit*. En considérant le cache vide au départ, on peut ainsi déduire :

- Le second accès est sur la même ligne que le premier.
- Le troisième accès est sur la même ligne que le premier.
- Le quatrième accès est sur une autre ligne que le premier.
- Le cinquième accès est soit sur la même ligne que le premier, soit sur la même que le quatrième.
- *etc.*

```
MHHM HMHM MMHM HHMH
```

En 2002, Page [15] prouva théoriquement que les traces de caches peuvent être utilisées pour la cryptanalyse d'implémentations de l'algorithme Data Encryption Standard (DES). Bertoni et al. [16] montrèrent qu'en ayant le contrôle sur les données présentes dans le cache au départ, un attaquant est capable de forcer des caches *misses* détectables. Ils sont alors utilisés pour retrouver partiellement une clé de chiffrement AES. D'autres travaux [17, 18] ont également prouvé la possibilité de fragiliser ce même algorithme en récupérant des traces de plusieurs chiffrements.

Les attaques de ce type souffrent néanmoins d'une limite majeure : elles nécessitent un accès physique à la cible. Il est sinon impossible d'effectuer les mesures physiques nécessaires. Les travaux les exploitant utilisent généralement des techniques à base de mesures électromagnétiques ou de consommation de courant [15-18] qui sont ici en dehors de notre modèle d'attaquant. Ainsi, ces traces ne sont pas récupérables à partir du logiciel. Si elles concernent les mémoires caches, ces attaques restent en dehors de notre modèle d'attaquant.

Attaques basées sur le temps Une deuxième technique mesure le temps d'exécution du programme visé. Elle exploite la dépendance entre les données et/ou le flot d'exécution du programme avec les variations temporelles induites au niveau des caches. À l'inverse de la technique précédente, celle-ci peut être totalement logicielle tant que des mécanismes permettent d'effectuer des mesures de temps.

En 2003, Tsunoo et al. [19] analysèrent les variations dans l'exécution d'un algorithme DES dues à des collisions internes menant à des caches *misses*. En ciblant le même processeur, mais utilisé dans un serveur, Bernstein [10] montra comment récupérer entièrement une clé AES à partir de plusieurs chiffrements connus comportant des variations temporelles (toujours dues à des collisions internes). Ses travaux montrèrent aussi à quel point cette problématique est généralisée, en observant des

résultats similaires sur des implémentations de différents constructeurs (AMD Athlon, Pentium III, Pentium M, IBM PowerPC RS64 IV or Sun UltraSPARC III).

Attaques basées sur les accès Le dernier type de techniques repose pleinement sur le partage des caches entre une victime et un attaquant pour permettre la détection d'accès mémoire. En déterminant quels ensembles ou lignes d'un cache ont été utilisés, il est possible de retrouver des informations partielles sur l'adresse visée.

En 2005, Percival [7] exploita les mémoires caches partagées entre des threads exécutés simultanément. Après avoir rempli le cache avec ses propres données, un espion laisse la victime effectuer des opérations. Plus tard, en accédant à nouveau à ses propres données, l'espion détecte lesquelles ont été évincées ou non. Selon les lignes toujours présentes, des informations peuvent être déduites menant à la récupération d'une clé RSA. C'est ce même principe qui fut exploité par Osvik, Shamir et Tromer [12] mais cette fois pour la récupération d'une clé AES.

Ce type d'attaque n'est cependant pas propre aux processeurs avec support du SMT. Neve et Seifert [13] ont montré la possibilité d'appliquer ces mêmes principes sur un processeur avec un seul thread. Dans ce cas-là, il est nécessaire d'influencer le système d'exploitation pour qu'il alterne régulièrement les exécutions de l'espion et de la victime. En 2010, Aciicmez, Brumley et Grabher [20] prouvèrent enfin que ce type d'attaques pouvait également s'appliquer aux caches d'instructions.

Finalement, cette classification montre une diversité dans les informations vulnérables à cause des mémoires caches, que ce soit des accès ou des temps d'exécution. Cependant, elle ne s'intéresse à l'exploitation des caches que dans le but de la cryptanalyse. Surtout, cette classification ne décrit pas les différentes méthodes pour exploiter un cache, et donc le pouvoir réel d'un attaquant sur le système.

Types d'attaques sur les caches

Une autre classification [21, 22] s'est imposée au cours du temps. Elle regroupe les attaques par stratégie d'exploitation des caches. Contrairement à la classification précédente, elle se concentre exclusivement sur les attaques logicielles. Nous allons ici décrire brièvement différentes méthodes connues pour l'exploitation des caches, essentiellement pour des attaques par canal auxiliaire.

Evict+Time Un premier type d'exploitation des caches est appelé Evict+Time [12, 23]. Son objectif est d'exécuter plusieurs fois un même programme pour détecter des variations du temps d'exécution selon que des lignes sont évincées ou non par l'attaquant. Cette technique passe par plusieurs itérations de ces trois étapes :

1. Trigger : le programme ciblé est exécuté (*e.g.* un chiffrement) et le temps mesuré comme référence,
2. Evict : certaines lignes du cache sont évincées par l'attaquant,
3. Time : l'exécution est relancée et le temps d'exécution mesuré puis comparé avec la référence.

Cette technique implique que l'attaquant et la victime partagent ici au moins une partie du cache. Elle permet à l'attaquant de connaître

l'influence d'une ligne de cache sur l'exécution de la victime.

Prime+Probe Un autre type d'attaques appelé Prime+Probe [12] utilise l'éviction pour déduire des informations sur la victime. Là encore, le procédé s'effectue en trois étapes :

1. Prime : l'attaquant remplit la mémoire cache avec ses propres données,
2. Trigger : le programme ciblé est exécuté,
3. Probe : l'attaquant identifie les lignes évincées.

L'attaquant est ainsi capable d'obtenir une empreinte mémoire d'un programme cible, c'est-à-dire quelles lignes celui-ci a du évincer pour s'exécuter. Comme pour Evict+Time, Prime+Probe implique donc un partage du cache entre la victime et l'attaquant.

Cette technique a aussi bien été démontrée sur des caches L1D [7, 12], que sur des L1I [9, 24] ou même des last-level cache (LLC) [25]. Ristenpart et al. [26] ont montré la possibilité d'utiliser cette technique entre différentes VM. Ge et al. [27] l'ont sélectionné pour leur étude des fuites temporelles dans de multiples processeurs modernes en 2017.

Flush+Reload En plus du partage d'une mémoire cache, il est possible d'exploiter le logiciel partagé, et notamment certaines librairies, entre l'attaquant et la victime. C'est le cas des attaques Flush+Reload [28, 29]. Leur objectif est de déduire quand une victime utilise ou non certaines plages mémoires partagées. Pour cela, trois phases distinctes sont nécessaires :

1. Flush : l'attaquant vide la mémoire cache,
2. Trigger : le programme ciblé est exécuté,
3. Reload : l'attaquant utilise certaines plages mémoires partagées et identifie celles présentent dans les caches.

La première version présentée [28] est exécutée sur un processeur avec un seul thread. L'ordonnanceur du système d'exploitation est alors responsable de l'alternance des exécutions de la victime et l'attaquant. Ces travaux montrent la possibilité pour un attaquant de récupérer une clé AES sur un Pentium M. La librairie partagée utilisée était alors OpenSSL. Cette technique a ensuite été améliorée [29] pour cibler le LLC et ainsi permettre à l'attaquant d'être situé sur un cœur physique différent. Elle a aussi été démontrée avec succès entre différentes VM, permettant l'extraction de clés RSA. Finalement, ce type d'attaques s'avère efficace contre d'autres d'algorithmes : AES [30], Elliptic Curve Digital Signature Algorithm (ECDSA) [31-33] ...

L'élément clé de ces attaques est la possibilité de vider le contenu des caches avec des instructions dites de *flush*. Ces instructions étant dépendantes de l'ISA, les attaques doivent être modifiées selon le système visé. Dans le cas du jeu d'instructions x86, l'instruction `clflush` (pour *cache line flush* justement) est parfaitement adaptée [29].

D'autres variantes de ces types d'attaques existent afin d'exploiter les mémoires caches. C'est le cas de Evict+Reload [34] qui évince des données au lieu de les effacer comme dans Flush+Reload. Elle ne dépend donc pas de l'utilisation d'une instruction de flush. Flush+Flush [35] est particulière puisqu'elle exploite le temps d'exécution de l'instruction de flush elle-même pour déduire la présence de données ou non. Finalement, ces

différentes attaques montrent bien les possibilités offertes à un attaquant pour exploiter les caches à partir du moment où ceux-ci sont partagés avec une potentielle victime. Nous allons voir que ce même constat peut également être fait avec d'autres mécanismes partagés du système.

Étude généralisée du système

Les mémoires caches représentent une cible importante des attaques exploitant les variations temporelles. Cependant, il est possible d'exploiter de la même façon d'autres mécanismes. Nous allons voir que ce problème touche finalement une grande partie de la microarchitecture, quel que soit l'endroit du système.

Autres mécanismes

Nous avons vu dans le chapitre 2 que de nombreux mécanismes, notamment pour la spéculation, accumulent des informations au cours des exécutions. Celles-ci représentent donc un état persistant qui, lorsqu'il est partagé, devient exploitable pour un attaquant.

Différentes mémoires caches Une grande partie des attaques sur les caches utilisent les accès mémoires pour les données. Il est important d'insister sur le fait que le principe est indépendant du type d'informations qu'elles contiennent. Ainsi le cas des caches d'instructions est également connu depuis de nombreuses années [9, 20]. De même, le cache des micro-opérations, utilisé notamment pour accélérer le décodage des instructions complexes x86, représente une ressource partagée exploitable [36]. Ces mêmes travaux ont d'ailleurs montré des différences entre les implémentations Intel et AMD. Les premiers ne sont pas vulnérables à cette attaque grâce à la séparation statique du cache entre les threads d'un cœur physique. Enfin, des attaques touchent les caches indépendamment de leurs niveaux dans la hiérarchie [25, 29, 37].

Prédicteur de branchement Les prédicteurs de branchement influencent différemment l'exécution d'un programme selon les informations qu'ils contiennent. Logiquement, un branchement correctement prédit n'aura aucun effet sur le temps d'exécution du programme. Le programme sera exécuté comme un programme linéaire. À l'inverse, une mauvaise prédiction ajoutera une pénalité de plusieurs cycles. Ces variations temporelles peuvent également être exploitées.

En 2007, Aciicmez, Koç et Seifert [38] ont présenté plusieurs attaques exploitables pour la cryptanalyse. Le programme ciblé doit pour cela être conçu avec un flot d'exécution dépendant de données secrètes. La première attaque nécessite de connaître précisément l'état du prédicteur de branchement avant l'exécution de la victime. Ensuite, son temps d'exécution est mesuré. Les prédictions étant déterministes, il est alors possible de comprendre l'impact du prédicteur sur les variations observées, afin d'en déduire des informations. Une deuxième attaque force le prédicteur à toujours effectuer la même prédiction. Pour cela, l'attaquant doit être capable de modifier l'état du BTB régulièrement depuis son propre processus (*e.g.* en utilisant le SMT). Enfin, la troisième attaque présentée reprend le principe des attaques de caches exploitant les traces. L'espion remplit le BTB avec ses propres données et ensuite détecte les utilisations par la victime pour en déduire des informations. Une seule

exécution de l'espion [39] peut même être suffisante pour la récupération d'une clé RSA.

Si le SMT est utilisé ici pour espionner l'état du BTB, Aciicmez, Koç et Seifert [38, 40] mentionnent le fait qu'il ne semble pas essentiel. L'attaquant doit alors s'assurer que le système d'exploitation intervertit régulièrement les exécutions des processus victime et espion.

TLB Comme vu précédemment, le TLB n'est finalement qu'une mémoire cache dédiée à l'accélération de la traduction de la mémoire virtuelle. Ainsi, il génère de la même manière des disparités temporelles : une entrée de page connue permettra une traduction d'adresse plus rapide. Ces variations ont également été prouvées comme exploitables.

Van Bulck et al. [41] ont constaté que le contenu des TLB est implicitement partagé avec les enclaves Intel Software Guard Extensions (SGX)[1]. Ces structures sont vidées avant et après l'exécution d'une enclave. Cependant, elles utilisent également le reste de la hiérarchie des caches qui elle reste non-impactée. Des informations persistent donc.

1: Intel SGX est une extension x86 pour la création d'enclaves, des programmes isolés de tout autre logiciel. Une brève description d'Intel SGX est faite dans la section 5.2.

Prefetcher Les prefetchers visent à accélérer les accès mémoires au sein de la hiérarchie. Nous avons vu qu'ils s'appuient eux aussi sur des métadonnées accumulées au cours du temps pour maximiser leur efficacité. Là encore, il est possible d'exploiter les différents états persistants.

Cronin et Yang [42] ont montré la possibilité de monter un canal caché sur un processeur Intel. Dans le système ciblé, le prefetcher est partagé entre les processus d'un même cœur physique. Il possède également un nombre d'entrées limité par page mémoire. Ainsi, un processus cheval de Troie peut modifier l'état de certaines entrées avant qu'un espion ne le détecte. Comme avec les caches, cette détection est possible en observant les variations temporelles : une donnée dans le prefetcher est plus rapidement accessible.

Protocole de cohérence mémoire Dans les implémentations multicœurs, il est possible que chaque cœur ait sa propre copie d'une ligne mémoire. C'est par exemple le cas si une donnée est utilisée par chacun d'entre eux : elle est ramenée dans chaque cache L1. Ainsi, si l'un d'eux modifie cette donnée, il est nécessaire de propager l'information aux autres caches. Cette transmission d'informations s'effectue par le biais d'un protocole de cohérence mémoire.

Ferraiuolo et al. [43] ont montré la possibilité d'établir un canal caché de communication à partir de ce protocole. Celui-ci est responsable de variations temporelles lors de l'accès aux adresses partagées. En exploitant ces variations, il est donc possible de transmettre des données d'un cœur vers un autre.

Pipeline et unités d'exécution De la même manière, un pipeline de processeur et ses mécanismes internes influent sur le temps d'exécution des programmes.

En 1996, Kocher [8] évoquait déjà les temps d'exécution variables pour des instructions comme les multiplications ou divisions. Hachez et Quisquater [44] expliquaient plus tard que sur certains processeurs Arm, des optimisations permettaient d'accélérer l'exécution de multiplications si les bits d'opérandes étaient à 0. En 2009 finalement, Grossschädl et al. [45] décrivaient comment ce genre de mécanismes pouvait être exploité

sur des systèmes embarqués afin d'extraire des informations. Finalement, Andrysco et al. [46] ont montré que ces variations ne sont pas propres aux unités de traitement des multiplications et divisions. Ainsi, des variations de nombre de cycles peuvent aussi être observées pour le traitement de certaines valeurs dans les unités de traitement des nombres flottants (FPU). En effectuant des mesures temporelles, retrouver des informations devient donc possible en fonction des instructions exécutées.

Schwarz et al. [47] ont également exploité les variations temporelles au sein de l'unité dédiée aux vecteurs dans les processeurs Intel (unité AVX pour *Advanced Vector Extensions*). Cette unité opère normalement sur 256 bits, mais certaines instructions n'utilisent que les 128 bits de poids faibles. Pour des gains en consommation, la partie responsable des bits supérieurs est donc ralentie en cas d'une longue période d'inactivité ($\approx 1ms$). Ainsi, si une instruction opérant sur les 256 bits doit être exécutée, un ralentissement général est observé le temps de relancer l'intégralité de l'unité. Schwarz et al. exploitent alors ces variations pour la conception d'un canal caché.

Un autre mécanisme appelé *store-to-load (STL) forwarding* permet au processeur avec exécution dans le désordre de transférer les valeurs de `store` non rangées à des `load` dépendants. Ce mécanisme cherche à anticiper les dépendances entre ces opérations. Cependant, les adresses utilisées au niveau du processeur sont des adresses virtuelles. Dans le cas d'un processeur avec SMT, celui-ci n'est donc pas toujours capable de savoir si deux adresses virtuelles de threads différents correspondent à la même adresse physique. Ainsi, la spéculation s'effectue dans un premier temps sur les bits d'offset (généralement 12 bits communs à l'adresse virtuelle et physique). Si ces bits sont les mêmes, alors le résultat d'un `store` non rangé est transféré spéculativement à un `load`. À l'inverse, si un `load` plus ancien est en cours d'exécution, alors un `store` sera retardé pour conserver l'ordre du programme.

C'est cette dernière dépendance qui est utilisée par Yarom, Genkin et Heninger [48]. Le principe d'une des attaques proposées est d'espionner les adresses chargées par un processus victime sur un processeur avec SMT. Pour cela, l'espion réalise une suite de `store` à des adresses consécutives et mesure le temps pour chacun d'entre eux. Si un d'eux est retardé, alors cela implique que la victime a effectué un `load` vers une adresse avec le même offset. À défaut d'une implémentation d'attaque, Yarom, Genkin et Heninger ont montré que ces variations temporelles sont réellement présentes.

Partage des ressources au niveau matériel

Les systèmes modernes peuvent se décomposer en différents niveaux de partage. Chacun définit les entités matérielles entre lesquelles les ressources sont partagées. Il est intéressant de voir dans quelles mesures les fuites d'informations temporelles varient entre chaque niveau.

Le contrôle du logiciel sur le système par exemple diminue à chaque niveau. Les instructions sont exécutées au niveau des threads. Ainsi, le logiciel a logiquement un plus grand contrôle sur les mécanismes partagés proches des threads. Cependant, il existe également des ressources partagées à d'autres niveaux de la hiérarchie, comme entre plusieurs cœurs ou même plusieurs puces électroniques (une puce correspondant à tous les éléments regroupés dans un même *package*). Des attaques restent possibles à ces niveaux-là, mais plus complexes à mettre en place.

Les interférences d'un plus grand nombre d'entités sont notamment à prendre en compte. Ge et al. [21] présentent un récapitulatif intéressant des différents niveaux de partages concernés, illustré ici par la Figure 3.1

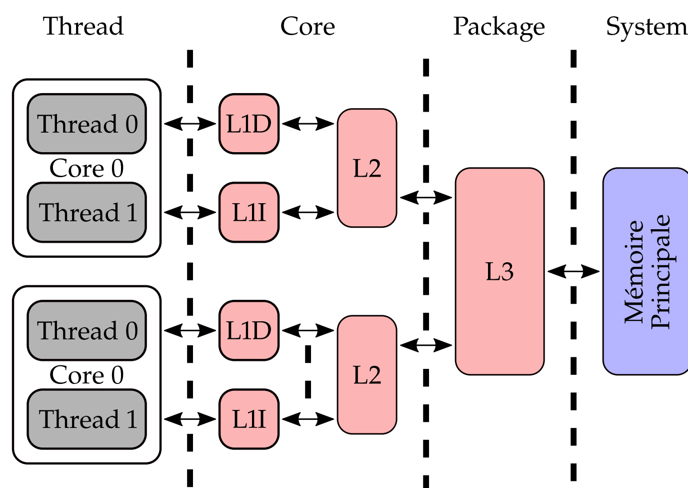


Figure 3.1. – Différents niveaux de partage. Le système peut être décomposé en différents niveaux représentant l'échelle à laquelle les ressources sont partagées. Par exemple, les tables de prédictions sont propres à chaque thread. Les deux premiers niveaux de cache sont eux partagés au sein d'un même cœur *etc*.

Le premier de niveau de partage se situe au niveau d'un thread matériel. Il concerne certaines tables de prédictions comme les BHTs. Seuls les processus exécutés sur ce thread peuvent donc accéder ces mécanismes et les exploiter [38-40].

À l'échelle d'un cœur physique, les premiers niveaux de mémoire (L1D, L1I et les TLB) sont partagés. Les threads d'un même cœur utilisent généralement les mêmes mémoires caches. Nous avons vu précédemment que de nombreuses attaques existent aussi à ce niveau [9, 12, 20].

Le partage est également possible au niveau d'une même puce électronique. Cela peut concerner les mécanismes mis en commun entre différents cœurs physiques. C'est généralement le cas du LLC (ou à partir du L3). Ici aussi, certaines attaques ont montré la possibilité d'exploiter ce niveau de partage [25, 29].

Finalement, on associera au dernier niveau l'ensemble des éléments communs à tout le système comme le bus d'interconnexion à la mémoire principale. Comme pour les niveaux précédents, des attaques ont montré la possibilité d'exploiter les partages existants [49].

Pour conclure, l'ensemble des éléments partagés sont susceptibles d'être exploités afin d'obtenir des informations. Le niveau de partage et donc de contrôle sur les ressources n'est pas suffisant pour empêcher une exploitation par un attaquant.

3.4. Attaques par contention dynamique de ressources

Notes : Dans la littérature, le terme de contention peut évoquer le fait de posséder une donnée statiquement. Par exemple, une ligne ramenée dans un cache par un utilisateur illustre la contention de cette ligne par l'utilisateur. On parle alors de contention statique (déduite d'un état persistant). Dans ce manuscrit, le terme de contention se réfère uniquement à la contention dynamique (déduite d'un état transitoire).

Les attaques étudiées jusqu'à présent utilisaient l'état persistant des ressources. Des variations temporelles étaient générées en fonction des valeurs stockées par les différents mécanismes. Or, certaines attaques exploitent l'état transitoire des ressources et plus particulièrement leur disponibilité par le biais de la contention dynamique.

Principes

On appelle contention dynamique d'une ressource l'état transitoire indiquant son utilisation. Par exemple, lorsqu'une ressource est utilisée et qu'elle n'est capable d'effectuer qu'une opération à la fois, elle apparaît alors comme indisponible pour les autres utilisateurs. C'est le cas décrit sur la Figure 3.2.

Au sein de la microarchitecture, il n'est pas toujours possible de dupliquer chaque ressource pour satisfaire tous les utilisateurs possibles (*e.g.* il y a moins de ressources que de threads). Cela peut mener à des cas de contention de ressources : les utilisateurs sont en conflit pour utiliser les ressources. Certains doivent donc attendre, ce qui implique le ralentissement de leurs exécutions. C'est cette variation temporelle qui est détectable et exploitable. Nous allons à présent voir comment ces situations peuvent être exploitées par un attaquant à plusieurs endroits du système.

Impact sur les timings

Il existe plusieurs mécanismes dans la microarchitecture pouvant mener à de la contention dynamique. C'est le cas des unités d'exécution, des banques mémoires, des bus mémoires et des contrôleurs mémoires. Deux d'entre eux ont particulièrement été exploités : les unités d'exécution et les contrôleurs mémoires.

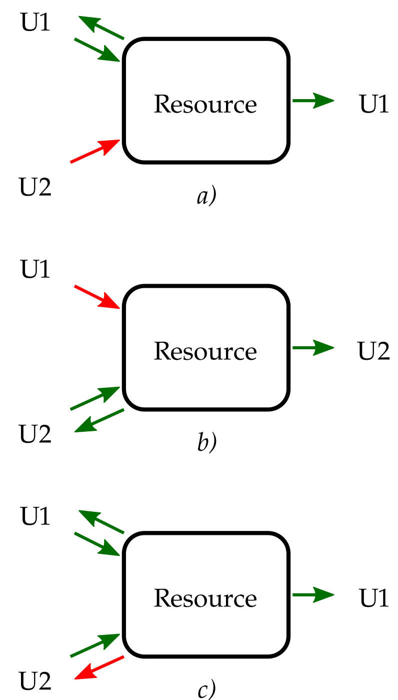


Figure 3.2. – Contention d'une ressource U1 et U2 sont deux utilisateurs de la ressource qui ne peut accepter qu'une requête simultanément. Dans les cas a) et b), U1 ou U2 font respectivement seuls une requête qui est acceptée. Dans c), U1 et U2 font simultanément une requête. La ressource ne peut en choisir qu'une seule : elle sélectionne ici U1. U2 doit donc attendre. Cette attente crée un délai dans l'exécution de U2, symptomatique de la contention de la ressource par U1.

Unités d'exécution Dans le cas d'un processeur supportant le SMT, les unités d'exécution sont partagées entre les threads. Ainsi, si deux d'entre eux cherchent à utiliser une même ressource au même moment, un cas de contention se crée. Le thread n'ayant pas la ressource voit son exécution ralentie jusqu'à la libération de la ressource convoitée. En mesurant le temps d'exécution, il devient possible de retrouver la contention.

Ce principe est connu depuis 2007 avec les travaux de Aciicmez et Seifert [50]. Ils ont alors prouvé la possibilité pour un espion de détecter l'utilisation d'une ressource par un processus de cryptographie. La précision de leur attaque leur permet seulement de distinguer différentes opérations mathématiques.

Cette approche a ensuite été actualisée en 2019. Aldaya et al. [51] ont tout d'abord montré la possibilité d'utiliser la contention afin de monter un canal de communication caché. Le même principe a également été exploité pour monter une attaque par canal auxiliaire dans lequel l'espion sature les ports d'exécution et effectue des mesures de temps régulières. Ce scénario leur a permis de démontrer la possibilité de retrouver une clé secrète ECDSA depuis un serveur, mais aussi l'utilisation contre des enclaves SGX.

La contention dynamique de port peut aussi être utilisée pour monter des attaques plus complexes [52] comme les attaques transitoires décrites ultérieurement.

Banques mémoires Les mémoires caches sont généralement divisées en une ou plusieurs petites mémoires indépendantes appelées banques mémoires. Cette stratégie permet d'effectuer plusieurs opérations d'écriture ou lecture en parallèle, chaque banque disposant de ses propres ports d'accès. Bernstein [10] et Osvik, Shamir et Tromer [12] mentionnaient

déjà en 2005 l'existence de conflits. Yarom, Genkin et Heninger [48] ont exploité la contention de ces banques pour détecter les accès réalisés par une victime. Ces informations ont ensuite été utilisées pour la récupération d'une clé RSA.

Bus mémoire Les bus mémoire sont des ressources partagées entre les différents utilisateurs souhaitant accéder à une mémoire. Ils sont généralement présents en nombre limité ce qui mène à des cas de contention. Ce scénario a notamment était mentionné en 2009 par Ristenpart et al. [26]. Wu, Xu et Wang [49] ont eux utilisé un bus comme canal de communication caché entre plusieurs VM.

Contrôleurs mémoire Les contrôleurs mémoires sont partagés entre les différents processus qui s'exécutent. Mais à la différence des unités d'exécutions, leur contention n'est pas spécifique aux processeurs avec SMT. Wang, Ferraiuolo et Suh [53] ont monté un canal caché entre deux cœurs de cette manière. Un scénario d'observation par canal auxiliaire a également été démontré en retrouvant des informations sur une clé RSA.

Finalement, Ristenpart et al. [26] ont aussi mentionné la possibilité d'utiliser la contention d'un disque dur.

3.5. Attaques par exécution transitoire

Les attaques présentées jusqu'à présent se focalisaient généralement sur une ressource partagée / un type de variation temporelle afin de mettre en place un canal d'observation ou de communication. En 2018, des travaux ont montré la possibilité de monter des attaques encore différentes au sein des microarchitectures modernes : c'est la découverte des célèbres attaques Spectre [54] et Meltdown [55]. Dans cette partie, nous étudions leur fonctionnement général. L'objectif est de prouver que si elles sont plus complexes, elles font toujours en grande partie appel aux mêmes principes fondamentaux. Nous distinguons trois types d'attaques par exécution transitoire : celles de types Spectre, celles de types Meltdown et celles par échantillonnage de la microarchitecture.

Principes

Exécution transitoire Lors de l'exécution d'un programme, les processeurs essayent d'anticiper les futures instructions afin d'accélérer l'exécution. Cette spéculation sur la future exécution ne s'avère cependant pas toujours correcte : le processeur peut avoir mal anticipé les prochaines instructions. Après la détection de cette erreur, le processeur passe alors dans une phase de restauration, où il revient dans le dernier état correct rencontré. On appelle *exécution transitoire* la fenêtre temporelle durant laquelle un processeur exécute des instructions de manière incorrecte, de la première erreur jusqu'à sa détection et la restauration. Par extension, on nomme *instruction transitoire* les instructions exécutées durant cette fenêtre temporelle. Ce phénomène, qui s'observe déjà dans n'importe quel pipeline à plusieurs étages, est d'autant plus accentué par des mécanismes comme l'exécution dans le désordre. Il est important de souligner qu'en cas de succession d'erreurs dans la même fenêtre

temporelle, le processeur reviendra ensuite uniquement à la première d'entre elles, les autres n'en étant que des conséquences.

Lors de la restauration, la théorie voudrait que le processeur soit remis exactement et entièrement dans le même état que si la spéculation avait été effectuée correctement. En pratique, ceci n'est vrai que du point de vue de l'architecture (les éléments spécifiés par l'ISA et visibles par le logiciel). Du point de vue de la microarchitecture, il reste parfois des traces de cette mauvaise spéculation. C'est par exemple le cas des accès mémoires effectués spéculativement vers les caches : les lignes ramenées sont généralement conservées. Ainsi, en générant une fenêtre d'exécution transitoire et en choisissant minutieusement les instructions transitoires, il devient possible pour un attaquant de faire fuir des informations et de les récupérer par le biais des traces.

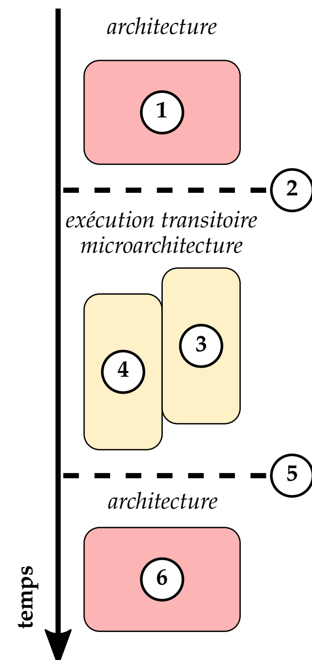


Figure 3.3. – Évolution d'une attaque par exécution transitoire. Durant la phase **1**, l'attaquant prépare l'environnement d'exécution en remplissant certaines ressources partagées. Plus tard, l'exécution transitoire est lancée (phase **2**) : le processeur est en train de spéculer. Durant cette période, l'objectif est donc d'effectuer l'accès aux données (phase **3**) et ensuite de les encoder dans un état de la microarchitecture (phase **4**). Ces deux étapes doivent être effectuées avant la phase **5** où le processeur corrige son erreur de spéculation en annulant les effets visibles au niveau de l'architecture. Enfin, dans l'étape **6**, l'attaquant cherche à récupérer les valeurs depuis l'état de la microarchitecture.

Différentes phases des attaques Plusieurs années se sont écoulées depuis la découverte de ce type d'attaques. Leur compréhension ainsi que les possibilités qu'elles offrent sont maintenant mieux évaluées. La Figure 3.3 illustre leur découpage en 6 phases [56, 57].

Les trois types d'attaques détaillés ci-après se distinguent principalement sur la manière dont est créée la fenêtre transitoire et la récupération des données (phases 1 à 3). Toutes utilisent ensuite un canal caché pour l'exfiltration des données. Ce canal peut changer, ce qui constitue une variante de l'attaque de base.

Finalement, les attaques par exécution transitoire peuvent être vues comme un modèle d'attaquant alternatif. L'exécution d'un programme est détournée pour forcer l'utilisation d'un canal caché afin de transmettre des informations à un attaquant.

Spectre

Spectre est le premier type d'attaque transitoire exploitant la microarchitecture. C'est aussi celui considéré comme le plus critique pour la sécurité des systèmes. Contrairement aux deux autres types présentés plus tard, celui-ci touche les processeurs de la plupart des fabricants (notamment Intel, AMD, Arm [54, 58-60]).

Principe général

Les attaques de type Spectre exploitent les mécanismes de spéculation. Au cours de l'exécution, des instructions de saut ou de branchement viennent rediriger l'exécution. Pour éviter les pénalités sur les performances, nous avons vu que différents mécanismes (notamment BHT, PHT, BTB et RSB) servent à les anticiper. Cependant, il arrive que leurs prédictions soient erronées.

C'est dans ces cas-là que se crée la fenêtre d'exécution transitoire exploitée par les attaques Spectre. La première variante est celle appelée **Spectre-PHT**. Comme son nom l'indique, elle exploite les erreurs de prédictions des branchements conditionnels par le PHT.

Un exemple est donné avec le Code 3.2 issue de l'article de Spectre [56]. Une vérification (le `if`) assure que normalement aucun accès au tableau `tab1` n'est hors limite. La valeur de x est pour cela comparée. L'objectif de l'attaque va être d'outrepasser cette vérification temporairement durant une fenêtre d'exécution transitoire. Pour cela, l'attaquant va dans un

Code 3.2: Exemple d'une attaque Spectre-PHT.

```
if (x < tab1_size) {
  y = tab2[tab1[x] * 4096];
}

```

premier temps entraîner le prédicteur (le PHT) à considérer la condition comme vraie. C'est la phase 1. Plus tard, il va réexécuter ce même code, mais avec une valeur de `x` qui ne respecte plus la condition. Cependant, le prédicteur va spéculer que la condition est vraie : une fenêtre d'exécution transitoire s'ouvre (phase 2). Pendant cette période, le processeur va accéder à une valeur en dehors du tableau `tab1[x]` (phase 3). Pour la retrouver ultérieurement, celle-ci est encodée dans l'état du cache : une ligne d'un second tableau est ramenée dépendamment de cette valeur (phase 4). Après cela, le processeur corrige son erreur et restaure l'état du processeur, mais pas du cache (phase 5). Finalement, l'application de l'attaque Flush+Reload ou de sa variante Evict+Reload permet de retrouver la donnée (phase 6).

Cette version de Spectre-PHT se déroule entièrement dans le processus de l'attaquant. Il entraîne le prédicteur de branchement puis abuse de son fonctionnement pour exfiltrer une donnée. Le problème ici est donc un pur problème de respect des frontières d'isolation : le processeur donne accès à des plages mémoires spéculativement. Canella et al. [56] ont utilisé le même principe entre un processus victime et un processus attaquant. Ils ont exploité les différentes associations d'adresses virtuelles/physiques entre plusieurs exécutions. L'attaquant utilise les différents mécanismes et laisse volontairement des traces. Plus tard, l'exécution de la victime se voit influencée par celles-ci. Indirectement, il est alors possible pour l'attaquant de forcer la microarchitecture à exécuter certaines instructions. Notamment, il peut rediriger l'exécution pour transmettre des informations *via* une ressource partagée, souvent une mémoire cache. La victime utilise donc un canal caché à son insu. Schwarz et al. [47] ont montré la possibilité d'exfiltrer différemment des données à travers un réseau (ici avec l'unité des vecteurs) . Ces possibilités étendent les moyens de l'attaquant en intégrant le fonctionnement des ressources partagées. Cela s'applique de la même manière pour les autres variantes de Spectre.

Variantes

D'autres variantes utilisent les erreurs de spéculation pour créer une fenêtre d'exécution transitoire. Elles s'appuient cependant sur des mécanismes différents de la microarchitecture.

Spectre-BTB La seconde variante de Spectre [54], et révélée en même temps que Spectre-PHT, est celle exploitant le BTB. Pour rappel, ce composant est responsable de la prédiction des sauts indirects : il enregistre l'adresse de destination associée à celle de l'instruction de saut. Les valeurs stockées sont des adresses virtuelles.

L'idée de Spectre-BTB est d'entraîner le BTB afin de pouvoir rediriger le flot de contrôle. Pour cela, l'attaquant l'entraîne à prédire une adresse de destination pour un saut à une adresse précise. Quand le processeur exécutera ultérieurement un saut placé à la même adresse virtuelle, il prédira également un saut vers la destination entraînée. Si l'attaquant a bien choisi ce morceau de code spéculativement exécuté (aussi appelé *gadget*), alors une exfiltration des données est possible.

Spectre-RSB Suite à la découverte de Spectre-BHT et Spectre-BTB, de nouveaux travaux [61, 62] ont prouvé l'existence d'une troisième variante. Appelée Spectre-RSB, elle exploite le mécanisme de prédiction

des adresses de retour : le RSB. Pour rappel, ce composant est responsable de la prédiction des retours et fonctionne comme une pile d'une taille fixe.

Sous certaines conditions, il est possible que le RSB devienne incohérent et prédise mal l'adresse de retour d'une fonction. C'est notamment le cas lors de changement de contexte entre le noyau, une enclave ou un utilisateur. Un attaquant peut alors en tirer profit pour rediriger le flot de contrôle.

Notes : Ces travaux ont également montré que certaines implémentations Intel utilisent le BTB à la place du RSB dans certains cas. Il est alors possible de mettre en place des attaques semblables à Spectre-BTB en utilisant des instructions de retour.

Spectre-STL Comme vu précédemment, la spéculation ne concerne pas seulement le flot de contrôle, mais aussi les accès mémoires. Ainsi, la dernière variante de Spectre appelée Spectre-STL [60, 63] exploite le mécanisme *store-to-load (STL) forwarding*.

Contrairement au cas précédent [48], l'idée est ici d'exploiter directement la prédiction d'un `load` suivant un `store`. Une des spéculations possibles est celle de considérer un `load` dont l'adresse est connue comme non-dépendant d'un `store` dont l'adresse est inconnue. La valeur est alors directement récupérée en mémoire. En considérant que la valeur est utilisée comme pointeur, il est ainsi possible d'exécuter spéculativement un gadget placé à l'ancienne valeur (celle en mémoire).

Notes : Des variantes de Spectre-STL, liées à de nouvelles spéculations sur les accès mémoires, ont également été présentées par AMD [64]

Meltdown

Meltdown est le second type d'attaque transitoire exploitant la microarchitecture. Au départ classé comme une variante de Spectre, il est finalement considéré comme un type distinct à part entière. On notera que les processeurs Intel sont davantage concernés par ces attaques.

Principe général

Lors de l'exécution d'un programme, il arrive que certaines opérations mènent à la levée d'une exception. C'est le cas quand une instruction illégale est détectée. Plusieurs facteurs existent et varient selon les architectures :

- une erreur de page, car une page mémoire n'est pas disponible,
- une utilisation d'une unité d'exécution indisponible,
- une violation des protections d'un registre spécial,
- *etc.*

Une fois une exception levée, le processeur est responsable de rediriger l'exécution vers une adresse spécifiée. Cette rupture du flot d'exécution, selon sa gestion, peut mener à de mauvaises exécutions de la microarchitecture. Une fenêtre d'exécution transitoire est ainsi créée et exploitée par les attaques de type Meltdown.

La première variante de Meltdown appelée **Meltdown-US** (User/-Supervisor) est présentée sur le Code 3.3. Son objectif est de lire des données du noyau qui sont présentes dans le processus de l'attaquant. Avant l'application de certaines contremesures comme KAISER [65], c'était le cas d'une grande partie des secrets du noyau. Le fonctionnement de cette attaque est direct. Un accès mémoire est effectué vers la donnée visée (et normalement protégée) ce qui lève une exception : la fenêtre d'exécution transitoire débute. Avant que l'état ne soit restauré, la donnée est encodée dans l'état du cache. Comme pour Spectre, il est alors possible de la retrouver en utilisant les attaques sur les caches.

Code 3.3: Exemple d'une attaque Meltdown-US.

```
1 p_addr = KERNEL_ADDR;
2
3 tab[(*p_addr) * 4096];
4
```

Variantes

Comme dit précédemment, il existe plusieurs manières au sein des processeurs de lever des exceptions. Cela mène à différentes variantes de Meltdown [56], qui visent également à récupérer des données bien différentes.

Meltdown-P Surnommée Foreshadow [66] dans la littérature, Van Bulck et al. ont démontré la possibilité d'utiliser des principes similaires à Meltdown-US pour viser Intel SGX. L'idée est ici de permettre la fuite des données d'une enclave disponible dans le cache de données L1. Cette variante a finalement était étendue [67] pour passer outre l'isolation du système d'exploitation ou de l'hyperviseur.

Meltdown-NM Tous les processus exécutés sur un processeur n'utilisent pas une FPU. Ainsi, quand un changement de contexte est opéré, le système d'exploitation peut décider de bloquer la FPU pour le nouveau contexte. Ceci permet notamment de réduire le coût du changement en n'ayant pas à sauvegarder les différents registres. Dans ce cas, la FPU est marquée comme indisponible. Une exception sera levée si le nouveau processus cherche à utiliser l'unité.

Il est possible d'exploiter cette exception Stecklina et Prescher [68]. Dans un premier temps, la victime charge des données dans des registres de la FPU. Le système d'exploitation effectue plus tard un changement de contexte vers l'attaquant. Celui-ci essaye alors d'accéder à une valeur d'un des registres : l'exception correspondante est levée. Durant la fenêtre transitoire, l'attaquant devient alors libre d'utiliser cette valeur.

Meltdown-GP Une autre variante dévoilée par Arm [60] et Intel [69] concerne les registres privilégiés. Appelée Meltdown-GP, cette variante consiste à accéder à ces registres et utiliser spéculativement les valeurs contenues. La finalité est donc de pouvoir retrouver des valeurs de configuration du système normalement secrètes.

Autres Chaque source d'exception est susceptible d'être exploitée par une attaque de type Meltdown. Il existe donc d'autres variantes [56]. Meltdown-RW [70] (originellement considérée comme une variante de Spectre 1.2) permet d'écrire spéculativement des plages d'adresses normalement en lecture seulement. Meltdown-PK [56] permet de passer outre un système de protection mémoire disponible sur certains processeurs Intel.

Échantillonnage des données microarchitecturales

Depuis les publications de Spectre et Meltdown, de nombreux travaux se sont concentrés sur les attaques par exécution transitoire. Cela a notamment mené à la découverte d'une nouvelle catégorie, affectant essentiellement les processeurs Intel : les attaques par échantillonnage des données microarchitecturales (ou *Microarchitectural Data Sampling (MDS)*). Ces attaques, dont la trame générale reste la même que Spectre et Meltdown (exploitation de l'exécution transitoire + exfiltration des données par un canal caché), soulignent d'autant plus la problématique posée par les ressources partagées.

Principe général

Il existe au sein de la microarchitecture de nombreux buffers, des structures composées de registres. Ils mémorisent des informations (état persistant) lorsqu'ils sont utilisés. Ces traces restent donc présentes tant qu'elles ne sont pas écrasées par de nouvelles valeurs. Sous certaines conditions, l'objectif des attaques par échantillonnage est de permettre à un attaquant de récupérer les données laissées par la victime. Ainsi, en prélevant des traces régulièrement, il devient possible d'avoir une connaissance claire des opérations récemment effectuées ou des données manipulées.

La Figure 3.4 décrit le fonctionnement classique d'une attaque par échantillonnage. On considère qu'un processus victime dispose d'un secret à partir duquel il souhaite effectuer des opérations. Au cours de l'une d'entre elles, la valeur secrète est utilisée et stockée dans un buffer cible (phase 1). Une fois son exécution terminée, le processus victime laisse sa place au processus attaquant : la valeur est toujours dans le buffer cible. Cet attaquant va volontairement effectuer une opération qui est susceptible d'utiliser ce même buffer cible. Le processeur spécule que la valeur à utiliser est celle déjà présente dans le buffer cible. Une fenêtre d'exécution transitoire est ouverte durant laquelle l'attaquant va encoder temporairement la donnée dans un autre buffer (phase 2). Le processeur se rend compte ultérieurement de son erreur et la corrige. Finalement, l'attaquant décode la valeur secrète (phase 3).

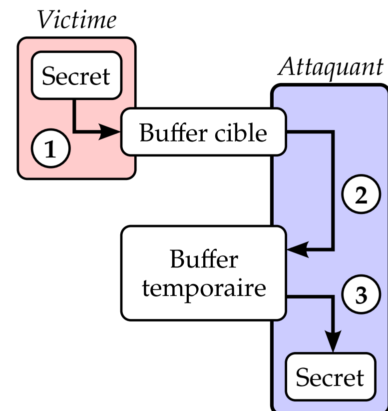


Figure 3.4. – Déroulement d'une attaque par échantillonnage microarchitectural.

Certains des buffers particulièrement ciblés sont les *line fill buffers* qui servent aux processeurs à suivre et gérer les *misses* de cache. Ils servent également à effectuer des optimisations, comme par exemple de la spéculation sur les adresses physiques/virtuelles des `load/store`. Ainsi, toutes les données échangées entre la mémoire et le processeur sont susceptibles de se trouver dans ce buffer. Des travaux [71, 72] officialisés par Intel [73] ont montré la possibilité d'en extraire des informations en effectuant des `load` de manière transitoire. Ils se placent à la limite entre Spectre et Meltdown, en exploitant l'utilisation de fautes (comme Meltdown) et la spéculation (comme Spectre). Ces fuites sont aussi bien exploitables avec ou sans SMT.

Variantes

Là encore, plusieurs variantes sur les buffers exploités [71, 73, 74], les données récupérées [75-78] ou les méthodes [72, 79] existent. Par exemple, les *store buffers*, utilisés pour la gestion des `store` et assurer leur ordre en mémoire, ont également été prouvés comme sensibles à ce genre d'attaques [73, 74, 80]. L'étendue du problème est d'autant plus importante si l'on considère que, jusqu'à présent, les modifications opérées par Intel ne semblent pas suffisantes. Comme détaillé dans le chapitre 4, un changement de l'instruction `verw` a été effectué afin de modifier le contenu des buffers exploités avant/après l'exécution d'un processus sensible. Cependant, de nouvelles attaques [75, 76] ont montré que cela n'était pas suffisant, en transférant des données depuis le cache L1D qui n'est lui-même en rien modifié.

Notes : Il est intéressant de noter que les attaques par échantillonnage exploitent la spéculation sur les adresses mémoires. Elles se rapprochent donc de Spectre-STL alors que les autres variantes utilisent la spéculation sur le flot de contrôle.

Toutes ces attaques se concentrent jusque-là au niveau d'un seul cœur physique entre un ou plusieurs thread(s). Cependant, cette problématique se pose également au niveau des systèmes multicœurs [81-83]. Ainsi, il

existe des buffers partagés entre les cœurs et notamment ceux appelés *staging buffers*. Là encore, un attaquant sur un cœur physique peut-être capable de retrouver des données laissées par une victime depuis un autre cœur. C'est notamment le cas des valeurs issues du générateur de nombres aléatoires (RNG) matériel qui passent par ces buffers.

Injection

Les attaques par échantillonnage présentées jusque-là permettent de récupérer des données depuis un processus victime. De la même manière que les autres ressources partagées peuvent être manipulées pour influencer ou espionner une victime, l'échantillonnage peut donc être renversé.

C'est ce principe qui est exploité par Van Bulck et al. [84]. Ainsi, dans ce scénario, l'attaquant injecte une valeur dans un des buffers microarchitecturaux qui sera ensuite spéculativement utilisée par la victime. Cela peut mener à rediriger temporairement le flot d'exécution vers un gadget exploitable. Intel a également reconnu ce type de failles [85], déclarant cependant qu'il est difficilement exploitable en pratique à cause des nombreuses conditions à remplir.

3.6. Attaques par canal contrôlé

Un scénario commun pour protéger une application est de considérer tous les éléments extérieurs comme non sécurisés. C'est ce modèle qui est notamment appliqué par les enclaves Intel SGX. Ainsi, même le système d'exploitation est considéré comme non sûr et n'a pas accès aux données d'applications moins privilégiées. Paradoxalement, il conserve généralement un rôle de gestion de certains mécanismes, même pour les enclaves. Cette particularité peut être exploitée comme nouveau levier.

Principes Le système d'exploitation est responsable de la gestion du système. C'est notamment vers lui qu'est renvoyé le flot d'exécution en cas d'exception ou d'interruption. Cette organisation reste la même dans le cas d'enclaves même si des mécanismes supplémentaires sont utilisés pour éviter des fuites de données (*e.g.* suppression de valeurs ou chiffrement). Dans le cas d'un attaquant, on appelle un "canal contrôlé" un mécanisme influant sur le fonctionnement de l'enclave et qui est sous le contrôle du système d'exploitation. Dans le cas où l'attaquant a pris le contrôle de ce dernier, il est donc capable d'interagir avec une potentielle enclave victime.

Faute de page L'une des exceptions intervenant le plus souvent lors de l'exécution de programme est la faute de page. Dans les systèmes avec mémoire virtuelle, elle indique que le processus en cours d'exécution ne trouve pas de traduction valide pour une adresse virtuelle. Le système d'exploitation est alors responsable de gérer cette situation.

Xu, Cui et Peinado [86] ont retrouvé des informations d'une victime en utilisant ce procédé. Pour cela, il est nécessaire que la victime effectue des accès à des pages (qui génèreront des fautes) dépendamment d'une valeur secrète. L'inconvénient majeur de cette approche est sa granularité : les informations sont obtenues à l'échelle des pages. Pour remédier à cela, des séquences spécifiques d'accès menant à des fautes de pages sont

identifiées dans le code. Il est alors possible de précisément savoir où en est l'exécution de la victime. Ces accès peuvent être aussi bien pour des données et du code. Ensuite, à partir des traces récupérées, il devient possible de retracer le flot d'exécution ou même de retrouver la valeur secrète. Il a été montré [87] que ce type d'attaques fonctionne également dans le cas de VM.

Van Bulck et al. [41] ont montré la possibilité de se passer des fautes de page pour récupérer des clés ECDSA. Dans le cas de la mémoire virtuelle, pour aider le support logiciel de la gestion de la mémoire, des informations sont associées aux tables de pages. Pour les processeurs Intel x86, deux bits indiquent notamment si une page a été utilisée (bit A pour *accesssed*) ou modifiée (bit D pour *dirty*). En récupérant ces valeurs après l'exécution d'une enclave, il est possible de déterminer des informations sur l'utilisation d'une page. Pour obtenir une granularité plus fine d'informations, des stratégies sont également proposées pour vérifier régulièrement les valeurs de ces bits. Elles utilisent alors les interruptions pour interrompre régulièrement l'exécution d'une enclave, ou un autre processus exécuté simultanément et effectuant des vérifications régulières.

Interruption Le système d'exploitation est également responsable de la gestion des interruptions, ces évènements déclenchés par le matériel. Ainsi, Van Bulck, Piessens et Strackx [88] ont exploité les mécanismes d'interruption. Toutes les instructions ne mettent pas le même nombre de cycles à être exécutées. Ainsi, selon l'instruction en cours d'exécution au moment de la levée d'une interruption, la latence de celle-ci peut différer. Avec cette attaque appelée Nemesis, l'objectif est donc de provoquer des interruptions régulièrement à l'aide d'un timer[1] pour détecter des variations. Cette attaque est efficace contre des applications où les instructions exécutées sont dépendantes de la valeur d'un secret. Il est de plus intéressant de noter que de par la généralisation des interruptions, les processeurs très simples comme complexes sont potentiellement concernés. Au-delà de la latence, Van Bulck, Piessens et Strackx mentionnent également la possibilité de détecter et exploiter le nombre d'instructions interruptibles, qui là encore peut varier selon le flot d'exécution.

1: On appelle un *timer* un registre ou périphérique qui déclenche une interruption au bout d'un certain nombre de cycles.

3.7. Synthèse et conclusion

Nous avons vu dans ce chapitre quatre grandes classes d'attaques logicielles exploitant la microarchitecture. La première est celle des attaques utilisant les variations temporelles lors de l'exécution de programmes. Au sein de la microarchitecture, des informations sont accumulées pour accélérer les futures opérations. Il est possible d'exploiter ces variations pour retrouver des informations sur les exécutions passées. La deuxième est celle des attaques utilisant la contention dynamique de ressources. Sous certaines conditions, quand plusieurs programmes s'exécutent simultanément, il devient possible de déduire des informations à partir des ressources bloquées. La troisième est celle des attaques par exécution transitoire. Ces dernières tirent profit des nombreuses optimisations et de la complexité des microarchitectures modernes pour exfiltrer des données. Enfin, celle des attaques exploitant les canaux contrôlés, ces mécanismes sous contrôle du système d'exploitation et influant une potentielle victime. Finalement, comme illustré sur la Figure 3.5, la pro-

blématique de sécurité posée par ces attaques peut être résumée en trois notions.

Les **ressources partagées** sont exploitées par des attaques des différentes classes. Dans leur fonctionnement actuel, elles permettent des interactions entre des programmes normalement indépendants (*e.g.* un processus attaquant et un victime). Pour les variations temporelles, ce sont les ressources partagées qui permettent l'échange des informations stockées générant les variations temporelles. Pour la contention dynamique, elle n'est détectable que sur les ressources qui sont partagées. Enfin, dans le cas de l'exécution transitoire, les ressources partagées sont exploitées de plusieurs manières. Cela peut aussi bien être pour entraîner les mécanismes, avoir accès à des données ou les exfiltrer. Les ressources partagées posent donc un problème d'isolation entre leurs différents utilisateurs.

Les attaques par exécution transitoire soulignent également un problème d'isolation entre les exécutions, et notamment le **respect des frontières d'isolation**. Les microarchitectures modernes abusent de la spéculation même au-delà de ces frontières. Cela mène inévitablement à des fuites normalement comblées par l'architecture. C'est aussi le respect des frontières d'isolation qui est remis en cause avec les attaques par canaux contrôlés. Le système d'exploitation, pourtant non sécurisé, reste responsable de la gestion de certains mécanismes. Il a donc accès à des informations normalement confidentielles et propres à un processus.

Enfin, même dans le cas d'un processus isolé, il reste tout de même possible d'observer un temps d'**exécution variable**. Pour être exécuté, chaque programme a besoin d'un certain temps perceptible. En créant des dépendances entre les données d'un programme et son temps d'exécution, la microarchitecture donne alors un moyen indirect de retrouver ces données. Un attaquant uniquement capable de mesurer ce temps peut donc simplement en tirer profit. C'est également en partie ce point qui est exploité par certaines des attaques par canaux contrôlés (*e.g.* la latence d'une interruption variant selon l'instruction exécutée).

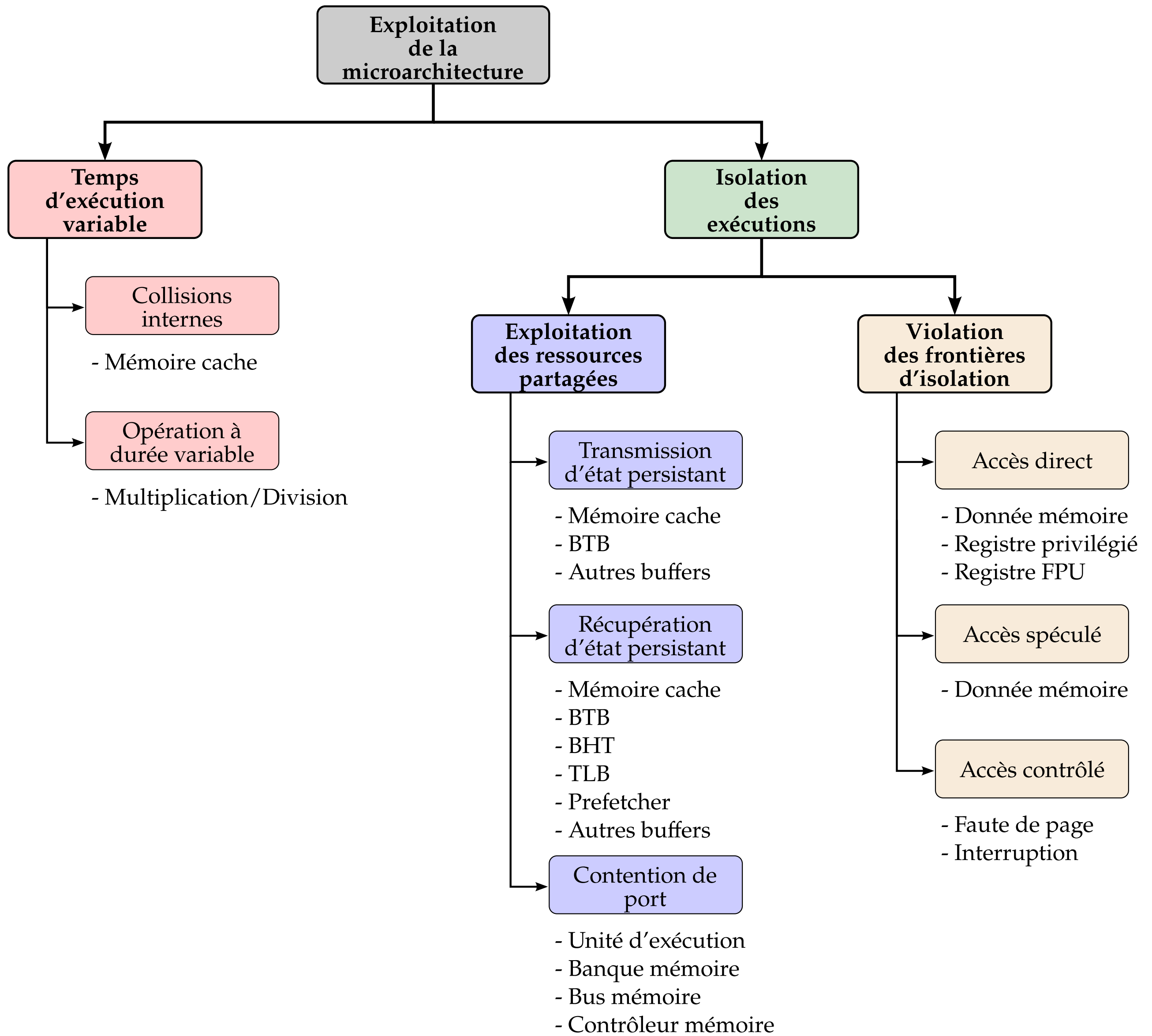


FIGURE 3.5. – **Classification des attaques logicielles exploitant la microarchitecture.** Chacune de ces attaques s'appuie sur au moins une des notions suivantes : le temps d'exécution variable d'un programme, l'exploitation des ressources partagées ou la violation des frontières d'isolation. Différentes utilisations de ces notions sont ensuite possibles. Par exemple, les ressources partagées peuvent être utilisées pour transmettre ou récupérer des états persistants, mais aussi créer de la contention de port.

Solutions existantes contre l'exploitation logicielle. 4.

Résumé du chapitre : Dans ce chapitre, nous nous intéressons aux approches de luttes contre les fuites temporelles. On en distingue ici trois grands types qui diffèrent de par leurs stratégies. Le premier concerne la gestion et l'isolation des ressources partagées. Étant la source de nombreuses fuites, ces mécanismes donnent lieu à un grand nombre de modifications. Le second type concerne la gestion de la spéculation dans les processeurs modernes. Enfin, le troisième type s'attaque directement à la notion de temps, centrale dans les attaques temporelles. À partir de cette vision globale des solutions existantes, nous effectuons un bilan afin de définir la meilleure stratégie pour la conception de processeurs sécurisés.

4.1 Solutions pour la gestion des ressources partagées . . 49
Suppression des mécanismes 49
Effacement des traces 50
Partitionnement des ressources 52
4.2 Solutions pour la gestion de la spéculation 55
Modifications du logiciel . . 55
Renforcement du matériel . 57
4.3 Solutions de gestion du temps et des évènements . . 59
Modification des évènements microarchitecturaux . 60
Indéterminisme du fonctionnement 60
Altération des mesures de temps 61
Exécution en temps constant 62
4.4 Stratégie globale 63
Prise en compte ciblée des failles. 63
Rôles des couches d'abstraction 64
4.5 Conclusion et suite 67

4.1. Solutions pour la gestion des ressources partagées

Nous avons vu dans le chapitre 3 que les ressources partagées telles qu'implémentées actuellement représentent un problème de sécurité majeur. Cette problématique est connue depuis plusieurs années. Ainsi, une approche possible est de modifier leur utilisation et/ou fonctionnement pour assurer des garanties d'isolation.

Suppression des mécanismes

La première méthode, la plus drastique, est la pure et simple suppression des mécanismes concernés. La plupart sont non essentiels, car non nécessaires à la conception d'un processeur fonctionnel. Cependant, nous avons vu dans les chapitres précédents qu'ils contribuent en grande partie à l'amélioration des performances des processeurs. Une suppression ne serait donc pas sans impact.

Mémoires caches La mémoire principale étant bien plus lente que le processeur, la suppression définitive des caches aurait un impact considérable [12]. Durant l'exécution, le processeur serait contraint d'attendre les réponses de la mémoire. De ce point de vue, les mémoires caches sont donc considérées comme essentielles. Des stratégies de suppression partielle ont cependant été suggérées. Page [89] évoque la possibilité de remplir les caches avec toutes les données potentiellement utiles avant leur utilisation. Cela mène néanmoins, du point de vue des performances, à accumuler les *misses*. Osvik, Shamir et Tromer [12] mentionnent la possibilité que les programmes sécurisés (*e.g.* des opérations de chiffrement) n'utilisent les caches qu'en cas de cache *hit*. Dans les autres cas, la mémoire principale serait consultée sans modification des caches.

Le même type de constat s'applique finalement aux prédicteurs de branchement [90] ou des prefetchers [42]. Leur impact sur l'exécution est trop important pour que leur suppression soit réaliste.

Multithreading simultané Certains mécanismes semblent cependant moins importants pour les performances. C'est le cas par exemple du

SMT. Une approche couramment mentionnée [12, 48, 91] contre les attaques l'exploitant est tout simplement de le désactiver. Pour Microsoft Azure [92], le choix effectué est d'utiliser des processeurs où seul le dernier niveau de cache est partagé. Des études [93] ont tenté d'évaluer l'impact sur des processeurs Intel de la désactivation du SMT. Le résultat est clair : une perte jusqu'à 20% des performances.

Une approche moins radicale est de limiter l'exécution d'utilisateurs différents sur un même cœur physique. Le système d'exploitation [91] (ou un autre moniteur dédié) doit alors assurer que des programmes qui s'exécutent ensemble se font mutuellement confiance. C'est par exemple le cas des différents programmes d'un même utilisateur ou de programmes sans contrainte de sécurité.

Adaptation de l'ISA Pour certaines attaques, l'espion et la victime doivent être exécutés simultanément. À défaut de multithreading simultané, cela passe généralement par des exécutions entrecoupées des deux processus. Le système d'exploitation fait généralement appel à des interruptions prévues par l'ISA pour gérer cela. Ainsi, une solution également proposée [12] est de les désactiver durant l'exécution d'un processus sensible. Dans le cas de Meltdown-NM, une alternative est de ne plus lever d'exception au niveau de la FPU et de toujours sauvegarder le contenu des registres (au détriment de la vitesse de changement de contexte).

Schwarzl et al. [94] propose une approche plus fondamentale contre les attaques de type Spectre. Au lieu de modifier ou supprimer les prédicteurs de branchements, ils proposent directement de ne plus utiliser de branchements conditionnels ou de sauts indirects. Cela passe par une modification du compilateur qui, au moment de générer le programme, doit être capable de ne pas utiliser ces instructions pour les programmes sensibles. Ceux-ci sont donc linéarisés : leur exécution correspond à une suite d'instructions placées consécutivement en mémoire. Par exemple, le corps d'une boucle est copié autant de fois que nécessaire plutôt que de rejouer les mêmes instructions en utilisant des sauts. Si cette approche reste possible dans certains scénarios, l'impact sur les performances peut cependant être drastique (de l'ordre d'un facteur 1000).

Afin de laisser le choix aux applications exécutées, Intel et AMD [95, 96] ont intégré la possibilité de désactiver la prédiction sur les `store`. Cette fonctionnalité est rendue disponible sur certains modèles de processeurs après une mise à jour du microcode[1].

1: Un microcode est un mécanisme permettant de définir les opérations correspondant à une instruction dans une mémoire. Le contenu de cette mémoire peut lui être modifié même après fabrication du processeur. C'est donc un moyen intéressant pour les fabricants d'ajouter ou modifier le fonctionnement de certaines instructions avec de simples mises à jour.

Finalement, la suppression définitive de tous les mécanismes problématiques semble irréaliste dans la plupart des cas. C'est généralement une solution théorique suggérée, mais non mise en pratique. Elle est donc réservée pour des mécanismes plus marginaux comme la désactivation par défaut d'Intel Transactional Synchronization Extensions (TSX) [97]. Ce mécanisme est notamment exploité par certaines attaques [79] d'échantillonnage microarchitectural. Ainsi, les autres approches cherchent à modifier l'utilisation ou les ressources elles-mêmes afin d'atteindre des compromis acceptables.

Effacement des traces

Dans le chapitre 2, nous avons vu qu'un grand nombre de mécanismes enregistre des informations durant les exécutions des différents

programmes pour optimiser les exécutions futures. Ces informations, qui correspondent à l'état persistant des ressources, sont utilisées par un grand nombre d'attaques pour récupérer des informations sur les exécutions passées. En réponse à cela, une stratégie largement suggérée et explorée [12, 42, 91, 98, 99] est l'effacement des traces. Cette opération est communément appelée *flush*. Ainsi, l'idée illustrée sur la Figure 4.1 est de marquer chaque changement d'utilisateur par une remise à un état par défaut du système. Chaque nouvel utilisateur n'a ainsi aucun moyen de retrouver des traces des précédentes opérations réalisées. La mise en place de cette stratégie implique donc de répondre à deux questions : quand et comment réaliser l'effacement des traces ?

En 2013, Godfrey et Zulkernine [100] proposent une modification de l'hyperviseur Xen. Celui-ci est utilisé pour la gestion de machines virtuelles sur des serveurs. Dans ce cas précis, le lancement du flush est alors géré par l'ordonnanceur responsable du lancement des VM. Deux versions de flush sont proposées. Une première plus efficace utilise directement les instructions x86. Une seconde plus lente, mais portable (indépendante du jeu d'instructions) cherche à remplacer les lignes de cache.

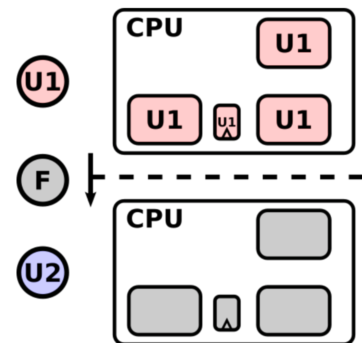


Figure 4.1. – Opération de flush. Dans un premier temps, un utilisateur U1 est en cours d'exécution. À la fin, avant de lancer un nouvel utilisateur U2, ses traces sont effacées. Quand U2 commence son exécution, il se retrouve dans un système avec des états persistant par défaut.

Le jeu d'instruction x86 fournit une instruction non privilégiée `clflush`. Celle-ci est utilisée pour effacer les lignes correspondant à une adresse spécifique. De son côté, l'ISA d'Arm fournit des instructions privilégiées pour vider certains éléments comme les caches, les TLB ou les tables de prédiction.

La solution d'écrasement des données est recommandée par AMD [101] pour certains buffers. Ainsi, ils suggèrent de remplacer les valeurs non sécurisées par des 0 dans chaque buffer avant d'exécuter un processus privilégié. Ils préconisent également d'effectuer une série de `call` afin de remplir le RSB. De la même manière, Intel [73] recommande d'écraser les valeurs des buffers dès qu'un programme sécurisé a terminé son exécution. L'instruction `verw` (et un registre `md_clear`) a été modifiée afin de réécrire les buffers exploités par les attaques par échantillonnage. Dans le cas des processeurs ne bénéficiant pas de cette modification, des suites d'instructions sont précisées pour atteindre le même objectif (au détriment des performances).

De manière générale, cette stratégie d'effacement est largement utilisée avant/après l'exécution de programmes sensibles. Elle est donc employée dès lors que le logiciel considère qu'un programme doit être protégé des autres programmes exécutés [43, 98, 99].

Les attaques présentées dans le chapitre 3 ont prouvé que n'importe quel état persistant est exploitable. Cette stratégie de nettoyage des ressources n'est donc efficace que si toutes les ressources peuvent être gérées de la sorte. Or, Ge et al. [27, 91] ont souligné que ce n'était pas le cas dans les processeurs modernes. Selon les implémentations, il existe des mécanismes matériels incontrôlables par le logiciel. Certains états persistants sont toujours conservés. Le logiciel n'a donc que des moyens limités pour effacer ses traces au niveau de la microarchitecture.

En réponse à cela, Bourgeat et al. [99] introduisent une nouvelle instruction `purge`. Cette instruction agit comme une barrière temporelle en effaçant les traces laissées par les exécutions précédentes. Le matériel est responsable de choisir les ressources impactées. Wistoff et al. [102, 103] ont montré l'efficacité d'une instruction RISC-V similaire appelée `fence.t`. Dans ce cas, les mécanismes impactés sont directement spécifiés

par le biais d'un paramètre. Le déclenchement d'une opération de flush peut aussi être la conséquence d'opérations architecturales. Ferraiuolo et al. [43] associent l'exécution sur chaque cœur du système à une notion de compartiment temporel. Si le compartiment temporel spécifié est modifié par le logiciel, alors le matériel est responsable d'effectuer les effacements nécessaires.

Finalement, l'effacement des traces est une stratégie efficace. Il permet notamment à un programme de ne pas laisser d'informations susceptibles d'être retrouvées. De plus, sa mise en place est simple : il suffit d'invalider ou réécrire des données. Cependant, son impact sur les performances est non-négligeable. Après effacement des traces, le processeur se retrouve dans un état dit "froid" : il n'a plus aucune information pour accélérer l'exécution. Multiplier les opérations de flush peut donc pénaliser de manière importante l'exécution. C'est particulièrement le cas lorsque les exécutions de plusieurs utilisateurs s'entrecoupent mutuellement, et notamment lors de la gestion d'exceptions, où un processus appelle régulièrement un plus haut niveau de privilège.

Vougioukas et al. [104] proposent alors de conserver certaines informations des prédicteurs de branchement dans des buffers dédiés. Chacun de ces buffers est ensuite associé à un processus : seul celui-ci est apte à l'utiliser. Le même genre de stratégie est mentionné par Cronin et Yang [42] dans le cas des prefetchers. Ces dernières approches se distinguent en privilégiant les performances et la sécurité au surcoût matériel (de nouveaux buffers). Elles s'appuient également sur une autre stratégie appelée partitionnement.

Partitionnement des ressources

Nous avons vu que différents programmes peuvent s'exécuter simultanément notamment grâce à des mécanismes comme le multithreading ou le multicœur. Cela mène à de nombreuses attaques détaillées dans le chapitre 3 [50, 51] contre lesquelles l'effacement des traces n'est pas suffisant. Des solutions complémentaires ont donc été proposées [91] afin de partitionner les ressources, c'est-à-dire les compartimenter entre les différents utilisateurs.

Partitionnement spatial

La première forme de partitionnement, et la plus intuitive, est le partitionnement spatial. Comme illustré sur la Figure 4.2, il consiste en un cloisonnement physique des ressources partagées entre les différents utilisateurs. Chacun n'a donc accès qu'à certaines ressources ou parties de ressources. Tout comme l'effacement des traces, le partitionnement spatial [12, 42] est étudié depuis de nombreuses années. Les implémentations proposées concernent généralement les mémoires caches, pour restreindre leur utilisation par chaque thread ou cœur.

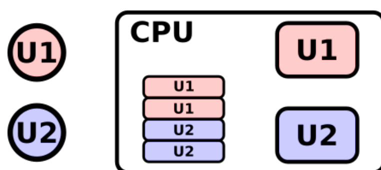


Figure 4.2. – **Partitionnement spatial des ressources.** Chaque ressource ou partie de ressource est assignée à un utilisateur U1 ou U2. Seul l'utilisateur correspondant est alors capable d'utiliser cette ressource et d'accéder aux informations qu'elle contient.

Partitionnement en logiciel Certaines solutions implémentent un partitionnement basé sur du logiciel en détournant le fonctionnement de la microarchitecture. Kim, Peinado et Mainar-Ruiz [105] proposent par exemple un mécanisme pour le LLC. Un hyperviseur attribue à chaque VM des pages privées. Durant l'exécution, il s'assure que leurs lignes ne seront jamais évincées du LLC. Pour cela, il exploite le fonctionnement déterministe de la politique de remplacement pseudo-LRU.

La solution proposée par Costan, Lebedev et Devadas [98] implique une légère modification du matériel. Généralement, chaque processus n'utilise que certaines régions mémoires continues. L'idée est d'utiliser les bits d'adresses correspondant à ces zones mémoires (en effectuant un décalage des bits d'adresse) pour effectuer le placement dans les caches. Ainsi, chaque processus voit ses lignes toujours placées au même endroit, ce qui correspond à une forme de partitionnement.

Effectuer un partitionnement sans mécanisme prévu à cet effet est rapidement limité. Les solutions proposées s'appuient donc généralement sur des modifications de l'ISA pour offrir un meilleur contrôle au logiciel.

Partitionnement en utilisant l'ISA En modifiant l'ISA, il est possible d'ajouter des instructions ou registres pour contrôler le partitionnement depuis le logiciel. Les mécanismes nécessaires à ce partitionnement sont eux assurés par le matériel.

Page [106] permet le partitionnement des caches en les rendant directement visibles au niveau de l'ISA. Pour cela, de nouveaux registres sont ajoutés permettant de créer des partitions rassemblant des emplacements du cache. Ces nouveaux registres contiennent un identifiant (choisi par le logiciel) pour chaque partition. Des instructions `addpar` et `delpar` permettent de créer et effacer ces partitions. Pendant l'exécution d'un programme, chaque accès mémoire a lui aussi un identifiant associé : il ne peut alors utiliser que les lignes des partitions ayant le même identifiant. Une instruction `invar` est également utilisée pour lancer l'effacement de la partition avec le même identifiant.

Wang et Lee [107] proposent PLCache (Partition-Locked Cache), un mécanisme de partitionnement permettant de bloquer certaines lignes. Lors d'un accès, la ligne est associée au processus qui l'a manipulé en utilisant un identifiant. De nouvelles instructions mémoires permettent de marquer la ligne comme "bloquée". Seul le même processus est alors apte à débloquer cette ligne pour la remplacer. Une autre granularité pour bloquer une ligne est proposée en utilisant les entrées de pages. Un nouveau bit pour chaque entrée indique si les accès de cette page mènent à des lignes bloquées ou non. Kong et al. [108] renforcent les garanties offertes en ajoutant des mécanismes de pré-chargement des données. Le temps d'exécution ne dépend ainsi plus des données sensibles utilisées (toutes sont tout le temps pré-chargées) et les accès ne sont plus détectables. Une fois dans les caches, les données sont bloquées durant toute l'exécution du processus propriétaire. En cas d'exception ou d'interruption, un registre dédié conserve l'information sur un pré-chargement déjà effectué ou non.

Ferraiuolo et al. [43] proposent une méthode de partitionnement du LLC dans les implémentations multicœurs. De nouveaux registres permettent d'attribuer un numéro de compartiment à chaque emplacement du cache par le biais d'un registre. Durant l'exécution, seuls les cœurs avec le même numéro de partitionnement ont le droit d'utiliser ces emplacements et les lignes qu'ils contiennent.

DAWG [70] est une nouvelle méthode de partitionnement permettant un meilleur contrôle par le logiciel. Comme précédemment, le principe est de taguer tous les accès mémoires et les lignes mémoires avec un identifiant de domaine. De nouveaux registres sont également définis pour indiquer précisément pour chaque emplacement :

- si la ligne peut être utilisée lors d'un *hit*,
- si la ligne peut être remplacée lors d'un *miss*.

Le logiciel est ainsi capable de gérer directement les ressources. Il peut ainsi permettre certains partages entre des domaines différents.

Dans MI6 [99], chaque cœur a un registre où chaque bit est associé à une région de la mémoire principale. Ce registre est accessible uniquement par le plus haut niveau de privilège. Un cœur ne peut ensuite accéder qu'aux régions qui lui sont indiquées. Chaque région ne pouvant aller que dans certaines parties du LLC, ces bits permettent donc un partitionnement de l'ensemble. En cas de tentative d'accès illégale, une exception est levée.

Finalement, le partitionnement est applicable à d'autres mécanismes que le contenu des caches (*e.g.* toute ressource avec un fonctionnement associatif [70]). Ren et al. [36] ont montré que le cache des micro-opérations est partagé statiquement entre les threads sur certains processeurs Intel. Les fuites entre les threads par le biais de ce mécanisme ne sont donc pas possibles. Il est également possible de partitionner les registres responsables de la gestion des *misses* [99]. Ainsi, chaque cœur n'a droit qu'à un nombre limité de *misses* en cours de gestion simultanément, ce qui empêche les cas de contention.

Partitionnement temporel

Une deuxième forme de partitionnement est le partitionnement temporel, efficace contre la contention dynamique [48]. Il permet de ne donner aux différents utilisateurs un accès qu'à tour de rôle comme présenté sur la Figure 4.3. On se retrouve alors dans le cas similaire d'un processeur partagé temporellement.

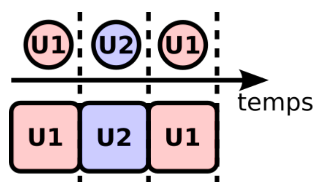


Figure 4.3. – Partitionnement temporel d'une ressource. Chacun à leur tour, les utilisateurs U1 et U2 ont le droit d'utiliser la ressource partagée. La durée de ce tour peut être d'un ou plusieurs cycles selon la latence de la ressource.

Wang, Ferraiuolo et Suh [53] utilisent le partitionnement temporel pour empêcher la contention au sein d'un contrôleur mémoire. Pour cela, les requêtes sont rangées dans des buffers dédiés à chacun des utilisateurs existants (partitionnement spatial localement). Un autre mécanisme est mis en place pour n'autoriser les requêtes que d'un seul utilisateur (considéré comme actif) à chaque instant. Des compteurs permettent de sélectionner ce dernier. La période dédiée à chaque utilisateur est statique et déterminée selon le temps de réponse des banques mémoires. Ce principe a été repris dans plusieurs implémentations [43, 99] pour le partage des contrôleurs de cache et des différents ports mémoire entre plusieurs cœurs.

Townley et Ponomarev [109] appliquent le partitionnement temporel aux ports d'exécution et aux unités fonctionnelles d'un processeur avec SMT. Celles-ci ne sont alors disponibles que certains cycles pour chaque thread. Le partitionnement temporel n'est appliqué que pour les ressources non dupliquées : les autres sont réparties spatialement entre les threads. Le partitionnement est effectif durant un nombre de cycles déterminé par le logiciel (*via* des registres dédiés). Plusieurs modes d'activation sont suggérés : lorsqu'une ressource est utilisée une première fois (matériel), lorsqu'une mesure de temps est détectée (matériel) ou quand une application le demande (logiciel).

Notes : L'activation en cas d'une première contention est plus intéressante pour les performances : elle ne s'active que quand c'est nécessaire. Cependant, d'un point de vue sécurité, elle ne permet de masquer des informations que pour un scénario d'observation par canal auxiliaire. Dans le cas d'un canal caché, cette différence peut théoriquement toujours être exploitée.

Finalement, les stratégies de partitionnement sont efficaces pour isoler des exécutions simultanées. Elles sont complémentaires à celles d'effacement des traces pour la gestion des ressources partagées. Ensemble, elles permettent de s'assurer que des exécutions simultanées ou différées

ne partagent pas de traces microarchitecturales. Elles reprennent pour cela le problème du partage d'informations dès les fondements. Ainsi, elles sont souvent associées [43, 91, 99, 110] pour la mise en place d'une isolation complète des exécutions.

Du point de vue des performances en revanche, elles réduisent l'impact des mécanismes d'optimisation. Les traces des précédentes exécutions ne sont pas conservées. Des mécanismes comme les prédicteurs de branchement perdent en efficacité. Le nombre de ressources utilisables est lui diminué par le biais du partitionnement, comme la taille des mémoires caches vue par chaque utilisateur. Ainsi, ces solutions ne doivent être appliquées que lorsqu'il est nécessaire de trouver un compromis entre sécurité et performance. Enfin, pour être efficaces, ces approches doivent être généralisées : aucune ressource ne doit être oubliée sous peine de laisser une faille exploitable.

4.2. Solutions pour la gestion de la spéculation

Les attaques par exécution transitoire exploitent la spéculation pour générer des traces microarchitecturales. Certaines d'entre elles utilisent les ressources partagées pour la transmission d'informations. Les solutions vues précédemment peuvent alors être appliquées. Cependant, il existe des cas où elles ne sont pas suffisantes. C'est par exemple le cas lorsque la spéculation est faite à travers les frontières d'isolation. Des solutions logicielles et matérielles ont donc été proposées afin de limiter la portée de la spéculation.

Modifications du logiciel

La spéculation au niveau matériel est normalement invisible pour le logiciel, car elle s'opère sans que celui-ci n'ait à intervenir. De nombreuses solutions [111] ont donc été proposées pour modifier le logiciel afin de gérer la spéculation.

Détournement de la microarchitecture Une première approche logicielle consiste à détourner le fonctionnement de la microarchitecture. L'objectif est de concevoir des programmes anticipant la manière dont ils seront exécutés. L'exemple le plus représentatif de ce genre de technique est retpoline [95, 101, 112, 113].

Retpoline, dont le fonctionnement est décrit sur le Code 4.1, cherche à anticiper et bloquer les prédictions sur les sauts indirects. Cette technique vise à forcer la microarchitecture à ne pas utiliser le BTB mais à passer par le RSB. Pour cela, chaque saut indirect sensible doit être remplacé par une séquence d'instructions. Cette méthode est efficace dans la majorité des cas pour se protéger d'attaques de type Spectre-BTB. Cependant, certains processeurs Intel [113], en cas de RSB vide, réutilisent le BTB pour la prédiction. Dans le cas de retpoline, cela revient à ajouter des instructions en vain. La solution pour cela est donc d'également ajouter par défaut une adresse de retour dans le RSB après chaque remise à zéro. Cette adresse de retour doit évidemment pointer vers un gadget non exploitable.

Une stratégie bien différente pour bloquer la spéculation est la création de dépendances de données artificielles. Cela peut prendre la forme d'accès mémoires dépendants du résultat de branchements condition-

Code 4.1: **Exemple d'implémentation de retpoline en x86.** Cette séquence vise à remplacer un `call *%r11`. Dans un premier temps, un appel de fonction *via* un saut direct est fait vers une adresse connue `ret_target` (l.2). En parallèle, le RSB sauvegarde l'adresse de retour `capture_spec`. La séquence `ret_target` est responsable d'effectuer le saut indirect. Pour cela, l'adresse de destination est déplacée dans le resgitre `rsb` utilisé pour les retours de fonction (l.7). Le saut en lui-même est ensuite effectué en utilisant `ret` (l.8). Dans le même temps, la microarchitecture spécule que le retour va vers l'adresse `capture_spec`. Une instruction de `pause` et une boucle infinie permettent alors de piéger les prédictions (l.5-6).

```
1 start_jmp:
2   call ret_target;
3 capture_spec:
4   pause;
5   jmp capture_spec;
6 ret_target:
7   mov %r11, (%rsp);
8   ret;
9
```

nels [101, 114]. Le processeur est alors contraint de sérialiser partiellement l'exécution. Une autre stratégie contre les attaques Spectre-PHT consiste à restreindre globalement l'index utilisable du tableau [101, 115]. Comme présenté sur le Code 4.2, un masque est appliqué sur l'index avant son utilisation (*e.g.* pour remettre les bits de poids forts à 0). Une variante possible est de créer une dépendance de données directement dans le calcul de l'adresse d'un `load` [116] . Ainsi, toute adresse calculée spéculativement est forcément erronée. La méthode proposée implique cependant d'utiliser des instructions conditionnelles dites "sans branchement". Ces instructions modifient un opérande uniquement si la condition est respectée. Enfin, l'empoisonnement de pointeurs est également utilisé dans le cas de WebKit [115]. Il modifie les pointeurs avant leur utilisation par des valeurs réversibles et déterminées à la compilation. Avant l'exploitation de chaque pointeur, un attaquant doit donc déterminer la valeur correspondante pour la prendre en compte.

Code 4.2: **Exemple d'implémentation d'un masque sur un index.** Le masque restreint ici l'index du tableau à ses 4 derniers bits (0xf = 1111 en binaire).

```
tab[addr & 0xf]

```

Une autre manière d'utiliser la microarchitecture est de réorganiser les processus. Généralement, les systèmes d'exploitation spécifient une partie des adresses du noyau directement dans les processus utilisateurs, notamment afin d'accélérer les changements de privilège. Une solution appelée KAISER [65] vise à ne plus mettre les adresses mémoires du noyau (ou le moins possible) dans le même espace mémoire que l'utilisateur. Au départ pensée pour d'autres menaces logicielles, cette stratégie s'avère efficace pour réduire la portée d'attaques comme Meltdown-US. Une approche mise en place dans Google Chrome [117] est de réaliser une isolation par site internet. Le navigateur internet utilise alors un processus différent pour chacun des sites. Les garanties de sécurité entre processus peuvent ainsi être appliquées : les attaques de type Spectre ne sont plus utilisables. En revanche, cette stratégie dépend pleinement des contremesures mises en place contre certaines variantes de Meltdown. Nous avons vu que ces dernières donnaient la possibilité à un attaquant de récupérer des informations du noyau depuis un programme non privilégié. Si elles sont utilisables, le problème n'est finalement que déplacé.

Utilisation des instructions dédiées disponibles Les architectures x86 et ARM contiennent des instructions pour la gestion globale de la spéculation. Une deuxième approche logicielle consiste donc à les utiliser.

L'instruction x86 `lfence` [95, 101] permet de sérialiser l'exécution. La microarchitecture assure que toutes les instructions précédant le `lfence` sont exécutées avant le `lfence` lui-même et les instructions suivantes. Ainsi, AMD et Intel [95, 101] préconisent l'utilisation de cette instruction contre les attaques de type Spectre. Dans le cas de la prédiction de branchement, l'idée est alors de protéger les sauts et branchements sensibles en les précédant d'un `lfence`. Ces opérations ne sont alors effectuées qu'avec les valeurs réelles. Dans le cas de la prédiction d'adresse, placer un `lfence` en amont peut bloquer la spéculation sur son adresse et le placer en aval peut bloquer la transmission de la donnée en cas de `load`. Cette stratégie a cependant des limites. Une utilisation abusive de `lfence` [84, 113] peut mener à des pertes de performances conséquentes, le processeur étant sans cesse en train d'attendre. Or, pour être efficace, chaque potentiel saut ou gadget [85] doit être protégé. L'implémentation en elle-même de cette instruction a également des limites. Des travaux [47,

118] ont montré que si l'exécution spéculative est stoppée, cela n'affecte pas la récupération d'instruction ou les modifications sur le TLB. Des attaques sont donc toujours envisageables en modifiant le canal caché utilisé.

En réponse, de nouvelles fonctionnalités ont été ajoutées dans le microcode de certains processeurs récents par AMD et Intel [95, 101] pour x86 :

- *IBPB - Indirect Branch Prediction Barrier* permet la mise en place d'une barrière de spéculation. Les exécutions avant la barrière n'influencent pas la prédiction des branchements indirects après la barrière.
- *IBRS - Indirect Branch Restricted Speculation* permet de restreindre la prédiction de branchement indirect. Les branchements indirects non-privilégies n'influencent plus les branchements indirects des enclaves ou des niveaux privilégiés.
- *STIBP - Single Thread Indirect Branch Predictors* permet de restreindre la prédiction de branchement indirect au niveau de chaque cœur logique. On revient ici à une forme de partitionnement spatial.

De son côté, Arm [60] a mis en place d'autres instructions barrières pour son jeu d'instructions. `csdb` crée une barrière à partir de laquelle seules les instructions de branchements peuvent être exécutées spéculativement en utilisant des données d'avant la barrière. `ssbb` et `pssbb` sont elles des barrières pour la spéculation sur les adresses, respectivement virtuelles et physiques.

Si la gestion de la spéculation par le logiciel est possible, elle reste cependant dépendante des outils mis à disposition par l'ISA. Les stratégies de contournement peuvent fonctionner dans certains cas, mais elles peuvent aussi se heurter à la complexité de la microarchitecture (*e.g.* comme Retpoline). L'utilisation d'instructions dédiées pose également le problème de la complexité logicielle, où il est nécessaire d'appliquer les contremesures adéquates à chaque saut potentiellement exploitable. Finalement, en multipliant l'ajout de nouvelles instructions, ce sont les performances qui sont impactées. Chaque application sécurisée nécessite l'exécution d'un plus grand nombre d'instructions. D'autres approches visent donc à gérer la problématique posée par la spéculation directement à sa source.

Renforcement du matériel

La spéculation étant un mécanisme purement matériel, certaines solutions académiques cherchent à revoir directement son implémentation. Une première solution envisagée est de la désactiver. C'est une approche utilisée par Bourgeat et al. [99] durant l'exécution du plus haut niveau de privilège. Celui-ci ayant accès à des informations d'autres applications, il est essentiel de s'assurer que les frontières d'isolation sont respectées. Les autres approches présentées visent donc principalement à implémenter différemment la spéculation.

Limiter les opérations spéculatives Une première approche vise directement à réduire la portée de la spéculation. L'idée générale est d'empêcher le processeur d'encoder spéculativement des données. C'est le principe des `store` dans les processeurs avec exécution dans le désordre [90]. Ceux-ci ne sont pas exécutés spéculativement, de par les difficultés à

revenir en arrière.

Lowe-Power et al. [90] proposent de n'exécuter spéculativement les `load` que lorsqu'ils génèrent un *hit*. Ainsi, aucune ligne n'est ramenée spéculativement. Cependant, même un cache *hit* peut altérer certains états d'un cache. Par exemple, des mises à jour sont effectuées à chaque accès pour des politiques de remplacement comme la LRU. Dans ces cas-là, ces états ne doivent être mis à jour que lorsque le `load` n'est plus spéculatif.

Dans leur publication originale de Spectre [54], Kocher et al. suggèrent de tracer les informations issues de la spéculation. Ainsi, l'objectif est de les empêcher d'être utilisées. Cette idée est reprise par NDA [119] et vise à empêcher la propagation de données spéculatives au sein d'un processeur avec exécution dans le désordre. Généralement, le résultat d'une opération est disponible et propagé aux opérations dépendantes dès que le calcul est effectué. Avec NDA, cette propagation n'est effectuée que quand l'opération source est marquée comme sûre. Les opérations considérées comme non sûres sont :

- les instructions suivant un branchement dont la cible et la direction ne sont pas connues,
- un `load` suivant un `store` dont l'adresse n'est pas déterminée,
- tout `load` qui n'est pas prêt à être retiré.

Finalement dans les autres cas, la propagation de données peut avoir lieu normalement. Une alternative allégée est de continuer à exécuter spéculativement les instructions ne menant pas à une potentielle source de fuite (*e.g. certaines opérations arithmétiques*).

Séparation de la spéculation Une autre approche vise à séparer la spéculation des états validés du processeur. Ainsi, ces derniers ne sont modifiés qu'une fois la spéculation totalement vérifiée. Aucun état dû à de mauvaises prédictions ne doit donc être observable *a posteriori*. C'est un principe déjà utilisé par les processeurs avec exécution dans le désordre pour les états architecturaux [90]. Ceux-ci ne sont transformés en état définitif que lorsque la spéculation est valide.

Khasawneh et al. [120] proposent SafeSpec qui modifie la microarchitecture pour séparer les états spéculés des états définitifs. Pour cela, des buffers cachés (invisibles depuis l'ISA) sont ajoutés et contiennent les états spéculés des caches et les TLB. Seules les instructions spéculées utilisent les informations contenues dans ces structures. Finalement, lorsqu'une instruction est définitivement validée, les états spéculatifs dépendants sont rendus visibles. À l'inverse, si l'instruction est annulée, alors ces états sont effacés. Yan et al. [121] et Gonzalez et al. [122] utilisent une approche semblable pour sécuriser les `load` spéculatifs. Ils se concentrent sur les caches de données, ressources nécessaires dans la plupart des variantes de Spectre publiées. Un buffer dédié est utilisé pour ces accès mémoires. Dans le cas d'InvisiSpec [121], il est mentionné que les autres modifications d'états (la mise à jour du TLB des données, du reste de la hiérarchie mémoire, *etc.*) sont retardées jusqu'à la validation de la spéculation.

Annulation de la spéculation Une variante de l'approche précédente est finalement d'être capable de restaurer directement l'état validé en cas de mauvaise spéculation. Lowe-Power et al. [90] mentionnent la possibilité d'annuler les traces de la spéculation. Cependant, ils ajoutent

de nouveaux mécanismes comme un buffer contenant les load spéculatifs. Cette stratégie est donc à la frontière entre annulation et séparation de la spéculation.

En revanche, Saileshwar et Qureshi [123] proposent une réelle stratégie d'annulation de la spéculation. Plutôt que de séparer systématiquement les mécanismes de spéculation, ils annulent leurs traces en cas d'erreur. Pour cela, ils évincent directement les lignes ramenées à cause de mauvaises prédictions. Ensuite, une restauration des lignes évincées est effectuée pour les caches L1. D'autres techniques basées sur l'indéterminisme ou l'altération des mesures sont utilisées pour protéger les autres états spéculatifs.

Autres approches D'autres approches ont été explorées afin de bloquer les effets négatifs de la spéculation.

Koruyeh et al. [124] considèrent que les attaques Spectre-BTB et Spectre-RSB sont des violations du flot d'exécution. Autrement dit, le processeur ne devrait jamais pouvoir rediriger certains des sauts vers n'importe quelle adresse. Ainsi, ils proposent d'appliquer des techniques connues pour assurer l'intégrité du flot d'exécution. Pour cela, l'ISA est modifiée afin de leur associer des étiquettes à chaque saut. Une nouvelle instruction est ajoutée afin d'effectuer une vérification de l'étiquette. Cette instruction est ensuite placée à chaque adresse de destination possible. Lors de l'exécution, le processeur n'aura donc le droit de rediriger spéculativement que vers les adresses spécifiées. En cas d'erreur détectée, l'exécution forcée d'une instruction de type `lfence` est lancée. Une structure cachée est également utilisée pour maintenir l'état du RSB. Située en mémoire, elle mémorise les informations de retour et prend en compte les changements d'exécution qui peuvent perturber les prédictions du RSB.

La gestion de la spéculation directement au niveau de la microarchitecture permet de traiter le problème à sa source. Ces mécanismes étant purement matériels, il est possible de les modifier sans impacter le reste du système. Contrairement aux approches logicielles, un contrôle fin des différents mécanismes impliqués y est possible. Cela implique cependant d'adapter de l'implémentation pour gérer tous les états spéculatifs.

Finalement, ces solutions de gestion de la spéculation cherchent à garantir qu'en toutes circonstances, la spéculation respecte certaines frontières. Elles ne visent donc que les premières phases des attaques par exécution transitoire. L'exfiltration des données en utilisant des ressources partagées reste possible. Dans le but d'une isolation complète, elles doivent être associées aux solutions pour la gestion des ressources partagées.

4.3. Solutions de gestion du temps et des évènements

Toutes les attaques présentées dans le chapitre 3 s'appuient sur des mesures de temps ou des détections d'évènements. Elles s'en servent pour identifier des motifs propres à des données ou exécutions. Une idée générale est donc de brouiller les éléments perceptibles depuis l'extérieur d'un processus.

Modification des évènements microarchitecturaux

L'une des conséquences est de directement modifier les évènements perceptibles pour qu'un attaquant ne soit pas capable de les exploiter.

Une approche radicale est de réduire autant que possible les traces laissées lors de l'exécution. Osvik, Shamir et Tromer [12] suggèrent par exemple de supprimer les accès mémoires sensibles. Ainsi, aucune modification dépendante n'est induite au niveau des caches. Pour cela, ils proposent notamment de remplacer les tables de correspondances utilisées en cryptographie par des opérations logiques. Si cette approche est intéressante, car elle implique de ne modifier que le logiciel, elle paraît difficilement applicable à tous les programmes.

Modifier la perception des accès aux caches est également possible en matériel. Cette stratégie est d'ailleurs mise en place par plusieurs solutions déjà présentées [90, 123], notamment contre l'exécution transitoire. Saileshwar et Qureshi [123] forcent la microarchitecture à générer un *miss* lors du premier accès d'un processus à une ligne ramenée spéculativement et partagée spatialement (*e.g.* dans le LLC). Ainsi, un attaquant devient incapable de détecter si une ligne a déjà été manipulée précédemment par un autre utilisateur. Ojha et Dwarkadas [125] généralisent ce concept et proposent de systématiquement transformer le premier accès d'un processus à une ligne de cache en *miss*. Pour cela, le matériel est modifié pour associer un bit pour chaque process à chaque ligne. Ce bit est mis à 1 une fois que le processus a effectué un premier accès : les cache *hits* sont alors possibles. Bien qu'efficace contre certaines attaques utilisant des librairies partagées (*e.g.* les attaques Evict+Reload), cette approche ne considère donc qu'une partie des attaques ciblant les mémoires caches.

Finalement, si modifier les évènements microarchitecturaux perceptibles peut être efficace dans certains cas (les *misses* ou *hits* dans les mémoires caches), cette approche reste limitée et spécifique à certaines attaques. D'autres informations (*e.g.* le remplacement d'une ligne) restent partagées et exploitables.

Indéterminisme du fonctionnement

L'une des caractéristiques essentielles des processeurs exploitée par les attaquants est leur déterminisme. En connaissant l'état du processeur à un instant précis, il est possible de connaître précisément son évolution au cours d'une exécution. Réciproquement, il est possible de déduire certaines composantes de son état à partir de l'évolution d'une exécution. Une approche de protection explorée est donc de casser ce déterminisme en y ajoutant de l'aléa au cours de l'exécution.

La plupart de ces solutions concernent le fonctionnement des mémoires caches. Page [89] propose par exemple de simplement maximiser la taille des lignes. L'objectif est alors de masquer les parties réellement utilisées : plus de données différentes étant contenues sur chaque ligne, il est plus difficile de savoir lesquelles ont précisément été manipulées. Une variante est de réduire la taille des données contenues (*e.g.* 8 bits au lieu de 32 bits). Ainsi, chaque ligne regroupe plus de données différentes et l'effet obtenu est similaire.

Wang et Lee [107, 126] proposent RPCache (Random Permutation Cache). Au lieu d'être fixe, l'index associé à chaque ensemble du cache est stocké dans un registre. Lorsqu'un *miss* entraînant un remplacement est

détecté, l'ensemble correspondant et un autre sont choisis aléatoirement, vidés puis leurs index échangés. Un attaquant n'est donc plus capable de déduire des informations sur la position des *misses*. De la même manière, NewCache [127] tient à jour une table par processus protégé qui associe de manière aléatoire une ligne à un index. Kong et al. [108] renforcent les propriétés de RPCache avec l'ajout d'instructions appelées *chargements informatifs (informing loads)*. Lors d'un accès mémoire, l'information d'un *hit* ou *miss* est directement retournée au logiciel. Cela permet alors aux programmes de s'adapter en conséquence. Kong et al. proposent également de protéger un cache standard (sans modification matérielle) en utilisant ces chargements informatifs ainsi que de la permutation aléatoire logicielle. Les accès mémoires sont alors randomisés directement par le programme lui-même.

Des techniques de chiffrement peuvent être utilisées pour briser le déterminisme du processeur. L'objectif de CEASER [128] est de rendre les accès aux caches aléatoires afin d'éviter les attaques basées sur les conflits. Avant d'accéder le LLC, l'adresse physique est chiffrée et la valeur obtenue est utilisée à la place. Des lignes normalement placées dans un même ensemble se retrouvent donc éparpillées dans l'ensemble du cache. Une version améliorée vient modifier régulièrement la clé de chiffrement (au bout d'un certain nombre de cycles) pour éviter un apprentissage de la part d'un attaquant. L'emplacement de chaque ligne doit alors être recalculé selon la nouvelle clé.

Au sein d'un cache, la manière la plus simple de forcer de l'indéterminisme reste cependant de modifier la politique de remplacement. En utilisant une politique aléatoire, il n'est plus possible d'exploiter les informations de remplacement. Cette stratégie, qui impacte peu les performances, est généralement privilégiée pour les caches L1 [123].

Le déterminisme n'est pas propre aux mémoires caches. Il est possible d'utiliser des principes similaires sur d'autres mécanismes. Zhao et al. [129] rendent aléatoire le partage des BHT et PHT entre des processus différents. Pour cela, des clés secrètes propres à chacun sont utilisées afin d'encoder le contenu ou l'index des tables. Le partage entre les processus est altéré.

Finalement, l'ajout d'indéterminisme est particulièrement efficace pour la gestion de ressources dupliquées (*e.g.* les lignes de cache). Les différents utilisateurs se retrouvent alors dans l'impossibilité de déduire précisément des informations d'utilisation des autres utilisateurs. Fondamentalement, cette approche ne fait donc que réduire la précision des informations perçues (*e.g.* l'utilisation d'un ensemble de cache précis). Elle ne traite pas le problème dès la source comme les mécanismes de gestion des ressources partagées. D'autres renseignements peuvent toujours être extraits (*e.g.* l'utilisation du cache en général si une ligne évincée est détectée).

Altération des mesures de temps

Pour fonctionner, de nombreuses attaques effectuent régulièrement des mesures de temps. Les résultats obtenus dépendent donc de la précision de ces dernières. De ce fait, une autre approche de sécurisation consiste en leur altération.

L'ajout d'indéterminisme, comme vu précédemment, est une méthode possible. Les mesures effectuées ne sont alors plus représentatives du fonctionnement réel. D'autres solutions ont montré comment les

altérer plus précisément.

Une première méthode est simplement d'ajouter du bruit aléatoirement aux mesures de temps [12]. Certaines des attaques ont besoin de mesures précises, à quelques cycles près. Le bruit serait donc capable de brouiller certaines informations sensibles. Les stratégies basées sur le bruit ne sont cependant pas fiables, un attaquant pouvant les contourner à partir de mesures répétées.

Page [89] propose une autre méthode pour tromper un attaquant en générant de faux accès. Pour cela, en parallèle d'un "vrai" tableau, les données d'un "faux" tableau sont manipulées afin de brouiller l'impact sur la mémoire. Seul le logiciel doit donc être modifié.

Une stratégie largement reprise pour les navigateurs internet [115, 130-132] est la réduction de la précision des horloges disponibles. Cependant, Schwarz et al. [133] ont montré qu'il existait d'autres moyens pour un attaquant d'obtenir une source de temps précise. Cette même stratégie peut également être appliquée directement au niveau du matériel et des compteurs de performance. Ge et al. [134] proposent par exemple de réduire la précision des compteurs quand des opérations de flush (nécessaires à certaines attaques sur les caches) sont détectées.

Une solution radicale pourrait être de supprimer la référence de temps au sein du circuit : le signal d'horloge [89]. L'idée serait alors de concevoir des circuits complètement différents dits "asynchrones". Cependant, même avec un changement si drastique, tous les problèmes ne seraient pas résolus, comme celui de la détection des *hits* et *misses*. De plus, une référence de temps est essentielle pour de nombreuses opérations. C'est notamment le cas pour les interactions et la synchronisation avec des éléments extérieurs comme des écrans ou d'autres périphériques.

Exécution en temps constant

Les solutions présentées jusqu'à présent s'intéressaient généralement à l'isolation entre les différentes exécutions. Leur objectif était de s'assurer que la microarchitecture ne peut être utilisée pour partager des informations. Une dernière approche s'attaque directement à la notion de temps au niveau de la microarchitecture. L'objectif est de supprimer toute variation temporelle en exécutant les programmes sensibles en temps constant. Cela implique que, quel que soit l'état du processeur, l'exécution d'un programme donné devra toujours mettre le même temps.

Une idée proposée par Osvik, Shamir et Tromer [12] est de fixer le temps d'un accès mémoire. Ainsi, que celui-ci génère un cache *hit* ou *miss*, le temps d'accès serait le même. Aucune information ne serait alors exploitable. En pratique, cette approche est cependant très contraignante pour les performances : le pire scénario doit toujours être considéré. Cela revient finalement à considérer que chaque accès est un *miss*.

Plus généralement, une stratégie est de concevoir des programmes dont le temps d'exécution ne dépendra pas des données sensibles [12, 48]. Cela passe notamment par la suppression des accès mémoires ou branchements dépendants de ces données.

L'exécution en temps constant remet en cause les fondamentaux du fonctionnement de la microarchitecture. Elle vise à bannir les variations temporelles intentionnellement introduites par les mécanismes d'optimisation des performances. Pour chaque opération, cela implique que le processeur doit se placer dans le cas le plus problématique : celui où l'opération aurait été la plus lente. Du point de vue des performances,

un tel fonctionnement peut s'avérer dramatique : comment faire par exemple dans le cas des accès mémoire ? Faut-il se passer des mémoires caches, au prix de ralentissements extrêmes en accédant directement à la mémoire principale ?

Du point de vue de la microarchitecture, l'exécution en temps constant ne résout pour autant pas le problème d'isolation. Certaines informations restent partagées et les mécanismes de spéculation peuvent toujours être abusés. C'est donc une problématique différente qui est ici visée : comment ne pas extraire d'informations du temps d'exécution d'un programme cible ?

Finalement la plupart de ces approches de gestion du temps et des évènements cherchent seulement à masquer le problème d'isolation. Les mesures d'un attaquant ou les évènements qu'il perçoit sont brouillés afin d'être difficilement exploitables. Cependant, les fuites ne sont pas réellement traitées à leurs sources comme pour les solutions de gestion des ressources partagées et de la spéculation. Certaines sont donc toujours présentes dans le système.

On distingue les solutions cherchant l'exécution en temps constant. Au-delà de l'isolation entre processus, elles considèrent le problème du temps d'exécution global d'un processus. Selon les optimisations appliquées, celui-ci peut s'avérer dépendant de données sensibles. L'idée est de s'assurer que ce temps d'exécution reste identique, quelles que soient les données utilisées.

4.4. Stratégie globale

Nous avons vu précédemment les différentes stratégies proposées contre les attaques logicielles exploitant les mécanismes microarchitecturaux. Si la plupart présentent des avantages pour offrir des protections bien précises sur des systèmes définis, aucune ne considère le problème dans son entièreté. L'objectif ici est d'en tirer des leçons afin de définitivement supprimer ce type d'attaques.

Prise en compte ciblée des failles.

Nous avons vu précédemment qu'il existait trois grands types d'approches contre ces attaques. À l'exception des attaques temporelles pures, elles peuvent être regroupées sous une même problématique : l'isolation du logiciel dans la microarchitecture. Que cela soit par le biais des ressources partagées ou des frontières non respectées, la microarchitecture permet actuellement des échanges d'informations entre des exécutions normalement indépendantes.

Chacun des types de solutions ne se concentre que sur une partie de cette problématique d'isolation. Le premier cherche à modifier la gestion des ressources partagées dans la microarchitectures. Le deuxième vise à restreindre les libertés prises par la spéculation dans les implémentations modernes. Le dernier type cherche lui à masquer les informations perçues par un attaquant, sans éliminer les fuites à la source. Si certains travaux proposent des combinaisons de ces approches, ils ne considèrent cependant que des implémentations précises. Cela peut être un type de mécanismes (*e.g.* les mémoires caches) ou une catégorie de processeur (*e.g.* les processeurs avec exécution dans le désordre). Aucune approche

générique et globale n'est appliquée.

Dans le cas présent, considérer les composants problématiques un à un mène inévitablement à un accroissement de la complexité du système. Pour autant, cela n'est pas synonyme d'efficacité. Pour preuve, nous avons vu que ces problèmes d'isolation sont connus depuis de nombreuses années. Pire encore, certaines solutions ciblées peuvent mener à l'aggravation d'autres sources de fuites (*cf.* Fallout [74] et les contremesures Meltdown).

Ainsi, la meilleure stratégie à adopter face à l'ampleur du problème semble de repartir d'une feuille blanche. Les processeurs modernes sont le résultat de nombreuses années d'optimisations pour les performances où les contraintes de sécurité n'existaient pas. Il est à présent nécessaire de les modifier depuis leur base, afin d'intégrer au plus tôt ces besoins de sécurité. La conception d'un système sécurisé ne passe évidemment pas par la suppression de tout ce qui existe, mais par une adaptation complète dès ses fondements. Les solutions proposées jusqu'à présent montrent bien que chaque niveau du système a un rôle à jouer, que ce soit le logiciel, le matériel ou le jeu d'instructions.

Il est important de rappeler qu'une reprise à zéro du système n'est évidemment pas toujours envisageable. De grands acteurs industriels comme Intel, Arm ou AMD ne peuvent envisager une telle stratégie à court ou moyen terme, notamment pour des contraintes économiques et de compatibilité de leurs produits. Ainsi, les solutions qu'ils proposent visent logiquement à apporter une réponse rapide à un problème donné. De la même manière, un grand nombre de propositions académiques cherchent avant tout à conserver une rétrocompatibilité avec les systèmes existants. Dans la suite de ce manuscrit, nous nous plaçons dans un scénario de liberté totale vis-à-vis des systèmes existants. C'est donc une approche sur le long terme, afin d'étudier les bases pour de futurs processeurs.

Rôles des couches d'abstraction

Tout système informatique est composé des trois niveaux d'abstraction suivants : le logiciel, le matériel et l'ISA. Repenser entièrement un système pour la sécurité implique de revoir le rôle de chacun de ces niveaux. Or, des propositions de modifications ont déjà été suggérées pour chacun d'entre eux. Elles représentent une base d'études intéressante pour bien comprendre les éléments à prendre en compte. Notre objectif ici est donc de partir de ces travaux existants pour déterminer précisément le rôle de chaque niveau.

Logiciel Le plus haut niveau d'abstraction est le logiciel. C'est là que sont définis les applications et utilisateurs du système. Ainsi, seul le logiciel connait au moment de l'exécution les contraintes réelles des applications, comme dans notre cas les frontières d'isolation à respecter. Celles-ci peuvent complètement varier d'une exécution à une autre. Comme utilisée par certains travaux, une approche possible est de mettre en place l'ensemble de la stratégie de sécurité en logiciel. Un premier avantage évident est la flexibilité puisqu'il est possible d'adapter le logiciel selon les contraintes à appliquer.

Cependant, ne modifier que le logiciel n'est pas efficace : les solutions de ce type sont directement dépendantes des outils à leurs dispositions, *i.e.* les instructions disponibles. Sans cela, elles sont "aveugles" vis-à-vis des

phénomènes se déroulant au niveau matériel. Ge, Yarom et Heiser [91] ont montré que les processeurs modernes manquent cruellement de moyens d'interagir avec le matériel. Certains états microarchitecturaux restent inaccessibles directement par le logiciel (*e.g.* les prefetchers d'instructions). Ils peuvent donc être librement exploités pour faire fuir ou récupérer des données. Pire encore, le cas des attaques par échantillonnage [75, 76] prouve également que les solutions adoptées restent insuffisantes. Le logiciel se retrouve sans moyen d'agir. Le cas de retpoline [112] est particulièrement représentatif de ce genre de cas. Pour rappel, cette solution logicielle essaye de détourner le fonctionnement du BTB. Cependant, cela ne fait parfois que déplacer le problème (du BTB vers le RSB dans ce cas précis). De plus, la complexité et la portabilité doivent également être considérées.

Ne considérer la sécurité qu'au niveau logiciel, c'est implémenter l'ensemble des contremesures à ce niveau-là. Or, cela passe par la considération de toutes les fuites potentielles. Ceci semble difficilement envisageable si l'on cumule les contremesures à prendre en compte : retpoline pour certains sauts, stopper la spéculation, vider les buffers. . .

Combler des fuites matérielles au niveau logiciel mène aussi à un problème de portabilité. Le principe de base d'un système est qu'à partir d'une architecture définie, il soit possible de concevoir indépendamment le logiciel de l'implémentation matérielle. Cela a des avantages, comme la possibilité de réutiliser une majorité de codes d'un processeur à un autre. Seulement, les fuites existantes sur un système varient selon la microarchitecture. Les contremesures logicielles, pour être efficaces, doivent donc prendre en compte le processeur sur lequel elles sont exécutées. De par la complexité du fonctionnement des microarchitectures modernes, le nombre de mécanismes à prendre en compte et leurs possibles interactions sont considérables. Ces solutions doivent être à nouveau adaptées sur chaque cible différente pour remplir leur rôle.

Sur le long terme, baser l'ensemble d'une politique d'isolation sur le logiciel semble donc une mauvaise stratégie. En revanche sur des systèmes existants, le matériel ne pouvant être modifié[1], le logiciel reste la meilleure solution.

1: À l'exception des instructions implémentées à l'aide de microcodes. Les modifications possibles restent certes intéressantes pour certaines contremesures, mais restreintes.

Matériel Le plus bas niveau d'abstraction est le matériel. C'est celui de la microarchitecture elle-même. À ce niveau, les différents mécanismes microarchitecturaux sont parfaitement contrôlables et modifiables. Cela correspond également au niveau d'abstraction où apparaissent les fuites elles-mêmes. Une approche radicalement opposée à la précédente est donc de ne modifier que le matériel. L'isolation est entièrement assurée à ce niveau-là en comblant les fuites à la source.

Le point fort principal de cette approche est bien évidemment la maîtrise du système et des fuites. Les solutions peuvent être parfaitement adaptées aux mécanismes rencontrés : pour un processeur donné, les contremesures nécessaires seront directement appliquées [90, 120, 123]. La complexité ajoutée se trouve entièrement dans la microarchitecture. Le matériel étant invisible pour le logiciel, les modifications le sont également. Le problème de portabilité ne se pose donc pas pour les applications exécutées.

Par définition, une approche purement matérielle est en revanche peu flexible : elle s'applique à chaque exécution et ne peut être modifiée après fabrication. Le matériel seul n'a aucun moyen de distinguer les différentes

frontières d'isolation que le logiciel utilise. Ainsi, pour n'importe quel logiciel exécuté, les garanties d'isolation doivent être assurées. Cela implique donc de toujours se placer dans le pire des scénarios, où tout doit être isolé. Aucun compromis n'est possible. Si l'on considère l'ensemble des contremesures matérielles nécessaires, l'impact sur les performances peut s'avérer important.

Au final, il "suffit" idéalement de changer un processeur non sécurisé par un sécurisé pour profiter des protections. La possibilité d'appliquer des "patchs" *a posteriori* si de nouvelles fuites sont découvertes est impossible (ou très limitée). Pour être efficace, cette approche demande donc en amont une analyse de risque basée sur une parfaite connaissance des menaces actuelles et futures.

ISA Le troisième niveau d'abstraction est l'ISA. Elle sert d'interface entre le matériel et le logiciel. C'est également le premier élément défini dans un système informatique. Elle spécifie les instructions implémentées par le processeur et utilisables par le logiciel. C'est à ce niveau que sont définies des règles générales de fonctionnement comme le modèle mémoire (comment apparaissent les opérations sur des mémoires).

Le point fort essentiel de ces approches est la vision globale du système. En modifiant l'ISA, il est aussi bien possible d'influer la conception du matériel que du logiciel [43, 70, 99]. Du point de vue de l'implémentation, cela permet d'exploiter la connaissance fine de la microarchitecture. Celle-ci peut-être adaptée différemment au sein de chaque processeur, afin de correspondre au mieux à l'ISA. Du point de vue de l'application exécutée, cela permet une adaptation en fonction des contraintes. Les nouvelles fonctionnalités offertes par l'ISA sont utilisables librement.

Évidemment, certaines contreparties sont à prendre en compte. Le première est qu'elle impacte aussi bien le logiciel que le matériel. L'ensemble doit donc être modifié pour s'adapter, ce qui est souhaitable pour de nouvelles propriétés de sécurité, mais moins pour des fonctionnalités existantes. Enfin, une modification de l'ISA jouant également sur le matériel, elle est difficilement envisageable sur des systèmes existants. Comme pour les approches purement matérielles, il est nécessaire d'avoir anticipé les besoins en amont. Ce genre de solution n'est donc pas (toujours) envisageable sur des systèmes existants. Elles doivent davantage être vues comme un moyen de gérer définitivement un problème sur les processeurs futurs.

La manière de modifier l'ISA est un choix important. Les solutions vues auparavant montrent que l'ajout d'instructions peut avoir des impacts différents. Par exemple, certaines modifications privilégieront la flexibilité à la portabilité du logiciel, quand d'autres préfèreront garder une grande partie de la complexité au niveau matériel. L'ISA peut donc être vue comme une ligne directrice pour l'ensemble du système. Il est ainsi difficile d'évaluer directement l'impact général des approches modifiant l'ISA : les conséquences sont spécifiques à chaque proposition.

Un récapitulatif des points forts et points faibles de chaque approche est présenté sur la Table 4.1. Il confirme que les approches purement logicielles et matérielles ont des caractéristiques distinctes, là où la modification de l'ISA apparaît comme un compromis.

Pour conclure, on distinguera trois scénarios où les rôles de chaque niveau d'abstraction peuvent être clairement définis :

Caractéristiques	Logiciel	ISA	Matériel
Flexible	+ +	+ +	- -
Efficace	-	+ +	+ +
Portable	- -	N/D	+ +
Ajustable	+ +	- -	- -
Complexité HW	+ +	N/D	- -
Complexité SW	- -	N/D	+ +

Table 4.1. – Comparaisons des approches par niveau d'abstraction. Les caractéristiques considérées sont la flexibilité du système, l'efficacité des contremesures en place, la portabilité du logiciel, la possibilité d'ajuster après implémentation, l'impact sur la complexité du matériel et sur celle du logiciel. Une caractéristique peut être un point fort important (+ +) d'un niveau d'abstraction, un point fort (+), un point faible (-), un gros point faible (- -) ou non défini (N/D).

1. **Court terme/ système existant :** le logiciel est le seul élément paramétrable (hors systèmes avec microcodes). Il est donc nécessaire, à partir des outils offerts par l'ISA et implémentés par le processeur, de faire "au mieux". Cela passe généralement par des compromis : pertes de performances importantes ou possibles failles de sécurité.
2. **Moyen terme/ base existante :** L'ISA est en grande partie déjà définie. Certaines nouvelles fonctionnalités peuvent être ajoutées tout en considérant la rétrocompatibilité. En conséquence, le matériel et le logiciel peuvent être adaptés.
3. **Long terme/ systèmes à concevoir :** tous les niveaux sont encore modifiables. Le logiciel, selon les applications exécutées, indique ses contraintes au système. Le matériel les applique et les respecte au sein de la microarchitecture. Enfin, la communication entre les deux se fait par le biais de l'ISA modifiée.

4.5. Conclusion et suite

Dans ce chapitre, nous avons vu les différentes stratégies utilisées contre les problèmes de variations temporelles. Après un bilan de ces solutions, de leurs avantages, mais aussi de leurs faiblesses, nous établissons l'approche adoptée pour la suite de ces travaux.

Bilan Les problèmes d'isolation et de variations temporelles étant connus depuis de nombreuses années, plusieurs stratégies ont été proposées. Elles peuvent être regroupées en trois grandes catégories.

La première est celle des contremesures dédiées à la gestion des ressources partagées. Ces ressources affaiblissent les frontières d'isolation entre les différents utilisateurs. De nombreuses solutions visent donc à rétablir ces frontières. Cela peut passer par la suppression des mécanismes incriminés, des traces persistantes ou bien par le partitionnement des ressources.

Une deuxième catégorie de contremesures vise à limiter les fuites dues à la spéculation. Celle-ci, selon les implémentations, passe parfois outre les frontières d'isolation. L'idée derrière ces solutions est donc de continuer à assurer les garanties durant l'intégralité de l'exécution des programmes. Les modifications proposées s'articulent autour de deux axes : la modification du logiciel ou le renforcement des frontières au niveau du matériel.

Enfin, la dernière catégorie de solutions s'attaque directement aux éléments perceptibles par un attaquant : le temps et les évènements. Le temps est la grandeur physique exploitée par le biais des variations temporelles. En modifiant sa perception ou en brouillant les mesures

effectuées par un attaquant, les informations recueillies ne sont plus fiables. De la même manière, il est possible d'empêcher la détection d'évènements microarchitecturaux liés à l'exécution d'instructions et données précises.

Bien qu'efficaces, ces stratégies ne ciblent que partiellement la problématique d'isolation. Cela peut être un mécanisme (*e.g.* les mémoires caches), un type de partage (*e.g.* l'utilisation des états persistants) ou un type de fuites (*e.g.* les variations dues aux cache *misses*). Finalement, on assiste à une course permanente entre les attaquants et les concepteurs. À chaque nouvelle faille d'isolation, une modification spécifique est proposée. En réponse, une variante (même légère) de l'attaque est proposée ce qui mène à de nouvelles modifications, un accroissement de la complexité sans pour autant obtenir une efficacité globale.

Pour envisager de réelles protections, le constat [135] est clair : il semble essentiel de repenser le système depuis ses fondements. L'ensemble doit être conçu en intégrant une même logique de sécurité. C'est dans cette idée que se placent donc les travaux de ce manuscrit.

Approche explorée Pour la suite de ces travaux, nous nous concentrons sur la problématique d'isolation des exécutions. Elle concerne la quasi-intégralité des attaques présentées précédemment, que cela soit l'utilisation de ressources partagées, le respect des frontières lors de la spéculation ou l'utilisation de canaux cachés. Le cas des attaques temporelles mesurant le temps d'exécution globale est bien distinct de cette problématique. Nous avons vu qu'elles nécessitent la mise en place de techniques bien particulières pour une exécution en temps constant. Cette partie des attaques affectant la microarchitecture est ainsi laissée pour de futurs travaux.

Notre objectif est la conception de processeurs avec des mécanismes d'isolation. Cette approche correspond à une vision sur le long terme du problème. Pour cela, l'idée est de reprendre la problématique dès la définition de l'ISA. Les travaux effectués dans le cadre de cette thèse s'organisent en deux axes.

Le premier est la définition de l'ISA elle-même. Chacune des propositions vues précédemment influe différemment sur l'ensemble du système. L'objectif est donc dans un premier temps d'effectuer une étude des différentes possibilités de modification de l'architecture. Un choix sera ensuite fait afin de définir un nouveau jeu d'instructions considérant la problématique d'isolation. C'est également à ce moment-là que nous pourrons détailler l'impact sur le logiciel lui-même.

Le second axe de travail est l'implémentation du processeur. À partir des informations reçues par le logiciel en utilisant notre nouvelle ISA, le matériel doit être capable de réagir en conséquence. Là encore, nous avons vu que des solutions ont été présentées indépendamment pour les différents mécanismes microarchitecturaux. L'objectif est donc de les généraliser, afin d'être capable de supporter cette nouvelle ISA dans tous types de processeurs, quels que soient les mécanismes implémentés.

À terme, la finalité sera de disposer d'une ISA complète intégrant les problématiques de sécurité ainsi que de disposer des outils et méthodes nécessaires à son implémentation matérielle.

Repenser le jeu d'instructions

Modifier l’ISA pour la sécurité | 5.

Résumé du chapitre : Nous avons mis en évidence dans les chapitres précédents que l’ISA a un rôle à jouer pour la conception de processeurs avec des contraintes de sécurité. Maintenant, nous étudions en détail ce rôle et les différentes possibilités pour le mettre en place. Deux notions centrales utilisées pour l’ajout de propriétés de sécurité sont les domaines de sécurité et les politiques de sécurité. Après les avoir définies, nous verrons comment l’ISA peut les intégrer. Les propositions industrielles et académiques actuelles sont étudiées afin d’en déterminer les avantages et limites. En partant de ce point, nous définissons un objectif de ce que doit permettre l’ISA.

5.1 Principes de base 71
5.2 Organisation des domaines de sécurité 72
Hiérarchie statique 72
Hiérarchie hybride 74
Hiérarchie dynamique . . . 77
5.3 Politique d’isolation microarchitecturale dans l’ISA 78
Forme des modifications de l’ISA 78
Stratégies d’abstraction du matériel 79
5.4 Conclusion 80

5.1. Principes de base

La mise en place de garanties de sécurité nécessite de savoir où et quand elles s’appliquent. Tous les utilisateurs n’ont pas forcément les mêmes besoins. Ainsi, pour généraliser une approche, nous devons définir sous quelle forme ces informations seront disponibles. Ensuite seulement, le système devra être conçu pour les interpréter. On utilise communément les notions de *domaine de sécurité* et de *politique de sécurité* pour indiquer ces informations.

Définition Le National Institute of Standards and Technology (NIST) donne la définition suivante d’un domaine de sécurité :

Definition 5.1.1 ***Domaine de sécurité :*** *Un domaine implémentant une politique de sécurité et administré par une seule autorité.* [136]

Un domaine de sécurité définit donc une zone, aussi bien temporelle que spatiale, où sont appliquées des politiques de sécurité et dirigée par une entité ayant les droits nécessaires.

Definition 5.1.2 ***Politique de sécurité :*** *Un ensemble de critères pour la provision de services de sécurité.* [137]

Autrement dit, une politique de sécurité fixe les objectifs de sécurité et les contraintes correspondantes.

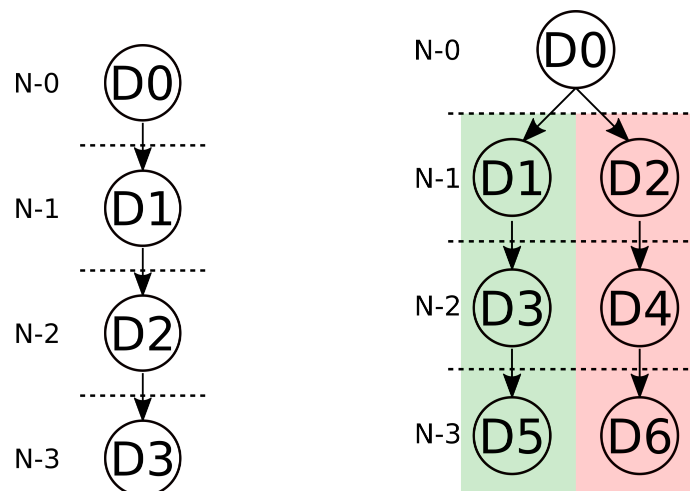
Appliqué au cas d’un processeur, le domaine de sécurité correspond à un environnement d’exécution où s’applique un ensemble de contraintes. Les objectifs de sécurité peuvent changer selon les programmes exécutés. L’ISA doit donc laisser au logiciel la possibilité de définir ses propres domaines de sécurité. Le matériel devra alors les interpréter. Une politique de sécurité va, elle, définir, pour un objectif de sécurité précis, des contraintes sur les mécanismes à utiliser. Par exemple, une politique de contrôle des droits d’accès autorisera l’utilisation de ressources à certains domaines seulement. Dans notre cas, nous nous intéressons à la mise en place d’une *politique d’isolation microarchitecturale*.

Politique d'isolation microarchitecturale On appelle ici une politique d'isolation microarchitecturale une politique de sécurité dont l'objectif est l'isolation des exécutions appartenant à un domaine. Ce dernier ne doit donc pas être capable de transmettre *via* la microarchitecture des informations observables par tout autre domaine.

La mise en place d'une telle politique implique deux éléments : le domaine où elle s'applique et les mécanismes nécessaires. Dans le cas d'un processeur, nous avons vu que les contraintes varient selon les programmes. Le domaine d'application de la politique n'est donc connu que du logiciel. Cependant, cette information doit être transmise au matériel pour savoir quand l'appliquer. En lien avec les conclusions du chapitre 4, c'est donc l'ISA qui doit permettre cette communication. L'application de la politique et la mise en place des mécanismes nécessaires reviennent, elles, entièrement au matériel.

5.2. Organisation des domaines de sécurité

Les besoins de sécurité dans les processeurs ont évolué avec le temps. Ainsi, on retrouve différentes propositions d'implémentation de la notion de domaine de sécurité au niveau de l'ISA. Au sein de chaque implémentation, les différents domaines sont organisés selon une hiérarchie. Dans cette section, nous proposons notre propre classement des implémentations de hiérarchie en différents types selon leurs caractéristiques et limites.



(a) Modèle des privilèges classiques avec : D0 = machine, D1 = hyperviseur, D2 = superviseur, D3 = utilisateur.

(b) Modèle de TrustZone. En rouge sont représentés les niveaux de privilèges classiques, en vert les sécurisés. Ainsi, on retrouve deux versions de l'hyperviseur, du superviseur et de l'utilisateur.

Figure 5.1. – **Hiérarchie statique des domaines de sécurité.** N- correspond au niveau de privilège sur le système : un nombre faible indique un plus grand contrôle du système. Une flèche indique une dépendance entre deux domaines : celui au-dessus peut gérer celui en dessous. Ces deux modèles sont figés. Ils ne s'adaptent donc pas au logiciel. À l'inverse, c'est au logiciel à se les approprier pour adapter sa structure.

Hiérarchie statique

Principes Le premier modèle de hiérarchie, et également le plus simple, est celui d'une hiérarchie statique. Comme présenté sur la Figure 5.1, les configurations de chaque domaine de sécurité sont figées. De même, leurs dépendances sont inflexibles. Sur la Figure 5.1a, D0 peut gérer D1

qui lui-même peut gérer D2 *etc.* Sur la Figure 5.1b, D0 peut gérer D1 et D2 qui respectivement gèrent D3 et D4 *etc.* En revanche, D1 n'a aucune influence sur D2 et D4, et D2 sur D1 et D3. Leur contrôle du système décroît en descendant dans la hiérarchie.

Sur la Figure 5.1b, plusieurs domaines peuvent être placés au même niveau de privilège. Certaines implémentations, comme ARM TrustZone (détaillée plus loin), permettent à plusieurs domaines d'avoir les mêmes droits sur le système. C'est ce que l'on appelle l'*horizontalité* de la hiérarchie. Cela correspond à la possibilité de distinguer plusieurs domaines avec les mêmes droits sur le système. Cette horizontalité est utile si plusieurs programmes doivent effectuer des tâches similaires, sans pour autant se faire confiance mutuellement. Par exemple, les différentes applications sont généralement exécutées avec un niveau utilisateur. Si le système permet une certaine horizontalité, alors il sera possible de les distinguer entre elles. Chacune aura ses propres politiques de sécurité appliquées au niveau du matériel.

De la même manière, on appelle *verticalité* la possibilité de distinguer plusieurs niveaux d'influence sur le système. Chaque niveau a des droits différents sur le système : le niveau le plus haut (ici D0) a tous les droits quand les autres ont des restrictions. Par exemple, le superviseur peut avoir accès aux registres pour paramétrer la mémoire virtuelle ou les exceptions, ce qui n'est pas le cas de l'utilisateur. Ce dernier est donc dépendant des niveaux supérieurs.

Privilèges La hiérarchie de la Figure 5.1a est celle utilisée pour les privilèges. Introduits par le projet MULTICS [138] il y a plusieurs dizaines d'années, ils constituent aujourd'hui la base de sécurité de la plupart des systèmes.

Pour rappel, les privilèges sont un ensemble statique de domaines de sécurité. Ils sont en nombre fixe et leurs propriétés sont établies par l'ISA. Dans le cas de l'ISA RISC-V [139], 4 niveaux de privilèges sont par exemple disponibles. Chacun de ces domaines est caractérisé par des politiques d'accès à des ressources particulières. Certains registres appelés CSR ne sont par exemple accessibles qu'à partir d'un certain niveau de privilège. Le matériel est responsable de la vérification des droits d'accès associés.

Parce qu'ils sont peu nombreux et statiques, les privilèges ont l'avantage d'être simples à implémenter. Ils sont de plus parfaitement adaptés aux contraintes de sécurité originelles : la restriction des accès à certaines ressources. Cependant, les possibilités offertes au logiciel demeurent réduites. Avec la complexification des systèmes et de leur utilisation, de nouveaux besoins de sécurité sont apparus. Le besoin d'isolation entre les applications en est un bon exemple. Or, les privilèges classiques n'offrent pas d'horizontalité pour distinguer ces applications. Ils proposent une granularité trop grosse : ils ne sont pas (ou peu) adaptés pour de nouvelles politiques de sécurité. De nouveaux domaines de sécurité ont donc été proposés au niveau des ISA.

ARM TrustZone Pour ses implémentations, Arm a introduit dans son ISA une notion d'environnement d'exécution de confiance (TEE) appelée TrustZone.

L'objectif d'un TEE est d'offrir un environnement d'exécution avec des garanties de sécurité. C'est une appellation standardisée par Glo-

balPlatform, reprise et adaptée par plusieurs fabricants comme Arm mais aussi Intel comme nous le verrons plus tard. Les TEE implémentent généralement des politiques de confidentialité et d'intégrité. Ces environnements peuvent être conçus entièrement en logiciel, ou bien assistés par des mécanismes matériels comme TrustZone.

Avec TrustZone, l'architecture permet la séparation du matériel en deux mondes : un sécurisé et un autre non sécurisé. Chacun de ces mondes n'a accès qu'à certains composants ou parties de composants matériels. Pour cela, un bit supplémentaire sur le bus mémoire vient identifier à quel monde appartient chaque requête. Chaque composant matériel est ainsi capable de réagir en conséquence, par exemple en autorisant ou non certains accès.

Du point de vue de l'ISA, cela s'illustre par l'ajout de trois niveaux de privilèges SEL0, SEL1 et SEL2. Ils sont les équivalents sécurisés des privilèges EL0, EL1 et EL2. Le passage d'un monde à l'autre s'effectue par le biais d'un moniteur logiciel sécurisé placé au niveau EL3. On retrouve donc un schéma équivalent à la Figure 5.1b.

L'objectif d'une telle organisation est de permettre une séparation renforcée par le matériel d'un environnement dit *riche* d'un autre *sécurisé*. Le code sensible doit être placé dans ce monde sécurisé quand le reste du logiciel doit utiliser les privilèges classiques. Ces nouveaux privilèges permettent donc de distinguer des domaines avec des politiques d'accès aux ressources différentes. Par exemple, il est possible de séparer certaines mémoires en deux espaces : riche ou sécurisé. Chaque privilège n'a ensuite accès qu'à la partie correspondante à son monde. Aussi, le bit supplémentaire utilisé est interprété comme un nouveau bit d'adresse (un 33^{ieme} bit). Du point de vue des caches, une distinction peut ainsi être faite entre une adresse manipulée par le monde riche ou par le monde sécurisé.

Limites Une hiérarchie statique n'offre que peu de flexibilité pour le logiciel. Celui-ci doit s'adapter avec les domaines qu'il a à sa disposition. Le manque d'horizontalité est particulièrement limitant. Il n'est par exemple pas (ou peu) possible de placer plusieurs applications dans des domaines différents. Même si le modèle de la Figure 5.1b reste plus flexible que celui sur la Figure 5.1a, il reste néanmoins limité. Dans le cas de TrustZone, l'ISA ne nous permet finalement de distinguer que 7 domaines différents dont les rôles sont fixés au niveau du matériel.

Les privilèges sont un pilier essentiel des ISA actuelles puisque les systèmes d'exploitation actuels sont tous basés sur ce modèle. D'autres solutions ont donc été proposées pour étendre les possibilités offertes tout en conservant cette base.

Hiérarchie hybride

Principes Une hiérarchie hybride est constituée d'une partie fixe et d'une partie paramétrable par le logiciel. Un exemple est illustré sur la Figure 5.2. Dans ce cas, les domaines D0, D1 et D2 sont fixés. Chacun d'entre eux a en revanche la possibilité de créer de nouveaux domaines de sécurité dont ils seront responsables.

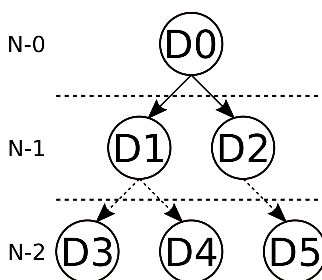


Figure 5.2. – Hiérarchie hybride des domaines de sécurité. Une flèche indique une dépendance entre deux domaines. Les flèches pleines sont des dépendances figées, celles pointillées sont des dépendances modifiables. Ce modèle, proche de celui d'Intel SGX ou de certaines propositions académiques, s'adapte donc partiellement au logiciel.

L'intérêt majeur d'un tel modèle hybride est d'ajouter de l'horizontalité dans le système. Il est à présent possible de créer plusieurs domaines avec les mêmes droits. Ceux-ci seront cependant indépendants les uns des autres : ils n'ont pas à se faire confiance mutuellement. Plusieurs

cas d'applications peuvent bénéficier de l'horizontalité. Par exemple, on peut imaginer la création de domaines de sécurité différents pour protéger des pilotes (ou *drivers*). Ces programmes communiquent avec des périphériques : ils ont donc besoin des droits correspondants (*e.g.* accéder à un bus mémoire). Ainsi, on retrouve des pilotes différents pour chaque périphérique, généralement fournis par le fabricant. En séparant chaque pilote dans un domaine différent, il est possible de restreindre ses capacités de nuisance sur les autres éléments du système. Il n'a ainsi accès qu'à ses propres informations et non à l'ensemble des pilotes. Cette même stratégie peut finalement être adaptée pour séparer des applications d'utilisateurs, des gestionnaires d'interruption ou d'exception *etc.*

En ce qui concerne la sécurité, pouvoir séparer le code en parties indépendantes facilite également son évaluation. Chacune peut ensuite être considérée seule : la complexité dont elle dépend est réduite au minimum. On appelle trusted code base (TCB) la base de code dont dépend la sécurité d'un programme. Plus un TCB est réduit, plus il est facile d'identifier les potentielles failles susceptibles de compromettre le programme correspondant.

Mémoire virtuelle La mémoire virtuelle permet la création de domaines de sécurité au niveau de la mémoire. Ces différents domaines ont des restrictions sur les adresses auxquelles ils peuvent accéder. Un domaine avec des droits supérieurs, celui où se situe le système d'exploitation, est responsable de leur gestion. Ce système n'offre finalement de l'horizontalité qu'au niveau de privilège le plus bas. La mémoire virtuelle constitue le mécanisme de base par excellence pour constituer des domaines de sécurité gérés par le logiciel.

Intel SGX Intel SGX est le nom de l'extension x86 d'Intel introduite en 2015 pour le support d'un TEE. Elle permet le support d'enclaves par le matériel.

On appelle une *enclave* un programme isolé (notamment sa mémoire) de tout autre programme du système. C'est donc un domaine de sécurité avec une politique d'isolation très stricte. L'intérêt d'Intel SGX est de pouvoir par exemple séparer un grand nombre d'applications sur un même système. Cette isolation s'applique également aux niveaux de privilèges supérieurs. Ceux-ci ne sont pas censés avoir accès aux informations propres à une enclave.

Dans le modèle d'Intel SGX, le code de chaque enclave est vérifié avant d'être exécuté. Pour cela, des instructions indiquent au matériel de vérifier l'intégrité de son code. L'exécution d'une enclave n'est finalement lancée par un moniteur que si la vérification est valide.

Grâce à cette nouvelle notion de domaine de sécurité, Intel SGX ajoute également le support de nouvelles politiques de sécurité. Par exemple, le code est chiffré en mémoire avec une clé différente pour chaque enclave. Les valeurs déchiffrées ne restent que dans les premiers niveaux de cache durant l'exécution de l'enclave. Cela permet notamment à un attaquant ayant accès au contenu de la mémoire principale d'être incapable de retrouver les informations de l'enclave. Cette stratégie bien connue offre des garanties de confidentialités à l'utilisateur.

Concernant l'isolation microarchitecturale, de nombreuses attaques ont montré que les implémentations d'Intel SGX étaient peu adaptées

face à ce genre de menace. Plusieurs exemples d'attaques exploitant le partage de ressources ou l'exécution transitoire existent. Même si nous avons vu que ce n'était pas le scénario idéal, on peut supposer que ces faiblesses puissent être corrigées par le logiciel grâce à des mises à jour du microcode. En revanche, l'organisation architecturale d'Intel SGX pose problème et ne peut être modifiée si facilement. Si ce système offre une certaine horizontalité au niveau de privilège le plus faible, il manque de flexibilité aux autres niveaux. Notamment, le système d'exploitation reste responsable de la gestion des interruptions et de certaines exceptions au sein de l'enclave. Cela mène donc à la possibilité d'attaques par canaux contrôlés comme vu dans le chapitre 3.

Propositions académiques d'ISA Des solutions académiques ont également été proposées pour l'ajout de domaines de sécurité au niveau de l'ISA. Dans ce but, une ISA libre et ouverte comme RISC-V est intéressante.

CURE [110] permet la création d'enclaves à différents niveaux de privilèges. Le moniteur sécurisé est lui-même une enclave fixée au plus haut niveau de privilège. Une certaine horizontalité est ainsi permise à tous les niveaux. Pour l'isolation microarchitecturale, le moniteur sécurisé reste responsable de la gestion des ressources. Il doit donc désactiver le SMT, vider les ressources et les partitionner si nécessaire. Il est également responsable de rediriger les exceptions et interruptions si nécessaires, pour éviter les attaques par canal contrôlé. Au sein du système, l'affiliation à une enclave est effectuée en propageant un identifiant local en même temps que les données et les opérations.

Alternatives logicielles À défaut de disposer de suffisamment d'horizontalité grâce à l'ISA, le logiciel peut donc essayer de distinguer des domaines au niveau logiciel. Si l'on considère le cas de l'isolation microarchitecturale, cela peut être fait en effectuant les opérations nécessaires (*e.g.* une opération de flush) entre deux exécutions supposées indépendantes. Même si elles sont à l'origine placées au même niveau de privilège, elles seront isolées par le logiciel l'une de l'autre.

Avec un objectif semblable à Intel SGX, Sanctum [98] permet la gestion d'enclaves isolées du reste du logiciel. Ces enclaves, seulement avec le niveau de privilège minimal, sont gérées avec un moniteur sécurisé placé au niveau de privilège le plus haut. Pour permettre cette nouvelle hiérarchie des domaines de sécurité, de nouveaux registres dédiés au support des enclaves sont ajoutés. Par exemple, chacune d'entre elles dispose de sa propre table des pages de base en plus de celle fournie par le système d'exploitation. Ainsi, les informations fournies par le moniteur de sécurité sont séparées de celles du système d'exploitation. Le moniteur sécurisé est responsable d'effectuer le partitionnement et l'effacement des traces pour isoler les enclaves.

KeyStone [140] est une alternative purement logicielle et basée sur la spécification RISC-V. Là encore, l'idée est d'avoir un moniteur sécurisé responsable de la gestion des enclaves et placé au niveau de privilège le plus élevé. Ce moniteur est également responsable de vider les ressources partagées et de leur partitionnement.

Comme discuté dans le chapitre 4, ces solutions logicielles sont donc dépendantes des outils fournis par l'ISA pour être efficaces. Par exemple, l'ISA RISC-V ne dispose pas à l'origine de mécanisme dédié

pour des opérations de flush ou de partitionnement. Ces solutions logicielles doivent donc trouver des alternatives dans l'ISA pour émuler ces opérations (*e.g.* en écrasant les données plutôt qu'en les invalidant). On retrouve alors le cas du logiciel devant s'adapter au matériel qui l'exécute.

Limites Les solutions proposées visent à conserver les privilèges tout en rendant la hiérarchie plus flexible. L'horizontalité est permise avec la possibilité de créer plusieurs enclaves à un même niveau de privilège. Cependant, elle reste limitée selon les niveaux de privilèges souhaités. Aucune solution ne permet par exemple de créer plusieurs moniteurs sécurisés au niveau de privilège maximal. Il n'est donc pas possible d'isoler entièrement des groupes d'applications bien différents. Des alternatives logicielles existent, mais elles ne s'appuient que sur une émulation des domaines de sécurité. Le logiciel est alors complètement responsable de la gestion, mais aussi de l'application des politiques de sécurité.

Les privilèges ayant des droits fixés, la verticalité est également peu adaptable. Sur cet aspect-là, le logiciel ne dispose que de certains mécanismes spécifiques comme la possibilité de rediriger ou non les exceptions. Les autres droits d'accès restent directement associés à des niveaux de privilèges.

Hiérarchie dynamique

Principes Les solutions proposées essayent de se rapprocher d'une hiérarchie entièrement dynamique. Dans un modèle idéal, la configuration des domaines (aussi bien leurs droits que leurs autres politiques de sécurité) et leur organisation seraient entièrement modifiables par le logiciel. Ce dernier bénéficierait d'une liberté totale aussi bien pour la verticalité que l'horizontalité.

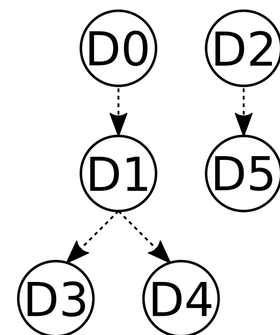


Figure 5.3. – Hiérarchie dynamique des domaines de sécurité. Les flèches pointillées sont des dépendances modifiables entre deux domaines. Ce modèle est un modèle idéal et complètement flexible pour le logiciel.

Une hiérarchie réellement dynamique doit donc pouvoir prendre n'importe quelle forme. Premièrement, le nombre de domaines existants doit être suffisamment grand. Leurs dépendances (qui constituent la hiérarchie) doivent également être modifiables. Par exemple, il doit être possible comme sur la Figure 5.3 d'avoir des branches complètement indépendantes, impliquant la mise en place de nouveaux mécanismes.

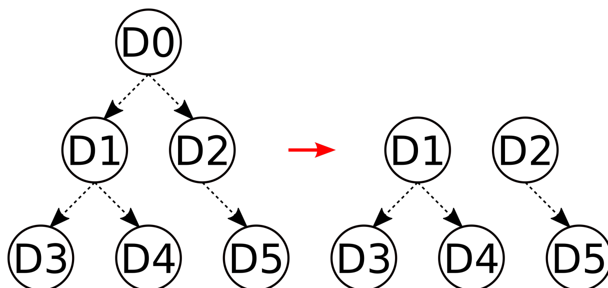


Figure 5.4. – Opération de séparation sur la hiérarchie. L'opération de séparation permet de rendre indépendants des domaines. Ainsi, ils n'ont plus aucun lien : contrôlé l'un ne permet pas à un attaquant d'atteindre l'autre.

Une hiérarchie dynamique suppose la possibilité de séparer des domaines comme sur la Figure 5.4. Ceux-ci appartiennent alors à des branches différentes de la hiérarchie et n'ont aucun contrôle les uns sur les autres.

En cas d'erreur ou de retour en arrière, il peut être utile pour le logiciel d'annuler une opération. L'existence d'un mécanisme de séparation implique donc la création d'un mécanisme de fusion comme

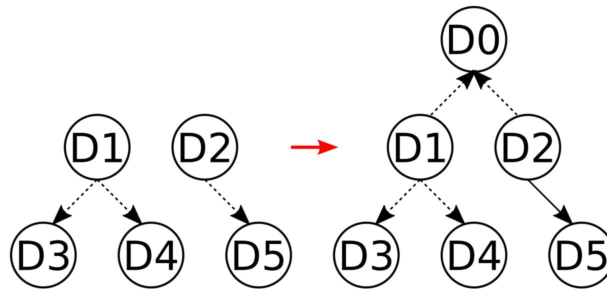


FIGURE 5.5. – Opération de fusion sur la hiérarchie. Pour rendre la hiérarchie réversible, il est utile d'avoir une opération inverse à la séparation : la fusion. Cela permet de recréer un lien de dépendance.

sur la Figure 5.5. En cas de séparation devenue inutile, le logiciel a ainsi la possibilité de simplifier la hiérarchie. La hiérarchie dynamique serait sinon fragmentée au cours du temps : tous les domaines seraient indépendants sans possibilité de retour en arrière.

Notes : La fusion n'est pas une opération essentielle. En cas de problème, on pourrait simplement envisager un redémarrage du système pour retrouver la hiérarchie de base. Cette opération offre simplement plus de flexibilité au logiciel. Cependant, comme toute opération, elle doit être implémentée afin de ne pas représenter une potentielle faille de sécurité.

Limites Une solution complètement dynamique implique de se passer des privilèges statiques. Un tel système marque une rupture complète avec les systèmes existants. Ainsi, à notre connaissance, aucune proposition d'ISA ne met en place ce genre de mécanisme.

Finalement, un passage d'un système statique à un système dynamique revient à rendre explicites et modifiables les domaines. Aucun domaine n'étant fixé, chacun doit également être configuré explicitement. Dans le cas statique, on sait dans tous les cas les droits correspondants aux privilèges. Des simplifications sont donc possibles au niveau de l'implémentation. Dans un cas dynamique (et même hybride dans une certaine mesure), un surcoût est nécessaire pour être capable de spécifier chaque domaine individuellement.

5.3. Politique d'isolation microarchitecturale dans l'ISA

Un domaine de sécurité est défini par ses politiques de sécurité. Ainsi, la manière de mettre en place ces politiques influe directement sur la création des domaines de sécurité. Pour être efficaces, ces politiques doivent être supervisées par le logiciel et implémentées par le matériel. L'ISA doit donc être modifiée pour intégrer cette possibilité.

Modifier l'ISA peut cependant être fait de différentes manières pour intégrer une politique d'isolation microarchitecturale. Chacune va influencer différemment le matériel et le logiciel comme nous l'avons évoqué à la fin du chapitre 4. Nous allons à présent voir quelles formes ces modifications peuvent prendre, et comment l'abstraction de l'ISA peut être prise en compte.

Forme des modifications de l'ISA

Les modifications de l'ISA peuvent être regroupées en deux grandes catégories.

Barrières La première catégorie est l'ajout d'instructions permettant la création de *barrières*. Une barrière correspond à un changement de politique de sécurité, et donc de domaine, lors de l'exécution. C'est une approche visant à distinguer deux domaines exécutés successivement

sans mémoriser d'état dans un registre (une barrière est une approche dite *sans état*). De nombreuses solutions présentées dans le chapitre 4 entrent dans cette catégorie. Les instructions de flush comme `purge` [99] ou `fence.t` [102, 103] séparent les états permanents de l'ancien domaine du nouveau. Les nouvelles opérations IBPB et IBRS [95, 101] constituent également des barrières, mais ici au niveau de la spéculation. L'ancien domaine exécuté n'est ainsi plus capable d'influer sur la spéculation du nouveau domaine.

Les barrières sont une méthode intéressante pour créer temporellement une politique d'isolation microarchitecturale. Elles permettent de limiter le partage d'états permanents entre des domaines successifs qui mènent à certaines fuites. Cependant, nous avons vu dans le chapitre 3 que les états permanents peuvent également être utilisés simultanément. De plus, il existe l'état de contention dynamique qui peut être partagé. Pour ces différents cas, les barrières sont inefficaces et une autre approche est nécessaire.

Contextualisation La deuxième catégorie de modification s'appelle la *contextualisation*. Contrairement aux barrières, celle-ci vise à ajouter un état pour décrire l'environnement dans lequel a lieu l'exécution. Ainsi, chaque opération ou donnée est associée à cet état et se propage avec lui dans l'ensemble de la microarchitecture. À tout moment, il est donc possible de savoir à qui appartient une information et si elle peut être partagée. Surtout, ce fonctionnement s'applique aussi bien à des domaines exécutés successivement que simultanément : l'ensemble du problème d'isolation peut être considéré.

Ce genre de modification est particulièrement utilisée par les solutions de partitionnement [70, 99, 110]. Déclencher une opération de barrière matérielle au moindre changement de contexte [110] est une forme de contextualisation. De manière générale, c'est également le principe mis en place par les privilèges. Le niveau actif définit les opérations qui peuvent être réalisées ou non. Le matériel s'adapte alors aux contextes exécutés.

Stratégies d'abstraction du matériel

L'ISA est conçue avec un rôle d'abstraction. Le logiciel n'a pas besoin de connaître en détail la constitution de la microarchitecture et doit pouvoir se contenter des éléments fournis par l'ISA. Cependant, cela mène également à la situation actuelle où les fuites d'informations microarchitecturales apparaissent dans des mécanismes non visibles par le logiciel. Une question se pose alors : faut-il conserver ce rôle d'abstraction ? Deux grandes approches s'opposent.

Gestion de la microarchitecture Une première approche vise à rendre possible une gestion fine de la microarchitecture par le logiciel. Pour cela, chaque état partagé au niveau de la microarchitecture doit être visible au niveau de l'ISA. Par exemple, les lignes de caches ou les tables de spéculation [90, 102] doivent être accessibles. L'ISA donne des méthodes pour gérer ces états (*e.g.* effacement et partitionnement). On peut alors parler d'une *ISA augmentée* [91].

Pour la sécurité, cette approche est particulièrement intéressante. Elle permet d'avoir un contrôle fin du matériel en adaptant le logiciel. En cas de faille détectée, si les mécanismes correspondants sont gérables en

utilisant l'ISA, il est alors possible de modifier le logiciel. De plus, celui-ci est libre d'appliquer les opérations qu'il souhaite dépendamment des domaines.

Cette approche a cependant des inconvénients. Tout d'abord, modifier l'ISA de cette manière revient à complètement briser l'abstraction qu'elle offre. Les mécanismes implémentés variant selon les microarchitectures, le logiciel devrait s'adapter aux différentes possibilités. Celui-ci ne serait alors plus portable. Une ISA trop précise revient également à limiter les choix de conception matérielle. Ceux-ci seront directement guidés par les états qui peuvent/doivent être visibles et gérables. Enfin, l'accès à de telles instructions de gestion doit être restreint. Sinon, on peut envisager qu'elles soient exploitées par un attaquant pour mieux contrôler les informations au niveau du matériel. Un exemple clair est le cas de l'instruction x86 `cflush`. Celle-ci permet d'évincer des lignes de cache sans privilège. Des attaques comme Flush+Reload [28, 29] ou Flush+Flush [35] en tirent directement profit.

Abstraction de la microarchitecture Une seconde approche vise à conserver l'abstraction tout en offrant des primitives de sécurité. Ce type de solutions applique donc le fonctionnement classique de l'ISA à la sécurité. Il permet au logiciel d'indiquer ce qu'il veut et laisse le matériel libre pour la réalisation.

Plusieurs solutions proposées découlent de ce postulat. Par exemple, la nouvelle instruction `purge` pour MI6 [99] permet d'effacer toutes les traces persistantes. Celle-ci est complètement abstraite : le matériel est responsable de savoir quelles ressources sont concernées. ConTExT [141] est également une approche abstraite. En indiquant pour chaque page mémoire si elle peut être exécutée/ utilisée spéculativement, le matériel est libre de la manière d'utiliser cette information. Il doit simplement s'assurer qu'aucune fuite due à de l'exécution transitoire n'est possible.

Ces solutions montrent bien la possibilité de lier abstraction et sécurité. Cependant, dans leur approche, elles ne considèrent que certaines des ressources partagées susceptibles de mener à des fuites microarchitecturales. En ciblant certains éléments de la microarchitecture, elles manquent finalement l'objectif principal. Celui-ci n'est pas d'être capable de gérer les mécanismes matériels un à un. Une telle approche mène à un empilement des contremesures, avec une complexité accrue aussi bien côté matériel que logiciel. L'objectif réel est un nouveau contrat formel logiciel/matériel, selon lequel le premier indique ses besoins indépendamment de tout élément microarchitectural et le second les applique.

5.4. Conclusion

Dans ce chapitre, nous avons vu les approches existantes pour modifier l'ISA et permettre des échanges d'informations entre le logiciel et le matériel. Les notions de domaine de sécurité et de politique de sécurité sont utilisées pour définir les garanties mises en place. On définit un domaine comme un endroit où sont appliquées des politiques de sécurité. Ces dernières établissent un ensemble de critères pour la mise en place de services de sécurité. Les droits d'accès à des ressources représentent une forme de politique de sécurité. De même, une politique d'isolation microarchitecturale met en place les mécanismes nécessaires

pour empêcher de potentielles fuites microarchitecturales.

Les systèmes sont basés depuis de nombreuses années sur un principe de privilège, qui sont des domaines statiques. Les privilèges sont essentiellement conçus pour restreindre les droits d'accès à des ressources. Par exemple, seuls certains niveaux peuvent gérer les exceptions ou la mémoire virtuelle. Aucune autre politique n'est strictement appliquée à ce niveau-là : les privilèges ne permettent pas une gestion assez fine. Ceci explique les problèmes d'isolation entre les privilèges : ils sont la base de sécurité tout en étant paradoxalement peu protégés.

En réponse à ces problématiques et au besoin de flexibilité du logiciel, de nouveaux domaines sont implémentés grâce à des hiérarchies hybrides. Cependant, même avec eux, les possibilités offertes restent limitées. Le nombre de niveaux de privilège (la verticalité) ou le nombre de domaines à un même niveau (l'horizontalité) sont souvent restreints.

Passer à un modèle dynamique serait un changement radical : les systèmes en seraient profondément bouleversés. Cependant, il serait alors possible de couvrir les cas d'usage actuels ainsi que ceux qui apparaîtront dans le futur. Pour la suite de ces travaux, notre objectif est de définir une ISA intégrant pleinement les besoins de sécurité. Le logiciel doit être libre de spécifier et paramétrer les domaines de sécurités ainsi que ses politiques appliquées. Dans le cas de l'isolation microarchitecturale, chaque programme doit pouvoir être distingué au niveau du matériel. De même, les droits attribués à chaque domaine doivent pouvoir être adaptés.

À partir de ces conclusions, il est possible de déduire des objectifs de conception. Une implémentation de domaines de sécurité devrait :

- Permettre un maximum de verticalité : le logiciel devrait être capable de choisir précisément les droits de chaque domaine sur le système.
- Permettre un maximum d'horizontalité : le logiciel doit pouvoir séparer des programmes qui ont les mêmes droits sur le système, mais sont indépendants.
- Permettre une indépendance entre les domaines : il doit être possible de choisir les dépendances entre chaque utilisateur d'un système.
- Renforcer les frontières entre les domaines : le passage d'un utilisateur à un autre doit être fait en respectant les politiques de sécurité de chacun. Aucune liberté ne doit être possible sur cet aspect-là par l'implémentation.
- Constituer une base pour n'importe quelle politique de sécurité : elle doit être utilisable pour la mise en place de droits d'accès à des ressources ou d'une politique d'isolation microarchitecturale (contrairement aux privilèges uniquement adaptés au premier cas).

À partir de cette base, l'ISA devra également intégrer une politique d'isolation microarchitecturale. Ces modifications devront :

- Considérer la problématique dans sa globalité. De ce fait, la simple mise en place de barrières n'est pas suffisante et la contextualisation sera donc privilégiée.
- Conserver l'abstraction offerte par l'ISA. Les modifications doivent être indépendantes de l'implémentation.

Pour rappel, nous visons une stratégie de conception des processeurs sur le long terme. Pour nos travaux, la compatibilité avec les systèmes existants (*e.g.* le support des privilèges) n'est donc pas une contrainte.

Domes 6.

Résumé du chapitre : Dans ce chapitre, nous modifions le jeu d'instructions RISC-V 32 bits non privilégié afin de définir de nouveaux domaines de sécurité. Pour cela, nous introduisons des privilèges dynamiques appelés domes. Nous définissons les nouveaux registres servant à représenter ces domaines ainsi que les instructions pour leur gestion. Nous discutons également de l'impact sur le logiciel et des changements entraînés par rapport aux modèles classiques. Enfin, nous établissons les possibilités d'extension de cette base pour de nouvelles politiques de sécurité.

6.1 Stratégie de modification de l'ISA 83
ISA de base : RISC-V 32 bits 83
Objectifs d'implémentation 83
6.2 Proposition d'implémentation 84
Modèles et représentation 84
Opérations et instructions 86
Utilisation des capacités : les exceptions 90
6.3 Support d'une politique d'isolation 92
6.4 Impact sur le logiciel . . . 93
Conception de la hiérarchie 93
Compilation 96
Scénario de `dome.switch` fréquents 96
6.5 Extension du modèle . . . 97
Accès aux compteurs de performances 97
Intégrité du flot d'exécution 97
Chiffrement et authentification de la mémoire . . . 98
Augmentation du nombre de domes 98
6.6 Version rétrocompatible avec les privilèges statiques 100
6.7 Conclusion 101

6.1. Stratégie de modification de l'ISA

Nous allons à présent justifier l'utilisation de l'ISA RISC-V comme base pour nos travaux. À la suite de cela, nous définissons des objectifs pour nos modifications.

ISA de base : RISC-V 32 bits

RISC-V est le nom d'un jeu d'instructions initialement développé en 2010 à l'Université de Berkeley. Cette architecture est libre et ouverte : ses spécifications [139, 142] sont accessibles librement sur internet. Initialement pensée pour la recherche et l'enseignement, elle est à présent utilisée dans de nombreux projets industriels [143, 144]. Une communauté, gérée par la fondation RISC-V, est responsable de son développement.

Comme son nom l'indique, le jeu d'instructions RISC-V est de type RISC. L'ensemble de ces instructions correspond donc à des opérations simples à interpréter par le matériel. Elle est également pensée pour être extensible. Ainsi, plusieurs extensions optionnelles peuvent être ajoutées à une base commune (elle-même disponible en deux versions 32 et 64 bits). Chacune de ces extensions définit un ensemble d'instructions offrant de nouvelles fonctionnalités. Au moment de la conception d'un processeur, les architectes sont donc libres d'utiliser la base et les extensions nécessaires. Aussi, il est possible d'ajouter des extensions dites "personnalisées" selon les besoins des projets. De la même manière, la spécification de base RISC-V [142] ne définit pas de privilège. Ainsi, une autre spécification optionnelle [139] définit les ajouts nécessaires pour un ensemble de privilèges statiques.

Ces différentes caractéristiques, en plus de l'intérêt toujours plus croissant et des nombreuses ressources documentaires disponibles, nous ont menés à utiliser cette ISA comme base pour nos travaux. Grâce à sa modularité, il est possible de partir d'une base simple et d'y ajouter les fonctionnalités voulues. Dans notre cas, nous avons donc choisi la base 32 bits appelée RV32I, sans le support des privilèges statiques. Cela implique également que seuls les CSR de base (les compteurs de performances) sont conservés.

Objectifs d'implémentation

De base, l'ISA RISC-V a été conçue selon certains objectifs et contraintes. Parmi eux, on note :

- La simplicité : le jeu d'instructions conditionne le développement du matériel et du logiciel. Une ISA simple permet de réduire la complexité des implémentations matérielles, leur vérification et limite également le coût final.
- Les performances : de par la simplicité de ces instructions, il est possible de réduire la période d'horloge ou le nombre de cycles nécessaires par instruction.
- L'abstraction : le jeu d'instructions est défini le plus indépendamment possible de toute implémentation. Les choix matériels sont laissés au concepteur.
- L'extensibilité : l'ISA doit pouvoir servir de base pour ajouter de nouvelles fonctionnalités.

Nos modifications doivent préserver autant que possible ces contraintes. Dans le cas des systèmes dédiés à la sécurité, il existe d'autres principes utilisés et compatibles avec ceux-ci. Ils permettent d'orienter nos choix lors de la définition du système. Certains sont directement en lien avec les réflexions menées dans les chapitres 4 et 5 :

- La simplicité : des systèmes plus simples à développer sont également plus faciles à analyser pour identifier de potentielles failles.
- Le principe du moindre privilège : n'importe quelle entité du système ne doit être capable de faire que ce pour quoi elle a été conçue, ni plus ni moins.
- La transparence : le système ne doit pas reposer sur son opacité (principe de la boîte noire).
- L'extensibilité : certaines notions, comme celle de domaine de sécurité, ne sont pas propres aux problématiques d'isolation. Les modifications effectuées doivent donc pouvoir être compatibles et réutilisées pour bâtir d'autres garanties de sécurité.

Enfin, les fonctionnalités fixées dans le chapitre 5, comme une hiérarchie dynamique des domaines, sont également un objectif majeur.

6.2. Proposition d'implémentation

Nous avons défini les différentes contraintes pour la modification de l'ISA. À présent, nous détaillons notre proposition de domaines de sécurité. Celle-ci doit être vue comme une extension de la base RV32I.

Modèles et représentation

On appelle un *dome* un domaine de sécurité (ou privilège) dynamique définit au niveau de l'ISA. Un dome est visible et géré par le logiciel. Ses propriétés sont assurées et garanties par le matériel. Pour cela, une représentation commune au logiciel et au matériel doit être définie au niveau de l'ISA.

Identifiant Un dome est caractérisé par une valeur qui lui est propre appelée *identifiant*. Deux identifiants différents correspondent à deux domes distincts. Dans ce modèle simple, notre ISA de base étant sur 32 bits, on définit la possibilité d'avoir 32 identifiants différents. Ce choix permet l'utilisation d'une table comme celle décrite dans le prochain paragraphe. Chaque identifiant est encodé sur $\log_2(32) = 5$ bits.

Table Pour établir une hiérarchie entre les domes, il est nécessaire de pouvoir les lier. Pour chaque `dome_0`, on doit donc pouvoir savoir s'il contrôle un `dome_1`, si le `dome_1` le contrôle ou s'ils sont simplement indépendants. On associe pour cela à l'identifiant une autre valeur appelée *table*. Cette table est une valeur sur 32 bits. Comme illustré sur la Figure 6.1, chaque bit de la table correspond à l'un des domes possibles (32 bits pour 32 identifiants différents, seuls 4 sont représentés ici). Il indique à chaque fois si le dome propriétaire de la table contrôle le `dome_i`, avec i le numéro du bit. Ainsi, uniquement à partir de ces deux informations, il est possible de décrire l'ensemble de la hiérarchie.

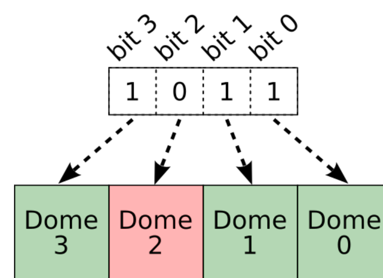


Figure 6.1. – Description de la table d'un dome. Chaque bit de la table décrit la possibilité de créer et gérer un dome avec l'identifiant associé. Ici, la table a les bits 0, 1 et 3 activés. Le dome propriétaire de cette table peut donc créer et gérer les domes avec les identifiants 0, 1 et 3. Par contre, il n'a pas l'autorisation de gérer le dome 2.

Point d'entrée L'appel d'un nouveau domaine d'exécution passe par un branchement vers une adresse précise. Ce domaine peut alors effectuer des vérifications et autres opérations pour s'assurer qu'il a été correctement appelé. Dans le cas classique des privilèges statiques, cela permet par exemple de savoir où rediriger le flot d'exécution (*e.g.* pour gérer une exception). Pour les domes, on associe donc à chaque identifiant et table une adresse appelée *point d'entrée*. Le lancement d'un dome passe par une redirection du flot de contrôle vers son point d'entrée.

Capacités Dans un système, chaque utilisateur ne doit pas être capable d'effectuer toutes les opérations possibles. D'après le principe du moindre privilège, il ne doit avoir que les droits qui lui sont nécessaires. Pour les privilèges statiques, cette notion est implicite : pour chaque niveau sont définis certains droits qui ne changent jamais. Dans le cas d'une organisation dynamique et adaptable par le logiciel, il doit être possible de définir précisément les droits de chacun des privilèges existants. Cette notion doit donc être explicite. Pour cela, chaque dome se voit également attribuer une valeur appelée *capacité*. Elle définit précisément le contrôle qu'a le dome correspondant sur le système.

Les domes sont avant tout des domaines de sécurité avec des politiques qui leur sont propres. De la même manière que pour les droits, les capacités sont utilisables pour définir les politiques de sécurité s'appliquant ou non. Un exemple d'utilisation des capacités est présenté dans la section 6.2. Comme illustré sur le Table 6.1, on retrouve deux types d'informations dans la valeur de capacités : les droits et les politiques de sécurité appliqués.

Table 6.1. – Description des capacités existantes. Le préfixe **sec-** correspond aux politiques de sécurité (bits 0 à 15) Le préfixe **fea-** correspond aux droits d'utiliser certains mécanismes (bits 16 à 31). Pour l'instant, seuls 2 bits sont utilisés et décrits ultérieurement (bit 0 et bit 16).

31 – 17	16	15 – 1	0
unused	mie	*unused*	exc
sec		fea	

Instance Avant le lancement de l'exécution d'un dome, il peut être intéressant de transmettre des informations supplémentaires au matériel. La valeur d'*instance* est prévue à cet effet. Elle peut être vue comme un argument transmis au matériel.

L'instance doit être vue comme une valeur optionnelle. Elle n'aide pas à décrire le dome lui-même, mais donne seulement des informations pour optimiser sa prochaine exécution (sa prochaine "instance"). Par exemple, nous verrons dans le cadre d'une politique d'isolation qu'il peut être intéressant d'indiquer le type de ressources utilisées. Il est tout à fait possible d'envisager de supprimer cette valeur (en mettant une valeur fixe en matériel) pour simplifier l'implémentation.

Table 6.2. – Détail des champs composant une configuration de dome. Chacun de ces champs est décrit sur 32 bits. L'offset permet de sélectionner un des champs.

offset	31 – 0
0x00	statut
0x01	identifiant
0x02	entrée
0x03	table
0x04	capacités
0x70	instance

Configuration L'ensemble des valeurs définissant un dome (son identifiant, sa table, son point d'entrée, ses capacités et l'instance) constitue une

structure appelée *configuration*. L'ensemble est résumé sur le Table 6.2. Chaque valeur détaillée précédemment représente un champ de la configuration qui peut être stocké dans des registres dédiés.

Pour définir l'état de chacune des configurations, des bits dits de *statut* leur sont également associés. Ils sont présentés sur le Table 6.3. Pour l'instant, 3 bits sont utilisés :

0. le bit V qui indique si la configuration est valide,
1. le bit L qui indique si la configuration est bloquée (et ne peut donc être modifiée),
2. le bit U qui indique si la configuration est en cours de modification. Nous verrons dans la section 6.2 qu'il est utilisé pour la mise en place de fusion de domes dans la hiérarchie.

Table 6.3. – Description des informations de statut. Les 3 bits V, L et U décrivent l'état de la configuration. Certains bits sont réservés pour une utilisation ultérieure. Les autres sont simplement inutilisés.

31 29	26 3	2	1	0
res.	*unused*	U	L	V

Opérations et instructions

À partir de ces configurations, il nous est possible de décrire à tout moment les domes existants. Nous allons à présent voir comment sont modifiées et gérées ces différentes configurations.

Registres des configurations

Les différentes configurations actives sont stockées dans des registres dédiés. Ils sont directement utilisés par les instructions opérant sur les domes, à l'image des GPR pour les entiers ou des registres contenant les nombres flottants.

Chaque configuration comporte pas moins de 6 valeurs sur 32 bits ($6 * 32 = 192$ bits)[1]. Mémoriser 5 configurations coûte donc presque autant que les 32 GPR, respectivement $5 * 192 = 960$ bits et $31 * 32 = 992$ bits. Ce coût en registres peut être rédhibitoire selon les contraintes lors de l'implémentation. Ainsi, le nombre de configurations disponibles est laissé libre pour chaque implémentation. Généralement, un système minimaliste pourra se contenter de deux configurations (une pour le dome en cours d'exécution, une autre pour paramétrer le prochain à exécuter) quand un système complexe pourra en contenir plus.

1: Cette valeur est vraie dans le cas où chaque bit d'une configuration serait réellement utilisé. Il faudrait alors tous les stocker dans des registres. Dans la version actuelle des domes, moins d'une centaine de bits sont réellement nécessaires. Des simplifications de l'implémentation sont donc possibles : certains bits peuvent être connectés à 0 par défaut (*e.g.* les *unused*). Cette organisation de configuration est cependant utile pour faciliter la distinction des types de champs (aussi bien en matériel qu'en logiciel). Elle facilite également d'éventuels ajouts.

Une alternative aux registres pourrait être de sauvegarder les configurations en mémoire. Cependant, sans nouveau mécanisme pour la protéger, la mémoire doit être considérée comme non sécurisée par défaut et non fiable. Par exemple, il faudrait séparer les adresses mémoires utilisées pour les domes de celles des autres opérations. Sinon, un utilisateur pourrait librement modifier les configurations des domes directement en mémoire. Une solution technique pourrait être d'utiliser des mécanismes de chiffrement et d'intégrité basés sur la cryptographie. Mais là encore, cela mènerait à un jeu d'instructions bien plus complexe à implémenter, loin de la simplicité visée. Le "ticket d'entrée" pour disposer uniquement de domaines de sécurité, sans même considérer les politiques de sécurité à implémenter, serait trop important.

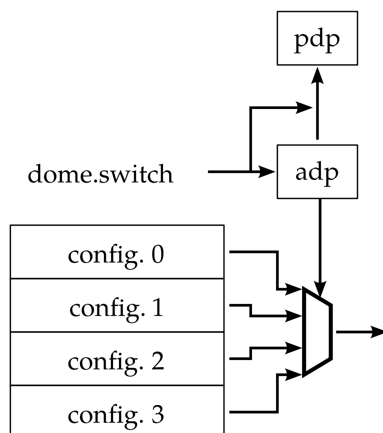


Figure 6.2. – Opérations sur les CSRs lors d'un `dome.switch` (modèle hybride).

Chaque cœur doit donc avoir ses propres registres de configuration. À tout moment, un seul dome est réellement exécuté par le processeur : c'est ce qu'on appelle le dome actif. Un registre dédié contient le numéro de la configuration décrivant le dome actif : c'est le *active dome pointer (adp)* (pour "pointeur du dome actif"). De la même manière, un registre contient également l'ancien dome exécuté : c'est le *previous dome pointer (pdp)* (pour "pointeur du précédent dome"). Ces registres constituent de nouveaux CSR en lecture seulement et accessibles par les instructions

dédiées. Le fonctionnement est décrit sur la Figure 6.2.

Statut et restrictions sur les modifications

Avant de détailler les instructions pour modifier les configurations, il est nécessaire de définir l'impact des bits de statut, mais aussi des restrictions sur les modifications possibles.

Les bits de statut représentent l'état d'une configuration. Ils ne sont modifiables que par le biais d'instructions dédiées. Les différents états possibles sont décrits dans le Table 6.4. Une configuration valide est une configuration dont tous les champs ont été vérifiés et qui est prête à être exécutée. Modifier un de ses champs entraîne son invalidation (V = 0). Une configuration valide et bloquée est seulement accessible en lecture. C'est toujours le cas de la configuration en cours d'exécution, pointée par le registre `adp`. Cet état permet de s'assurer qu'une configuration ne sera pas altérée même après changement du dome actif. Une configuration libre peut voir ses champs modifiés sans contrainte (hors bits de statut). Enfin, le dernier état possible est celui de "mise à jour".

Une configuration dans ce dernier état est considérée comme valide, mais encore modifiable. Des vérifications sont donc effectuées à chaque fois qu'un de ses champs est modifié. Elles assurent que le dome actif ne donne pas plus de pouvoir à un autre dome que ce qu'il en a lui-même. Notamment, un dome ne doit pas donner accès à des droits (*via* les bits de capacité) ou à la gestion d'autres domes (*via* les bits de la table) sans en avoir les droits. Ainsi, la vérification consiste à s'assurer que tout bit nouvellement mis à 1 dans la table ou les droits d'une configuration est également à 1 dans la configuration active. Sinon, la configuration modifiée est considérée comme non valide et son état est mis libre.

Cet état sert à fusionner les droits de plusieurs domes. Une fois mis à jour, les différents domes exécutés pourront à tour de rôle rajouter certains des droits qu'ils possèdent. Nous verrons par la suite des scénarios illustrant ce mécanisme.

Table 6.4. – Différents états d'une configuration de dome. Ils sont définis en fonction des valeurs des bits de statut V, L, et U.

V	L	U	État
1	1	X	Valide et bloquée
1	0	X	Valide
0	X	1	Mise à jour
0	X	0	Libre

Table 6.5. – Liste des instructions arithmétiques et logiques sur les domes. Chaque instruction est encodée sur 32 bits. `imm` correspond à une valeur immédiate. `rs` correspond à un registre d'entier source. `rd` correspond à un registre d'entier destination. `rs[confs]` correspond à un registre d'entier source contenant le numéro de la configuration utilisée. `offset` permet de sélectionner le champ d'une des configurations.

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
imm[6 :0]	rs2	rs1[confs1]	func3	rd	opcode	
7	5	5	3	5	7	
offset[6 :0]	rs2	rs1[confs1]	SET	rd	DOME-OP	`dome.set`
offset[6 :0]	rs2	rs1[confs1]	CLEAR	rd	DOME-OP	`dome.clear`
xxxxxxx	rs2[confs2]	rs1[confs1]	MV	rd	DOME-OP	`dome.mv`
offset[6 :0]	rs2	rs1[confs1]	CMV	rd	DOME-OP	`dome.cmv`
offset[6 :0]	xxxxx	rs1[confs1]	IMV	rd	DOME-OP	`dome.imv`

Opérations arithmétiques et logiques

De nouvelles instructions sont nécessaires pour accéder et modifier ces configurations. Elles sont détaillées sur le Table 6.5 en reprenant le modèle des spécifications RISC-V [139, 142].

Tout d'abord, il faut préciser que les configurations sont sélectionnées par un adressage indirect représenté par `rs[confs]`. Ainsi, le numéro

de configuration `confs` est stocké dans un registre entier `rs`, et non pas directement dans l'instruction. L'avantage principal de cet adressage est sa flexibilité. Selon les valeurs des bits de statut, nous allons voir que le résultat des opérations peut différer. Le logiciel peut ainsi facilement parcourir l'ensemble des configurations. Il doit pour cela modifier la valeur du registre `rs1` avant de réexécuter l'instruction. Cet adressage permet également au logiciel de s'adapter au nombre de configurations disponibles, qui peut différer selon les implémentations.

Les instructions `dome.set` et `dome.clear` permettent d'appliquer un masque sur le champ `offset` de la configuration `confs1`. Ce masque met les bits correspondant respectivement à 1 ou à 0. Ces instructions sont particulièrement utiles pour la gestion des capacités ou de la table, où chaque bit a une signification propre. L'instruction `dome.cmv` permet de transférer une valeur d'un registre entier `rs2` vers un champ d'une configuration `confs1`. À l'inverse, `dome.imv` permet de transférer une valeur d'un champ de `confs1` vers un registre d'entier `rd`. Enfin, `dome.mv` permet de copier l'ensemble d'une configuration `confs2` vers `confs1`.

Les instructions `dome.set`, `dome.clear` et `dome.cmv` ne modifient qu'un seul champ. Cependant, elles ne sont pas autorisées à directement modifier les bits de statut. Seul le matériel, si une configuration valide est modifiée (ou si une configuration mise à jour est modifiée illégalement), peut en conséquence changer les bits correspondants. Le cas de `dome.mv` est lui différent : en copiant l'intégralité d'une configuration, il ne modifie pas sa validité. Ainsi, les bits V et U sont conservés quand le bit L est toujours mis à 0. La valeur retournée vers rd (sauf pour `dome.imv`) indique si la modification souhaitée a pu être effectuée : 0 en cas de succès et une autre valeur sinon.

On notera donc qu'à la différence des instructions RISC-V de base, la destination n'est ici pas explicite. La configuration de destination est ici la même que la configuration source `confs1`. Ce choix permet de simplifier l'implémentation. La modification de la configuration de destination étant dépendante de ses bits de statut, il est nécessaire de la lire avant de la modifier. Si la destination et la source étaient distinctes, il faudrait alors effectuer deux lectures simultanément pour chaque instruction. Ici, les deux sont confondues. Cela permet également de libérer des bits de l'instruction pour retourner un entier vers `rd` indiquant le statut d'une opération. Il est ainsi possible de savoir si une configuration a pu être modifiée ou non. De plus, on notera que ce format s'adapte parfaitement aux instructions qui appliquent des masques comme `dome.set` et `dome.clear`.

Table 6.6. – Liste des instructions mémoires sur les domes.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
imm[6 :0]	rs2	rs1[confs1]	func3	rd	opcode	
7	5	5	3	5	7	
imm[11 :5]	rs2	rs1[confs1]	LOAD	imm[4 :0]	DOME-OP	`dome.load`
imm[11 :5]	rs2	rs1[confs1]	STORE	imm[4 :0]	DOME-OP	`dome.store`

Opérations mémoire

Les configurations utilisées sont généralement sauvegardées dans des registres. Cependant, ces derniers ne sont présents qu'en nombre limité. Il peut être intéressant de stocker et récupérer des valeurs de

configurations depuis la mémoire.

L'instruction `dome.cmv` vue précédemment permet de transférer une valeur d'un registre entier vers un champ. Couplée avec une instruction `load` de l'ISA de base RISC-V, il est donc déjà possible de récupérer une configuration en mémoire champ par champ. Cette méthode fonctionne également pour les `store`. Cependant, comme illustré sur le Code 6.1, cela demanderait de nombreuses opérations pour charger une configuration entière. Ainsi, de nouvelles instructions ont été ajoutées pour permettre le chargement ou le stockage de l'intégralité d'une configuration. Le processeur est libre de les implémenter en un ou plusieurs accès successifs. Dans les cas d'alignement mémoire, le premier accès doit être aligné sur une adresse de 256 bits (puissance de 2 supérieure au nombre de bits d'une configuration complète).

Code 6.1: Chargement et copie d'une configuration champ par champ. Les adresses mémoires de la configuration et du registre sont stockées (l.1-2). Puis, chaque champ est chargé puis copié vers le registre de la configuration. L'identifiant est chargé en premier (l.3-4), puis l'entrée (l.5-6) *etc.*

```
1  la t0, conf_addr
2  li t1, 1
3  lw t2, 4(t0)
4  dome.cmv zero, t1, t2, 0x01
5  lw t2, 8(t0)
6  dome.cmv zero, t1, t2, 0x02
7  lw t2, 12(t0)
8  dome.cmv zero, t1, t2, 0x03
9  ...
10
```

L'instruction `dome.load` charge la configuration placée à l'adresse `rs2 + imm` dans `confs1`. L'instruction `dome.store` stocke la configuration `confs1` à l'adresse `rs2 + imm`. Les deux instructions sont détaillées sur la Table 6.6.

À nouveau, aucun prérequis de sécurité n'est supposé sur la mémoire. Ainsi, elle est considérée comme un élément non sécurisé du système. Toutes les valeurs chargées depuis la mémoire doivent donc être vérifiées avant d'être utilisées. Pour cela, les 3 bits de statut V, L et U sont toujours mis à 0 lors du chargement d'une configuration par un `dome.load`.

Table 6.7. – Liste des instructions de statut des domes.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
imm[6 :0]	rs2	rs1[confs1]	func3	rd	opcode	
7	5	5	3	5	7	
0000001	xxxxx	rs1[confs1]	CHECK	rd	DOME-OP	`dome.check.v`
0000100	xxxxx	rs1[confs1]	CHECK	rd	DOME-OP	`dome.check.u`
0000011	xxxxx	rs1[confs1]	CHECK	rd	DOME-OP	`dome.check.l`
0000000	xxxxx	rs1[confs1]	CHECK	rd	DOME-OP	`dome.check.c`

Opérations de gestion des domes

Les bits de statut ont un rôle central dans la définition des domes. Ils doivent être modifiés avec une attention particulière. De nouvelles instructions sont donc dédiées à cette tâche. Elles sont détaillées sur le Table 6.7.

Les modifications possibles de l'état d'une configuration par ces instructions sont détaillées sur la Figure 6.3. L'instruction `dome.check.v` vérifie une configuration pour la rendre valide. Tout d'abord, sa table et ses droits sont comparés à ceux de la configuration active. Cette vérification est donc similaire à celle faite pour l'état de mise à jour. En plus de cela, l'implémentation s'assure que la configuration cible est comprise dans la table de la configuration active. Autrement dit, le dome actif doit avoir le droit de gérer ce nouveau dome. Pour créer un dome avec un identifiant I, le dome actif doit donc avoir le bit I de sa table à 1. En cas de succès de l'ensemble de ces vérifications, le bit V de la configuration est mis à 1 et U et L à 0. Une configuration dont les V ou U sont déjà à 1 est directement considérée comme valide.

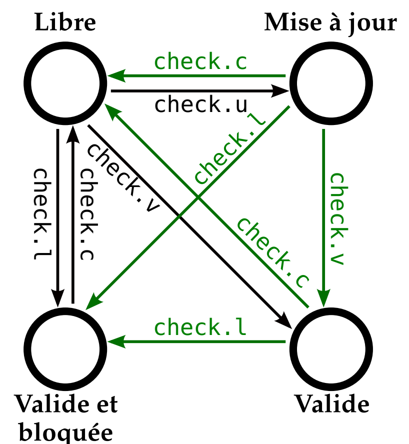


Figure 6.3. – Modifications de l'état d'une configuration avec `dome.check`. Les flèches vertes représentent les transitions où une vérification par le matériel n'est pas nécessaire.

Les deux autres instructions `dome.check.u` et `dome.check.l` effectuent des vérifications similaires. Pour `dome.check.u`, le bit U est mis à 1 en cas de succès, V et L à 0. Si de base, la configuration a ses bits V ou U

à 1, alors aucun changement ne s'opère. Pour `dome.check.l`, les bits V et L sont mis à 1 en cas de succès, U à 0. Une configuration dont les V ou U sont déjà à 1 est directement considérée comme valide et bloquée.

Enfin, `dome.check.c` remet les 3 bits à 0 en cas de succès. Dans le cas d'une configuration valide et bloquée, le changement ne s'opère que si le bit correspondant dans la table du dome actif est à 1. De plus, la configuration visée ne doit pas être celle active. `dome.check.c` est finalement le seul moyen de débloquer une configuration.

Changement du dome actif

Nous avons vu jusqu'à présent uniquement des instructions pour la modification des configurations. Or, une fois paramétrées, il est nécessaire de pouvoir les exécuter. Les instructions dédiées sont décrites dans le Table 6.8. Elles permettent de changer le dome en cours d'exécution.

Table 6.8. – Liste des instructions de switch des domes.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[6 :0]	rs2	rs1[confs1]	func3	rd	opcode	
7	5	5	3	5	7	
0000001	xxxxx	rs1[confs1]	SWITCH	rd	DOME	`dome.switch.v`
0000011	xxxxx	rs1[confs1]	SWITCH	rd	DOME	`dome.switch.l`
0000000	xxxxx	rs1[confs1]	SWITCH	rd	DOME	`dome.switch.c`

Le déroulement de ces instructions est décrit sur la Figure 6.4. Chacune d'elles effectue les mêmes vérifications que `dome.check.v`. De même, si la configuration a déjà un de ses bits U ou V à 1, alors elle est automatiquement considérée comme valide. En cas de succès, le flot d'exécution est redirigé vers le point d'entrée de la configuration `confs1`. Comme précédemment, la valeur retournée vers `rd` indique si l'opération peut avoir lieu : 0 en cas de succès sinon une autre valeur. Ensuite, le registre `adp` est mis à jour et contient le numéro de la configuration `confs1`. Le registre `pdp` récupère lui l'ancienne valeur de `adp`. Le dome décrit dans `confs1` devient donc le nouveau dome actif. Son état est maintenant valide et bloqué.

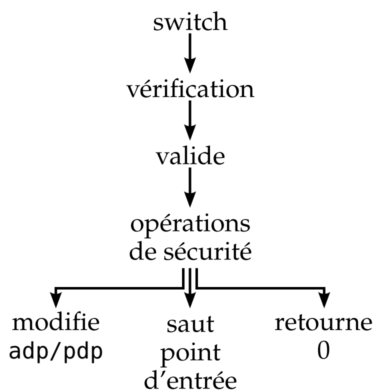


Figure 6.4. – Déroulement de l'exécution d'un `dome.switch`.

Les trois instructions `dome.switch.v`, `dome.switch.l` et `dome.switch.c` diffèrent dans leur gestion de l'ancien dome actif. `dome.switch.v` met son état à valide. Il pourra donc être conservé et relancé. `dome.switch.l` change son état à valide et bloqué. Cela signifie que sa configuration reste présente sur le système. Nous verrons que cela peut être utile pour la gestion de mécanismes comme les exceptions. Enfin, `dome.switch.c` modifie son état à libre. Ce dome a terminé ses exécutions et ne pourra plus être exécuté.

Du point de vue de la sécurité, ces instructions `dome.switch` sont essentielles. Elles marquent la rupture temporelle entre deux domaines de sécurité. Ainsi, c'est à ce moment-là, en cas de succès des vérifications, que doivent être effectuées les opérations nécessaires au lancement du nouveau dome.

Utilisation des capacités : les exceptions

Nous avons jusqu'à présent établi une base pour la mise en place des domes au niveau de l'ISA. Il est maintenant nécessaire de voir comment intégrer les mécanismes normalement gérés par les privilèges. L'un d'eux est la gestion des exceptions. Les privilèges étant peu nombreux

et statiques, ils disposent généralement de registres propres pour cette gestion. Cette organisation est cependant difficilement transposable au cas des domes. Notre hiérarchie est à présent dynamique et n'importe quel dome est susceptible de gérer les exceptions. Après une description simplifiée du mécanisme de gestion des privilèges dans l'ISA RISC-V, nous verrons comment transposer ce modèle statique à des domaines dynamiques.

Exceptions avec RISC-V

Pour rappel, on appelle une exception un évènement se déclenchant lors de l'exécution d'une instruction. Cela peut être dû à une instruction erronée, une faute de page, un appel système, *etc*. Des CSR dédiés sont prévus pour gérer ces évènements. Dans le cas de l'ISA RISC-V, chaque niveau de privilège a ses propres registres. On va considérer les registres pour le niveau M (Machine), le plus haut niveau de privilège. En voici une liste simplifiée :

- `mtvec` est paramétrable par le niveau M et sert à spécifier l'adresse de redirection pour l'exception.
- `mepc` est modifiée par l'implémentation et contient l'adresse de l'instruction ayant générée l'exception.
- `mcause` est modifiée par l'implémentation et contient un code décrivant l'exception rencontrée.
- `mtval` est modifiée par l'implémentation et donne plus d'informations sur l'exception (*e.g.* l'instruction erronée).

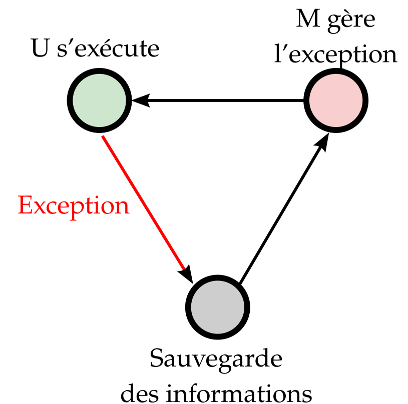


Figure 6.5. – Cycle de gestion d'une exception. U est le niveau utilisateur, M le niveau machine.

La gestion d'une exception est décrite sur la Figure 6.5. Son implémentation est illustrée sur la Figure 6.6. Elle passe par les étapes suivantes :

1. le niveau M configure `mepc` et laisse l'exécution à un autre niveau,
2. une exception est détectée,
3. `mepc`, `mcause` et `mtval` sont remplis avec les informations correspondantes par l'implémentation,
4. un changement de privilège est effectué (si le niveau en cours d'exécution est différent de M) avant un saut vers l'adresse dans `mtvec`,
5. le niveau M gère l'exception,
6. le niveau M enclenche la procédure de retour en modifiant le niveau de privilège et retournant vers l'adresse `mepc`.

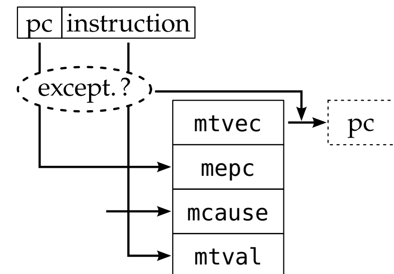


Figure 6.6. – Mécanisme d'exception dans l'ISA RISC-V. Lorsqu'une instruction génère une exception, des informations sont enregistrées. Dans le même temps, l'exécution est redirigée vers l'adresse contenue dans `mtvec`.

Dans cette situation, le rôle de chaque privilège est clair et défini. Le niveau M étant responsable des exceptions, alors le contrôle de l'exécution lui est directement confié.

Exceptions avec les domes

Dans le cas des domes, aucun n'est obligatoirement prédisposé à la gestion des exceptions. Ainsi, il est nécessaire de rendre cette information explicite.

Pour cela, un bit de capacité `fea-exc` est dédié dans le champ de capacité. Il correspond à la fonctionnalité "gestion des exceptions". Un dome avec ce bit à 1 est donc apte à gérer les exceptions en accédant aux registres correspondants. Dans notre cas, suivant le modèle RISC-V, on reprend les CSR `mtvec`, `mepc`, `mcause` et `mtval` qui deviennent respectivement `tvec`, `epc`, `cause` et `tval` (ils ne sont plus propres au

niveau M). On ajoute également un nouveau CSR appelé `excdome` en lecture seulement. Celui-ci contient le numéro de configuration du dome appelé en cas d'exception. Dès qu'un dome avec la capacité `fea-exc` modifie `tvec`, son numéro de configuration (contenu dans `adp`) est copié dans `excdome`. Ainsi, un dome peut seulement mettre son propre numéro de configuration dans ce registre. Il ne lui est donc pas possible de forcer une redirection erronée vers d'autres domes par le biais de `tvec`.

La gestion d'une exception est semblable au modèle de la figure Figure 6.5 : le dome avec `fea-exc` remplace le niveau M et un autre dome correspond au niveau U. La nouvelle implémentation est décrite sur la Figure 6.7. Les étapes nécessaires sont les suivantes :

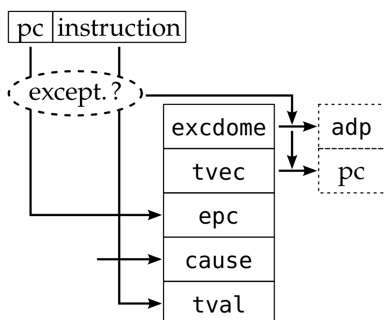


Figure 6.7. – Mécanisme d'exception avec les domes. Comme pour l'ISA RISC-V, des informations sont enregistrées lorsqu'une exception est détectée. Dans le même temps, le processeur force un `dome.switch` vers la configuration contenue dans `excdome`. Le point d'entrée utilisé est contenu dans `tvec`.

1. le dome avec `fea-exc` configure `epc` et `excdome`. Il utilise ensuite `dome.switch.l` pour s'assurer que sa configuration reste valide puis laisse l'exécution à un autre dome,
2. une exception est détectée,
3. `mepc`, `mcause` et `mtval` sont remplis avec les informations correspondantes par l'implémentation,
4. un changement de privilège est effectué (si le dome exécuté est différent de celui responsable des exceptions) avant un saut vers l'adresse dans `mtvec`. Cela revient à forcer un `dome.switch.v` vers la configuration indiquée par `excdome`. Le point d'entrée dans ce cas-là est `epc`,
5. le dome responsable gère l'exception,
6. le dome responsable enclenche la procédure de retour en effectuant un `dome.switch.l` vers la configuration pointée par `pdp`. Celui-ci sera responsable de rediriger l'exécution vers là où elle s'était interrompue.

Le fait d'effectuer des `dome.switch` à chaque fois rend le changement de domaine de sécurité explicite. Ainsi, les différents mécanismes pour garantir le respect des politiques de sécurité sont bien déclenchés.

6.3. Support d'une politique d'isolation

Nos changements actuels de l'ISA ne permettent pour le moment qu'un échange d'informations entre logiciel et matériel sur les domaines de sécurité utilisés. Le logiciel a la possibilité de répartir ses applications dans les différents domes, mais aucune garantie n'est encore assurée par le matériel. Il est nécessaire d'implémenter une politique de sécurité. Dans notre cas, nous nous intéressons à la mise en place d'une politique d'isolation. L'objectif est donc de déterminer quelles autres informations le logiciel peut transmettre au matériel et par quel moyen. L'implémentation de la politique elle-même est laissée libre au matériel. Cette problématique est étudiée dans les prochains chapitres 7, 8 et 9.

Utilisation de la politique d'isolation Une première information utile pour le matériel est de savoir si un domaine a besoin d'être isolé ou non. Les applications exécutées n'ont pas toutes les mêmes contraintes, aussi bien du point de vue des performances que des mécanismes matériels nécessaires ou que des garanties de sécurité exigées. Ainsi, une nouvelle capacité `sec-mie` (pour "isolation microarchitecture des exécutions") est ajoutée. Un dome avec le bit correspondant à 1 doit donc être isolé. Sinon, le matériel est libre d'effectuer les choix qu'il veut.

Indication des types d'instructions Un autre type d'information utile à transmettre au matériel correspond aux types d'instructions qu'effectuera le dome. Comme nous le verrons dans le chapitre 7, cela permet d'optimiser l'allocation des ressources. C'est particulièrement le cas dans des implémentations SMT où beaucoup de ressources sont partagées. Si certaines instructions sont essentielles pour toute exécution de dome (*e.g.* les instructions arithmétiques et logiques de base sur les entiers ou les branchements), d'autres dépendent des applications. On pense par exemple aux multiplications et divisions des entiers, aux instructions pour les nombres flottants, aux instructions cryptographiques, *etc*. Ces informations sont donc transmises par le biais de la valeur d'instance comme présentées sur la Table 6.9. Dans notre cas, le bit `md` indique la nécessité de pouvoir exécuter des instructions de multiplication et division.

Il est important de souligner que ces bits sont indépendants de l'implémentation et ne correspondent pas à des ressources précises. Ils ne font que donner au matériel une indication sur les instructions que voudra exécuter le dome. L'implémentation est donc libre d'interpréter ces informations pour allouer les ressources correspondantes. L'abstraction offerte par l'ISA est ainsi préservée.

Table 6.9. – Description des informations d'instance'. Deux bits sont utilisés pour indiquer le poids d'un dome et s'il utilise les instructions de multiplication et division.

31 … 3	2	1 0
unused	md	we

Indication du nombre de ressources Les implémentations contiennent parfois plusieurs instances d'une même ressource. Cela permet d'augmenter les performances d'exécution. Par exemple, un dome avec plus de lignes de cache sera susceptible de ramener plus de données, ce qui pourra accélérer son exécution. Ainsi, de la même manière que pour les types de ressources, il peut être intéressant de spécifier les besoins d'un dome au matériel. La valeur d'instance est donc utilisée pour transmettre une valeur appelée *poids* (ou *weight*) sur 2 bits. Un poids plus important signifie un besoin plus grand en ressources (*e.g.* pour améliorer les performances).

Ici encore, la valeur du poids ne correspond pas à un nombre précis de ressources. Il n'est qu'une indication pour le matériel et non une contrainte fixe. Cela n'impacte dont pas l'abstraction de l'ISA. Voici par exemple des interprétations possibles de chaque valeur de poids :

- 00 - le dome ne demande que le minimum de ressources pour chaque type,
- 01 - le dome ne demande qu'une répartition standard des ressources (*e.g.* pour N threads et R ressources, R/N ressources par thread),
- 10 - le dome demande toutes les ressources disponibles,
- 11 - le dome veut toutes les ressources, sinon le `dome.switch` est décliné.

6.4. Impact sur le logiciel

L'ajout des domes implique des changements au niveau du logiciel pour leur utilisation que nous allons à présent évoquer.

Conception de la hiérarchie

La conséquence directe des domes est la création d'une nouvelle hiérarchie des domaines de sécurité. Dans le cas statique, cette hiérarchie est fixe et donc définie dès le départ. Ici, le logiciel doit explicitement la

mettre en place. Nous décrivons les mécanismes qu'il peut utiliser et un exemple de scénario d'application.

Division et fusion des domes Dans le chapitre 5, deux mécanismes essentiels étaient évoqués pour constituer une hiérarchie dynamique : la division et la fusion des domaines. Pour rappel, la division permet de répartir les droits d'un domaine entre d'autres domaines, mais aussi l'influence sur le système. Si un des domaines est compromis, l'ensemble n'est pas forcément menacé. Cela permet également de créer des branches indépendantes de la hiérarchie. La fusion permet à plusieurs domaines de se rassembler en un seul. Une restructuration de la hiérarchie en cas de divisions excessives est ainsi possible : elle rend la hiérarchie réversible.

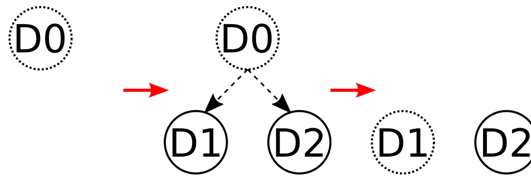


Figure 6.8. – Division d'un dome en logiciel. Le dome D0 répartit ses droits à D1 et D2 avant de se supprimer.

L'implémentation logicielle de la division des domes est illustrée sur la Figure 6.8. Elle s'appuie sur deux instructions : `dome.check.l` et `dome.switch.c`. Tout d'abord, le premier dome D0 prépare les configurations des autres domes D1 et D2. Une fois terminé, il utilise l'instruction `dome.check.l` pour les valider et les bloquer. Ainsi, cela garantit qu'ils ne pourront pas s'altérer mutuellement. Enfin, le dome actif D0 réalise un `dome.switch.c` vers un dome de son choix (ici D1). Sa propre configuration est donc invalidée : le dome D0 n'existe plus. Pour reproduire ultérieurement un dome avec des droits similaires, les nouveaux domes créés devront s'accorder pour réaliser une fusion.

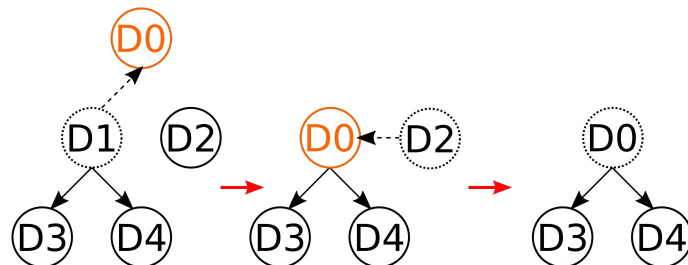


Figure 6.9. – Fusion d'un dome en logiciel. Les domes D1 et D2 donnent chacun leurs droits à D0 avant de se supprimer.

L'implémentation logicielle de la fusion des domes est illustrée sur la Figure 6.9. Ce cas est plus complexe, car il implique que plusieurs domes potentiellement indépendants s'accordent pour créer un dome commun. Ici, l'état de mise à jour d'une configuration joue un rôle clé. On considère le cas où D1 et D2 veulent rassembler leurs droits dans un dome D0. D1 commence par préparer une configuration libre et y ajoute les droits qu'il souhaite transmettre à D0. Ensuite, il effectue un `dome.check.u` sur cette nouvelle configuration : son état est "mise à jour". D1 effectue alors un `dome.switch` vers D2 pour lui laisser la main. À son tour, D2 transfère des droits à la configuration de D0 en modifiant cette configuration. Cette fois, le matériel vérifie à chaque fois que ces modifications sont valides. Finalement, une fois terminé, D2 réalise un `dome.switch` vers D0.

Exemple de scénario d'application Un cas d'application classique pour cette nouvelle hiérarchie dynamique est le scénario d'un démarrage sé-

curisé (ou *secure boot*). Le programme de démarrage est responsable de vérifier l'intégrité et l'authenticité des autres programmes avant leur lancement. Ceux-ci peuvent être des moniteurs responsables de l'exécution d'applications. Dans notre scénario, on considère un programme de démarrage responsable de lancer deux moniteurs : un pour des applications sécurisées et un autre pour des applications standards. Le premier moniteur doit donc être séparé du reste du système. Il est dans notre cas responsable de la gestion des exceptions des applications sécurisées. Il effectue également des opérations logicielles pour vérifier l'intégrité des applications qu'il gère. Le second moniteur gère lui les autres applications ainsi que leurs exceptions.

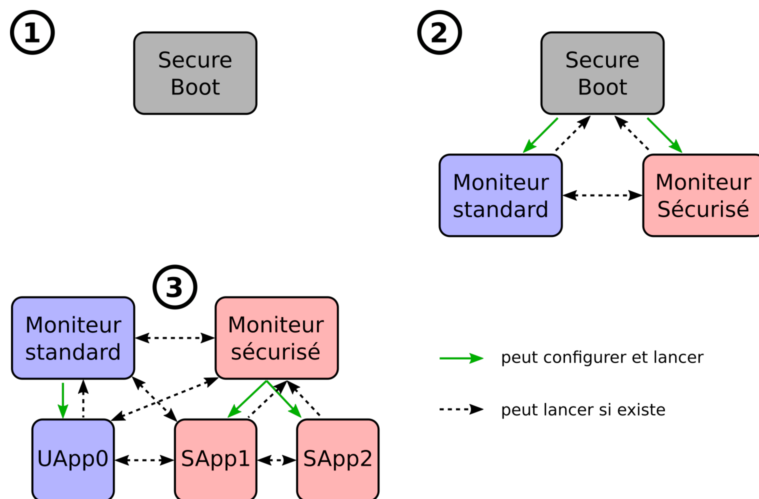


Figure 6.10. – Scénario d'application des domes : démarrage sécurisé. Le secure boot lance les différents moniteurs dans leurs propres domes. Chacun de ces moniteurs est ensuite responsable de la gestion de ses applications. Uapp correspond à des applications standards, SApp à des applications sécurisées.

Le scénario est décrit sur la Figure 6.10. Au départ, le programme de démarrage sécurisé est placé dans le dome par défaut (étape 1). Ce dome, dont la configuration est fixée par le matériel au démarrage, a tous les droits possibles. Cette condition est nécessaire pour pouvoir ensuite les transmettre, aucun dome n'ayant la possibilité de créer des droits. Le programme de démarrage sécurisé va alors préparer les configurations des domes pour les moniteurs (étape 2). Les deux ont des contraintes bien différentes : l'un est sécurisé et doit donc avoir des politiques de sécurité actives quand l'autre n'a aucun prérequis de sécurité. Le bit de capacité `sec-mie` (politique d'isolation microarchitecturale) est activé pour le dome du moniteur sécurisé. Les deux moniteurs ont également les droits nécessaires pour la gestion des exceptions, *via* le bit *fea-exc*. Après avoir vérifié l'intégrité (opérations effectuées librement par le logiciel) du code des moniteurs, le programme de démarrage lance le moniteur sécurisé. Ce dernier est à présent en cours d'exécution (étape 3). Il peut créer des domes selon les applications qu'il décide d'exécuter (tout en respectant sa table). Il paramètre également les registres correspondants pour la gestion des exceptions. Finalement, quand il le souhaite, il peut laisser la main au moniteur standard qui exécutera ses propres applications.

Un point important à souligner est la possibilité d'effectuer un `dome.switch` vers n'importe quelle configuration de dome valide (ou mis à jour). Ainsi, si la configuration de SApp1 est valide, rien n'empêche le moniteur standard d'effectuer un `dome.switch` vers ce dome. La seule garantie offerte par le matériel à ce niveau est que la configuration de SApp1 ne pourra être altérée par le moniteur standard (sauf si sa table lui permet, ce qui ici ne devrait pas être le cas). Il est donc de

la responsabilité de chaque application sécurisée SApp de vérifier au démarrage son environnement d'exécution (quel dome l'a lancé *via* `pdp`, quel dome gère les exceptions *via* `excdome` *etc.*).

Compilation

Modifier l'ISA implique d'adapter le compilateur pour un support complet. Lors de la génération de code, c'est lui qui sera donc capable d'utiliser les bonnes instructions et optimisations selon le code d'entrée.

Dans le cadre de ces travaux, la version RISC-V du compilateur gcc a été modifiée *a minima* pour le support des nouvelles instructions. Ainsi, aucune optimisation n'est réalisée et les blocs d'instructions utilisant les instructions pour les domes sont gérés manuellement. Des travaux supplémentaires sont nécessaires pour une automatisation complète.

Un nouvel aspect propre aux domes qui semble gérable par le compilateur est le type d'instructions utilisées (avec les bits d'instance). Ces informations dépendent complètement des contraintes indiquées au compilateur. Ainsi, il semble envisageable, lors de la compilation d'une application, de générer le masque correspondant pour indiquer les types des instructions générées. Avant de lancer un dome, il n'y aurait ainsi qu'à appliquer ce masque pour demander à allouer les ressources correspondantes. Dans le cas de l'ISA RISC-V, une organisation naturelle pourrait être d'attribuer un bit d'instance à chaque extension :

- `md` pour l'extension M des multiplications et divisions d'entiers,
- `fp` pour l'extension F de calculs sur les nombres flottants,
- *etc.*

Scénario de `dome.switch` fréquents

Comme nous le verrons dans les prochains chapitres, le renforcement et l'application stricte de la politique d'isolation a un coût. Une baisse des performances est observée à chaque changement de dome actif, donc durant l'exécution des `dome.switch`. Cet impact se fait notamment ressentir dans le cas de scénarios de `dome.switch` fréquents. L'exemple le plus représentatif de ces scénarios est le cas des exceptions. Le dome responsable est alors régulièrement appelé. Or, chaque appel implique deux `dome.switch` : un pour aller vers le dome responsable, un autre pour en revenir. Nous allons voir quelles alternatives peut offrir l'ISA au logiciel pour limiter ce surcoût.

Limitation des `dome.switch` La première solution est logiquement de limiter les changements de dome actif. Si cela n'est pas toujours possible, cette méthode peut être employée pour les exceptions. L'ISA RISC-V [139] propose par exemple une méthode de délégation des exceptions. Ainsi, à l'aide d'un nouveau CSR `mdeleg`, le niveau de privilège M peut choisir quelles exceptions il gère lui-même. Les autres doivent donc directement être gérées par les autres niveaux, qui disposent alors de leur propre registre pour indiquer l'adresse de redirection (un `tvec` dédié par niveau).

Dans notre cas, ce mécanisme de délégation peut être adapté comme le reste des exceptions. Ainsi, le dome responsable aura la possibilité de ne provoquer un `dome.switch` que pour les exceptions les plus importantes.

Considérer plusieurs domes actifs La limitation des `dome.switch` n'est cependant pas toujours applicable. Une autre approche peut être de spécifier directement au matériel les domes susceptibles d'être régulièrement appelés. Ainsi, le processeur peut s'adapter, par exemple en allouant des ressources pour des domes autres que celui en cours d'exécution. De son point de vue, cela revient à considérer plusieurs domes actifs sur une certaine période de temps.

Au niveau de l'ISA, cette information pourrait prendre la forme d'un nouveau bit de capacité appelé `fea-fr` (*fast recovery* ou reprise rapide). Il indiquerait la possibilité pour un dome de conserver des ressources et informations même lorsqu'il n'est pas exécuté. Celles-ci seraient donc plus rapidement accessibles lors du `dome.switch`. D'après les principes du chapitre 7, cela reviendrait à combiner allocation statique et partitionnement spatial.

6.5. Extension du modèle

Les domes se veulent une base intéressante pour bâtir des politiques de sécurité. Si les travaux présentés dans ce manuscrit sont axés autour de l'isolation microarchitecturale, il est possible d'imaginer des extensions pour offrir d'autres garanties de sécurité. Certaines de ces réflexions ont été présentées dans « *Recommendations for a Radically Secure ISA* » [145] et « *Electromagnetic Fault Injection against a Complex CPU, toward New Micro-Architectural Fault Models* » [146].

Accès aux compteurs de performances

Les compteurs de performances sont des registres responsables de dénombrer les évènements microarchitecturaux. Dans l'ISA de base RISC-V, on en retrouve pour indiquer le temps écoulé, le nombre de cycles de fonctionnement ou le nombre d'instructions exécutées. Il est commun de retrouver d'autres compteurs pour le nombre de *misses* de chaque cache, la précision des prédicteurs de branchements *etc.* Ils donnent directement des informations importantes sur les exécutions passées. Or, tous les programmes n'ont pas la nécessité de ces mécanismes. À l'échelle des domes, on peut imaginer l'utilisation de capacités pour limiter les accès ou la précision des informations qu'ils contiennent.

Intégrité du flot d'exécution

Lors de l'exécution d'un programme, le logiciel s'attend à ce que les instructions soient exécutées dans un ordre précis. Cependant, certains attaquants peuvent vouloir ignorer ou sauter certaines instructions. Des techniques matérielles et/ou logicielles cherchent donc à détecter ce genre de phénomène. Elles visent à préserver l'intégrité du flot de contrôle.

Pour cela, une approche possible est de réaliser une étude statique du programme durant sa compilation. L'objectif est alors de déterminer les différents scénarios de fonctionnement attendus du programme. Les sauts et branchements peuvent aussi être annotés avec les adresses cibles légitimes. Ainsi, durant l'exécution, le processeur est capable de savoir si un saut est légal ou non. Durant cette étude statique, les sauts indirects représentent un problème majeur [147]. Étant dépendant d'une valeur stockée dans un registre, leur destination n'est connue qu'au moment de

l'exécution. Ils sont susceptibles de rediriger l'exécution vers n'importe quelle adresse.

À l'échelle des domes, la mise en place d'une politique d'intégrité du flot de contrôle pourrait passer par la suppression des sauts indirects. Ceux-ci peuvent alors être remplacés par d'autres mécanismes au niveau de l'ISA. Notamment, `dome.switch` et le point d'entrée pourraient être utilisés pour transférer le flot d'exécution à un autre dome. Contrairement à un simple saut indirect, ce transfert considèrerait donc que la destination n'appartient pas au même domaine de sécurité. D'autres instructions de sauts et branchement pourraient également être ajoutées pour permettre plus de flexibilité à l'intérieur de chaque dome, tout en permettant l'étude statique.

Chiffrement et authentification de la mémoire

Par défaut, la mémoire n'est pas considérée comme un composant sécurisée du système. Des attaquants sont susceptibles d'accéder et de modifier depuis l'extérieur aux données qu'elle contient.

Pour la confidentialité, certaines techniques s'appuient sur du chiffrement. À partir d'algorithmes cryptographiques, l'idée est de modifier les données stockées pour que seul le propriétaire soit apte à les récupérer. Ce genre de mécanisme passe généralement par l'attribution d'une clé confidentielle de chiffrement à chaque utilisateur. Des techniques cryptographiques (mais basées sur des algorithmes différents) sont également utilisables pour assurer l'intégrité. Là encore, une clé est nécessaire afin d'authentifier les données.

La configuration des domes pourrait être modifiée pour supporter l'ajout d'une clé cryptographique. Comme illustré sur la Table 6.10, chaque dome aurait donc à la création une clé qui lui serait affectée. L'ISA ne définirait alors que les règles d'utilisations de ces clés, voir les algorithmes à utiliser selon les niveaux de sécurité requis. Par exemple, les registres contenant la clé ne seraient lisibles que par le dome lui-même ou les domes responsables (après vérification de la table). Le choix des mécanismes serait laissé libre pour l'implémentation.

Table 6.10. – Ajout d'une clé 128 bits à chaque configuration. La clé est divisée en champs de 32 bits adressables. On pourrait alors imaginer des restrictions pour les manipuler (illisibles pour les domes non autorisés, non-modifiables une fois le dome validé, *etc.*)

offset	31 0
0x00	statut
0x01	identifiant
0x02	entrée
0x03	table
0x04	capacités
0x05	clé0
0x06	clé1
0x07	clé2
0x08	clé3
0x70	instance

Augmentation du nombre de domes

Dans la version présentée jusqu'à présent, seuls 32 domes différents peuvent exister simultanément. Si c'est suffisant pour un grand nombre de systèmes, on peut imaginer que ce nombre soit limité pour d'autres. Par exemple, certains serveurs sont susceptibles de gérer un très grand nombre d'utilisateurs simultanément. On peut alors envisager une extension du modèle pour considérer plus de domes. Dans la version proposée ici, pour une ISA 32 bits, ce nombre pourrait être de 2^{32} domes différents. Cette valeur est évidemment une limite maximale pour permettre un maximum de flexibilité : le logiciel peut n'en utiliser qu'une infime partie.

Pour la suite, on définit :

- La *plage d'identifiants* correspond à l'ensemble des identifiants représentés par une table. Dans le modèle de base, il y avait un identifiant par bit donc une plage de 32 identifiants.
- La *plage d'identifiants globale* correspond à l'ensemble des identifiants existants. Dans le modèle de base, il y en avait également 32. Pour la suite, notre objectif est donc de 2^{32}.

- La *base d'une plage d'identifiants* est le plus petit identifiant représenté par la table. Dans le modèle de base, c'était l'identifiant 0.
- La *largeur d'une plage d'identifiants* est le nombre d'identifiants représentés par la table. Là encore dans le modèle de base, il y en avait 32.
- Un *sous-groupe d'identifiants* est le nombre d'identifiants représentés par chaque bit de la table. Dans le modèle de base, il y en avait 1 par bit.

Dans ce nouveau modèle, l'identifiant de chaque dome est utilisé dans son intégralité : il est donc encodé sur 32 bits et non plus uniquement 5 bits. Seulement, la table ne contient toujours que 32 bits. Une première approche est alors de diviser l'ensemble de l'espace des domes (2^{32}) en sous-groupes. Chaque bit de la table n'indique alors plus un seul dome mais un sous-groupe entier. Ainsi, la plage d'identifiants pour chaque dome est de 2^{32}, ce qui correspond à la plage d'identifiants globale.

Cependant avec ce système, les sous-groupes sont systématiquement de $2^{32}/32 = 2^{27}$, ce qui offre peu de flexibilité. Une alternative serait de rendre la plage d'identifiants flexible, qu'elle puisse représenter un nombre d'identifiants variables. Pour cela, on ajoute à chaque dome une valeur d'*index*. Pour chaque dome, cet index donne la largeur de la plage d'identifiants à l'aide de la formule : $2^{index \cdot 5} \cdot 32$ où $2^{index \cdot 5}$ est la taille d'un sous-groupe. L'ISA étant sur 32 bits, on aurait que 7 valeurs d'index possibles (de 0 à 6, avec seulement 4 sous-groupes de 2^{30} pour $index = 7$).

Le dernier élément à définir pour chaque dome est la base de sa plage d'identifiants. Actuellement, elle est toujours fixée à 0 ce qui ne permet pas d'utiliser l'ensemble de la plage d'identifiants globale. Une alternative est de déplacer la base selon la valeur de l'identifiant du dome lui-même. Par exemple :

- le dome 17, d'index 0 et donc avec une plage de largeur 32, aurait sa base à 0,
- le dome 76, d'index 0 et de largeur 32, aurait sa base à 64,
- le dome 78, d'index 0 et de largeur 32, aurait sa base à 64,
- *, etc.*

La formule pour déterminer la base est la suivante : $ID - (ID \mod L)$ avec ID l'identifiant et L la largeur. Du point matériel, cela revient à faire : $ID[(i_{max} + 1) * 5 : (i + 1 * 5)] * 2^{5*i}$ avec i l'index et i_{max} l'index maximal, ici 6. Finalement le Table 6.11 présente plusieurs exemples de domes et du calcul de leurs caractéristiques.

Table 6.11. – Exemples illustrant l'impact de l'index. Les bits de l'identifiant en rouge sont ceux qui seront identiques pour tous les identifiants de la plage. En noir sont donc les bits restants qui peuvent être modifiés pour constituer des identifiants valides.

Identifiant (décimal)	Identifiant (binaire)	Index	Base	Largeur
17	0000 0000 0000 0000 0000 0000 0001 0001	0	0	32
76	0000 0000 0000 0000 0000 0000 0100 1100	0	64	32
76	0000 0000 0000 0000 0000 0000 0100 1100	1	0	1024
0	0000 0000 0000 0000 0000 0000 0100 1100	6	0	2^{32}

L'ajout de l'index permet donc d'agrandir la plage globale tout en conservant une certaine flexibilité. Cependant, cela se répercute irrémédiablement sur le matériel. Les différentes vérifications, qui dans le premier modèle n'avaient qu'à considérer la table et un identifiant de

5 bits, doivent également être modifiées. Par exemple, quand un dome D0 configure un dome D1, il est nécessaire de s'assurer que l'ensemble de la plage attribuée à D1 est possédée par D0. Sinon, cela reviendrait à donner à D1 le contrôle de domes dont D0 n'a pas la possession.

6.6. Version rétrocompatible avec les privilèges statiques

Le modèle dynamique présenté jusqu'ici est la dernière version de nos travaux sur le support de nouveaux domaines de sécurité au niveau de l'ISA. Auparavant, une première version hybride sur le modèle de la Figure 5.2 avait été étudiée. C'est cette version qui a été utilisée pour l'étude de la mise en place d'une politique d'isolation microarchitecturale (décrite dans les prochains chapitres). Nous allons dans cette section étudier son fonctionnement général.

La version hybride des domes s'appuie sur de nouveaux CSR ainsi que les privilèges définis par l'ISA RISC-V. Contrairement au modèle dynamique, elle est donc compatible avec la version privilégiée. Une configuration d'un dome est constituée de seulement trois champs sur 32 bits :

- l'identifiant,
- le point d'entrée,
- les capacités. Ces dernières contiennent seulement les informations nécessaires à l'allocation des ressources, comme le poids ou la nécessité d'exécuter des multiplications et divisions. Les autres capacités ne sont ici pas nécessaires, les privilèges statiques étant conservés.

Pour chaque thread, on définit deux configurations accessibles par le biais des CSR. Chacun de leurs champs est accessible comme un registre individuel. La première est la configuration du dome actif `domecur`. Elle est accessible en lecture seulement. La seconde est une configuration de travail appelée `domework`. Elle est accessible en lecture et écriture et permet de charger le prochain dome qui sera exécuté.

Enfin, une seule nouvelle instruction `dome.switch rd` permet de modifier le dome actuel. Pour cela, le matériel effectue les différentes opérations de sécurité et renvoie le résultat. En cas de succès :

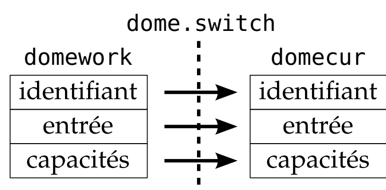


Figure 6.11. – Opérations sur les CSRs lors d'un `dome.switch` (modèle hybride).

- 0 est écrit dans le registre `rd`,
- un saut est forcé par le matériel vers le point d'entrée dans `domework`,
- le contenu de `domework` est copié dans `domecur` comme sur la Figure 6.11.

En cas d'échec des opérations de sécurité, le dome exécuté reste le même et une valeur différente de 0 est écrite dans le registre `rd`.

Le logiciel reste responsable du choix du dome actif. Cependant, la possibilité de modifier `domework` est laissée uniquement au privilège le plus haut M. Ainsi, c'est ce niveau qui choisit le domaine de sécurité des applications en cours d'exécution. Une application malveillante ne peut donc volontairement se placer dans le même dome qu'une autre application pour l'espionner.

6.7. Conclusion

Dans ce chapitre, nous avons vu comment modifier l'ISA afin d'introduire une nouvelle notion de domaines de sécurité. Appelés domes, ils se comportent comme des niveaux de privilèges dynamiques. Chacun possède une configuration qui lui est propre, décrivant notamment ses droits et les politiques d'isolation qu'il implique. La hiérarchie des domes est entièrement modulable par le logiciel. Pour cela, de nouvelles instructions dédiées ont été ajoutées et sont résumées dans l'Annexe B. Certaines vérifications sont effectuées par le matériel afin de garantir l'intégrité du modèle.

Les privilèges statiques représentent la base de sécurité des ISA modernes. En les supprimant, c'est donc une grande partie du jeu d'instructions qui doit être réadaptée. Nous avons ici vu comment un mécanisme comme les exceptions peut être géré par des privilèges dynamiques, sans pour autant en bouleverser le fonctionnement. À terme, pour constituer une ISA privilégiée complète, le support de la mémoire virtuelle devra également être considéré. D'autres modifications seront en plus nécessaires pour retrouver l'ensemble des garanties offertes par les implémentations modernes. Par exemple, il peut être intéressant d'adapter le support des domes dans les périphériques comme c'est le cas avec ARM TrustZone. L'ensemble du système serait alors capable d'associer les opérations en cours à un domaine de sécurité.

Avec ce mécanisme de domes, le logiciel est maintenant capable d'informer le matériel de ses contraintes de sécurité. Notre objectif, à présent, va être d'utiliser ces informations avec le matériel pour la mise en place d'une politique d'isolation microarchitecturale. Tous les niveaux d'abstraction seront alors capables de jouer leur rôle. Seule une version hybride des domes est utilisée dans les prochains chapitres. Ainsi, d'autres travaux seront également nécessaires afin d'implémenter entièrement le modèle dynamique. Certains éléments comme le coût et la complexité matérielle, mais aussi l'utilisation par le logiciel doivent encore être évalués.

Adapter la microarchitecture

Conception des ressources partagées 7.

Résumé du chapitre : Dans ce chapitre, nous voyons comment concevoir des ressources partagées en prenant en compte des contraintes d'isolation. Après avoir détaillé la notion de partage de ressources, nous établissons un objectif de sécurité pour la conception. Nous déduirons ensuite plusieurs principes d'utilisation à partir des informations reçues par l'ISA. Trois méthodes pour la gestion des ressources partagées sont également détaillées, afin d'évaluer les choix possibles pour un concepteur matériel et les compromis qu'il doit considérer. Ces travaux sont également présentés dans « *Under the Dome : Preventing Hardware Timing Information Leakage* » [148].

7.1 Définitions 105
Ressource partagée 105
Problématique et objectifs 106
7.2 Allocation statique des ressources 108
Principe 108
Mécanismes pour l'implémentation 108
7.3 Séparation des ressources 109
Principes 109
Mécanismes pour l'implémentation 109
7.4 Suppression des traces . . 111
Principe 112
Mécanismes pour l'implémentation 112
7.5 Homogénéité 113
7.6 Modèle mémoire et renforcement des frontières . . . 114
Objectif 114
Accès contrôlé 114
Duplication 115
Cohérence 115
7.7 Conclusion 116

7.1. Définitions

Lors de la conception de la microarchitecture d'un processeur, nous avons vu que différentes contraintes entrent en jeu : les performances, le coût, la consommation et la sécurité. Des mécanismes ont dû être mis en place pour les concilier, et notamment le partage de ressources.

Ressource partagée

On appelle une ressource partagée un mécanisme pouvant être utilisé par plusieurs entités différentes, qu'il s'agisse de plusieurs threads matériels, de plusieurs cœurs entiers ou processus logiciels. Le partage est particulièrement utile dans deux cas précis.

Utilisation des ressources partagées Le premier cas est la mise à disposition de chaque entité d'un plus grand nombre de ressources pour améliorer les performances. Lors de l'exécution, avoir accès à plusieurs ressources simultanément permet d'effectuer plusieurs opérations en parallèle et donc d'accroître le débit global d'opérations exécutées. C'est par exemple valable pour les unités d'exécutions dans des implémentations avec multithreading. Comme présenté sur les Figure 7.1 et Figure 7.2, le partage s'oppose par définition à la duplication des ressources, c'est-à-dire l'implémentation de plusieurs ressources identiques. La duplication permet d'améliorer grandement les performances en permettant à chaque entité d'avoir ses propres ressources, mais a un impact significatif sur le coût du système et sur la consommation. En utilisant le partage, le système donne l'illusion à chaque entité qu'elle dispose de manière exclusive de plusieurs ressources. En contrepartie, cette illusion ne s'applique que lorsque ces ressources ne sont pas déjà utilisées. S'il tend a également améliorer le taux d'utilisation d'une ressource, le partage n'est efficace que dans le cas de conflits limités entre les entités. Sinon, la duplication devient essentielle pour améliorer les performances.

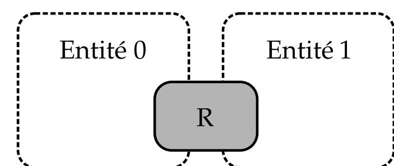


FIGURE 7.1. – Partage d'une ressource entre plusieurs entités.

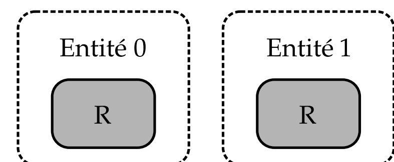


FIGURE 7.2. – Duplication d'une ressource pour chaque entité.

Le second cas d'utilisation concerne toujours les performances avec la mise en commun d'informations entre plusieurs entités. Lors de l'exécution des programmes, les processeurs accumulent au fur et à mesure des informations (ou métadonnées) qui leur permettent

d'accélérer la suite de l'exécution. Le partage de ressources permet alors de réaliser un échange d'"expérience" entre les différentes entités. C'est le principe même de la spéculation détaillé dans le chapitre 2.

Finalement, ces ressources partagées peuvent prendre de nombreuses formes différentes dans les microarchitectures modernes comme par exemple :

- Les mémoires caches accélèrent les accès mémoires en conservant une copie des données localement. Les données stockées dépendent des accès mémoires effectués précédemment. Ainsi, si plusieurs exécutions partagent ces données, elles peuvent y accéder plus rapidement.
- Les tables de prédictions conservent un historique des branchements et sauts passés. En partageant son propre historique, une exécution peut donc en aider une autre à prédire ses propres branchements afin de l'accélérer.
- Les buffers ou autres registres contenant des données ou informations, qui sont partagés par les utilisateurs successifs.
- Les bus qui sont partagés par les utilisateurs souhaitant réaliser des accès à des mémoires ou périphériques .
- Les machines à états finis (FSM) qui représentent l'état d'une ressource où d'un mécanisme.

États des ressources Pour décrire une ressource partagée, nous avons vu qu'on utilise généralement deux états différents. Le premier est l'**état persistant** qui correspond à un état qui se maintient dans le temps. C'est notamment le cas des ressources utilisant des registres ou des mémoires qui influencent leur fonctionnement (*e.g.* les mémoires caches). À l'inverse, l'**état transitoire** correspond à un état temporaire d'une ressource observable sous certaines conditions. Une forme de ce type d'état particulièrement exploitée est la contention dynamique, c'est-à-dire la détection de l'utilisation ou non d'une ressource. De par leurs natures différentes, nous verrons que différentes approches sont nécessaires afin de protéger ces états.

Types de partage Nous distinguerons dans la suite de ce document deux types de partage. Le **partage temporel** qui permet à deux entités d'utiliser une ressource, mais à des moments différents. C'est par exemple le cas pour deux processus s'exécutant l'un après l'autre sur un même processeur. En réutilisant les données ou informations du précédent processus (et donc l'état persistant), le second voit sa propre exécution influencée. Le **partage spatial** permet, lui, à deux entités d'accéder simultanément à une même ressource. On retrouve notamment ce type de partage dans le cas de microarchitectures permettant des exécutions en parallèle (multithreading simultané ou multicœurs). Chaque thread ou cœur peut alors effectuer en même temps une requête vers certaines ressources. Dans le cas d'une mémoire cache par exemple, les différentes lignes (état persistant pour les données) ou les ports d'accès (état transitoire pour la disponibilité) sont partagés.

Problématique et objectifs

Le partage de ressources est donc une notion essentielle dans les systèmes modernes. Cependant, nous avons vu dans le chapitre 3 qu'il

mène également à de nombreux problèmes d'isolation. Le partage tel qu'implémenté actuellement permet un échange excessif d'informations entre les utilisateurs, avec notamment l'accès aux métadonnées (accès à une adresse, exécution d'une instruction, utilisation d'une ressource *etc.*). Malgré les problèmes qu'elles soulèvent, la simple suppression de ces ressources partagées n'est pas réaliste. L'autre alternative est donc de les implémenter d'une manière différente et compatible avec des contraintes de sécurité. Afin de définir un objectif de conception précis, il est nécessaire de cerner précisément les limites du système.

Tout d'abord, un utilisateur de ressource partagée correspond dans notre cas à un domaine de sécurité (ou dome dans le cas de notre nouvelle ISA). Comme présenté dans les chapitres précédents, il représente notre seul élément de décision pour définir où se trouvent les frontières d'isolation. Ainsi, c'est uniquement à cette échelle que nos nouvelles garanties de sécurité doivent être construites, pour déterminer lorsqu'un partage est possible.

Ensuite, dans n'importe quelle implémentation, les ressources sont limitées en nombre, d'où la nécessité de leur partage. C'est la raison principale pour laquelle elles sont partagées. Un système avec plusieurs domaines de sécurité devra donc nécessairement gérer la disponibilité des ressources. Une ressource utilisée est obligatoirement indisponible pour un nouvel utilisateur qui peut alors le détecter. Par définition, cette situation est admise et indissociable du partage. Cependant, il est important de distinguer deux types de disponibilité :

1. La disponibilité dynamique est la possibilité d'utiliser une ressource à n'importe quel moment si elle n'est pas déjà requise par un utilisateur.
2. La disponibilité statique est la possibilité pour un utilisateur de bloquer une ressource pour une potentielle future utilisation. On appelle cela l'*allocation*.

La différence clé entre ces deux disponibilités est l'information qu'elles indiquent. Dans le cas statique, une ressource non disponible est seulement allouée, mais pas forcément utilisée, contrairement au cas dynamique. Ainsi, alors que la disponibilité dynamique fait fuiter des informations précises sur les temps d'exécution et d'utilisation (exploitées par les attaques par contention), la disponibilité statique ne permet pas de retrouver ce genre d'informations. Dans notre cas, nous ne considérons donc comme acceptable que cette dernière, ce qui nous mène à l'objectif de conception suivant :

Ressource partagée isolée

Les seules informations qu'un domaine de sécurité peut extraire d'une ressource partagée sont celles propres à ce domaine ou à la disponibilité statique de la ressource.

Les ressources partagées doivent donc être conçues et modifiées afin d'empêcher toute fuite d'une autre information. Nous allons par la suite décrire des principes de conception permettant d'atteindre cet objectif. Ceux-ci sont inspirés des propriétés présentées par Ge, Yarom et Heiser [91], mais appliquées à la conception matérielle en considérant les informations sur les domaines de sécurité fournies par l'ISA. Ils s'articulent autour de trois grandes méthodes : l'allocation statique des

[91]: Ge et al. (2018), « No Security Without Time Protection : We Need a New Hardware-Software Contract »

ressources, la séparation des ressources et la suppression des traces.

7.2. Allocation statique des ressources

L'allocation permet à un domaine de sécurité de bloquer plusieurs ressources pour une potentielle future utilisation. Elle permet de masquer la disponibilité dynamique entre plusieurs domaines isolés.

Principe

Afin de permettre la bonne exécution d'un domaine de sécurité, celui-ci doit avoir accès à l'ensemble des ressources dont il a besoin. Ceci nous mène donc à un premier principe de conception :

Principe de conception 7.2.1 ***: Allocation statique***
Les ressources minimales nécessaires à un domaine de sécurité doivent être allouées durant la création de ce domaine et bloquées jusqu'à sa suppression.

Chaque ressource ne peut supporter simultanément qu'un nombre limité de domaines de sécurité, qui peut aller d'un (seulement du partage temporel) à plusieurs (partage temporel et spatial). L'allocation statique permet donc d'avoir l'exclusivité d'une ressource pour être capable de l'utiliser sans générer de fuites temporelles. Évidemment, cela n'est nécessaire que dans le cas où plusieurs domaines sont exécutés simultanément et où des fuites spatiales comme la contention existent. Dans le cas inverse où un seul domaine est exécuté, cette isolation spatiale est assurée par construction : le domaine peut utiliser n'importe quelle ressource étant donné qu'il est le seul utilisateur.

Mécanismes pour l'implémentation

L'allocation statique de ressources est un concept diamétralement opposé à ce qui est fait dans les microarchitectures modernes. Le dynamisme est privilégié pour permettre un partage optimal des ressources : elles ne sont bloquées que lors de leur utilisation, ce qui rend le système plus flexible et peut améliorer les performances. De nouveaux mécanismes doivent donc être implémentés au niveau du matériel. L'allocation statique peut être décomposée en deux : l'attribution des ressources et son application.

Pour l'attribution, une stratégie consiste à concevoir des unités responsables de cette tâche, et affectant des ressources selon les requêtes reçues. C'est ce qui a été fait dans les travaux d'implémentation décrits dans le prochain chapitre. Une fois l'attribution effectuée, chaque ressource se voit associée à un domaine précis. Une stratégie commune dans ce genre de cas pour assurer le respect de l'allocation est l'utilisation d'un *tag*, une valeur interne représentant le domaine associé. Ainsi, si une opération tente d'utiliser une ressource, une simple comparaison des tags permet de définir si cela est possible ou non. Cette stratégie est bien connue pour le partitionnement des ressources comme les mémoires que nous verrons prochainement.

Implémenter l'allocation statique a évidemment un coût. En plus des nouvelles unités nécessaires, elle réduit logiquement les ressources

accessibles à un domaine d'exécution. Selon les configurations, les performances peuvent en être impactées.

7.3. Séparation des ressources

L'allocation statique permet d'allouer entièrement et exclusivement une ressource. Ce fonctionnement n'est cependant pas adapté à toutes les ressources partagées. C'est par exemple le cas des contrôleurs mémoires : tous les domaines exécutés doivent pouvoir accéder à la mémoire et donc utiliser le contrôleur. Il est alors nécessaire d'appliquer une séparation des domaines : la ressource devra être capable de gérer plusieurs utilisateurs simultanément.

Principes

Dans le cas où une ressource gère des requêtes de plusieurs domaines, elle doit assurer qu'aucune interaction n'est possible entre eux. Ceci nous mène au principe suivant :

Principe de conception 7.3.1 : *Partitionnement spatial*
Une ressource gérant simultanément des requêtes de plusieurs domaines de sécurité doit être capable de cloisonner chacun d'eux dans son propre compartiment.

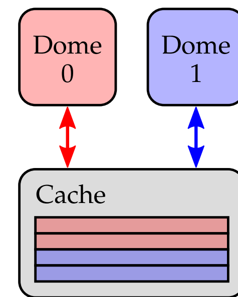


Figure 7.3. – Partitionnement spatial d'une mémoire. Chacun des domes ne peut finalement accéder qu'à certaines lignes de la mémoire. Les autres sont, de son point de vue, invisibles.

Ainsi, chaque ressource partagée spatialement doit être scindée entre les différents domaines de sécurité qui l'utilisent. D'un point de vue allocation, une telle ressource peut donc être vue comme ayant plusieurs places : chaque domaine en alloue un certain nombre.

Si le principe précédent s'applique particulièrement aux états persistants des ressources, le même raisonnement doit s'appliquer aux états transitoires et notamment la disponibilité :

Principe de conception 7.3.2 : *Séparation de la disponibilité*
Une ressource partagée doit assurer qu'à n'importe quel moment, sa disponibilité pour un domaine de sécurité est indépendante des autres domaines servis.

Finalement, gérer plusieurs domaines simultanément implique d'être capable d'imposer une séparation stricte entre eux.

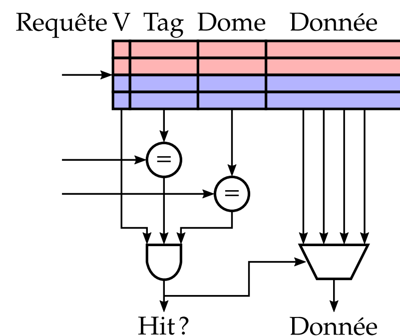


Figure 7.4. – Implémentation du partitionnement spatial d'un cache. Chacune des lignes étant considérée comme une ressource différente, elle se voit associée le numéro du dome auquel elle appartient. Lors d'une requête, en plus de la comparaison du tag, la comparaison des domes est également effectuée.

Mécanismes pour l'implémentation

Le partitionnement est une approche explorée depuis plusieurs années. Nous avons vu dans le chapitre 4 que le partitionnement spatial avait particulièrement été étudié dans le cas des mémoires caches, afin de permettre à plusieurs cœurs ou threads de fonctionner de manière isolée. Il peut finalement être appliqué de la même manière à toute ressource dupliquée. Son implémentation peut être semblable à celle de l'allocation, où chaque sous-partie de la ressource va être taguée par le domaine de sécurité associé comme représenté sur les Figure 7.3 et Figure 7.4. Cette technique est particulièrement intéressante pour isoler les états persistants comme mentionné par le Principe 7.3.1.

Nous avons vu dans le chapitre 4 qu'il existait également le partitionnement temporel (ou multiplexage temporel). Au lieu de scinder

spatialement une ressource, l'idée est de scinder son temps d'utilisation [53, 109]. Ainsi, si à un moment précis un seul domaine peut utiliser la ressource, plusieurs le peuvent sur une plus grande période de temps. La Figure 7.5 illustre le partitionnement temporel d'une ressource. La disponibilité de la ressource est divisée en *tour*. Durant chaque tour, un seul domaine de sécurité peut accéder à la ressource. Le domaine autorisé change statiquement et donc indépendamment des besoins.

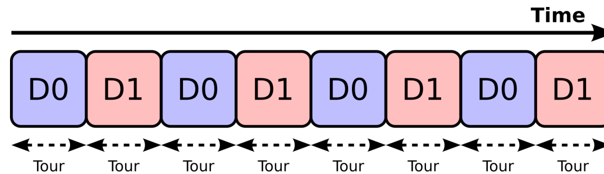
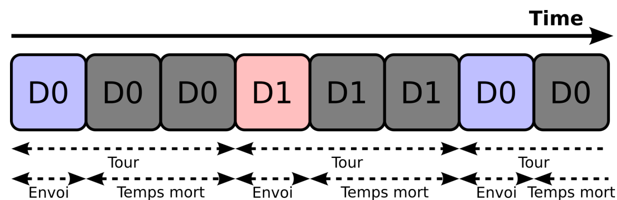
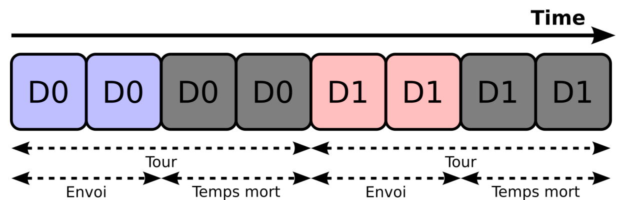


Figure 7.5. – Disponibilité d'une ressource avec un partitionnement temporel.

Le fonctionnement de la Figure 7.5 est le cas le plus simple d'une ressource sans cycle de latence. La réponse est reçue dans le même cycle que la requête. La Figure 7.6 illustre le cas d'une ressource avec 2 cycles de latence (réponse reçue 2 cycles après la requête). Durant ce temps supplémentaire, la ressource est déjà en train de traiter une requête : elle est donc indisponible. Si un domaine différent est capable d'accéder à la ressource durant cette période, il pourra détecter de la contention dynamique. Ainsi, l'intégralité de la latence doit être incluse dans la durée d'un tour comme sur la Figure 7.6a. Cette période correspond à un *temps mort*, où le domaine ne peut envoyer de nouvelles requêtes et attend seulement des réponses. Au final, un domaine n'est donc capable d'envoyer une requête que tous les $N_{dome} * (1 + L)$ cycles, avec L la latence et N_{dome} le nombre de domes actifs. Dans le cas de la Figure 7.6a, cela serait tous les $2 * (1 + 2) = 6$ cycles.



(a) Implémentation naïve d'un tour : une seule requête possible.



(b) Implémentation d'un tour adaptée à un pipeline : deux requêtes possibles.

Figure 7.6. – Partitionnement d'une ressource avec latence.

Du point de vue des performances, une telle implémentation n'est cependant pas adaptée aux ressources utilisant un pipeline. Ce dernier permet normalement de masquer la latence en gérant simultanément plusieurs opérations. Ce bénéfice serait perdu par l'implémentation naïve de la Figure 7.6a. Une alternative est alors de permettre l'envoi de plusieurs requêtes par tour. En moyenne, le domaine pourrait alors envoyer $\frac{N_{dome} * (R+L)}{R}$ avec R le nombre de requêtes par tour. 2 requêtes par tour sont autorisées sur la Figure 7.6b, ce qui correspond à une moyenne d'une requête tous les $\frac{2*(2+2)}{2} = 4$ cycles.

Finalement, le cas le plus avantageux du point de vue des perfor-

mances est celui d'une ressource sans latence. Une requête peut être envoyée tous les N_{dome} cycles. Un dernier mécanisme vise donc à cumuler duplication, partitionnement spatial et temporel au sein du pipeline. Le résultat est présenté sur la Figure 7.7. Chaque étage du pipeline est considéré comme une ressource à part entière. Les différents états persistants sont isolés.

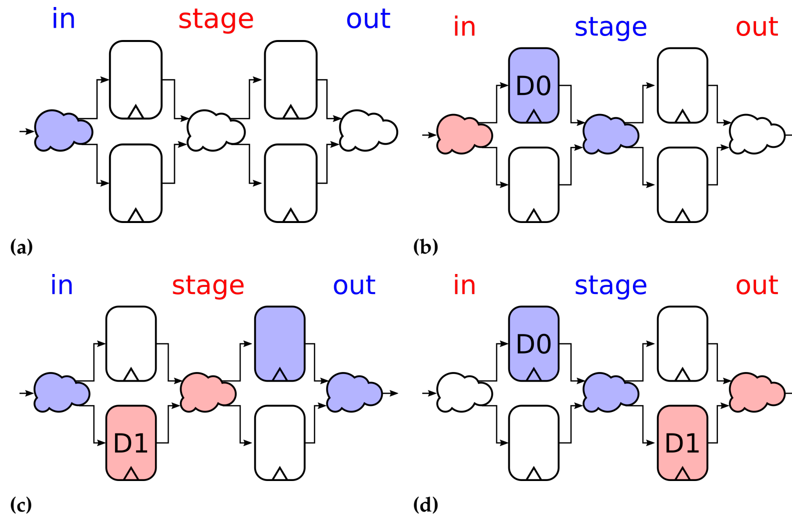


Figure 7.7. – Partitionnement temporel de deux étages d'un pipeline. À chaque cycle d'horloge, la possibilité d'utilisation d'un étage, considéré comme une ressource, alterne entre les différents domes utilisateurs. Dans la Figure 7.7a, seul le dome D0 est accepté en entrée. Au cycle suivant dans la Figure 7.7b, c'est la requête de D1 qui est acceptée. De même dans les étages suivants, la possession d'un étage alterne.

Ces deux mécanismes ont également des inconvénients à prendre en compte Du côté du partitionnement spatial, il réduit purement et simplement le nombre de ressources accessibles et utilisables par un domaine de sécurité. Le partitionnement temporel réduit lui le temps où une ressource peut être utilisée. De plus, les états persistants des ressources doivent toujours être gérés spécifiquement, que ce soit en les effaçant ou en les partitionnant. Finalement, on privilégiera donc ***le partitionnement spatial pour distinguer les états persistants***, et ***le partitionnement temporel pour éviter la contention dynamique***.

Historiquement, le partitionnement n'est finalement qu'une généralisation pour la sécurité des mécanismes de multithreading. Si le plus répandu est le SMT (ici associé à un partitionnement spatial), il existe aussi des techniques pour partager temporellement un même thread. C'est le cas des processeurs implémentant un multithreading à grain fin ou à grain grossier [2]. L'isolation entre les threads permet alors de ne pas partager les blocages du pipeline : une bulle créée par un thread ne doit pas impacter les autres.

Dans le cas de ressources partagées temporellement et spatialement, il est important de préciser que les techniques de relâchement et de séparation doivent être combinées. Cela se traduit notamment par des opérations de flush limitées aux partitions du même dome. Sinon, cela reviendrait à passer outre le partitionnement en interférant sur les autres domaines.

7.4. Suppression des traces

L'allocation ne dure pas éternellement et quand un domaine a fini d'être exécuté, il doit pouvoir relâcher ses ressources pour que d'autres puissent les allouer. C'est cette phase qu'on appelle la libération des

ressources. Elle doit être faite en assurant l'isolation, et notamment en supprimant les traces restantes.

Principe

Certaines ressources contiennent des états persistants. Ainsi, même si elles ne sont plus utilisées, des informations peuvent toujours y être conservées. En cas de partage temporel, si une ressource est allouée par un domaine, relâchée puis réallouée par un autre, il est donc nécessaire de supprimer les traces représentant de potentielles fuites. Ceci nous mène au principe suivant :

Principe de conception 7.4.1 ***: Suppression des traces***
Quand un domaine de sécurité est terminé, toutes ses ressources doivent être relâchées uniquement lorsque les traces persistantes ont été effacées.

Ainsi, tous les états persistants associés à un domaine de sécurité ne doivent plus exister une fois que ce même domaine n'est plus exécuté. Ensuite seulement, un nouveau domaine peut réallouer cette ressource. Toute ressource partagée temporellement et comportant un état persistant doit donc suivre ce principe.

Selon les ressources, le temps nécessaire à l'effacement des traces peut changer. Des variations peuvent même être observées selon les données contenues/les opérations effectuées. C'est par exemple le cas des mémoires caches dans le cas d'une politique d'écriture différée (*write-back*) : les modifications ne sont propagées qu'une fois la copie effacée. Ainsi, selon le nombre de lignes écrites, le temps de relâchement variera.

Principe de conception 7.4.2 ***: Relâchement temps constant***
Chaque ressource doit être relâchée après un temps indépendant de son état.

Le Principe 7.4.2 vise donc à rendre impossible l'interprétation du temps de relâchement. La mise en place d'un temps constant a cependant un coût : elle revient à toujours considérer le pire scénario.

Mécanismes pour l'implémentation

Comme vu dans le chapitre 4, l'efficacité de cette approche est bien connue avec l'implémentation d'opérations de *flush* [99, 102]. Son implémentation passe généralement par l'annulation des bits de validité dans des mémoires (*e.g.* caches ou tables de prédiction), la suppression d'opérations en cours et la réinitialisation des différentes FSM. Il est également possible d'écraser les valeurs en les remplaçant par d'autres par défaut.

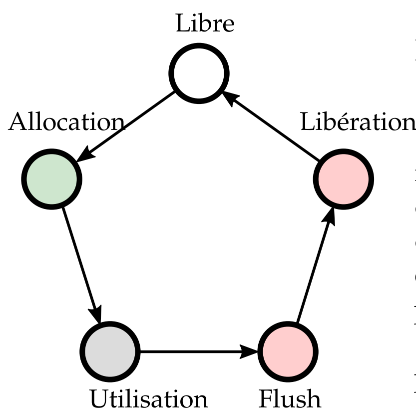


Figure 7.8. – Cycle de vie d'une ressource partagée. Une ressource, avant d'être utilisée, doit être allouée afin de respecter le Principe 7.2.1. À la fin de l'exécution du domaine de sécurité propriétaire, cette ressource voit ses états permanents effacés avant d'être relâchée. Elle est alors à nouveau libre. Le Principe 7.4.1 est ainsi respecté.

Finalement, la gestion de chaque ressource partagée revient à l'implémentation d'une FSM comme celle décrite sur la Figure 7.8. Les différentes phases d'allocation et de relâchement se succèdent donc selon les domaines de sécurité exécutés sur le système.

Si l'effacement des données persistantes est un mécanisme bien connu pour créer des barrières temporelles, il garde cependant un impact non négligeable sur les performances.

Premièrement, conserver le temps de relâchement constant implique de considérer le pire scénario. Cela peut donc directement influer sur les

mécanismes implémentés. À titre d'exemple, on privilégiera par exemple une politique d'écriture traversante (*write-through*) où les modifications sont effectuées au fur et à mesure. Cette notion de temps constant dépend également du modèle de menace envisagée. Si l'on considère un attaquant (ou un espion) comme incapable de mesurer précisément quand l'opération de relâchement commence, alors simplement libérer toutes les ressources simultanément est suffisant. D'un point de vue extérieur, il est alors seulement possible de détecter que le temps d'exécution globale d'un domaine de sécurité (comprenant temps d'allocation + temps d'utilisation + temps de relâchement). Dans le cas des domes présentés dans le chapitre 6, on peut très bien imaginer un bit de capacité pour activer ou non ce flush constant.

Le second aspect concerne le nouveau domaine qui sera exécuté. Après l'allocation des ressources, celui-ci est placé dans un système dit "froid" : toutes les tables et mémoires sont vides ou dans leur état par défaut. Il est alors nécessaire d'effectuer un certain nombre d'opérations afin que toutes les optimisations puissent fonctionner pleinement. Ceci implique des caches *misses*, des erreurs de spéculation *etc.*

Certains systèmes peuvent ne permettre que l'exécution d'un seul domaine de sécurité simultanément. Dans ce cas-là, seul du partage temporel existe : les domaines sont exécutés l'un après l'autre. Il n'est alors plus nécessaire d'utiliser des mécanismes d'allocation statique ou de séparation des ressources. Par défaut, toutes les ressources existantes peuvent être attribuées au domaine en cours d'exécution. En revanche, les mécanismes d'effacement restent nécessaires. Le cycle de vie d'une ressource partagée, présenté sur la Figure 7.8, est donc simplifié : une ressource est soit allouée, soit effacée.

7.5. Homogénéité

Comme dit précédemment, l'allocation statique empêche la détection de l'utilisation d'une ressource, mais pas sa complète disponibilité. Or, en influant sur la disponibilité de certaines ressources, un attaquant peut être capable de construire un canal caché de communication.

Comme illustré sur la Figure 7.9, un système est dit hétérogène lorsque tous ses utilisateurs ne disposent pas forcément des mêmes ressources. Cette organisation est commune aux microarchitectures modernes : tous les threads ou cœurs ne sont pas nécessairement équivalents, *e.g.* pour des raisons de performances ou de consommation. Ainsi, si un cheval de Troie est capable d'influer sur l'allocation en allouant certaines ressources, un message peut être encodé pour un espion capable d'observer la disponibilité des ressources. Dans un système considérant une isolation contre les canaux cachés, il devient alors nécessaire d'appliquer également le principe suivant :

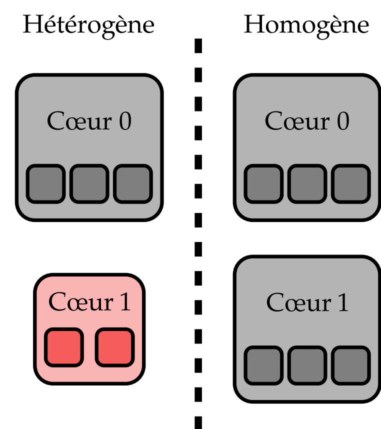


Figure 7.9. – Architecture hétérogène *vs.* homogène. Dans le cas hétérogène, les deux cœurs sont différents. En demandant l'allocation de l'un plutôt que de l'autre, un utilisateur peut créer un canal caché : un cœur indisponible correspond à une valeur (0 ou 1). Dans le cas homogène, les deux cœurs sont identiques : le matériel peut choisir sans contrainte lequel allouer. Un canal de communication n'est plus possible.

Principe de conception 7.5.1 *: Homogénéité*

Durant leur exécution, tous les utilisateurs doivent être égaux en allouant les mêmes ressources, que ce soit en nombre ou en type.

Ce principe est particulièrement restrictif : aucune flexibilité n'est permise sur l'allocation des ressources. Ainsi, selon les implémentations et contraintes du système, il peut être intéressant de distinguer deux

niveaux d'isolation : résistant à l'ensemble des attaques par canal caché via les ressources partagées ou seulement aux attaques par canal auxiliaire. Dans le premier cas, chacun des cœurs disponibles doit être strictement identique. Lors d'une demande d'allocation, chaque cœur est donc également susceptible d'être alloué selon sa disponibilité. Cela implique que les informations d'`instance` introduites dans le chapitre 6 ne s'appliquent pas : un nombre fixe et minimal de chaque type de ressources doit être alloué. Dans le second cas en revanche, on peut envisager des configurations différentes selon les besoins du domaine. Les valeurs d'`instance` prennent alors tout leur sens, en permettant d'allouer uniquement les ressources nécessaires.

7.6. Modèle mémoire et renforcement des frontières

Nous avons vu jusqu'à présent uniquement le partage au niveau des ressources matérielles. D'autres éléments du système peuvent cependant être partagés. C'est le cas des plages mémoires.

Si par défaut l'ensemble des adresses mémoires sont accessibles par tous les utilisateurs, nous avons vu dans le chapitre 2 que la mémoire virtuelle permet de définir des règles de partage. Au cours des travaux sur les ressources matérielles, certains liens avec la gestion des plages mémoires partagées par le matériel sont apparus. L'objectif de cette section est d'établir des principes afin de prolonger une politique d'isolation à travers la définition d'un modèle mémoire.

Objectif

Une plage mémoire est un ensemble de données accessibles à partir d'adresses précises. Dans le cas d'un système de partage mémoire, il est alors possible pour un nombre restreint d'utilisateurs d'accéder à chaque donnée. À partir de ce constat, l'objectif de sécurité visé pour une politique d'isolation est le suivant :

Plage mémoire partagée isolée

Les seules informations qu'un domaine de sécurité peut extraire d'une plage mémoire sont les données auxquelles il est autorisé à accéder.

Accès contrôlé

L'application stricte de cet objectif mène au contrôle de chaque accès à une plage mémoire. Le matériel doit alors s'assurer qu'il est légitime et agir en conséquence. En découle alors le principe de conception suivant :

Principe de conception 7.6.1 : *Plage mémoire possédée*

Une plage mémoire partagée n'est accessible à tout moment que par les domaines de sécurité autorisés. Ses données doivent être invisibles pour tous les autres.

Ce principe de conception ne vise finalement qu'à renforcer des frontières d'isolation déjà définies par l'ISA. Nous avons vu dans le

chapitre 3 que certaines implémentations fragilisaient cette garantie de sécurité pour optimiser les performances. Il est donc important d'insister sur le fait que ce respect de l'isolation doit être assuré pour chaque accès. Cela s'applique également aux processeurs effectuant des accès spéculatifs vers la mémoire et pour les différentes frontières prévues. Pour la mémoire virtuelle, on en distinguera deux :

- les entrées de page qui peuvent être utilisables ou non par un utilisateur/superviseur (*e.g.* à l'aide d'un bit dédié),
- les pages qui sont accessibles ou non selon les informations spécifiées par l'entrée de page.

Pour chacune, les données contenues doivent donc être vues comme inexistantes en cas d'accès non autorisé. Il ne doit pas être possible de savoir si une page non possédée existe ou non [74, 80].

Duplication

Les différents principes sur les ressources partagées matérielles modifient en profondeur leur fonctionnement. Plus particulièrement, le cas des mémoires caches a des conséquences plus larges au niveau de l'utilisation de la mémoire.

L'un des intérêts des mémoires caches est de permettre à plusieurs utilisateurs de mettre en commun les données ramenées afin d'y accéder plus rapidement. Cela est utile pour les plages mémoires partagées : une même copie en cache peut ainsi être réutilisée plusieurs fois, permettant un gain de performances, mais aussi de place. Cependant, en lien avec le Principe 7.3.1, cela mène également à des fuites d'information entre les utilisateurs de cette même mémoire partagée. Appliqué au cas des plages mémoires partagées, on obtient donc le principe suivant :

> **Principe de conception 7.6.2** ***: Plage mémoire dupliquée***
> *Si des mécanismes sont capables de dupliquer des plages mémoires, chaque donnée dupliquée ne doit pas être détectable par un autre domaine de sécurité.*

Ce principe n'est donc finalement qu'un renforcement dû Principe 7.3.1 en considérant la mémoire partagée.

Cohérence

L'utilisation de copies de la mémoire passe aussi par la mise en place de mécanismes de cohérence. Si l'un des utilisateurs modifie une des copies, il est alors nécessaire de répercuter le changement sur l'ensemble des copies. Cette opération prend un certain temps à se propager sur l'ensemble du système. Là encore, si ce temps varie selon l'utilisateur modifieur (*e.g.* selon le cœur ou thread où il est exécuté), un attaquant capable d'observer cette propagation temporelle pourrait retrouver des informations. On en déduit alors le principe suivant :

> **Principe de conception 7.6.3** ***: Cohérence mémoire***
> *Si une plage mémoire lisible et modifiable est partagée, une donnée modifiée par un domaine de sécurité doit être propagée de manière transparente aux autres domaines, indépendamment du contexte modifiant.*

Les mécanismes de cohérence étant déjà suffisamment complexes,

introduire une gestion de plusieurs copies au sein d'un même cache tout en assurant des propriétés de sécurité n'est pas souhaitable. Finalement, on en revient donc à un principe assez commun, qui est certes radical, mais essentiel pour assurer simplement l'isolation :

Principe de conception 7.6.4 *: Plage mémoire modifiable*
Une plage mémoire modifiable ne peut être dupliquée entre plusieurs domaines de sécurité.

La stricte application aurait elle aussi un impact sur les systèmes. Avant d'effectuer une copie locale d'une page modifiable, il faudrait vérifier qu'aucune autre copie n'existe pas. Alors seulement, une copie pourrait être faite et les demandes de nouvelles copies devraient être bloquées. En pratique, cela passerait donc par la mise en place d'un verrou à l'échelle du système.

Une autre solution encore plus radicale ferait appel à une gestion logicielle, et notamment par le système d'exploitation. Celui-ci serait responsable de garantir qu'en aucun cas plusieurs domaines de sécurité en cours d'exécution n'ont accès à une même plage mémoire modifiable.

7.7. Conclusion

À partir des informations reçues du logiciel, le matériel doit être capable de respecter les contraintes fixées et notamment ici celles d'isolation. Si nous avons vu que la disponibilité statique d'une ressource ne peut être masquée, les autres informations propres à un domaine de sécurité doivent rester confidentielles. Différents principes de conception permettent de respecter cet objectif, en se basant sur trois grandes méthodes.

L'allocation statique consiste à attribuer à un domaine de sécurité les ressources nécessaires à son exécution. Il n'est ainsi plus possible de détecter quand il utilise chacune d'entre elles. Le relâchement des ressources impose d'effacer les traces à la fin d'une exécution. Les prochains utilisateurs ne peuvent ainsi les retrouver. Enfin, le partitionnement des ressources permet de répartir les ressources entre différents utilisateurs simultanés. Aucun conflit d'entre eux n'est ainsi possible.

Dans le cas de processeurs simples n'exécutant qu'un domaine de sécurité simultanément, appliquer les principes de relâchement des ressources est suffisant. Il n'y a alors que le partage temporel à considérer : l'allocation statique est faite implicitement et le partitionnement est inutile. Combiner ces trois méthodes permet en revanche d'isoler des systèmes avec du partage spatial, comme dans des implémentations multicœurs ou avec multithreading. Dans tous les cas, chaque ressource partagée doit être adaptée en conséquence pour implémenter une politique d'isolation globale. Les principes de conception des ressources partagées sont résumés sur la Table 7.1.

L'objectif des prochains chapitres est de mettre en place les mécanismes pour implémenter ces principes au sein de différentes microarchitectures. Il sera alors possible d'évaluer leur impact réel, que ce soit pour la sécurité, les performances ou le coût matériel.

TABLE 7.1. – Récapitulatif des principes de conception et mécanismes.

Principe	Nom	Implémentation
7.2.1	Allocation statique	Tag et blocage des ressources
7.3.1	Partitionnement spatial	Séparation des ressources dupliquées
7.3.2	Séparation de la disponibilité	Partitionnement temporel ou allocation statique
7.4.1	Suppression des traces	Effacement ou écrasement des états persistants
7.4.2	Relâchement temps constant	Effacement ou écrasement réalisé en un temps fixe
7.5.1	Homogénéité	Implémentation de threads et cœurs équivalents
7.6.1	Plage mémoire possédée	Renforcement des frontières et de la spéculation
7.6.2	Plage mémoire dupliquée	Copie différente pour chaque domaine
7.6.3	Cohérence mémoire	Adaptation des protocoles de cohérence mémoire
7.6.4	Plage mémoire modifiable	Restriction de modification par un seul domaine

Implémentation d'une politique d'isolation | 8.

Résumé du chapitre : Dans ce chapitre, nous appliquons les principes de conception définis dans le chapitre précédent sur deux processeurs différents. Plusieurs mécanismes partagés temporellement et spatialement sont adaptés afin d'évaluer l'impact de notre approche sur le pipeline et le premier niveau de mémoire cache. Une version hybride des domes, présentée dans le chapitre 6 est également mise en place. Ces travaux sont publiés dans « *Under the Dome : Preventing Hardware Timing Information Leakage* » [148].

8.1 Contexte 119
8.2 Processeur Aubrac 120
Description de l'architecture de base 120
Support des domes 121
8.3 Processeur Salers 122
Description de l'architecture de base 122
Support des domes 123
8.4 Mémoire cache 125
Fonctionnement général des caches 125
Partie "mémoire" des caches 126
Partie "contrôleur" des caches 127
Partie "bus" des caches . . 127
8.5 Généricité de l'approche . 129
8.6 Autres mécanismes 130
8.7 Conclusion 130

8.1. Contexte

L'évaluation complète des principes de conception énoncés précédemment passe par la modification de systèmes existants, implémentant aussi bien du partage de ressource temporel que spatial. Pour ce dernier, nous avons vu que le cas le plus problématique est celui du multithreading simultané avec un partage du processeur en profondeur : un grand nombre de ressources est accessible par les différents threads. Pour une réelle étude de nos principes, une application sur un processeur avec SMT doit être considérée.

S'il existe un nombre croissant de processeurs RISC-V libres et ouverts, la quasi-totalité d'entre eux au moment de réaliser nos expérimentations se concentrait sur la simplicité ou l'exécution dans le désordre. En revanche, aucun à notre connaissance n'implémentait de multithreading simultané. Nous avons donc décidé de concevoir "en partant d'une feuille blanche" les processeurs utilisés pour la suite de notre étude. Un autre avantage de cette approche est la maîtrise complète des systèmes et mécanismes. Cet aspect est ici essentiel de par la profondeur des modifications qui sont effectuées.

Le choix de développer deux processeurs permet également d'étoffer notre étude. Ainsi, cela permet de mieux estimer l'impact général des principes de conception énoncés dans le chapitre 7. Dans notre cas, une montée en complexité est intéressante, avec une distinction claire des protections pour le partage temporel et celles pour le partage spatial.

Les différents circuits décrits ultérieurement sont basés sur le langage Chisel [149], un langage plus haut niveau que le Verilog ou le VHDL, et qui permet notamment un important paramétrage au moment de la génération du circuit. Les sources sont accessibles et utilisables librement sur le GitLab de l'Inria[1].

[149]: (2021), *Chisel/FIRRTL*

1: Répertoire GitLab : https ://gitlab.inria.fr/mescoute/hsc-eval

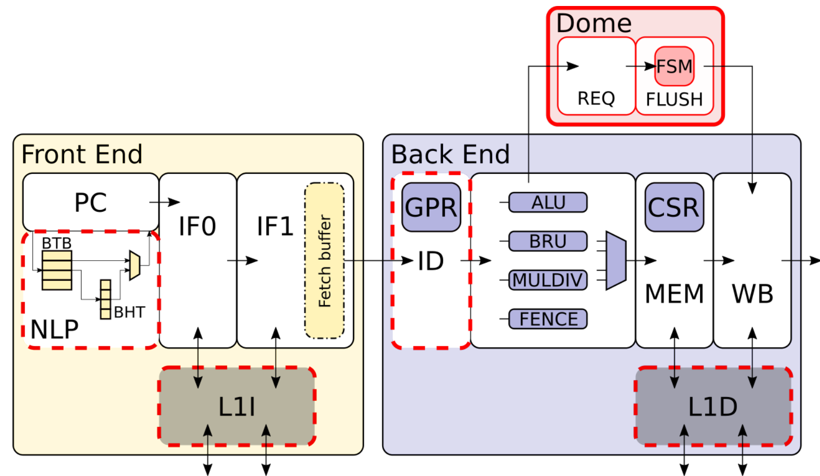


Figure 8.1. – Vue globale de la microarchitecture Aubrac. Le processeur de base est composé de 5 étages de pipeline. Il est décomposé en deux parties principales : le *front-end* responsable de la récupération des instructions et le *back-end* qui gère leur exécution. L'unité d'exécution *Dome* n'est nécessaire que pour le support des domes. Les éléments entourés par des pointillés rouges sont les principaux modules modifiés pour le support des domes, ici notamment pour le support du flush.

8.2. Processeur Aubrac

Nous nous concentrons dans un premier temps sur les modifications nécessaires pour permettre l'isolation lors de partages temporels. Le point de départ est donc le processeur Aubrac, développé spécialement à cet effet.

Description de l'architecture de base

Le processeur Aubrac est un processeur basé sur l'architecture RISC-V 32 bits [139, 142]. Ce processeur est basé sur un pipeline 5 étages avec exécution dans l'ordre, comme décrit précédemment dans le chapitre 2 et illustré ici sur la Figure 8.1. En plus du jeu d'instructions de base RV32I, il supporte également l'extension "M" pour les multiplications et divisions, "ZiCSR" pour l'utilisation de CSR et "ZiFencei" pour la synchronisation écriture/fetch mémoire. Plusieurs unités d'exécutions sont pour cela disponibles : une unité arithmétique et logique (ALU), une unité de branchement (BRU), une unité de multiplication et division (MULDIV) et une unité dédiée aux instructions de `fence` (gère des opérations sur les mémoires).

Cette microarchitecture implémente plusieurs mécanismes pour évaluer nos principes de conception. Tout d'abord, le processeur contient un prédicteur de branchement simple, aussi appelé prédicteur de la prochaine ligne (NLP), composé d'un BTB et d'un BHT. Les sauts directs et branchements conditionnels déjà rencontrés ont donc la possibilité d'être correctement anticipés. Le processeur comporte également un premier niveau de caches, avec deux mémoires différentes pour les instructions (L1I) et les données (L1D). La structure des caches ainsi que les modifications sont détaillées plus tard dans une section dédiée.

En utilisant certaines fonctionnalités du langage Chisel, ce circuit (ainsi que ceux détaillés ultérieurement) a été conçu de manière à être grandement paramétrable. Par exemple, la taille des tables de prédiction, des caches ou même leur simple utilisation peuvent être modifiées au moment de la création du design par les paramètres utilisés (voir Annexe A).

Support des domes

À partir du circuit décrit précédemment, certaines modifications sont nécessaires pour le support des domes.

Pipeline Afin d'évaluer le plus simplement possible nos principes de conception, la version des domes implémentée est celle utilisant les CSR. Pour rappel dans cette version, de nouveaux registres `domecur*` et `domework*` définissent respectivement le dome en cours d'utilisation et le prochain exécuté. Une seule nouvelle instruction `dome.switch rd` permet de basculer vers le prochain dome. Au niveau du pipeline, en plus de l'ajout des CSR, on distingue deux autres modifications principales : l'ajout d'une nouvelle unité d'exécution pour les `dome.switch rd`, ainsi que la modification de certaines ressources pour être capable de réaliser l'opération de flush. Ce processeur n'exécutant qu'un processus à la fois, il n'y a ici aucun partage spatial à prendre en compte.

Unité de gestion des domes Une unité d'exécution dédiée est responsable de la gestion des ressources pour les domes. Ainsi, quand un changement est détecté (lors d'un `dome.switch rd`), son rôle est de préparer les différentes ressources en déclenchant le mécanisme de flush. Pour cela, une FSM représente l'état du système : elle reprend celle illustrée sur la Figure 7.8. Deux versions de flush sont utilisables :

1. une avec temps variable qui s'arrête quand toutes les ressources sont marquées comme prêtes,
2. une avec temps constant qui, à l'aide d'un compteur matériel, effectue l'opération en un nombre fixe de cycles.

Dans le cas du processeur Aubrac, les ressources considérées sont aussi bien les tables de prédictions que les mémoires caches ou le pipeline lui-même. Lorsqu'un `dome.switch rd` est détecté, l'unité attend que les instructions précédentes soient terminées afin de ne pas les perturber, puis elle supprime toutes les instructions suivantes. Aucune trace ne persiste donc avant/après l'exécution de cette instruction, permettant l'isolation temporelle.

Notes : le support des domes est lui aussi implémenté (ou non) en utilisant un simple paramètre lors de la génération du circuit, de la même manière que précédemment (`useDome = true` pour les utiliser.). Ce paramétrage permettra lors de l'étape d'évaluation détaillée dans le chapitre 9 de simplement pouvoir quantifier le surcoût des modifications, en comparant des versions du processeur avec ou sans le support des domes.

Flush des ressources Dans le cas de ce processeur, le flush des ressources consiste simplement en l'invalidation des données, métadonnées ou instructions stockées. Au niveau des tables de prédiction, des bits de validité sont simplement remis à zéro. Pour les instructions chargées dans le pipeline, c'est là aussi le bit de validité associée à chacune d'entre elles dans l'étage de fetch qui est annulé.

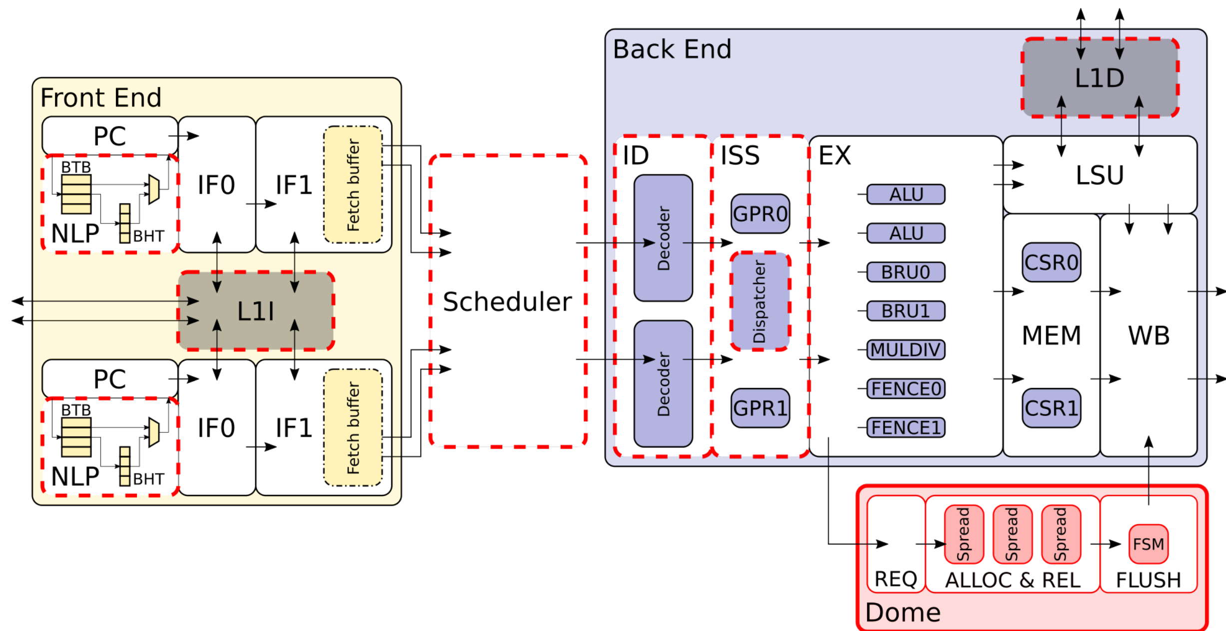


Figure 8.2. – Vue globale de la microarchitecture Salers. Le processeur de base est composé de 6 étages et supporte l'exécution de deux threads simultanément. Comme pour Aubrac, l'unité d'exécution *Dome* n'est nécessaire que pour le support des domes mais est ici utilisée par les deux threads. Les éléments entourés par des pointillés rouges sont les principaux modifiés pour le support des domes. Ici, cela comprend ceux implémentant le support du flush et aussi ceux supportant l'allocation ou le partitionnement.

8.3. Processeur Salers

Un processeur comportant du partage spatial était nécessaire pour notre évaluation. C'est donc dans ce but qu'a été conçu le processeur Salers.

Description de l'architecture de base

Tout comme Aubrac, le processeur Salers est basé sur l'architecture RISC-V 32 bits. Son pipeline est ici composé d'un étage supplémentaire appelé *issue stage* (ISS). Celui-ci comprend un répartiteur (ou *dispatcher*) dont le rôle est d'affecter une unité d'exécution à chaque instruction décodée. Les instructions sont exécutées dans l'ordre. Une représentation complète du processeur est détaillée sur la Figure 8.2. Les extensions "M", "ZiCSR" et "ZiFencei" sont également supportées.

Les deux différences majeures par rapport au processeur Aubrac sont la possibilité d'exécuter plusieurs instructions simultanément (Salers est un processeur dit *superscalaire*) et qu'elles puissent appartenir à des threads (support du SMT). Pour cela, un certain nombre de composants sont dupliqués :

- le *front-end* qui rassemble les éléments utilisés pour récupérer les instructions en mémoire,
- les GPR,
- les CSR,
- certaines unités d'exécution comme les ALU ou BRU qui sont essentielles à chaque programme.

Un CSR propre à ce processeur permet à chaque thread de se désactiver lui-même ou d'en activer un autre éteint en modifiant la valeur du registre. Dans le cas où un seul thread est actif, le processeur se comporte comme un processeur superscalaire classique : plusieurs instructions de

ce même thread sont exécutées simultanément si les dépendances du programme le permettent.

Salers implémente le même prédicteur de branchement NLP qu'Aubrac. Il est dupliqué entièrement pour chaque thread. Pour les mémoires caches L1I et L1D en revanche, celles-ci sont partagées entre les deux threads : une donnée utilisée par l'un est disponible pour l'autre. Différents paramètres permettent également de modifier les caractéristiques du circuit généré.

Support des domes

Tout comme pour le processeur Aubrac, le support des domes se résume en l'ajout d'une unité d'exécution dédiée et en la modification de certains éléments du pipeline.

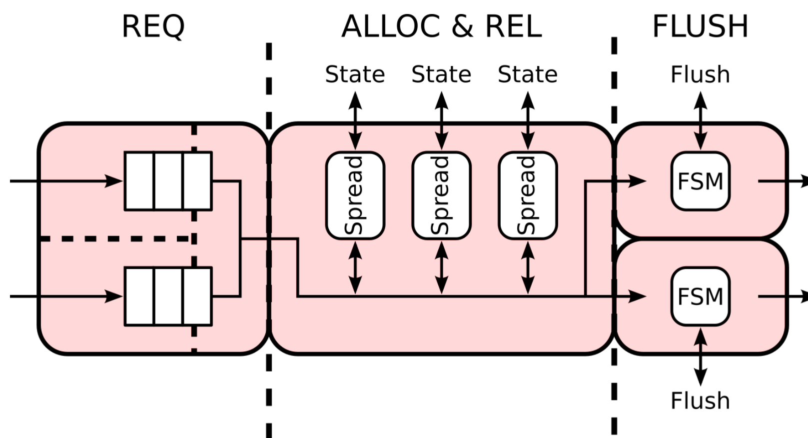


Figure 8.3. – Vue globale de l'unité de gestion des domes.

Unité de gestion des domes Dans Salers, cette unité d'exécution est partagée entre les deux threads. Ainsi, lorsque des opérations sur les domes sont nécessaires, les requêtes lui sont transmises. L'unité est conçue afin d'isoler les opérations des deux threads : aucune contention ne doit apparaître dans cette unité.

Pour cela, le premier étage REQ de l'unité sépare statiquement les requêtes dans deux buffers séparés. Le second étage, où est effectuée la gestion des ressources, est lui conçu en utilisant le partitionnement temporel. À tour de rôle, chacun des threads a la possibilité d'envoyer des requêtes d'allocation/ libération aux unités de répartition (*spread unit*). Théoriquement, une unité de répartition est nécessaire pour chaque type de ressource (ALU, BRU, ligne de cache *etc.*). Il est cependant possible de les regrouper dans le cas où certaines ressources sont en nombres proportionnels et nécessaires pour toutes les exécutions[1]. Le fonctionnement des unités de répartition est détaillé plus loin.

1: Dans le cas de Salers, au minimum une ALU, une BRU et une ligne de cache sont nécessaires pour chaque exécution. Ainsi, il est possible de rassembler les demandes d'allocation et libération dans une seule unité de répartition. Admettons que l'on ait 2 ALU, 2 BRU et 8 lignes de caches dans notre système. Les requêtes peuvent donc être gérées par groupe de 1 ALU, 1 BRU et 4 lignes de cache. Le circuit est ainsi simplifié. La contrepartie est une réduction de la flexibilité, même avec l'utilisation des poids des capacités.

Les identifiants des domes sont sur 32 bits. Propager l'ensemble des bits et effectuer sur ces valeurs des comparaisons peut être coûteux. Ainsi, une unité de répartition est également implémentée pour la gestion des emplacements des domes. À chaque fois qu'un `dome.switch rd` est détecté, une demande d'allocation d'emplacement est effectuée. Si le prochain dome est déjà exécuté par un autre thread, alors le même emplacement est utilisé. Toutes les opérations propres à un dome, qu'elles soient effectuées sur différents threads ou non, sont donc bien associées. Sinon, un nouvel emplacement est demandé. Pour exécuter potentielle-

ment deux threads dans deux domes différents, deux emplacements sont nécessaires.

Ce second étage détermine si un `dome.switch rd` peut avoir lieu, selon la disponibilité de l'ensemble des ressources demandées. Le troisième étage ne déclenche l'étape de flush qu'en cas de succès. Chaque thread a ici sa propre FSM pour gérer les opérations de flush. Celles-ci sont également taguées avec le dome correspondant : cela permet de n'impacter que les ressources qui lui appartiennent.

Unité de répartition La gestion des ressources étant similaire quelque soit le type de la ressource, une unité générique a été conçue spécialement. Appelée unité de répartition, elle reçoit des requêtes d'allocation/relâchement et gère l'état des ressources en conséquence.

Pour cela, des registres sont associés à chaque ressource. Ils contiennent plusieurs informations :

- la disponibilité de la ressource,
- le dome propriétaire,
- le thread responsable de l'allocation (nécessaire pour savoir si elle peut être relâchée),
- le numéro de port au sein du dome. Cette valeur permet de dissocier des ressources d'un même type.

Un scénario d'utilisation d'une unité de répartition est présenté sur la Figure 8.4.

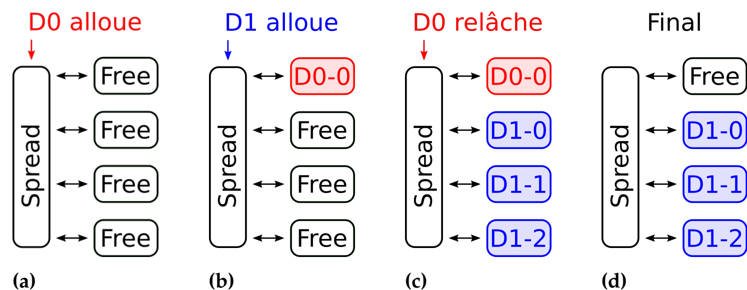


Figure 8.4. – Scénario d'utilisation d'une unité de répartition. L'unité de répartition (*spread unit*) gère les ressources d'un même type. Au départ dans la Figure 8.4a, toutes les ressources sont libres et une demande d'allocation du dome D0 arrive. Dans la Figure 8.4b, une ressource est attribuée à D0. Elle est alors taguée et un numéro de port au sein du dome lui est attribué (ici port 0 d'où D0-0). D1 effectue une demande à son tour. Le poids demandé par D1 étant plus grand que D0, plus de ressources lui sont attribuées dans la Figure 8.4c. Elles sont elles aussi taguées. D0 demande alors à relâcher ses ressources, ce qui est fait dans la Figure 8.4d.

Pipeline Chaque thread possède ses propres CSR. Les nouveaux registres pour le support des domes ont donc été ajoutés pour chacun d'eux. Les capacités sont également implémentées avec `domecurcap` et `domeworkcap`. Un des bits est utilisé pour définir l'allocation (ou non) de l'unité MULDIV, deux autres indiquent le poids du dome. Le support de la nouvelle instruction `dome.switch rd` a été rajouté dans chaque décodeur ainsi que l'unité d'exécution associée. En plus du support de l'opération de flush (semblable au processeur Aubrac), certaines modifications supplémentaires étaient nécessaires pour l'isolation spatiale.

La première concerne l'ordonnanceur (ou *scheduler* sur la Figure 8.2). Ce composant est responsable de la répartition des instructions depuis les différents fetch buffers vers les ports d'entrée des décodeurs. Sans dome, dès qu'un thread est désactivé, l'ordonnanceur réalloue dynamiquement les ports vers l'autre thread afin de maximiser l'utilisation des ressources. Avec le support des domes, l'ordonnanceur prend à présent en compte

l'allocation des ressources effectuée précédemment : seuls les ports alloués par le dome correspondant sont utilisables.

La seconde modification concerne le décodeur. Certains domes n'allouent pas l'unité MULDIV. Dans ce cas, le décodeur doit s'adapter s'il rencontre tout de même des instructions de multiplication ou division. Ici, pour simplifier l'étude, il a été décidé de remplacer ces instructions par des `nop` (pas d'opération) dans ce cas précis.

Enfin, le cœur du SMT repose sur le répartiteur. À l'origine, il prend en compte le type de l'opération et la disponibilité de l'ensemble des unités. Après modification pour le support des domes, celui-ci ne peut attribuer à chaque opération qu'une unité allouée par le même dome. La détection de contention entre les domaines de sécurité n'est donc plus possible.

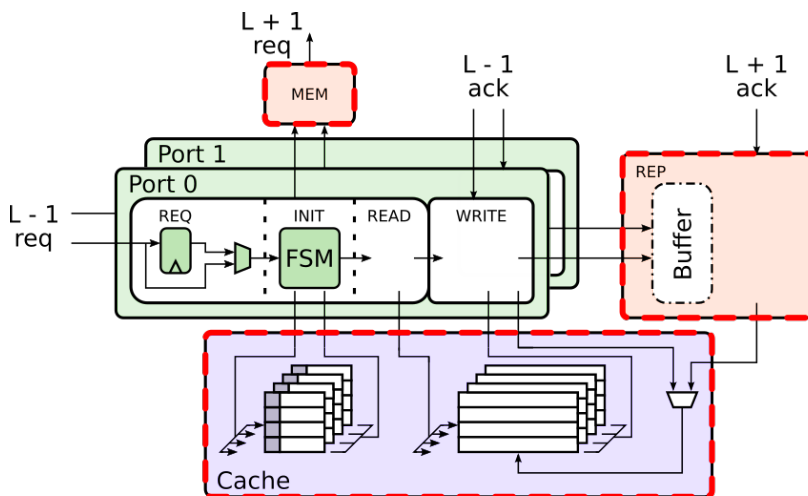


FIGURE 8.5. – Vue globale de la mémoire cache. La mémoire cache utilisée se décompose en trois éléments. Le cache lui-même contient les bits de validité, les tags et les données. Les ports sont les parties du contrôleur qui gèrent les requêtes venant du processeur (un port par thread). Enfin, les derniers éléments du contrôleur (MEM et REP) sont responsables de lancer les accès mémoire vers le niveau supérieur et d'effectuer les remplacements.

8.4. Mémoire cache

En plus des processeurs, l'application d'une politique d'isolation au sein de la microarchitecture passe aussi par la modification des mémoires et notamment des caches. Nous décrivons ici comment les différents éléments qui les composent peuvent être modifiés afin de répondre aux principes de conception sur les ressources partagées. La mise en place d'un nouveau modèle mémoire comme évoqué dans le chapitre 7 est laissée pour de futurs travaux avec la mémoire virtuelle. La Figure 8.5 présente l'architecture globale des mémoires caches utilisées.

Fonctionnement général des caches

Le cache implémenté a un fonctionnement classique. Les requêtes en provenance du processeur (requête du niveau L - 1 d'où *L - 1 req*) arrivent en entrée du port. L'étage REQ gère l'acquittement du bus (détaillé plus tard). Chaque requête va ensuite dans l'étage INIT. Un accès au cache est effectué pour voir si la donnée est présente :

- Si c'est un *hit*, la requête poursuit dans l'étage de lecture READ. Au cycle suivant, la donnée lue est disponible et peut être envoyée (L - 1 ack). En cas d'écriture, la donnée reçue est transmise au cache.
- Si c'est un *miss*, la FSM lance la procédure de remplacement. Une requête est envoyée à l'interface mémoire MEM du contrôleur pour

accéder au niveau supérieur (L + 1 req). Une autre est propagée à travers le port jusqu'à l'étage REP responsable de l'acquittement du niveau supérieur. Quand il reçoit les données pour le remplacement, il doit également écrire ces données dans le cache. Alors seulement la FSM se comporte comme pour un *hit*.

Le cache implémentant une politique d'écriture traversante, les interfaces vers le niveau de mémoire supérieur MEM et REP sont aussi utilisées en cas d'écriture.

Comme pour les processeurs, ce circuit est conçu d'une manière paramétrable : le nombre d'ensembles, de lignes ou la taille des buffers tout comme le nombre de ports peuvent être changés. Un seul port est nécessaire pour Aubrac et deux pour Salers.

Partie "mémoire" des caches

Le cœur même d'une mémoire cache sont ses éléments mémorisant où sont stockées les lignes et autres informations. Selon le Principe 7.3.1, elles doivent être réparties statiquement entre les différents domes exécutés simultanément. La stratégie de partitionnement retenue est le partitionnement par sous-ensemble vu précédemment sur la Figure 7.4. Chaque emplacement de ligne est associé à un dome par une unité de répartition. La logique de comparaison est modifiée pour prendre en compte cet élément. Une ligne n'est utilisée que si elle est valide, a le même tag et le même dome que la requête. La donnée qu'elle contient peut alors être manipulée. Sinon le système procède à un remplacement de ligne de manière classique.

La politique de remplacement doit, elle aussi, prendre en compte le partitionnement : une ligne d'un dome ne peut remplacer qu'une ligne de ce même dome. Les politiques associant des informations propres à chaque ligne se prêtent particulièrement bien à ce changement. Partitionner les lignes inclut les informations qui leur sont propres. Deux politiques de remplacement bien connues conviennent particulièrement :

- La politique LRU permet de choisir la ligne la plus anciennement utilisée à partir d'un compteur associé à chaque ligne. Il faut donc comparer le compteur, mais aussi le tag du dome pour déduire la ligne la plus ancienne de ce même dome.
- La politique de bit pLRU associe un bit de statut à chaque ligne. Il n'y a donc qu'à considérer ce bit en plus du tag du dome.

Dans notre cas, la politique pLRU est implémentée également pour son faible coût.

Une autre stratégie de partitionnement possible consiste à considérer un cache comme un groupe de sous-caches comportant un nombre réduit d'ensembles (*e.g.* diviser un cache de 8 ensembles en 4 sous-caches de 2 ensembles). Ainsi, l'attribution des ressources et la vérification des tags se font uniquement à l'échelle des sous-caches. Cependant, cette stratégie revient à modifier plus en profondeur l'organisation des caches. De par la taille réduite de nos caches (étant des caches de premier niveau), la première stratégie est parfaitement adaptée et a été utilisée. Il pourrait tout de même être intéressant d'étudier les compromis de chaque stratégie lors de la conception en considérant des mémoires caches avec des contraintes différentes (notamment des tailles plus importantes pour des L3 ou plus).

Le partitionnement n'assure finalement que l'isolation spatiale. Il

est également nécessaire de considérer l'isolation temporelle : il faut donc implémenter le principe de flush. Au niveau de la mémoire cache, cela revient simplement à remettre tous les bits de validité des lignes appartenant à un dome à 0. Ceci est effectué par un bus dédié contrôlé par l'unité de répartition.

Partie "contrôleur" des caches

Le contrôleur est le mécanisme responsable de la gestion du cache. Il est composé ici de deux parties :

1. Des ports (représentés en vert sur la Figure 8.5) qui gèrent les accès du processeur. Dans notre cas, on a un port différent pour chacun des deux threads.
2. Deux étages MEM et REQ qui gèrent les requêtes vers la mémoire supérieure ainsi que les remplacements.

Un flush pour l'isolation temporelle consiste ici à attendre que toutes les requêtes en cours soient terminées afin d'assurer que les instructions avant un `switch` aient bien été traitées. Au cours d'un flush, aucune nouvelle requête du même dome n'est acceptée.

Si le contrôleur est aussi partagé spatialement, plus de changements sont alors nécessaires. Chaque port du contrôleur est associé à un thread, qui lui-même n'exécute qu'un dome à la fois (un dome actif par thread). Localement, on se retrouve ainsi dans une situation de partage temporel uniquement : aucune modification supplémentaire n'est requise. En revanche, la deuxième partie du contrôleur doit être capable d'effectuer des accès pour n'importe quel dome exécuté : elle doit donc être partitionnée. On utilisera pour cela le mécanisme de partitionnement temporel dédié aux pipelines de la Figure 7.7. La logique de chaque étage de cette partie du contrôleur appartiendra successivement à un dome différent, et les registres sont eux dupliqués.

Notes : Le partage spatial ne s'applique évidemment que lorsque le nombre maximal de domes actifs est supérieur à 1. Autrement, ces modifications sont simplifiées/supprimées à la génération du code.

Comme évoqué précédemment, cette méthode à un impact sur la latence. Dans le cas de notre contrôleur, elle est détaillée dans la Table 8.1. Ces valeurs se retrouvent lors de l'évaluation présentée sur la Figure 9.8.

Table 8.1. – Latence pour des accès au cache L1. Dans le cas d'un *hit*, la réponse est toujours disponible au cycle suivant. Dans le cas d'un *miss*, le nombre de cycles varie selon le nombre d'accès A nécessaires pour récupérer une ligne entière et le nombre D de domes actifs. En supposant que la deuxième partie du contrôleur soit disponible dès le premier cycle, la latence suit alors la formule : $1 + 2 + (D * (A - 1)) + 1 + 1$. On prendra ici A = 4.

Domes actifs	1	2
Hit (cycles)	1	1
Miss (cycles)	8	11

Partie "bus" des caches

Afin d'échanger des données avec les autres niveaux de mémoires, les contrôleurs de cache communiquent en utilisant des bus. Dans le cas d'un partage spatial où plusieurs domaines de sécurité existent simultanément, les différents bus doivent donc être capables d'indiquer à quel domaine appartient une requête. De plus, le protocole de ces bus doit être conçu en considérant les cas de contention. Nous voyons ici les différentes réflexions liées à la modification du bus utilisé dans nos implémentations.

Le point de départ est un bus mémoire classique permettant d'effectuer des requêtes de lecture et d'écriture comme décrit sur la Figure 8.6 et la Figure 8.7. Les signaux sont nommés du point de vue du contrôleur maître du bus : *o_** indique un signal de sortie quand *i_** est une entrée. Ce bus permet la conception de contrôleurs ayant des temps de réponse variables (tous les caches ne répondent pas avec le même nombre de cycles). Ainsi *o_valid*, *o_rw* et *o_addr* indiquent la nature de l'opération souhaitée par le maître quand *o_wvalid* et *o_wdata* / *i_rvalid* et *i_rdata*

permettent respectivement l'envoi des informations pour une écriture ou la réception des données pour une lecture. *i_ready* permet de savoir si le contrôleur esclave est prêt pour une nouvelle requête.

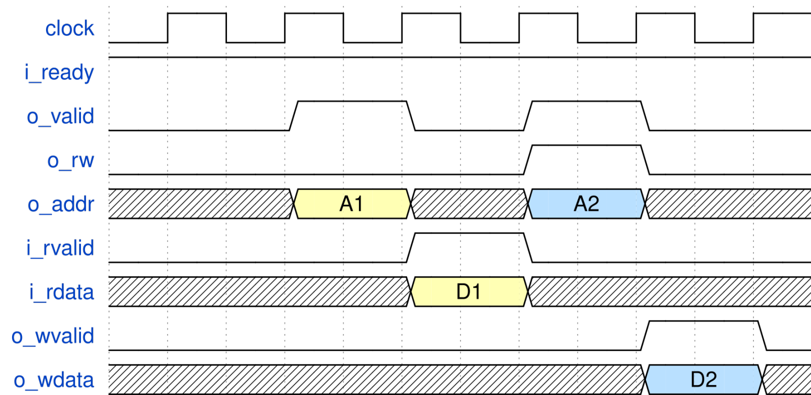


Figure 8.6. – Bus mémoire version 1 : lecture et écriture. La première requête en jaune correspond à une lecture : le bus dédié à cette opération renvoie une réponse au cycle suivant. La seconde requête en bleu est une demande d'écriture : la donnée à écrire est envoyée également au cycle suivant.

Dans ce protocole, aucune des requêtes n'est distinguée au niveau de l'utilisateur : ainsi, si l'une d'elles met plus de temps que prévu pour être traitée (*e.g.* un cache *miss*), elle retarde également celles qui suivent. C'est le cas classique de contention menant à de potentielles fuites temporelles. Sur la Figure 8.7, on voit que la réponse de la deuxième requête est retardée à cause du ralentissement sur la première requête.

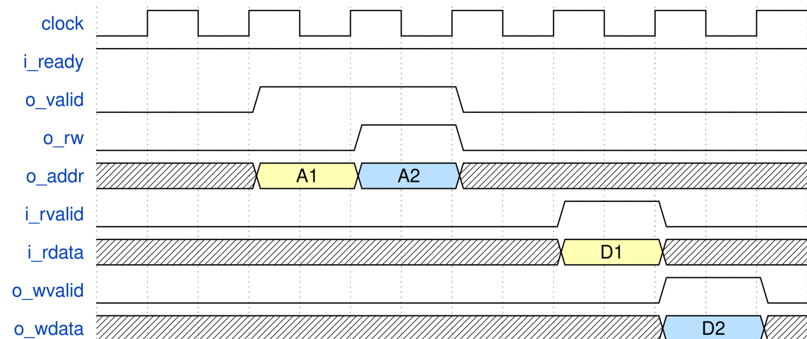


Figure 8.7. – Bus mémoire version 1 : contention

Pour modifier ce protocole, la notion de dome doit donc y être intégrée. Pour cela, plusieurs changements ont été effectués :

- L'identifiant du dome (ou une valeur interne propre) est utilisé pour taguer la requête.
- L'identifiant du dome est également utilisé pour taguer les réponses (écriture dans un sens, lecture dans l'autre).
- Le signal *i_ready* est découplé pour chaque dome pouvant être exécuté simultanément. Dans notre cas, on considère qu'il n'y a que deux domes simultanément au maximum (un pour chaque thread dans le processeur Salers) d'où les deux *i_ready*. Ainsi, le contrôleur maître sait à tout moment si la requête dans un dome particulier sera acceptée ou non.

La Figure 8.8 illustre ainsi le fonctionnement de ce protocole. **_-REQ* / **_ACK* indiquent le dome traité au sein de l'étage du contrôleur correspondant en utilisant le principe de partitionnement temporel. Une première requête est envoyée dans le dome Do1 puis une seconde dans le dome Do2. La première est retardée, mais cela n'impacte pas la seconde, les deux étant dans des domes différents. En revanche, la troisième est

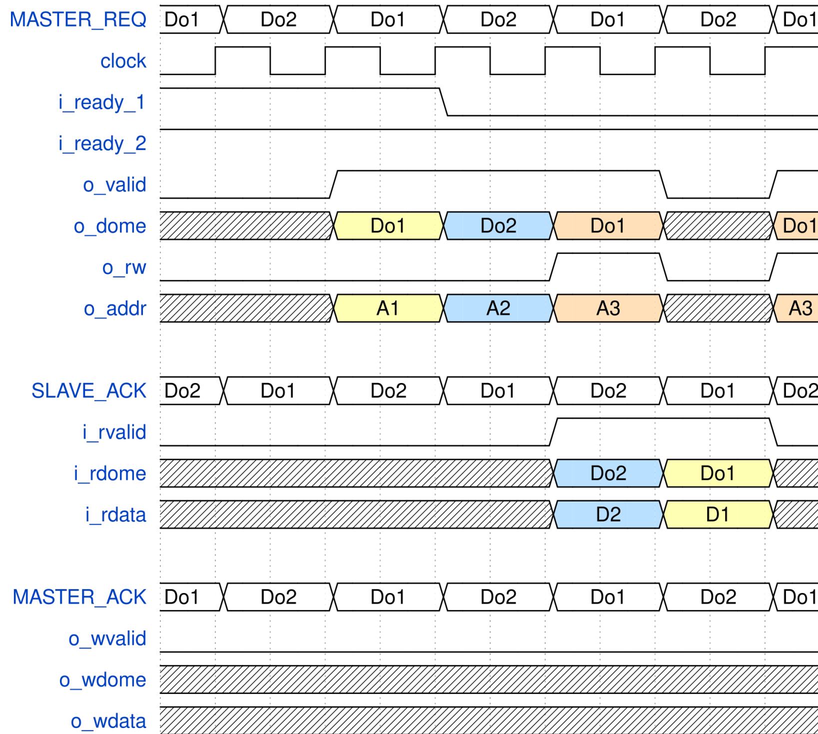


Figure 8.8. – Bus mémoire version 2 : support des domes

retardée, car elle se trouve dans le même dome Do1. Ainsi, la contention est interne à chaque dome.

On note que dans ce bus, les signaux de réponse (les signaux propres à l'écriture ou la lecture) sont de base totalement découplés. Ainsi, comme il ne peut y avoir qu'une requête à la fois, les bus de réponse sont sous-utilisés (au mieux à 50%). Une solution, pour allier performances et isolation, serait donc d'également découpler la partie requête. Ainsi, des contrôleurs avec un support complet du protocole seraient capables d'effectuer une écriture et une lecture à la fois (pour un même dome ou deux différents) afin de limiter les pertes de performances. C'est finalement une approche assez semblable à celle du protocole AXI, en simplifiée, et basée avant tout sur la sécurité.

8.5. Généricité de l'approche

Une grande partie de la complexité des modifications se trouve dans l'unité de gestion des domes. Tous les mécanismes de contrôle des flush et de gestion des ressources y sont implémentés. Dans le cas du support de plusieurs domes, les principes d'isolation spatiale doivent également y être appliqués.

Heureusement, cette partie est grandement indépendante du reste du processeur. Seules les interfaces suivantes doivent être présentes :

- les ports de gestion des ressources pour indiquer leur état et déclencher le flush,
- le port de requête de l'unité,
- le port de réponse de l'unité.

À partir du nombre de domes à supporter (un ou plusieurs), il est donc possible d'envisager de réutiliser l'unité adéquate. Enfin, si le support de nouvelles ressources doit être ajouté (*e.g.* une ou plusieurs unités pour les nombres flottants), il suffit de réutiliser l'unité de répartition.

8.6. Autres mécanismes

De par la variété et la complexité des microarchitectures modernes, des essais n'ont pas pu être effectués sur tous les types de mécanismes. Il reste cependant possible d'imaginer l'impact de l'application des principes de conception. Voici certains des principaux mécanismes à étudier.

Exécution dans le désordre Nous nous sommes jusqu'à présent uniquement concentrés sur l'exécution dans l'ordre. Cependant, l'exécution dans le désordre est très utilisée dans les microarchitectures modernes : il faut donc également être capable de l'adapter. Pour cela, le point principal est qu'aucune spéculation ne doit avoir lieu à travers les frontières d'isolation. Cela s'applique aussi bien pour les ressources partagées que pour les plages mémoires partagées. Ainsi, un `dome.switch` doit représenter une barrière temporelle de spéculation. De même, les tags utilisés pour le partitionnement doivent représenter des barrières spatiales de spéculation.

Prefetch Un prefetch étant également une forme de spéculation, les mêmes règles s'appliquent. Cependant, les prefetchers conservent un rôle essentiel au niveau de la hiérarchie mémoire. Dans le cas d'un partage spatial, nous avons vu que certaines parties des contrôleurs sont scindées et la disponibilité des bus est répartie entre les domes. Un dome n'a donc ces ressources que durant un temps limité qui doit être exploité au maximum. Avec cet objectif, les prefetchers peuvent avoir un rôle d'autant plus important. Leur rôle serait de maximiser la disponibilité des ressources. En cas de spéculation exacte, les gains ne seraient que plus importants.

Mémoire virtuelle Le cas de la mémoire virtuelle ne fait pas exception et doit également être considéré. Ainsi, les différentes tables utilisées pour accélérer la traduction (*cf.* les TLB) doivent être partitionnées entre les domes et vidées lors des `dome.switch rd`. Dans ce cas-là en revanche, on peut envisager un impact d'autant plus important au lancement d'un nouveau domaine de sécurité. Celui-ci se trouvera alors sans donnée préchargée ni pour la mémoire virtuelle, ni pour les accès mémoires avec les caches.

Le cas de la mémoire virtuelle est étroitement lié à la mise en place d'un modèle mémoire pour les plages mémoires partagées. C'est actuellement ce mécanisme qui permet d'établir les règles de partage et de restriction sur les accès mémoires. Pour un support complet de l'ensemble des principes du chapitre 7, des travaux plus importants pourraient donc être nécessaires.

8.7. Conclusion

Dans ce chapitre, nous avons vu comment adapter la microarchitecture au support d'une politique d'isolation. À partir des principes définis précédemment, deux processeurs ont été modifiés pour nous permettre d'évaluer l'impact à différents niveaux de complexité. L'ensemble des ressources partagées modifiées est décrit sur la Table 8.2.

Le processeur Aubrac permet maintenant une isolation temporelle entre les différents domaines exécutés. Le principe de relâchement est appliqué à chaque changement de domaine sur l'ensemble des ressources partagées. Aucun état persistant n'est ainsi conservé.

En plus de l'isolation temporelle, le processeur Salers ajoute l'isolation spatiale entre ses deux threads. Cela passe dans un premier temps par la mise en place d'un système d'allocation statique des ressources grâce à des unités dédiées. Les unités d'exécution sont allouées de manières exclusives : elles ne sont utilisables que par un seul dome. La même stratégie est appliquée aux lignes de cache afin de permettre un partitionnement spatial. Enfin, le contrôleur des caches étant essentiel pour tous les domes en cours d'exécution, il est conçu en utilisant le partitionnement temporel. Chaque thread, même si tous ne sont pas placés dans un même dome, peut ainsi accéder aux niveaux de mémoire supérieurs sans générer de contention.

Le support des domes, pour Aubrac ou Salers, passe par l'ajout d'une unité d'exécution dédiée à la gestion des ressources. Cette unité est en grande partie indépendante du reste du processeur. Une réutilisation pour d'autres processeurs peut donc être envisagée.

Ces deux implémentations doivent à présent être évaluées pour vérifier leur apport du point de vue de la sécurité. De plus, il est également nécessaire d'estimer le surcoût dû à ces modifications, autant sur le plan des performances que sur celui des ressources matérielles.

Table 8.2. – Récapitulatif des ressources partagées modifiées. T indique un partage temporel, S un partage spatial. Pour Aubrac, les ressources ne sont partagées que temporellement : seul l'effacement des traces est appliqué. Pour Salers, le cas de chaque ressource est détaillé.

Aubrac

Ressource	Partage	Mécanisme
*	T	Effacement

Salers

Ressource	Partage	Mécanisme
Pipeline	T + S	Effacement + Allocation statique + Partitionnement spatial
Buffers	T + S	Effacement + Partitionnement spatial
Unité d'exécutions	S	Allocation statique
Prédicteur de branchement	T	Effacement
BHT	T	Effacement
BTB	T	Effacement
Mémoire cache	T + S	Effacement + Partitionnement spatial + Partitionnement temporel
Emplacements mémoires	T + S	Effacement + Partitionnement spatial
Contrôleur	T + S	Effacement + Partitionnement spatial + Partitionnement temporel
Port mémoire	S	Partitionnement temporel

Évaluation 9.

Résumé du chapitre : Dans ce chapitre, nous évaluons les implémentations décrites dans le chapitre 8. Notre premier objectif est de tester les garanties de sécurité offertes par l'application des principes de conception du chapitre 7. Pour cela, nous introduisons une nouvelle suite de tests destinée à la détection des fuites temporelles dues à l'utilisation de ressources partagées. Un autre aspect étudié est le surcoût de ces principes selon deux métriques : les performances et les ressources matérielles nécessaires. Pour cela, nous réalisons plusieurs implémentations sur une cible FPGA. Ces travaux sont également présentés dans « *Under the Dome : Preventing Hardware Timing Information Leakage* » [148].

9.1 Timesecbench : une suite de tests pour la sécurité . . 133
Objectif et contraintes de développement 133
Scénario d'attaquant 134
Vulnérabilités couvertes par Timesecbench 134
9.2 Mesure d'efficacité des contremesures 136
Comparaison des implémentations vulnérables et sécurisées 136
Isolation temporelle 137
Isolation spatiale 139
9.3 Analyse des performances et du coût 141
Performances 141
Coût en ressources matérielles 144
9.4 Bilan et conclusion 145

9.1. Timesecbench : une suite de tests pour la sécurité

Lors de la conception d'un processeur, une phase de tests est nécessaire : elle permet de s'assurer que les fonctionnalités sont correctement implémentées. Cela passe entre autres par l'exécution de codes dont le résultat final correct est connu. Dans notre situation, en plus des fonctionnalités elles-mêmes, il est également nécessaire de pouvoir tester le système pour en évaluer les garanties de sécurité offertes. Pour cela, une suite de tests appelée *Timesecbench*[1] a été mise en place.

1: Répertoire GitLab : https ://gitlab.inria.fr/rlasherm/timesecbench

Objectif et contraintes de développement

Les principes de conception détaillés dans le chapitre 7 visent à guider un concepteur matériel pour concevoir des ressources partagées. La prochaine étape est donc de vérifier la bonne application de ces règles. Avec *Timesecbench*, l'objectif est de disposer d'un moyen d'évaluer l'implémentation de ressources partagées et des contremesures disponibles. Pour cela, différentes attaques sont reproduites pour tenter d'exfiltrer des informations, similairement à ce qui a pu être fait dans d'autres travaux d'évaluation [27]. Pour être réutilisable, *Timesecbench* doit être simple à mettre en œuvre et exécutable sur n'importe quelle implémentation. Trois contraintes doivent ainsi être respectées pour atteindre cet objectif.

Test des ressources partagées Nous cherchons à détecter les fuites temporelles dues à l'utilisation de ressources partagées. Bien que leurs implémentations puissent varier selon les processeurs, leur fonctionnement reste essentiellement le même. Les différents tests doivent donc être suffisamment génériques pour s'adapter. Ainsi, dans un premier temps, nous avons choisi de cibler les ressources partagées les plus communes.

Détection des vulnérabilités Avec cette suite de tests, notre objectif est la détection des sources de fuite temporelle. Ainsi, on se focalise ici sur les vulnérabilités potentielles. Le but final est d'avoir, pour un processeur donné, l'ensemble des sources de fuite existantes. Cette recherche d'exhaustivité s'oppose ici à la notion d'exploitabilité. On

ne cherche pas uniquement à détecter les fuites exploitables par un attaquant.

Indépendance du jeu d'instructions Être capable d'évaluer et comparer n'importe quelle implémentation implique de s'abstraire du jeu d'instructions. Le seul prérequis pour le fonctionnement d'un test doit être la présence de la ressource partagée ciblée. Pour cela, les différents codes ont besoin d'être développés dans un langage de plus haut niveau que l'assembleur : le compilateur sera alors responsable d'adapter chaque test au jeu d'instructions visé. Les différents codes ont donc ici été développés en langage C le plus indépendamment possible de tout paramètre matériel. À l'image de ce qui est fait avec d'autres suites comme Embench [150], certaines informations génériques restent nécessaires.

[150]: (2021), *Embench : A Modern Embedded Benchmark Suite*

Scénario d'attaquant

Timesecbench vise à détecter un maximum des fuites présentes. Le scénario d'attaquant doit faciliter autant que possible la réussite d'une attaque. Dans le cas des fuites temporelles, ce scénario correspond à celui d'un canal caché de communication. Pour rappel, l'attaquant contrôle alors deux domaines de sécurité qui cherchent à communiquer en utilisant des moyens détournés. Contrairement au scénario de canal auxiliaire, l'attaquant contrôle le canal de communication de bout en bout : il est libre de choisir la forme des informations transmises. Dans notre cas, les moyens mis à disposition de l'attaquant sont l'utilisation des différentes ressources partagées à sa disposition.

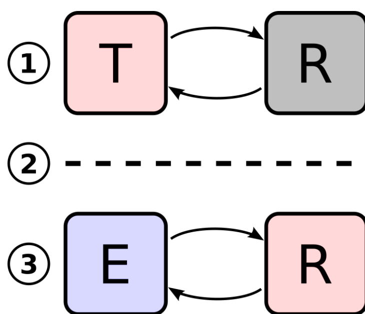


Figure 9.1. – Scénario utilisé pour Timesecbench. T correspond au cheval de Troie encodant l'information, E à l'espion la décodant et R à la ressource utilisée. R est ici partagée temporellement entre T puis E.

Ce scénario est illustré sur la Figure 9.1. Il correspond à l'utilisation d'une ressource partagée temporellement. Chaque test de ce type se décompose en trois étapes :

1. Le code d'un cheval de Troie est exécuté : il encode une donnée dans l'état d'une ressource partagée.
2. L'exécution du cheval de Troie se termine et celle de l'espion commence.
3. L'espion essaye de retrouver la valeur encodée dans l'état de la ressource.

Dans le cas du partage spatial d'une ressource, ce scénario diffère un peu. L'étape 2 n'est alors plus nécessaire et les étapes 1 et 3 s'exécutent simultanément.

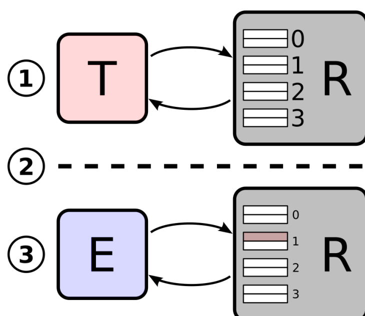


Figure 9.2. – Scénario utilisé pour les caches L1 partagés temporellement. T correspond au cheval de Troie encodant l'information, E à l'espion la décodant et R à la ressource utilisée. Une ligne est ramenée dans un des ensembles par le cheval de Troie (ici l'ensemble 1). Cette même ligne est ensuite détectée par l'espion qui peut en déduire la valeur 1.

Vulnérabilités couvertes par Timesecbench

Dans la version actuelle de *Timesecbench*, six tests différents ont été développés conformément aux ressources partagées disponibles sur nos implémentations. Ils concernent aussi bien des cas de partages temporels que spatiaux.

Exploitation temporelle du L1D La première ressource testée est le cache L1D partagé temporellement. La Figure 9.2 décrit le scénario du test. L'objectif de ce test est de vérifier si une donnée ramenée par le cheval de Troie est détectable ultérieurement par l'espion. Le cheval de Troie et l'espion partagent une plage mémoire assez grande pour remplir le cache L1D. Elle est dans un premier temps utilisée pour remplir et initialiser

l'état du cache. Pour un nombre E d'ensembles dans le cache, le cheval de Troie est capable de transmettre n'importe quelle valeur entre 0 et E - 1. Pour cela, il est responsable de ramener une ligne de cette plage dans l'ensemble correspondant. L'espion accède alors à une ligne à son tour. Si le temps d'accès correspond à un *hit*, alors la ligne correspondante est celle qui a été ramenée par le cheval de Troie. Le numéro de l'ensemble où se trouve la ligne indique la donnée transmise.

Exploitation temporelle du L1I La seconde ressource partagée ciblée est le cache L1I lui aussi partagé temporellement. La description vue précédemment pour le L1D sur la Figure 9.2 est toujours valable. L'objectif est de vérifier qu'une instruction ramenée par le cheval de Troie est détectable ultérieurement par l'espion. Le cheval de Troie et l'espion partagent toujours une plage mémoire, mais contenant maintenant des instructions à exécuter. Cette plage est assez grande pour remplir le cache L1I et est utilisée pour l'initialisation de la ressource. Elle est constituée d'instructions de saut qui, une fois dans le cache, sont placées dans chacun des ensembles. Le cheval de Troie exécute un saut situé dans l'ensemble S pour transmettre une donnée S. Plus tard, l'espion exécute également un des sauts : si le temps d'exécution correspond à un *hit*, alors le même saut a été exécuté en amont par le cheval de Troie.

Exploitation temporelle du BTB Un autre test exploite le partage temporel du BTB. Pour rappel, ce composant vu dans le chapitre 2 est responsable de la prédiction des sauts indirects. Pour son exploitation, un morceau de code est partagé entre le cheval de Troie et l'espion. Il doit être suffisamment grand pour couvrir l'ensemble du BTB. Ce bout de code est composé uniquement de sauts (vers *addr0*). Chacun de ces sauts est exécuté pour remplir le BTB et l'initialiser. Puis, pour transmettre la valeur S, le cheval de Troie modifie la destination du S-ième saut (vers *addr1*) réécrivant l'instruction, puis il l'exécute. La valeur du BTB pour ce saut est mise à jour. Enfin le cheval de Troie restaure l'état de la plage mémoire. Quand l'espion est exécuté, il sélectionne un des sauts, modifie la destination (vers *addr1*) puis l'exécute. Si la mesure du temps d'exécution indique qu'une prédiction correcte (la redirection vers *addr1* a été anticipée), alors c'est ce même saut qui avait été sélectionné par le cheval de Troie.

Exploitation temporelle du BHT Le quatrième test exploite le partage temporel du BHT. Pour rappel, ce composant est responsable de la prédiction des branchements conditionnels. Le cheval de Troie et l'espion possèdent à nouveau une plage mémoire partagée assez grande pour remplir le BHT (N instructions consécutives pour N entrées du BHT). Chaque instruction est un même branchement conditionnel répété plusieurs fois. Tous sont exécutés une première fois en s'assurant que la condition est fausse. Le BHT est ainsi initialisé. Pour transmettre la valeur S, le cheval de Troie s'assure que la condition est vraie et saute directement vers le S-ième saut. L'état du compteur associé à ce branchement dans le BHT est mis à jour. Quand l'espion est exécuté, il sélectionne un des branchements et le réalise. Si la mesure du temps indique que la prédiction est correcte, alors ce même branchement avait été sélectionné par le cheval de Troie.

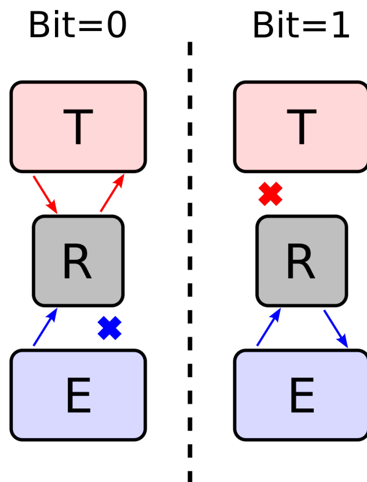


Figure 9.3. – Scénario utilisé pour la contention d'unité d'exécution. Le cheval de Troie T et l'espion E accède à une même ressource R. T est prioritaire sur E en cas de conflit. Si T et E accèdent à la ressource en même temps, E est bloqué et détecte la contention dynamique. Cela correspond à un bit de valeur 0. À l'inverse, si aucune contention n'est détectée, alors le bit est de valeur 1.

Exploitation spatiale du L1D Le cinquième test inclus dans *Timesecbench* exploite le partage spatial du cache L1D entre deux threads. Là encore, le principe est le même que pour l'exploitation temporelle du L1D. La différence majeure est que le cheval de Troie et l'espion sont ici exécutés simultanément sur deux threads différents.

Les tests d'exploitation spatiale (du L1D et des unités d'exécution) n'ont d'intérêt que sur un processeur supportant l'exécution de threads en simultané (SMT). Ils n'ont donc été mis en place que sur le cœur Salers.

Exploitation spatiale des unités d'exécution Le dernier des tests intégrés dans *Timesecbench* utilise le partage spatial d'une unité d'exécution entre deux threads. L'unité choisie est celle responsable de l'exécution des opérations de multiplications MULDIV. L'objectif est ici de créer une situation de contention comme sur la Figure 9.3. La valeur transmise est donc sur 1 bit :

0 le cheval de Troie bloque la ressource,
1 le cheval de Troie n'utilise pas l'unité.

De son côté, l'espion n'a qu'à réaliser successivement des multiplications. Des variations dans le temps d'exécution indiquent de la contention et donc la réception d'un 1.

9.2. Mesure d'efficacité des contremesures

Une fois les scénarios d'attaques définis, nous devons les exécuter pour détecter la présence de fuites temporelles. L'analyse s'effectue en deux étapes : l'isolation temporelle puis spatiale.

Comparaison des implémentations vulnérables et sécurisées

Notre objectif principal est de vérifier les propriétés d'isolation offertes par les domes. Chaque test est donc exécuté deux fois : une fois dans un cas "sans isolation", et un autre "avec isolation". Dans le premier cas, le cheval de Troie et l'espion sont placés dans un même domaine de sécurité/dome. Les frontières d'isolation, aussi bien spatiale que temporelle, ne s'appliquent donc pas. Dans le second cas, le cheval de Troie et l'espion sont placés dans des domes différents.

Pour chaque test, l'information mutuelle (IM) est calculée. Cette métrique est reprise de précédents travaux [91] et permet d'évaluer la quantité d'informations pouvant être envoyée par le biais du canal de communication. Prenons le cas d'un canal transmettant 3 bits d'informations. Une IM de 46% indique que l'espion est capable de retrouver en moyenne $3 * 0,46 = 1,37$ bits pour chaque valeur encodée. Un canal parfait a une IM de 100% alors qu'un canal inexistant a 0%.

Les résultats présentés par la suite ont été obtenus après exécution sans système d'exploitation. Ce choix permet de supprimer une grande partie du bruit pouvant altérer la communication cheval de Troie/espion. Les capacités de l'attaquant sont donc renforcées.

Isolation temporelle

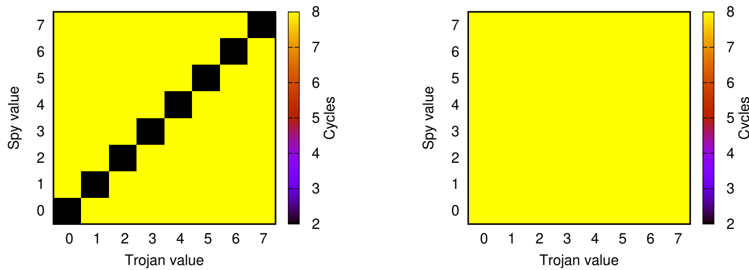
Dans le cas de l'isolation temporelle, on considère que le cheval de Troie et l'espion sont exécutés l'un après l'autre sur un même thread. C'est le modèle utilisé pour les quatre premiers tests. Pour le système dit "protégé" ou "avec isolation", un `dome.switch` est exécuté entre temps pour indiquer le changement de dome. Cette situation est illustrée sur le Code 9.1.

Code 9.1: Isolation temporelle entre un cheval de Troie et un espion. Le cheval de Troie s'exécute. Quand il a fini, une indication est envoyée à la microarchitecture pour assurer le changement de dome. Finalement, l'espion est exécuté à son tour.

```
trojan:
  ...
  ...
frontiere:
  ...
  dome.switch rd
spy:
  ...
  ...

```

Caches L1 Nous évaluons tout d'abord les fuites au niveau des caches. Aussi bien pour le L1D que le L1I, on considère le cas où chacun contient 8 ensembles. Une valeur de $\log_2(8) = 3$ bits par symbole peut donc être transmise.

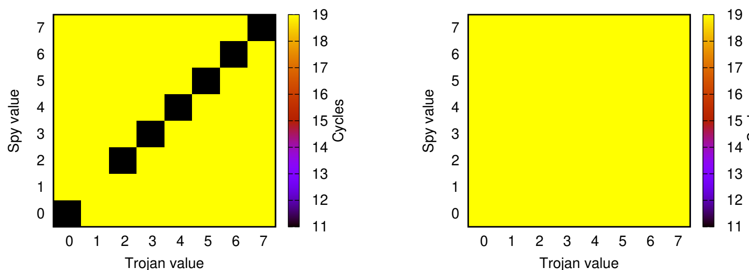


(a) Cas non protégé. IM = 100%(3,0 bits) **(b)** Cas protégé. IM = 0%(0,0 bits)

Figure 9.4. – Évaluation de l'isolation du L1D avec partage temporel. L'axe des abscisses donne l'ensemble utilisé par le cheval de Troie, celui des ordonnées l'ensemble utilisé par l'espion. La couleur indique le nombre de cycles pour un accès du cheval de Troie. Le processeur ciblé est Aubrac.

Les résultats obtenus pour le cache L1D sont présentés sur la Figure 9.4. Dans le cas non protégé, l'IM est maximale : l'espion est à chaque fois capable d'identifier la valeur envoyée par le cheval de Troie. Cela se caractérise par des variations temporelles observables. Une ligne présente (car ramenée en amont par le cheval de Troie) est accessible en 2 cycles, ce qui diffère des 8 cycles pour une non présente.

Dans le cas protégé en revanche, aucune variation temporelle n'est observable. Toutes les lignes ont été effacées à la fin de l'exécution du cheval de Troie lors du `dome.switch`. Cette isolation s'exprime par la disparition des variations sur l'axe horizontal : on ne distingue plus de dépendance entre la donnée envoyée par le cheval de Troie et les temps perçus par l'espion. Le canal caché n'est donc plus exploitable.



(a) Cas non protégé. IM = 46%(1,37 bits) **(b)** Cas protégé. IM = 0%(0,0 bits)

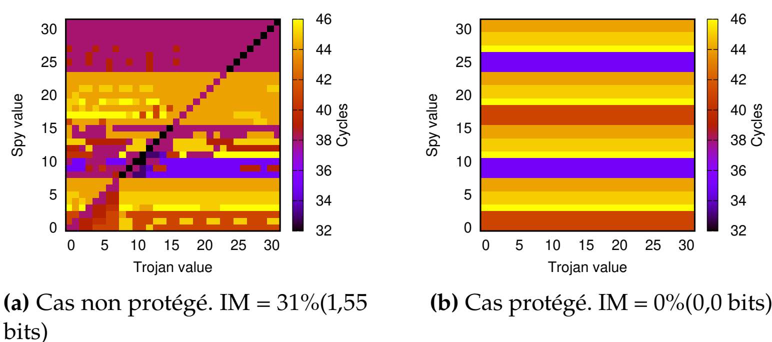
Figure 9.5. – Évaluation de l'isolation du L1I avec partage temporel. L'axe des abscisses donne l'ensemble utilisé par le cheval de Troie, celui des ordonnées l'ensemble utilisé par l'espion. La couleur indique le nombre de cycles nécessaire pour un accès du cheval de Troie. Le processeur ciblé est Aubrac.

On retrouve des résultats similaires pour le L1I sur la Figure 9.5. Dans le cas non protégé, des fuites existent même si le canal qui en découle est moins efficace. Par exemple, aucune transmission n'est détectée avec l'ensemble 1 sur la Figure 9.5a. Cela s'explique notamment par la pollution

du L1I par le code en cours d'exécution. Durant son exécution, le cheval de Troie ramène des lignes de code dans le cache en plus de celles visées. Là encore dans le cas protégé, on constate que la stricte application de l'opération de flush élimine les fuites.

Finalement au niveau des caches, les vider au moment d'un changement de domaine de sécurité *via* le `dome.switch` est efficace. On observe également que l'espion se retrouve à chaque fois dans une situation de cache "froid" : les premiers accès mettent toujours le temps maximal dû à un *miss*.

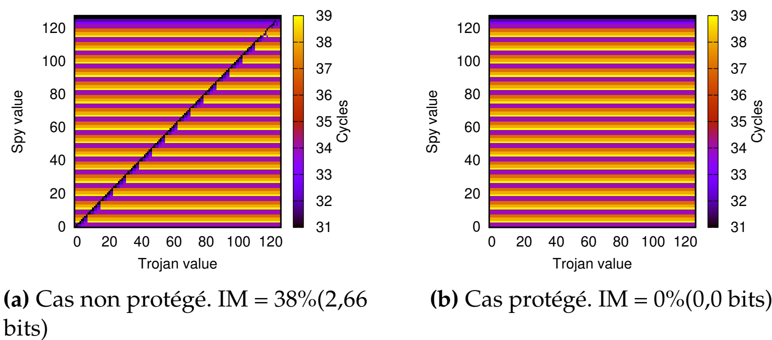
BHT et BTB Les deux autres tests sur le partage temporel concernent le BTB et le BHT. Les résultats obtenus sont présentés sur la Figure 9.6 et la Figure 9.7.



(a) Cas non protégé. IM = 31%(1,55 bits)

(b) Cas protégé. IM = 0%(0,0 bits)

Figure 9.6. – **Évaluation de l'isolation du BTB avec partage temporel.** L'axe des abscisses donne le saut indirect exécuté par le cheval de Troie, celui des ordonnées le saut exécuté par l'espion. La couleur indique le nombre de cycles pour une exécution du cheval de Troie. Le processeur ciblé est Aubrac.



(a) Cas non protégé. IM = 38%(2,66 bits)

(b) Cas protégé. IM = 0%(0,0 bits)

Figure 9.7. – **Évaluation de l'isolation du BHT avec partage temporel.** L'axe des abscisses donne le branchement conditionnel exécuté par le cheval de Troie, celui des ordonnées le branchement exécuté par l'espion. La couleur indique le nombre de cycles pour une exécution du cheval de Troie. Le processeur ciblé est Aubrac.

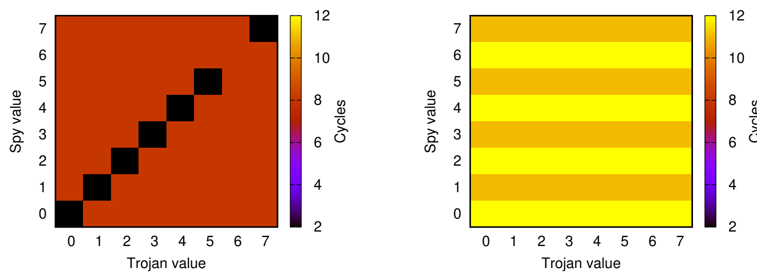
Les valeurs encodées sont respectivement de 5 et 7 bits par le biais du BTB et du BHT. Pour les deux tests, on retrouve finalement des résultats similaires. Sur les Figure 9.6a et Figure 9.7a, on distingue le même type de diagonale représentative d'une fuite par un canal caché. Cette forme est cependant moins nette dans ces cas-là. Là encore, la raison est la pollution des structures de prédiction générée par l'exécution du test lui-même. Dans les deux cas protégés, on constate néanmoins que plus aucune dépendance n'existe entre l'exécution du cheval de Troie et de l'espion.

Finalement, on peut conclure que la stratégie de flush pour les ressources partagées temporellement est bien efficace sur les implémentations testées. Cela implique que l'ensemble des états persistants utilisés n'est pas conservé entre deux domaines isolés.

Isolation spatiale

Les deux derniers tests sont ensuite appliqués sur le cœur Salers, nous permettant d'évaluer les principes dans le cas du partage spatial de ressources.

Cache L1D Le premier composant partagé spatialement est le cache L1D. Les résultats obtenus sont présentés sur la Figure 9.8.



(a) Cas non protégé. IM = 46%(1,37 bits)

(b) Cas protégé. IM = 0%(0,0 bits)

Figure 9.8. – **Évaluation de l'isolation du L1D avec partage spatial.** L'axe des abscisses donne l'ensemble utilisé par le cheval de Troie, celui des ordonnées l'ensemble utilisé par l'espion. La couleur indique le nombre de cycles. Le processeur ciblé est Salers.

Sans surprise, les résultats sont similaires à ceux pour le partage temporel du L1D. Dans le cas non protégé, on retrouve ces variations horizontales dépendantes des accès du cheval de Troie. On observe une dégradation du canal à cause des accès simultanés du cache par le cheval de Troie et l'espion.

Dans le cas protégé en revanche, toute variation horizontale disparaît. Le partitionnement est donc efficace pour éliminer les fuites entre le cheval de Troie et l'espion. Il est cependant intéressant de noter l'accroissement de la latence des *misses* et l'apparition de variations verticales. Ces phénomènes s'expliquent par la structure du code de l'espion et surtout l'application de la stratégie de partitionnement temporel du contrôleur de cache.

Pour rappel, nous avons ici deux domes en cours d'exécution. Le contrôleur du cache est donc utilisé cycliquement par un dome ou l'autre. Nous avons vu dans la Table 8.1 que la latence en cas de *miss* devenait au mieux de 11 cycles. Ici, elle atteint même 12 cycles dans le cas où le dome correspondant n'est pas celui immédiatement traité par le contrôleur. 1 cycle supplémentaire d'attente est nécessaire pour la synchronisation requête/contrôleur. Ainsi, l'espion alterne les cas où sa requête est immédiatement traitée par le contrôleur (11 cycles) ou mise en attente (12 cycles). Cette variation étant indépendante du cheval de Troie, elle n'est pas symptomatique d'une fuite.

Notes : Reprenons la formule de la Table 8.1 : latence = $3 + (D * (A - 1)) + 2$. Pour D domes, le pire cas pour synchroniser une requête et le contrôleur est D - 1. Il faut alors attendre de parcourir tous les autres domes un à un avant de finalement traiter la requête. Le pire cas de latence avec les domes est donc donné par la formule : latence = $3+(D-1)+(D*(A-1))+2$.

Unité d'exécution Le dernier test évalue la contention d'unité d'exécution. Les résultats obtenus pour Salers et l'unité MULDIV sont présentés sur la Figure 9.9. Dans notre cas d'étude, il n'y a qu'une seule unité de ce type pour les deux threads.

On retrouve finalement une durée d'exécution différente pour chacun des quatre cas. La valeur minimale est logiquement obtenue lorsqu'aucune contention n'apparait (espion et cheval de Troie à 1). À l'inverse, la valeur maximale est obtenue quand les deux threads se disputent mutuellement l'unité MULDIV. Il est donc clairement possible de distinguer un bit transmis à 1 d'un bit transmis à 0. Les deux autres cas sont des

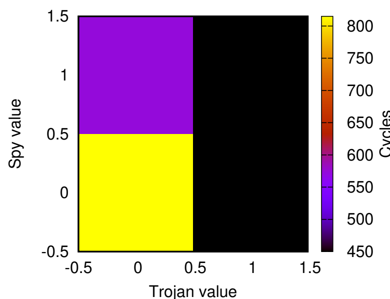


Figure 9.9. – Évaluation de l'isolation de la contention dynamique d'une unité d'exécution. Cas non protégé. IM = 10%(0,1 bits). L'axe des abscisses indique si le cheval de Troie utilise l'unité d'exécution, celui des ordonnées si l'espion utilise l'unité d'exécution. Dans les deux cas, 0 signifie que l'unité visée est utilisée, 1 signifie qu'une autre unité est demandée. La couleur indique le nombre de cycles. Le processeur ciblé est Salers.

valeurs de temps intermédiaires qui varient uniquement selon la latence des opérations exécutées.

Le cas protégé n'est pas représenté dans ce cas. Cela s'explique par le fait que le scénario de contention n'est tout simplement pas réalisable. Grâce à l'allocation statique des ressources, un seul dome peut posséder la ressource à un instant précis. Aucun conflit ne peut exister : il n'y a plus de fuite due à la contention de ressource.

Pour conclure sur cette partie de l'évaluation, on constate que l'application des principes de conception permet bien la mise en place d'une politique d'isolation complète. Aucune dépendance temporelle entre les actions du cheval de Troie et le temps d'exécution de l'espion n'est observable, quel que soit le type de partage. Pour confirmer la pertinence de nos tests, des travaux sont également en cours afin de les exécuter sur un Raspberry Pi 3. Les résultats actuels, bien qu'intermédiaires, montrent déjà des vulnérabilités dues à certaines ressources partagées (notamment le cache L1I).

Malgré ces résultats, une question demeure : pouvons-nous garantir pleinement qu'aucune fuite temporelle n'est présente ? Malheureusement, à l'heure actuelle, la réponse est non.

Tout d'abord, nous sommes dans une situation où le même groupe de personnes est en charge du développement des implémentations, mais aussi de la suite de tests. Dans un environnement de test idéal, ces responsabilités devraient être réparties entre des équipes distinctes afin d'éviter des biais dans les jugements et prises de décision. En ce sens, *Timesecbench* représente par contre un outil intéressant : il fournit un environnement de tests indépendant pour d'autres concepteurs. Il vise donc à répondre à un besoin qu'aucun outil de tests de référence ne satisfait à l'heure actuelle.

Enfin, *Timesecbench* nous permet seulement de prouver que les ressources partagées testées ne génèrent pas de fuite. Dans le cas où il existerait une autre ressource partagée non testée, rien ne garantit qu'elle n'assure correctement l'isolation. Pour cela, un nouveau test pour la cibler serait nécessaire. À terme, une autre stratégie pour garantir l'absence de fuite serait l'utilisation de techniques de formalisation. À partir de contraintes clairement définies, il faudrait analyser les processeurs dans leur ensemble. Cependant, à notre connaissance, aucun outil n'est aujourd'hui capable d'effectuer de telles analyses. *Timesecbench* reste donc un outil intéressant pour guider un concepteur et augmenter la confiance en une implémentation.

9.3. Analyse des performances et du coût

Les principes de conception du chapitre 7 modifient la microarchitecture. En plus de la sécurité, les processeurs doivent répondre à des contraintes de performances et de coût. Dans cette section, nous évaluons l'impact de l'application des principes de conception en prenant les processeurs Aubrac et Salers comme exemples.

Notes : Plusieurs versions des cœurs Salers et Aubrac ont été utilisées. Différentes tailles de caches, la présence du NLP (ou non) et l'implémentation des domes sont les principaux paramètres variés. Comme dit précédemment, cela est notamment facilité par le paramétrage permis par le langage Chisel.

Les résultats présentés ont été obtenus après synthèse et implémentation sur le logiciel Vivado 2019.2. Ce logiciel permet la programmation sur circuit logique programmable (field-programmable gate array ou FPGA). Dans notre cas, la carte de développement visée était la Xilinx ZCU104, comportant le FPGA XCZU7EV-2FFVC1156.

Performances

Deux métriques sont généralement utilisées pour évaluer les performances d'un processeur. La première est la fréquence de fonctionnement. Nous avons vu dans le chapitre 2 que les processeurs sont des composants dits synchrones. Ils utilisent un signal d'horloge qui vient activer régulièrement certains éléments du système. Plus cette cadence est rapide, plus les opérations se font donc souvent. Ainsi, une fréquence d'horloge plus grande implique un circuit plus rapide. La seconde métrique est le nombre de cycles nécessaires à la réalisation d'un programme donné. Indépendamment de la fréquence, elle permet de prendre en compte tous les mécanismes internes d'optimisations comme le pipeline, les caches *etc.* Ces deux métriques permettent de donner le temps nécessaire à l'exécution d'un programme. Pour connaître l'impact de l'application des domes, nous distinguons donc le surcoût sur la fréquence de fonctionnement et celui sur le nombre de cycles d'horloge nécessaires.

Nombre de cycles Pour évaluer l'impact sur le nombre de cycles, nous utilisons la suite de tests Embench [150]. Dédiée aux systèmes embarqués, elle vise à être exécutée sans système d'exploitation sur les processeurs cibles. Dans notre cas, nous l'avons compilée et exécutée sur des versions simulées de nos processeurs.

Cache	1 kB	4 kB	1 kB	4 kB
	Aubrac		Salers (1 thread)	
	réf.	0,86	0,95	0,90
Dome	1,00	0,86	1,16	0,92
NLP	0,92	0,78	0,95	0,81
NLP Dome	0,92	0,78	1,07	0,82

Table 9.1. – Moyennes géométriques normalisées obtenues pour Embench La configuration Aubrac avec 1 kB de taille des caches (sans dome ni NLP) est utilisée comme référence.

Les résultats obtenus sont présentés sur la Table 9.1. Dans le cas d'Aubrac, on constate que le support des domes n'a finalement aucun impact. L'ensemble des tests étant à chaque fois exécuté dans le même domaine de sécurité, aucun changement n'est perceptible. Il n'y a aucune opération de gestion des domes qui altère l'exécution des programmes.

En revanche, le cas de Salers est différent. Ici, les deux threads sont placés dans des domaines d'exécution différents. Les ressources disponibles sont donc partitionnées. Avec des caches d'1 kB, on constate une dégradation de $\approx$ 20% (1,16 au lieu de 0,95) dans le pire cas. Cela est particulièrement dû au partitionnement des caches. Initialement, les caches L1 sont déjà de petites tailles. Le partitionnement réduit encore plus leur capacité et entraîne un nombre important de remplacements/*misses*.

Chaque thread n'a finalement accès qu'à 0.5kB. On remarque que ce phénomène est ici en partie atténué avec le support du NLP. En plus des mauvaises prédictions, il permet également de supprimer certains des remplacements qu'elles entraînent. L'impact des caches se confirme si on les agrandit (4kB au lieu de 1kB). Le surcoût des domes n'est alors que de ≈ 2% dans le pire des cas. Enfin, le NLP modifié garde tout son intérêt (0,82 avec NLP + dome contre 0,90 de base)

Table 9.2. – Moyennes géométriques normalisées obtenues pour Embench (version SMT) La configuration Aubrac avec 1 kB de taille des caches (sans dome ni NLP) de la Table 9.1 est utilisée comme référence. Pour cette évaluation, certains tests d'Embench ont été modifiés pour supporter le SMT.

Cache	1 kB	4 kB	1 kB	4 kB
Salers	1 thread actif		2 threads actifs	
	1,05	0,91	0,56	0,51
Dome	1,07	0,88	0,76	0,62
NLP	1,03	0,88	0,55	0,51
NLP Dome	1,05	0,86	0,75	0,61

De même, il est intéressant de savoir si le support des domes préserve l'intérêt du SMT. Pour cela, trois des tests d'Embench ont été modifiés afin d'être optimisés pour l'exécution en parallèle. Dans les cas avec deux threads, chacun exécute une partie différente d'un même programme. Le résultat final est donc connu à la fin de l'exécution des deux parties. Les temps d'exécution sont de la même manière présentés sur la Table 9.2.

On constate qu'avec 2 threads, les domes induisent à nouveau une baisse significative des performances. Comme précédemment, cela s'explique par le partitionnement des caches. Bien qu'exécutant des parties différentes d'un même test, chacun est placé dans un dome différent. Cette différence s'atténue en augmentant la taille des mémoires caches.

Finalement, l'intérêt du SMT est préservé. Si le nombre de cycles n'est plus divisé par 2, le gain reste néanmoins important (0,75 au lieu de 1,05 soit ≈ 30%).

Table 9.3. – Nombre de cycles par test de Timesecbench.

	Non-protégé (cycles)	Protégé (cycles)	Surcoût (%)	Surcoût par `dome.switch` (cycles)
L1D (temporel)	27256	35880	+32%	67,4
L1I	57416	64264	+12%	53,5
BHT	1756202	1796774	+2%	19,8
BTB	445544	463443	+4%	35,0
L1D (spatial)	188026	134395	−29%	N/A
Contention	59250	N/A	N/A	N/A

Durant l'exécution de la suite Embench, aucun `dome.switch` n'est exécuté. Son impact n'est donc pas mesuré. Pour cela, nous avons repris les tests de sécurité *Timesecbench*. L'idée est de comparer le nombre de cycles nécessaires pour le cas protégé et non protégé. Les résultats sont présentés sur la Table 9.3.

Dans le cas des tests sur le partage temporel (les 4 premiers), on constate une dégradation des performances. Il y a deux raisons à cela. La première est que les `dome.switch` ajoutés prennent un certain temps à s'exécuter. Comme dit précédemment, cela dépend notamment de la stratégie de flush. Dans notre cas, la plupart de ces opérations est instantanée ou quasi-instantanée (une politique d'écriture traversante est utilisée). La seconde raison est l'état "froid" dans lequel se trouve

le nouveau dome exécuté. Celui-ci fait face à un nombre important de *misses* et de mauvaises prédictions en début d'exécution. Finalement, un `dome.switch` peut avoir un impact allant jusqu'à 64 cycles. Ce résultat est obtenu pour le test L1D temporel, car il sollicite de nombreuses fois plusieurs ressources : le L1D en plus du L1I et du prédicteur de branchement.

Dans le cas du L1D spatial, on observe une nette amélioration du temps d'exécution (≈ 30%). Étonnamment, cela s'explique par le partitionnement des caches. Ce test étant exécuté par deux threads, des opérations sont utilisées pour les synchroniser. Celles-ci affectent les caches et ont tendance à ralentir les threads mutuellement. Dans le cas protégé, grâce à l'isolation, ce ralentissement mutuel disparaît.

Enfin, aucun résultat n'est disponible pour le cas de contention dynamique, le programme ne pouvant être exécuté sur la cible protégée.

Fréquence d'horloge Pour évaluer l'impact du support des domes sur la fréquence d'horloge, les implémentations de plusieurs configurations d'Aubrac et Salers ont été réalisées.

Cache	1 kB		4 kB	
Aubrac				
	4,70 ns	réf.	4,93 ns	×1,05
Dome	4,85 ns	×1,03	4,91 ns	×1,04
NLP	4,69 ns	×1,00	4,93 ns	×1,05
NLP Dome	4,76 ns	×1,01	5,05 ns	×1,07

Table 9.4. – Période et fréquence d'horloge pour Aubrac. La configuration la plus simple sert de référence. Ces résultats sont obtenus en ciblant un FPGA Xilinx XCZU7EV-2FFVC1156.

Les résultats pour Aubrac sont présentés sur la Table 9.4. Globalement, l'impact observé peut-être considéré comme du bruit lié au logiciel et au FPGA lui-même. On constate aussi bien des augmentations que des réductions de la fréquence (de ≈ −1% à ≈ +3%). Ces résultats sont finalement cohérents dans le sens où le chemin critique[1] n'a pas été modifié ici.

1: Pour rappel, le chemin critique correspond au chemin le plus long utilisé par un signal électrique entre deux coups d'horloge. Cette notion a été introduite dans le chapitre 2.

Cache	1 kB		4 kB	
Salers				
	6,90 ns	réf.	7,48 ns	×1,08
Dome	6,82 ns	×0,99	7,50 ns	×1,09
NLP	7,57 ns	×1,10	7,69 ns	×1,11
NLP Dome	7,44 ns	×1,08	7,50 ns	×1,09

Table 9.5. – Période et fréquence d'horloge pour Salers. La configuration la plus simple sert de référence. Ces résultats sont obtenus en ciblant un FPGA Xilinx XCZU7EV-2FFVC1156.

Les résultats pour Salers sont eux présentés sur la Table 9.5. On observe à nouveau des variations quasiment nulles, car le chemin critique n'a pas non plus été modifié.

Finalement, l'application des principes de conception du chapitre 7 ne dégrade pas la fréquence de fonctionnement. Ces résultats sont semblables à ceux d'autres travaux [43, 99]. Bien évidemment, l'impact observé reste très dépendant du processeur visé et des technologies utilisées. Sur certains processeurs, on peut supposer qu'un chemin critique plus restreint puisse être impacté, impliquant des changements plus importants pour préserver les performances.

[43]: Ferraiuolo et al. (2017), « Full-Processor Timing Channel Protection with Applications to Secure Hardware Compartments »
[99]: Bourgeat et al. (2019), « MI6 »

Coût en ressources matérielles

Enfin, nous évaluons à présent le surcoût en ressources matérielles impliqué par les domes. Pour cela, les mêmes configurations que celles pour étudier la fréquence d'horloge ont été réutilisées. Les implémentations étant faites pour une cible FPGA, on distinguera deux types de ressources :

1. les lookup tables (LUT) qui sont les éléments configurables utilisés pour représenter la logique combinatoire du circuit,
2. les registres (abrégés FF pour flip-flops) qui sont donc les éléments mémorisant du circuit.

Table 9.6. – Ressources du FPGA utilisées pour Aubrac. La configuration la plus simple sert de référence.

Cache	1 kB		4 kB	
Aubrac - without SMT				
	9370 LUT	réf.	24590 LUT	×2,62
	4408 FF	réf.	8378 FF	×1,90
Dome	9651 LUT	×1,03	24530 LUT	×2,62
	4715 FF	×1,07	8703 FF	×1,97
NLP	12168 LUT	×1,30	27410 LUT	×2,93
	5267 FF	×1,19	9237 FF	×2,10
NLP Dome	12146 LUT	×1,30	26984 LUT	×2,88
	5575 FF	×1,26	9562 FF	×2,17

Les résultats pour le cœur Aubrac sont présentés sur la Table 9.6. Pour rappel, celui-ci ne gère que du partage temporel. Le surcoût en LUTs est ici seulement de quelques pour cent (de 0 à +3%). Ce chiffre est finalement similaire à ceux présentés pour l'implémentation d'une instruction de flush (1% pour `fence.t` [103]). En revanche, on note une augmentation plus importante du nombre de FF (jusqu'à 7%). Ceci s'explique en grande partie par l'ajout de nouveaux CSR pour les domes. Dans ce cas précis cependant, une simplification possible pourrait être de supprimer ces registres non essentiels. L'opération de flush suffit en cas de partage temporel uniquement.

[103]: Wistoff et al. (2020), « Prevention of Microarchitectural Covert Channels on an Open-Source 64-Bit RISC-V Core »

Table 9.7. – Ressources du FPGA utilisées pour Salers. La configuration la plus simple sert de référence.

Cache	1 kB		4 kB	
Salers - with SMT				
	21000 LUT	réf.	46943 LUT	×2,24
	8270 FF	réf.	13676 FF	×1,65
Dome	22919 LUT	×1,09	44852 LUT	×2,14
	10889 FF	×1,32	16498 FF	×1,99
NLP	26731 LUT	×1,27	54278 LUT	×2,58
	10011 FF	×1,21	15566 FF	×1,88
NLP Dome	28167 LUT	×1,34	55977 LUT	×2,67
	12599 FF	×1,52	18188 FF	×2,20

Les résultats pour le cœur Salers sont présentés sur la Table 9.7. Ici, l'impact du support des domes est bien plus important. De la logique combinatoire est par exemple nécessaire au sein de l'unité de gestion des domes ou à différents endroits pour effectuer des comparaisons de tags. Le surcoût atteint jusqu'à +9% du système de référence.

Il est encore plus important sur les FF avec une augmentation jusqu'à +32%. Là encore, cela est principalement dû à l'ajout de CSR (pour les deux threads), mais aussi à la duplication des registres des contrôleurs de caches et au pipeline plus complexe de l'unité de gestion des domes.

Il est cependant nécessaire de nuancer ce surcoût. Tout d'abord, il n'est pas possible de le comparer directement avec celui obtenu sur

d'autres travaux. Généralement, les ressources partagées considérées ne sont pas les mêmes : c'est particulièrement le cas du SMT qui n'est souvent pas modifié, mais seulement désactivé ou ignoré (en considérant un domaine de sécurité par cœur physique). Des comparaisons avec les travaux de Townley et Ponomarev [109] sont également impossibles : ils ne considèrent que le cas du SMT et non pas celui de l'ensemble des ressources partagées. De plus, il est nécessaire de rappeler que le processeur Salers est un processeur à la structure assez simpliste. Or, le SMT est généralement implémenté avec des mécanismes bien plus coûteux en ressources comme l'exécution dans le désordre. Le nombre de CSR présents de base augmente également lorsque l'on ajoute des fonctionnalités (ajout des privilèges, support de la mémoire virtuelle *etc.*). Dans ce genre de processeurs, on peut facilement imaginer que le surcoût matériel avec l'ajout des domes serait dans des proportions encore plus raisonnables.

[109]: Townley et al. (2019), « SMT-COP : Defeating Side-Channel Attacks on Execution Units in SMT Processors »

9.4. Bilan et conclusion

Dans ce chapitre, nous avons évalué l'impact de l'application des principes de conception du chapitre 7 sur la microarchitecture. Pour l'évaluation de sécurité, nous avons introduit une nouvelle suite de tests appelée *Timesecbench*. Elle permet d'éprouver des ressources partagées connues et de détecter les fuites temporelles qu'elles engendrent. Nous avons ainsi pu démontrer que la politique d'isolation implémentée est bien efficace. Aucune fuite n'a été détectée entre des domaines de sécurité distincts.

Nous avons ensuite évalué le surcoût de ces modifications du point de vue des performances et de l'utilisation de ressources matérielles. Dans le premier cas, l'intérêt des ressources partagées est globalement préservé pour la diminution du nombre de cycles d'exécution. Le cas du partitionnement des caches doit cependant être surveillé selon les implémentations. Partitionner un cache d'une taille trop réduite peut grandement réduire les performances. Concernant la fréquence, le support des domes n'a pas d'impact sur le chemin critique. Des contraintes de performances et d'isolation ne sont donc pas forcément contradictoires. Une étude générale intégrant une couche logicielle plus complète (*e.g.* avec un système d'exploitation adapté) reste tout de même souhaitable pour définitivement confirmer ces résultats.

En revanche, un surcoût important est à souligner dans le cas de l'isolation dans des processeurs avec SMT. De la logique et des registres supplémentaires sont nécessaires pour séparer les domaines exécutés simultanément. Cet impact doit tout de même être nuancé : le cœur Salers modifié est un modèle processeur avec SMT très simple. En cas de contraintes trop fortes sur les ressources utilisées, une alternative à explorer reste les microarchitectures multicœurs. De par leur structure, elles permettent également un parallélisme entre les programmes sans pour autant engendrer un partage des ressources profond. Le scénario classique serait alors de ne tolérer qu'un domaine de sécurité par cœur. La mise en place d'une politique d'isolation pourrait donc y être implémentée à un coût réduit.

Si la suite *Timesecbench* nous a permis d'évaluer un certain nombre de ressources partagées, des travaux restent cependant nécessaires pour étoffer cet outil. Il existe de nombreuses autres ressources partagées au

sein des microarchitectures modernes qui peuvent être exploitées. Sur le même modèle, il serait intéressant de compléter l'ensemble des tests afin de cibler d'autres sources de fuite possibles. De même, des travaux pour viser d'autres jeux d'instructions (*e.g.* ARM sur les Raspberry Pi 3) doivent être terminés.

Conclusion 10.

10.1. Résumé des travaux

10.1 Résumé des travaux . . . 147
10.2 Bilan synthétique 149
10.3 Travaux futurs et perspectives 149

L'objectif de cette thèse est la définition de nouvelles bases pour la conception de processeurs avec des contraintes de sécurité. Nous nous intéressons plus particulièrement à la problématique d'isolation posée par certaines attaques logicielles. À partir de leur exécution sur un processeur, certains programmes sont capables de retrouver des informations sur des exécutions passées ou toujours en cours, en exploitant les différents mécanismes de la microarchitecture.

En impliquant aussi bien le logiciel que le matériel, ces attaques remettent en cause l'organisation complète des systèmes informatiques. Ainsi, l'approche étudiée consiste à adapter le jeu d'instructions afin de permettre une solution globale. Pour assurer des garanties de sécurité, chaque niveau d'abstraction du système a un rôle bien défini. Le logiciel est responsable de définir les contraintes de sécurité des différents programmes. Le matériel applique ces contraintes aux différentes exécutions à l'aide de mécanismes dédiés. Enfin, le jeu d'instructions assure l'échange d'informations entre logiciel et matériel. Les travaux présentés dans ce manuscrit reprennent les étapes menant à la conception d'un processeur sécurisé.

Rôle et caractéristiques de l'ISA Premièrement, le rôle et les caractéristiques de notre nouvelle ISA ont été définis. Le matériel doit recevoir deux types d'informations de la part du logiciel : quels mécanismes de sécurité doivent être appliqués et quand ? L'ISA doit donc permettre l'envoi d'informations sur les domaines de sécurité actifs et les politiques de sécurité utilisées. Pour être capable de s'adapter à tous les logiciels, l'ISA doit permettre une grande flexibilité. Chaque domaine de sécurité ainsi que l'organisation globale doivent être dynamiques.

Modifier l'ISA implique de tenir compte de l'impact sur le logiciel et le matériel. Notamment, le jeu d'instructions a un rôle d'abstraction. Il permet de rendre indépendant le matériel du logiciel en définissant une spécification commune. Si l'abstraction est utile pour séparer la conception de ces deux parties, elle empêche également le logiciel d'avoir une vision claire des mécanismes présents au niveau du matériel. Dans le cas de certaines contraintes de sécurité, les mécanismes exploités par les attaques sont, du point de vue du logiciel, invisibles et incontrôlables. Mais à l'inverse, briser cette abstraction implique le développement de code dépendant du processeur ciblé. Un compromis doit donc être trouvé pour que notre ISA conserve son rôle d'abstraction, tout en permettant au logiciel de communiquer ses besoins au matériel.

Proposition d'ISA À partir de ces besoins, une proposition d'extension de l'ISA est présentée. Elle détaille l'ajout d'une nouvelle notion de domaines de sécurité dynamiques appelés domes. Chaque dome est défini par un identifiant, un point d'entrée et un ensemble de capacités définissant ses droits et politiques de sécurité. Ils peuvent être vus comme des contextes d'exécution explicites et configurables avec des propriétés

de sécurité. Ils représentent une alternative aux privilèges classiques, n'offrant qu'une flexibilité limitée et impliquant l'ajout de mécanismes supplémentaires pour assurer des garanties de sécurité.

Cette proposition d'extension du jeu d'instructions RISC-V détaille le fonctionnement des domes et leur représentation au niveau de l'ISA. Cela passe par l'ajout de registres dédiés pour stocker les configurations de chaque dome. De nouvelles instructions sont également détaillées pour la gestion de ces domaines de sécurité.

Principes de conception À partir des informations reçues du logiciel, le matériel doit s'adapter pour respecter les contraintes de sécurité. Des principes de conception génériques ont été définis pour l'implémentation de microarchitectures capable d'isoler les différentes exécutions. Ils se veulent indépendants des mécanismes ciblés et applicables à n'importe quelle microarchitecture.

Une première partie de ces principes s'intéresse à la gestion des ressources partagées. Ils s'articulent autour de trois méthodes pour contrôler le partage temporel et spatial de chaque ressource. L'allocation statique des ressources permet d'attribuer à chaque domaine d'exécution les ressources dont il a besoin. Le partitionnement des ressources permet de répartir les ressources entre les différents utilisateurs en cours d'exécution. L'effacement des traces assure qu'aucune information propre à un domaine d'exécution ne reste accessible après son exécution. Des mécanismes pour mettre en place ces méthodes sont également présentés.

D'autres principes s'intéressent au partage des plages mémoires. Ils visent notamment à s'assurer que les frontières d'isolation prévues par l'ISA sont respectées en tout point.

Implémentation des principes Les différents principes ainsi qu'une version allégée des domes ont été implémentés. L'objectif est uniquement d'évaluer leur impact sur la partie matérielle. Deux processeurs ont pour cela été modifiés. Un processeur simple appelé Aubrac n'implémente que des ressources partagées temporellement. Cela comprend le pipeline lui-même, deux mémoires caches et un prédicteur de branchement. Un seul domaine de sécurité étant exécuté à la fois, seuls les principes d'effacement des traces sont nécessaires. Le second processeur appelé Salers implémente également un mécanisme de SMT. Les différentes ressources sont donc partagées non seulement temporellement, mais aussi spatialement. En appliquant l'ensemble des mécanismes sur les ressources partagées, il permet l'exécution simultanée de deux domaines de sécurités isolés.

Timesecbench et évaluation Comme pour garantir la fonctionnalité, il est nécessaire de tester un processeur pour détecter de potentielles fuites d'informations. Pour cela, une nouvelle suite de tests appelée *Timesecbench* a été développée. Elle vise à tester les fuites d'informations dues à l'utilisation de ressources partagées communes comme les mémoires caches ou les tables de prédiction. Elle a été appliquée avec succès sur nos deux implémentations : les résultats ont montré que les fuites présentes dans les systèmes d'origine disparaissaient avec l'utilisation des domes.

L'impact sur les performances et le coût en ressources a également été évalué en exécutant la suite de tests *Embench* et en réalisant une

implémentation sur FPGA. Les résultats montrent que le partage temporel de ressources n'implique pas de surcoût important. En revanche, dans le cas du support du SMT avec Salers, plusieurs points sont à noter. Tout d'abord, le partitionnement des caches entre les threads a un impact sur les performances (−20%), notamment pour des caches de tailles réduites. Cependant, même dans cette situation, le gain offert par le SMT reste intéressant par rapport à un processeur sans SMT. Le support du SMT permet le partage en profondeur d'un processeur, et donc d'un grand nombre de ressources. Ainsi, l'application des domes pour permettre une isolation implique de nombreuses modifications. Chacune d'entre elles nécessite l'utilisation de nouvelles ressources, menant à un surcoût total non négligeable (jusqu'à +30%).

10.2. Bilan synthétique

Pour conclure, plusieurs leçons sont à tirer de ses travaux.

La première concerne la modification de l'ISA. Nous avons vu qu'il était possible de construire efficacement des garanties de sécurité à ce niveau. Aussi, les modifications peuvent intervenir tout en préservant l'abstraction offerte par le jeu d'instructions. Abstraction et sécurité ne sont donc pas forcément incompatibles : l'ajout d'instructions de gestion de la microarchitecture n'est pas la seule voie possible. La contrepartie est que chaque niveau du système à un rôle précis à assumer entièrement, sous peine de fragiliser la totalité du système.

Deuxièmement, chaque mécanisme matériel doit finalement être reconsidéré. Considérer des contraintes de sécurité dès leurs fondements influe directement sur la conception des processeurs. Elles s'ajoutent aux contraintes de performances et de coût qui guidaient la plupart des choix jusque-là. La politique d'écriture utilisée au sein de la hiérarchie mémoire (écriture traversante ou écriture différée) est un bon exemple d'évolution des choix. De base, l'écriture différée est préférée pour accélérer les écritures en mémoire. La mise en place de garanties d'isolation implique de régulièrement effacer le contenu des caches. Dans ce genre de situation, l'écriture traversante peut s'avérer plus performante pour réduire les temps d'effacement.

Finalement, concernant l'utilisation du SMT, il est généralement admis que ce mécanisme ne peut être modifié pour assurer une isolation entre les threads. Nous avons vu que cela reste possible, mais au prix de certaines modifications. Pour être pertinentes, ces dernières doivent avoir un coût inférieur à celui des ressources partagées elles-mêmes. Autrement, une duplication complète du cœur peut être plus intéressante (multicœur plutôt que SMT). C'est le cas des microarchitectures ayant des unités d'exécution très coûteuses à dupliquer (*e.g.* une FPU).

10.3. Travaux futurs et perspectives

Les travaux présentés dans ce manuscrit modifient en profondeur les processeurs. Ainsi, d'autres sont encore nécessaires pour mieux comprendre les implications sur différents systèmes.

Développement de Timesecbench *Timesecbench* permet la détection de fuites au niveau de la microarchitecture. Pour cela, cette suite de tests cible certaines des ressources partagées les plus communes.

À l'heure actuelle, ces tests ont seulement été exécutés avec succès sur nos propres processeurs (base RISC-V) ainsi que sur un Raspberry Pi 3 (ISA Armv8). D'autres essais d'exportation sur de nouvelles cibles doivent être effectués pour améliorer cet outil. De même, de nouveaux tests doivent être mis en place pour éprouver le fonctionnement de nouvelles ressources partagées. Des mécanismes comme les FPU, les TLB, les prefetchers ou le RSB représentent notamment des cibles communes et donc intéressantes à prendre en compte.

À terme, l'objectif est de disposer d'une collection de tests capable d'éprouver n'importe quel processeur, indépendamment du jeu d'instructions implémenté. Un concepteur matériel ou un développeur logiciel pourrait alors évaluer et comparer plusieurs implémentations et contre-mesures.

Il est important de garder à l'esprit que *Timesecbench* vise uniquement à simplifier l'évaluation de failles connues. Ainsi, même en ajoutant un nombre considérable de tests, ceux-ci ne sont pas supposés découvrir des fuites jamais observées. *Timesecbench* ne garantit pas qu'aucune fuite n'existe dans un processeur. Par exemple, des cas spécifiques à une seule implémentation ne seront pas forcément détectés. Sur le long terme, une évaluation exhaustive des processeurs passera probablement par des techniques de formalisation. À partir de modèles des processeurs, il faudra alors être capable d'éprouver tous les cas possibles pour garantir qu'aucune fuite n'existe. Le but serait alors d'être capable de générer automatiquement les différents tests à partir des modèles d'attaquants retenus. À l'heure actuelle cependant, aucun outil ne permet ce niveau d'évaluation et *Timesecbench* représente donc une alternative.

Adaptation du logiciel Les travaux détaillés dans ce manuscrit sont principalement axés sur deux niveaux d'abstraction : la modification de l'ISA et l'adaptation du matériel. Une suite logique de ces travaux est de s'intéresser plus en détail à la prise en compte de ces changements par le logiciel.

L'élément logiciel principal impacté par l'ajout des domes est le système d'exploitation. Originellement, celui-ci est conçu pour fonctionner avec des privilèges statiques. Une prise en compte des domes implique donc de revoir la répartition des droits sur le système. La notion de politique de sécurité étant maintenant présente au niveau de l'ISA, elle doit être considérée pour bénéficier des protections offertes par le matériel. La modification de l'ISA influe également sur les outils utilisés pour le logiciel. Les compilateurs doivent notamment être modifiés pour profiter au mieux des nouvelles fonctionnalités du jeu d'instructions. À l'heure actuelle, la version RISC-V du compilateur gcc a été modifiée uniquement pour reconnaître nos nouvelles instructions en langage assembleur. D'autres modifications sont nécessaires pour que les registres contenant les configurations des domes soient reconnus. De même, l'automatisation de la gestion des capacités (savoir à la compilation quel programme demande quelle(s) instruction(s)) doit être prise en compte.

Modification de processeurs complexes Les processeurs Aubrac et Salers ciblés dans ce manuscrit restent des systèmes relativement simples. Bien que le second ait le support du SMT, le pipeline implémenté reste assez commun et les exécutions sont toujours effectuées dans l'ordre. De nouveaux travaux sont donc nécessaires pour évaluer l'impact des

principes de conception sur des systèmes plus complexes.

L'exécution dans le désordre est un mécanisme particulièrement répandu dans les processeurs modernes. Ce fonctionnement complexifie énormément la microarchitecture et intensifie la spéculation. Dans le chapitre 7, nous avons défini des principes pour limiter la spéculation au niveau des frontières d'isolation. Pour être évalués, il est nécessaire de modifier des processeurs utilisant l'exécution dans le désordre. En plus d'être des frontières pour le partage temporel ou spatial, les domes agiraient alors comme des frontières de spéculation. Lors de l'exécution, le processeur ne pourrait spéculer que sur des états appartenant au dome en cours d'exécution.

Les architectures multicœurs sont également des cibles répandues qui doivent être considérées. De nouvelles ressources sont partagées entre les différents cœurs et doivent donc être modifiées. Notamment, dans le cas d'architectures avec de nombreux cœurs, la latence mémoire peut devenir une problématique essentielle. Par exemple, si un trop grand nombre de domes différents est simultanément exécuté, alors chaque contrôleur mémoire devra répartir son temps d'utilisation. L'application des principes sur les plages mémoires partagées, pour éviter les fuites au niveau de la cohérence mémoire, doit également être considérée.

Étude des compromis matériels La modification du processeur Salers pour permettre une isolation au niveau des threads a mis en avant le compromis entre performances, sécurité et coût des modifications. Ainsi, il serait intéressant d'étudier l'impact de chaque mécanisme. À l'image de précédentes études sur l'implémentation des ressources partagées pour les performances dans un processeur SMT [151], ces résultats pourraient guider les concepteurs matériels pour de futurs processeurs. Le cas du processeur Salers soulève en lui-même deux questions :

[151]: Raasch et al. (2003), « The Impact of Resource Partitioning on SMT Processors »

1. Quand faut-il privilégier le multicœur au SMT ?
2. Isoler les différents threads a-t-il toujours un surcoût aussi important ?

Répondre à la première question passe par l'implémentation de processeurs multicœurs afin de comparer les coûts d'isolation. Dans ces microarchitectures, le partage de ressources est fait moins en profondeur (entre cœurs et pas entre threads) : le coût d'isolation doit donc logiquement être moins important. Pour répondre à la seconde question, il serait intéressant de modifier un processeur avec support du SMT, mais plus complexe que Salers. Par exemple, ce processeur devrait implémenter l'exécution dans le désordre. Si la complexité ajoutée était toujours importante, elle serait probablement relativisée par rapport à la taille du processeur d'origine.

Plus fondamentalement, la manière d'isoler chaque type de ressource partagée se pose. Dans le chapitre 7, plusieurs mécanismes génériques (allocation statique, partitionnement temporel et partitionnement spatial) sont proposés. L'objectif serait donc de savoir précisément sur quels éléments baser la décision d'appliquer l'un d'entre eux. Par exemple, un cas concret est celui des unités d'exécution. Est-il plus pertinent de les allouer exclusivement ou de les partitionner temporellement ? La réponse à cette question varie-t-elle selon le type d'unité d'exécution ? Faut-il partager différemment une MULDIV qu'une FPU ? Toutes ces questions sont autant de choix que devra faire un concepteur matériel au moment de l'implémentation de la microarchitecture.

Publications et communications

Certains des travaux détaillés dans ce manuscrit ont été publiés dans des conférences ou journaux internationaux listés ci-dessous :

- [145] Mathieu Escouteloup, Jacques Fournier, Jean-Louis Lanet et Ronan Lashermes. « Recommendations for a Radically Secure ISA ». In : *4th Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. Mai 2020
- [146] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes et Guillaume Bouffard. « Electromagnetic Fault Injection against a Complex CPU, toward New Micro-Architectural Fault Models ». In : *Journal of Cryptographic Engineering* (2021)
- [148] Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier et Jean-Louis Lanet. « Under the Dome : Preventing Hardware Timing Information Leakage ». In : *20th Smart Card Research and Advanced Application Conference (CARDIS2021)*. Lübeck, Germany, oct. 2021

Ces travaux ont également été présentés dans différents colloques :

- Mathieu ESCOUTELOUP, Jacques FOURNIER, Jean-Louis LANET et Ronan LASHERMES "Microarchitecture security", *Workshop on Practical Hardware Innovation in Security and Characterization (PHISIC 2019)*, octobre 2019
- Mathieu ESCOUTELOUP, Jacques FOURNIER, Jean-Louis LANET et Ronan LASHERMES "Preventing timing information leakages from the microarchitecture", *3rd RISC-V Meeting*, mars 2021

Bibliographie

[1] « Moore's Law ». In : *Wikipedia* (sept. 2021) (cf. p. 1).

[2] John L. Hennessy et David A. Patterson. *Computer Architecture : A Quantitative Approach*. 5th Edition. San Francisco, CA, USA : Morgan Kaufmann, sept. 2011 (cf. p. 7, 111).

[3] David A. Patterson et John L. Hennessy. *Computer Organization and Design RISC-V Edition : The Hardware Software Interface*. 1st Edition. Cambridge, Massachusetts, USA : Morgan Kaufmann, avr. 2017 (cf. p. 7).

[4] Butler W. Lampson. « A Note on the Confinement Problem ». In : *Communications of the ACM* 16.10 (oct. 1973), p. 613-615. doi : 10.1145/362375.362389 (cf. p. 28).

[5] Jakub Szefer. *Principles of Secure Processor Architecture Design*. Sous la dir. de Margaret Martonosi. Morgan & Claypool, oct. 2018 (cf. p. 28).

[6] Wei-Ming Hu. « Lattice Scheduling and Covert Channels ». In : *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy 1992*. Oakland, CA, USA : IEEE Computer Society, 1992, p. 52-61. doi : 10.1109/RISP.1992.213271 (cf. p. 28, 30).

[7] Colin Percival. « Cache Missing for Fun and Profit ». In : (2005), p. 13 (cf. p. 28, 30-32).

[8] Paul C Kocher. « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ». In : *Annual International Cryptology Conference*. Springer, 1996, p. 104-113 (cf. p. 28, 30, 34).

[9] Onur Aciicmez. « Yet Another Microarchitectural Attack : Exploiting I-Cache ». In : *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*. 2007, p. 11-18 (cf. p. 30, 32, 33, 36).

[10] Daniel J Bernstein. « Cache-Timing Attacks on AES ». In : (2005), p. 37 (cf. p. 30, 37, 38).

[11] Joseph Bonneau et Ilya Mironov. « Cache-Collision Timing Attacks against AES ». In : *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006, p. 201-215 (cf. p. 30).

[12] Dag Arne Osvik, Adi Shamir et Eran Tromer. « Cache Attacks and Countermeasures : The Case of AES ». In : *Cryptographers' Track at the RSA Conference*. Springer, 2006, p. 1-20 (cf. p. 30-32, 36, 37, 49-52, 60, 62).

[13] Michael Neve et Jean-Pierre Seifert. « Advances on Access-Driven Cache Attacks on AES ». In : *International Workshop on Selected Areas in Cryptography*. T. 4356. Berlin, Heidelberg : Springer, 2006, p. 147-162. doi : 10.1007/978-3-540-74462-7_11 (cf. p. 30, 31).

[14] Onur Aciicmez et Cetin Kaya Koç. « Microarchitectural Attacks and Countermeasures ». In : *Cryptographic Engineering*. Springer, 2009, p. 475-504 (cf. p. 30).

[15] Dan Page. « Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel ». In : *IACR Cryptol. ePrint Arch.* Citeseer, 2002, p. 1-23 (cf. p. 30).

[16] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero et Gianluca Palermo. « AES Power Attack Based on Induced Cache Miss and Countermeasure ». In : *International Conference on Information Technology : Coding and Computing (ITCC'05)*. T. II. Las Vegas, NV, USA : IEEE, 2005, p. 586-591. doi : 10.1109/ITCC.2005.62 (cf. p. 30).

[17] Onur Aciicmez et Çetin Kaya Koç. « Trace-Driven Cache Attacks on AES (Short Paper) ». In : *International Conference on Information and Communications Security*. Springer, 2006, p. 112-121 (cf. p. 30).

[18] Jean-François Gallais, Ilya Kizhvatov et Michael Tunstall. « Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations ». In : *International Workshop on Information Security Applications*. T. 6513. Berlin, Heidelberg : Springer, 2010, p. 243-257 (cf. p. 30).

[19] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri et Hiroshi Miyauchi. « Cryptanalysis of DES Implemented on Computers with Cache ». In : *International Workshop on Cryptographic Hardware and Embedded Systems*. T. 2779. Berlin, Heidelberg : Springer, 2003, p. 62-76. doi : 10.1007/978-3-540-45238-6_6 (cf. p. 30).

[20] Onur Aciicmez, Billy Bob Brumley et Philipp Grabher. « New Results on Instruction Cache Attacks ». In : *Cryptographic Hardware and Embedded Systems, CHES 2010*. Sous la dir. de David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Stefan Mangard et François-Xavier Standaert. T. 6225. Berlin, Heidelberg : Springer, 2010, p. 110-124 (cf. p. 31, 33, 36).

[21] Qian Ge, Yuval Yarom, David Cock et Gernot Heiser. « A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware ». In : *Journal of Cryptographic Engineering* 8.1 (avr. 2018), p. 1-27. doi : 10.1007/s13389-016-0141-6 (cf. p. 31, 36).

[22] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Bhatti et Guy Gogniat. « Winter Is Here ! A Decade of Cache-Based Side-Channel Attacks, Detection & Mitigation for RSA ». In : *Information Systems* 92 (2020), p. 36 (cf. p. 31).

[23] Eran Tromer, Dag Arne Osvik et Adi Shamir. « Efficient Cache Attacks on AES, and Countermeasures ». In : *Journal of Cryptology* 23.1 (2010), p. 37-71 (cf. p. 31).

[24] Yinqian Zhang, Ari Juels, Michael K Reiter et Thomas Ristenpart. « Cross-VM Side Channels and Their Use to Extract Private Keys ». In : *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 2012, p. 305-316 (cf. p. 32).

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser et Ruby B. Lee. « Last-Level Cache Side-Channel Attacks Are Practical ». In : *2015 IEEE Symposium on Security and Privacy*. San Jose, CA : IEEE, mai 2015, p. 605-622. doi : 10.1109/SP.2015.43 (cf. p. 32, 33, 36).

[26] Thomas Ristenpart, Eran Tromer, Hovav Shacham et Stefan Savage. « Hey, You, Get Off of My Cloud : Exploring Information Leakage in Third-Party Compute Clouds ». In : *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. New York, NY, USA : ACM, 2009, p. 199-212. doi : 10.1145/1653662.1653687 (cf. p. 32, 38).

[27] Qian Ge, Yuval Yarom, Frank Li et Gernot Heiser. « Your Processor Leaks Information - and There's Nothing You Can Do About It ». In : *arXiv :1612.04474 [cs]* (sept. 2017) (cf. p. 32, 51, 133).

[28] David Gullasch, Endre Bangerter et Stephan Krenn. « Cache Games – Bringing Access-Based Cache Attacks on AES to Practice ». In : *2011 IEEE Symposium on Security and Privacy*. Mai 2011, p. 490-505. doi : 10.1109/SP.2011.22 (cf. p. 32, 80).

[29] Yuval Yarom et Katrina Falkner. « FLUSH+RELOAD : A High Resolution, Low Noise, L3 Cache Side-Channel Attack ». In : *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA, USA : USENIX Association, août 2014, p. 719-732 (cf. p. 32, 33, 36, 80).

[30] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth et Berk Sunar. « Wait a Minute ! A Fast, Cross-VM Attack on AES ». In : *International Workshop on Recent Advances in Intrusion Detection*. Sous la dir. d'Angelos Stavrou, Herbert Bos et Georgios Portokalidis. T. 8688. Cham : Springer International Publishing, 2014, p. 299-319. doi : 10.1007/978-3-319-11379-1_15 (cf. p. 32).

[31] Naomi Benger, Joop van de Pol, Nigel P. Smart et Yuval Yarom. « "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way ». In : *International Workshop on Cryptographic Hardware and Embedded Systems*. Sous la dir. de David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Camille Salinesi, Moira C. Norrie et Óscar Pastor. T. 7908. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, p. 75-92. doi : 10.1007/978-3-662-44709-3_5 (cf. p. 32).

[32] Yuval Yarom et Naomi Benger. « Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-Channel Attack ». In : *IACR Cryptol. ePrint Arch.* 2014 (2014), p. 11 (cf. p. 32).

[33] Joop Van de Pol, Nigel P Smart et Yuval Yarom. « Just a Little Bit More ». In : *Cryptographers' Track at the RSA Conference*. 2015, p. 3-21 (cf. p. 32).

[34] Daniel Gruss, Raphael Spreitzer et Stefan Mangard. « Cache Template Attacks : Automating Attacks on Inclusive Last-Level Caches ». In : *Proceedings of the 24th USENIX Security Symposium*. Août 2015, p. 897-912 (cf. p. 32).

[35] Daniel Gruss, Clémentine Maurice, Klaus Wagner et Stefan Mangard. « Flush+Flush : A Fast and Stealthy Cache Attack ». In : *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA 2016. Berlin, Heidelberg : Springer, 2016, p. 279-299. doi : 10.1007/978-3-319-40667-1_14 (cf. p. 32, 80).

[36] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen et Ashish Venkat. « I See Dead *M*ops : Leaking Secrets via Intel/AMD Micro-Op Caches ». In : 2021, p. 14 (cf. p. 33, 54).

[37] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen et Richard Schlichting. « An Exploration of L2 Cache Covert Channels in Virtualized Environments ». In : *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. Chicago, Illinois, USA : ACM Press, 2011, p. 29. doi : 10.1145/2046660.2046670 (cf. p. 33).

[38] Onur Aciicmez, Çetin Kaya Koç et Jean-Pierre Seifert. « Predicting Secret Keys via Branch Prediction ». In : *Cryptographers' Track at the RSA Conference*. T. 4377. Berlin, Heidelberg : Springer, 2007, p. 225-242. doi : 10.1007/11967668_15 (cf. p. 33, 34, 36).

[39] Onur Aciicmez, Cetin Kaya Koç et Jean-Pierre Seifert. « On the Power of Simple Branch Prediction Analysis ». In : *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. ASIACCS '07. Singapore : ACM, mar. 2007, p. 312-320. doi : 10.1145/1229285.1266999 (cf. p. 34, 36).

[40] Onur Aciicmez, Shay Gueron et Jean-Pierre Seifert. « New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures ». In : *IMA International Conference on Cryptography and Coding*. Springer. 2007, p. 185-203 (cf. p. 34, 36).

[41] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens et Raoul Strackx. « Telling Your Secrets without Page Faults : Stealthy Page Table-Based Attacks on Enclaved Execution ». In : *26th USENIX Security Symposium (USENIX Security 17)*. 2017, p. 1041-1056 (cf. p. 34, 45).

[42] Patrick Cronin et Chengmo Yang. « A Fetching Tale : Covert Communication with the Hardware Prefetcher ». In : *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. Mai 2019, p. 101-110. doi : 10.1109/HST.2019.8741033 (cf. p. 34, 49, 51, 52).

[43] Andrew Ferraiuolo, Yao Wang, Rui Xu, Danfeng Zhang, Andrew Myers et G Edward Suh. « Full-Processor Timing Channel Protection with Applications to Secure Hardware Compartments ». In : (avr. 2017), p. 12 (cf. p. 34, 51-55, 66, 143).

[44] Gaël Hachez et Jean-Jacques Quisquater. « Montgomery Exponentiation with No Final Subtractions : Improved Results ». In : *International Workshop on Cryptographic Hardware and Embedded Systems*. T. 1965. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 293-301 (cf. p. 34).

[45] Johann Grossschädl, Elisabeth Oswald, Dan Page et Michael Tunstall. « Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications ». In : *International Conference on Information Security and Cryptology*. T. 5984. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 176-192 (cf. p. 34).

[46] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner et Hovav Shacham. « On Subnormal Floating Point and Abnormal Timing ». In : *IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA : IEEE, mai 2015, p. 623-639. doi : 10.1109/SP.2015.44 (cf. p. 35).

[47] Michael Schwarz, Martin Schwarzl, Moritz Lipp et Daniel Gruss. « NetSpectre : Read Arbitrary Memory over Network ». In : *arXiv :1807.10535 [Cs]*. Juil. 2018 (cf. p. 35, 40, 56).

[48] Yuval Yarom, Daniel Genkin et Nadia Heninger. « CacheBleed : A Timing Attack on OpenSSL Constant Time RSA ». In : *Journal of Cryptographic Engineering* 7.2 (2017), p. 99-112 (cf. p. 35, 38, 41, 50, 54, 62).

[49] Zhenyu Wu, Zhang Xu et Haining Wang. « Whispers in the Hyper-Space : High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud ». In : *IEEE/ACM Transactions on Networking* 23.2 (avr. 2015), p. 603-615. doi : 10.1109/TNET.2014.2304439 (cf. p. 36, 38).

[50] Onur Aciicmez et Jean-Pierre Seifert. « Cheap Hardware Parallelism Implies Cheap Security ». In : *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. Vienna, Austria : IEEE, sept. 2007, p. 80-91. doi : 10.1109/FDTC.2007.16 (cf. p. 37, 52).

[51] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida Garcia et Nicola Tuveri. « Port Contention for Fun and Profit ». In : *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA : IEEE, mai 2019, p. 870-887. doi : 10.1109/SP.2019.00066 (cf. p. 37, 52).

[52] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer et Anil Kurmus. « SMoTherSpectre : Exploiting Speculative Execution through Port Contention ». In : *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, p. 785-800. doi : 10.1145/3319535.3363194 (cf. p. 37).

[53] Yao Wang, Andrew Ferraiuolo et G. Edward Suh. « Timing Channel Protection for a Shared Memory Controller ». In : *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Fév. 2014, p. 225-236. doi : 10.1109/HPCA.2014.6835934 (cf. p. 38, 54, 110).

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz et Yuval Yarom. « Spectre Attacks : Exploiting Speculative Execution ». In : *40th IEEE Symposium on Security and Privacy (S&P'19)*. Los Alamitos, CA, USA : IEEE Computer Society, mai 2019, p. 1-19. doi : 10.1109/SP.2019.00002 (cf. p. 38-40, 58).

[55] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom et Mike Hamburg. « Meltdown : Reading Kernel Memory from User Space ». In : *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, USA : USENIX Association, août 2018, p. 973-990 (cf. p. 38).

[56] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin et Daniel Gruss. « A Systematic Evaluation of Transient Execution Attacks and Defenses ». In : *28th USENIX Security Symposium (USENIX Security 19)*. Nov. 2018, p. 249-266 (cf. p. 39, 40, 42).

[57] Claudio Canella, Khaled N. Khasawneh et Daniel Gruss. « The Evolution of Transient-Execution Attacks ». In : *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. Virtual Event China : ACM, sept. 2020, p. 163-168. doi : 10.1145/3386263.3407583 (cf. p. 39).

[58] Jann Horn. *Reading Privileged Memory with a Side-Channel*. Jan. 2018. url : https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html (cf. p. 39).

[59] *Intel Analysis of Speculative Execution Side Channels*. Jan. 2018 (cf. p. 39).

[60] *Cache Speculation Side-Channels*. Rapp. tech. Arm, 2018 (cf. p. 39, 41, 42, 57).

[61] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song et Nael Abu-Ghazaleh. « Spectre Returns ! Speculation Attacks Using the Return Stack Buffer ». In : *12th {USENIX Workshop on Offensive Technologies ({WOOT 18)*. 2018 (cf. p. 40).

[62] Giorgi Maisuradze et Christian Rossow. « Ret2spec : Speculative Execution Using Return Stack Buffers ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. Toronto, Canada : ACM Press, 2018, p. 2109-2122. doi : 10.1145/3243734.3243761 (cf. p. 40).

[63] Jann Horn. *Speculative Execution, Variant 4 : Speculative Store Bypass*. Fév. 2018. url : https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (cf. p. 41).

[64] *Security Analysis of AMD Predictive Store Forwarding*. Rapp. tech. Advanced Micro Devices, mar. 2021 (cf. p. 41).

[65] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice et Stefan Mangard. « KASLR Is Dead : Long Live KASLR ». In : *Engineering Secure Software and Systems*. Sous la dir. d'Eric Bodden, Mathias Payer et Elias Athanasopoulos. T. 10379. Bonn, Germany : Springer International Publishing, juil. 2017, p. 161-176 (cf. p. 41, 56).

[66] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom et Raoul Strackx. « Foreshadow : Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution ». In : *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, USA : USENIX Association, août 2018, p. 991-1008 (cf. p. 42).

[67] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch et Yuval Yarom. « Foreshadow-NG : Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution ». In : août 2018, p. 7 (cf. p. 42).

[68] Julian Stecklina et Thomas Prescher. « LazyFP : Leaking FPU Register State Using Microarchitectural Side-Channels ». In : *arXiv :1806.07480 [Cs.OS]*. Juin 2018 (cf. p. 42).

[69] *Q2 2018 Speculative Execution Side Channel Update*. 2018. url : https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html (cf. p. 42).

[70] Vladimir Kiriansky et Carl Waldspurger. « Speculative Buffer Overflows : Attacks and Defenses ». In : *arXiv :1807.03757 [Cs]*. Juil. 2018 (cf. p. 42, 53, 54, 66, 79).

[71] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos et Cristiano Giuffrida. « RIDL : Rogue In-Flight Data Load ». In : *40th IEEE Symposium on Security and Privacy (S&P'19)*. San Francisco, CA, USA, mai 2019, p. 18 (cf. p. 43).

[72] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher et Daniel Gruss. « ZombieLoad : Cross-Privilege-Boundary Data Sampling ». In : *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom : ACM, nov. 2019, p. 753-768. doi : 10.1145/3319535.3354252 (cf. p. 43).

[73] *Microarchitectural Data Sampling (Fallout/Zombieload/RIDL)*. Mai 2019. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html (cf. p. 43, 51).

[74] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck et Yuval Yarom. « Fallout : Leaking Data on Meltdown-Resistant CPUs ». In : *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM. London United Kingdom : ACM, nov. 2019, p. 769-784. doi : 10.1145/3319535.3363219 (cf. p. 43, 64, 115).

[75] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin et Yuval Yarom. « CacheOut : Leaking Data on Intel CPUs via Cache Evictions ». In : *S&P*. Mai 2021, p. 16 (cf. p. 43, 65).

[76] Stephan van Schaik, Andrew Kwong, Daniel Genkin et Yuval Yarom. *SGAxe : How SGX Fails in Practice*. 2020. url : https://sgaxeattack.com/ (cf. p. 43, 65).

[77] *L1D Eviction Sampling*. Jan. 2020. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/l1d-eviction-sampling.html (cf. p. 43).

[78] *Vector Register Sampling / CVE-2020-0548 , CVE 2020-8696 /...* Jan. 2020. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/vector-register-sampling.html (cf. p. 43).

[79] *Intel® Transactional Synchronization Extensions (Intel® TSX)...* Déc. 2019. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/intel-tsx-asynchronous-abort.html (cf. p. 43, 50).

[80] Michael Schwarz, Claudio Canella, Lukas Giner et Daniel Gruss. « Store-to-Leak Forwarding :Leaking Data on Meltdown-Resistant CPUs ». In : *arXiv preprint arXiv :1905.05725* (2019), p. 14 (cf. p. 43, 115).

[81] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos et Cristiano Giuffrida. « CROSSTALK : Speculative Data Leaks Across Cores Are Real ». In : *IEEE Symposium on Security & Privacy (S&P)*. 2021, p. 16 (cf. p. 43).

[82] *Special Register Buffer Data Sampling / CVE-2020-0543 / INTEL-SA-00232*. Sept. 2020. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/special-register-buffer-data-sampling.html (cf. p. 43).

[83] *Snoop-Assisted L1 Data Sampling / CVE-2020-0550 / INTEL-SA-00330*. Oct. 2020. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/snoop-assisted-l1-data-sampling.html (cf. p. 43).

[84] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss et Frank Piessens. « LVI : Hijacking Transient Execution through Microarchitectural Load Value Injection ». In : *2020 IEEE Symposium on Security and Privacy (SP)*. Mai 2020, p. 54-72. doi : 10.1109/SP40000.2020.00089 (cf. p. 44, 56).

[85] *Load Value Injection*. Jan. 2020. url : https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/load-value-injection.html (cf. p. 44, 56).

[86] Yuanzhong Xu, Weidong Cui et Marcus Peinado. « Controlled-Channel Attacks : Deterministic Side Channels for Untrusted Operating Systems ». In : *2015 IEEE Symposium on Security and Privacy*. Mai 2015, p. 640-656. doi : 10.1109/SP.2015.45 (cf. p. 44).

[87] Mathias Morbitzer, Manuel Huber, Julian Horsch et Sascha Wessel. « SEVered : Subverting AMD's Virtual Machine Encryption ». In : *Proceedings of the 11th European Workshop on Systems Security*. EuroSec'18. New York, NY, USA : Association for Computing Machinery, avr. 2018, p. 1-6. doi : 10.1145/3193111.3193112 (cf. p. 45).

[88] Jo Van Bulck, Frank Piessens et Raoul Strackx. « Nemesis : Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA : Association for Computing Machinery, oct. 2018, p. 178-195. doi : 10.1145/3243734.3243822 (cf. p. 45).

[89] D Page. « Defending against Cache-Based Side-Channel Attacks ». In : *Information Security Technical Report* 8.1 (mar. 2003), p. 30-44. doi : 10.1016/S1363-4127(03)00104-3 (cf. p. 49, 60, 62).

[90] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King et Christopher J. Nitta. « Position Paper : A Case for Exposing Extra-Architectural State in the ISA ». In : *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. Los Angeles California : ACM, juin 2018, p. 1-6. doi : 10.1145/3214292.3214300 (cf. p. 49, 57, 58, 60, 65, 79).

[91] Qian Ge, Yuval Yarom et Gernot Heiser. « No Security Without Time Protection : We Need a New Hardware-Software Contract ». In : *Proceedings of the 9th Asia-Pacific Workshop on Systems - APSys '18*. Jeju Island, Republic of Korea : ACM Press, 2018, p. 1-9. doi : 10.1145/3265723.3265724 (cf. p. 50-52, 55, 65, 79, 107, 136).

[92] Andrew Marshall, Michael Howard, Grant Bugher, Brian Harden, Charlie Kaufman, Martin Rues et Vittorio Bertocci. « Security Best Practices For Developing Windows Azure Applications ». In : *Microsoft Corp* 42 (2010), p. 27 (cf. p. 50).

[93] Michael Larabel. *Intel Hyper Threading Performance With A Core I7 On Ubuntu 18.04 LTS - Phoronix*. Juin 2018. url : https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4 (cf. p. 50).

[94] Martin Schwarzl, Claudio Canella, Daniel Gruss et Michael Schwarz. « Specfuscator : Evaluating Branch Removal as a Spectre Mitigation ». In : *Financial Cryptography and Data Security 2021* (2021), p. 18 (cf. p. 50).

[95] *Speculative Execution Side Channel Mitigations*. Mai 2018 (cf. p. 50, 55-57, 79).

[96] *Speculative Store Bypass Disable*. Mai 2018 (cf. p. 50).

[97] Michael Larabel. *Intel To Disable TSX By Default On More CPUs With New Microcode*. Juin 2021. url : https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode (cf. p. 50).

[98] Victor Costan, Ilia Lebedev et Srinivas Devadas. « Sanctum : Minimal Hardware Extensions for Strong Software Isolation ». In : *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, USA : USENIX Association, août 2016, p. 857-874 (cf. p. 51, 53, 76).

[99] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind et Srinivas Devadas. « MI6 : Secure Enclaves in a Speculative Out-of-Order Processor ». In : *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA : ACM, oct. 2019, p. 42-56. doi : 10.1145/3352460.3358310 (cf. p. 51, 54, 55, 57, 66, 79, 80, 112, 143).

[100] Michael Godfrey et Mohammad Zulkernine. « A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud ». In : *2013 IEEE Sixth International Conference on Cloud Computing*. Juin 2013, p. 163-170. doi : 10.1109/CLOUD.2013.21 (cf. p. 51).

[101] *Software Techniques for Managing Speculation on AMD Processors*. Sept. 2020 (cf. p. 51, 55-57, 79).

[102] Nils Wistoff, Moritz Schneider, Frank K Gürkaynak, Luca Benini et Gernot Heiser. « Microarchitectural Timing Channels and Their Prevention on an Open-Source 64-Bit RISC-V Core ». In : *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. Fév. 2021, p. 627-632. doi : 10.23919/DATE51398.2021.9474214 (cf. p. 51, 79, 112).

[103] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini et Gernot Heiser. « Prevention of Microarchitectural Covert Channels on an Open-Source 64-Bit RISC-V Core ». In : *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. Mai 2020 (cf. p. 51, 79, 144).

[104] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi et Geoff V. Merrett. « BRB : Mitigating Branch Predictor Side-Channels. » In : *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Fév. 2019, p. 466-477. doi : 10.1109/HPCA.2019.00058 (cf. p. 52).

[105] Taesoo Kim, Marcus Peinado et Gloria Mainar-Ruiz. « STEALTHMEM : System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud ». In : *21st {USENIX Security Symposium ({USENIX Security 12)*. 2012, p. 189-204 (cf. p. 52).

[106] D Page. « Partitioned Cache Architecture as a Side-Channel Defence Mechanism ». In : (2005) (cf. p. 53).

[107] Zhenghong Wang et Ruby B. Lee. « New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks ». In : *Proceedings of the 34th Annual International Symposium on Computer Architecture - ISCA '07*. San Diego, California, USA : ACM Press, 2007, p. 494. doi : 10.1145/1250662.1250723 (cf. p. 53, 60).

[108] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert et Huiyang Zhou. « Hardware-Software Integrated Approaches to Defend against Software Cache-Based Side Channel Attacks ». In : *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Fév. 2009, p. 393-404. doi : 10.1109/HPCA.2009.4798277 (cf. p. 53, 61).

[109] Daniel Townley et Dmitry Ponomarev. « SMT-COP : Defeating Side-Channel Attacks on Execution Units in SMT Processors ». In : *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2019, p. 43-54. doi : 10.1109/PACT.2019.00012 (cf. p. 54, 110, 145).

[110] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi et Emmanuel Stapf. « CURE : A Security Architecture with CUstomizable and Resilient Enclaves ». In : *30th USENIX Security Symposium (USENIX Security 21)*. 2021, p. 18 (cf. p. 55, 76, 79).

[111] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss et Khaled N. Khasawneh. « Evolution of Defenses against Transient-Execution Attacks ». In : *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. Virtual Event China : ACM, sept. 2020, p. 169-174. doi : 10.1145/3386263.3407584 (cf. p. 55).

[112] Paul Turner. *Retpoline : A Software Construct for Preventing Branch-Target-Injection*. Jan. 2018. url : https://support.google.com/faqs/answer/7625886 (cf. p. 55, 65).

[113] *Retpoline : A Branch Target Injection Mitigation*. Juin 2018 (cf. p. 55, 56).

[114] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein et Christof Fetzer. « You Shall Not Bypass : Employing Data Dependencies to Prevent Bounds Check Bypass ». In : *arXiv :1805.08506 [cs]* (oct. 2018) (cf. p. 56).

[115] Filip Pizlo. *What Spectre and Meltdown Mean For WebKit*. Jan. 2018. url : https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (cf. p. 56, 62).

[116] Chandler Carruth. *[Llvm-Dev] RFC : Speculative Load Hardening (a Spectre Variant #1 mitigation)*. Mar. 2018. url : https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html (cf. p. 56).

[117] Charles Reis, Alexander Moshchuk et Nasko Oskov. « Site Isolation : Process Separation for Web Sites within the Browser ». In : *28th {USENIX Security Symposium ({USENIX Security 19)*. 2019, p. 1661-1678 (cf. p. 56).

[118] Roberto Guanciale, Musard Balliu et Mads Dam. « InSpectre : Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis ». In : *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event USA : ACM, oct. 2020, p. 1853-1869. doi : 10.1145/3372297.3417246 (cf. p. 56).

[119] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch et Baris Kasikci. « NDA : Preventing Speculative Execution Attacks at Their Source ». In : *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA : ACM, oct. 2019, p. 572-586. doi : 10.1145/3352460.3358306 (cf. p. 58).

[120] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev et Nael Abu-Ghazaleh. « SafeSpec : Banishing the Spectre of a Meltdown with Leakage-Free Speculation ». In : *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC16)*. Las Vegas, NV, USA : ACM Press, juin 2019, p. 1-6. doi : 10.1145/3316781.3317903 (cf. p. 58, 65).

[121] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher et Josep Torrellas. « InvisiSpec : Making Speculative Execution Invisible in the Cache Hierarchy ». In : *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka : IEEE, oct. 2018, p. 428-441. doi : 10.1109/MICRO.2018.00042 (cf. p. 58).

[122] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis et Krste Asanović. « Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture ». In : *CARRV 2019*. 2019, p. 7 (cf. p. 58).

[123] Gururaj Saileshwar et Moinuddin K. Qureshi. « CleanupSpec : An "Undo" Approach to Safe Speculation ». In : *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA : ACM, oct. 2019, p. 73-86. doi : 10.1145/3352460.3358314 (cf. p. 59-61, 65).

[124] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song et Nael Abu-Ghazaleh. « SPECCFI : Mitigating Spectre Attacks Using CFI Informed Speculation ». In : *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. Juin 2019, p. 39-53 (cf. p. 59).

[125] Divya Ojha et Sandhya Dwarkadas. « TimeCache : Using Time to Eliminate Cache Side Channels When Sharing Software ». In : *International Symposium on Computer Architecture (ISCA)*. Virtual, juin 2021, p. 375-387 (cf. p. 60).

[126] Zhenghong Wang et Ruby Lee. « Covert and Side Channels Due to Processor Architecture ». In : *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. Miami Beach, FL, USA : IEEE, déc. 2006, p. 473-482. doi : 10.1109/ACSAC.2006.20 (cf. p. 60).

[127] Zhenghong Wang et Ruby B. Lee. « A Novel Cache Architecture with Enhanced Performance and Security ». In : *2008 41st IEEE/ACM International Symposium on Microarchitecture*. Nov. 2008, p. 83-93. doi : 10.1109/MICRO.2008.4771781 (cf. p. 61).

[128] Moinuddin K. Qureshi. « CEASER : Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping ». In : *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka : IEEE, oct. 2018, p. 775-787. doi : 10.1109/MICRO.2018.00068 (cf. p. 61).

[129] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian et Dan Meng. « A Lightweight Isolation Mechanism for Secure Branch Predictors ». In : *arXiv :2005.08183 [cs]* (mai 2020) (cf. p. 61).

[130] *Mitigating Speculative Execution Side-Channel Attacks in Microsoft Edge and Internet Explorer*. Jan. 2018. url : https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/ (cf. p. 62).

[131] *Mitigating Side-Channel Attacks - The Chromium Projects*. 2018. url : https://www.chromium.org/Home/chromium-security/ssca (cf. p. 62).

[132] Luke Wagner. *Mitigations Landing for New Class of Timing Attack*. Jan. 2018. url : https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack (cf. p. 62).

[133] Michael Schwarz, Clémentine Maurice, Daniel Gruss et Stefan Mangard. « Fantastic Timers and Where to Find Them : High-Resolution Microarchitectural Attacks in JavaScript ». In : *Financial Cryptography and Data Security*. Sous la dir. d'Aggelos Kiayias. Lecture Notes in Computer Science. Cham : Springer International Publishing, 2017, p. 247-267. doi : 10.1007/978-3-319-70972-7_13 (cf. p. 62).

[134] Jingquan Ge, Neng Gao, Chenyang Tu, Ji Xiang et Zeyi Liu. « AdapTimer : Hardware/Software Collaborative Timer Resistant to Flush-Based Cache Attacks on ARM-FPGA Embedded SoC ». In : *2019 IEEE 37th International Conference on Computer Design (ICCD)*. Nov. 2019, p. 585-593. doi : 10.1109/ICCD46524.2019.00085 (cf. p. 62).

[135] Jan Tobias Mühlberg et Jo Van Bulck. « Reflections on Post-Meltdown Trusted Computing : A Case for Open Security Processors ». In : *;Login : the USENIX magazine* 43.3 (2018), p. 1-4 (cf. p. 68).

[136] CSRC Content Editor. *Security Domain - Glossary | CSRC*. url : https://csrc.nist.gov/glossary/term/security_domain (cf. p. 71).

[137] CSRC Content Editor. *Security Policy - Glossary | CSRC*. url : https://csrc.nist.gov/glossary/term/security_policy (cf. p. 71).

[138] Elliott I Organick. *The Multics System : An Examination of Its Structure*. MIT press, 1972 (cf. p. 73).

[139] Andrew Waterman et Krste Asanovic. *The RISC-V Instruction Set Manual, Volume II : Privileged Architecture*. Juin 2019 (cf. p. 73, 83, 87, 96, 120).

[140] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song et Krste Asanović. « Keystone : An Open Framework for Architecting TEEs ». In : *arXiv :1907.10119 [Cs]*. Sept. 2019 (cf. p. 76).

[141] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl et Daniel Gruss. « ConTExt : A Generic Approach for Mitigating Spectre ». In : *Proceedings of the Network and Distributed System Security Symposium*. T. 10. 2020. doi : https://doi.org/10.14722/ndss (cf. p. 80).

[142] Andrew Waterman et Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I : User-Level ISA*. Déc. 2019 (cf. p. 83, 87, 120).

[143] *The Heart of RISC-V Development Is Unmatched - SiFive*. Oct. 2020. url : https://www.sifive.com/blog/the-heart-of-risc-v-development-is-unmatched (cf. p. 83).

[144] *Nios® V Processor - Intel® FPGA*. 2021. url : https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor/v.html (cf. p. 83).

[145] Mathieu Escouteloup, Jacques Fournier, Jean-Louis Lanet et Ronan Lashermes. « Recommendations for a Radically Secure ISA ». In : *4th Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. Mai 2020, p. 7 (cf. p. 97, 153).

[146] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes et Guillaume Bouffard. « Electromagnetic Fault Injection against a Complex CPU, toward New Micro-Architectural Fault Models ». In : *Journal of Cryptographic Engineering* (2021), p. 1-15 (cf. p. 97, 153).

[147] Alexandre Gonzalvez et Ronan Lashermes. « A Case against Indirect Jumps for Secure Programs ». In : *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering - SSPREW9 '19*. San Juan, Puerto Rico : ACM Press, 2019, p. 1-10. doi : 10.1145/3371307.3371314 (cf. p. 97).

[148] Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier et Jean-Louis Lanet. « Under the Dome : Preventing Hardware Timing Information Leakage ». In : *20th Smart Card Research and Advanced Application Conference (CARDIS2021)*. Lübeck, Germany, oct. 2021 (cf. p. 105, 119, 133, 153).

[149] *Chisel/FIRRTL : Home*. 2021. url : https://www.chisel-lang.org/ (cf. p. 119).

[150] *Embench : A Modern Embedded Benchmark Suite*. 2021. url : https://www.embench.org/ (cf. p. 134, 141).

[151] S.E. Raasch et S.K. Reinhardt. « The Impact of Resource Partitioning on SMT Processors ». In : *Oceans 2002 Conference and Exhibition. Conference Proceedings (Cat. No.02CH37362)*. New Orleans, LA, USA : IEEE Comput. Soc, 2003, p. 15-25. doi : 10.1109/PACT.2003.1237998 (cf. p. 151).

Annexes

A. Paramètres des processeurs

Les processeurs Aubrac et Salers sont basés sur le langage Chisel. Avant de générer un circuit (*e.g.* en Verilog), certains paramètres peuvent être spécifiés. Ci-dessous sont listés l'ensemble des paramètres utilisés respectivement pour les deux processeurs.

A.1. Processeur Aubrac

```
// GLOBAL PARAMETERS
def debug = false                        // Utiliser elements de debug
def pcBoot: Int = 0x40                   // Adresse au demarrage
def useDome: Boolean = true              // Utiliser les domes
def useDomeConstFlush: Boolean = true    // Utiliser flush constant
def nDomeFlushCycle: Int = 10            // Duree flush constant

// PIPELINE - FRONT-END
def nFetchInstr: Int = 2                 // Nombre d'instructions recuperees simultanement
def useIMemSeq: Boolean = false          // Ajouter registres pour le acces du fetch
def useIf1Stage: Boolean = false         // Ajouter un etage dans le front-end
def nFetchFifoDepth: Int = 4             // Nombre d'instructions dans le fetch buffer
def useNlp: Boolean = false              // Ajouter NLP (BHT + BTB)
def nBhtSet: Int = 8                     // Nombre d'ensembles dans le BHT
def nBhtSetEntry: Int = 128              // Nombre de compteurs par ensemble du BHT
def nBhtBit: Int = 2                     // Taille des compteurs du BHT
def nBtbLine: Int = 16                   // Nombre de lignes dans le BTB
def nBtbTagBit: Int = 10                 // Taille du tag dans le BTB

// PIPELINE - BACK-END
def useMulDiv: Boolean = false           // Ajouter les multiplications et divisions
def useFencei: Boolean = true            // Ajouter l'instruction FENCE.I
def useMemStage: Boolean = true          // Ajouter l'etage MEM du pipeline
def useBranchReg: Boolean = true         // Ajouter des registres pour les branchements

// CACHE - L1I
def nL1ISetReadPort: Int = 1             // Nombre de ports de lecture par ensemble
def nL1ISetWritePort: Int = 1            // Nombre de ports d'ecriture par ensemble
def nL1IWordByte: Int = 16               // Octet par mot L1I <-> Memoire
def slctL1IPolicy: String = "BitPLRU"    // Police remplacement
def nL1ISubCache: Int = 1                // Nombre de sous-caches (groupe d'ensembles)
def nL1ISet: Int = 4                     // Nombre d'ensembles
def nL1ILine: Int = 4                    // Nombre de ligne par ensemble
def nL1IWord: Int = 4                    // Nombre de mots par ligne

// CACHE - L1D
def nL1DSetReadPort: Int = 1             // Nombre de ports de lecture par ensemble
def nL1DSetWritePort: Int = 1            // Nombre de ports d'ecriture par ensemble
def nL1DWordByte: Int = 16               // Octet par mot L1D <-> Memoire
def slctL1DPolicy: String = "BitPLRU"    // Police remplacement
def nL1DSubCache: Int = 1                // Nombre de sous-caches (groupe d'ensembles)
def nL1DSet: Int = 4                     // Nombre d'ensembles
def nL1DLine: Int = 4                    // Nombre de ligne par ensemble
def nL1DWord: Int = 4                    // Nombre de mots par ligne
```

A.2. Processeur Salers

```
// GLOBAL PARAMETERS
def debug: Boolean = false                // Utiliser elements de debug
def pcBoot: Int = 0x40                    // Adresse au demarrage
def useDome: Boolean = true               // Utiliser les domes
def useDomeConstFlush: Boolean = true     // Utiliser flush constant
def nDomeFlushCycle: Int = 10             // Duree flush constant

// PIPELINE - FRONT-END
def nFetchInstr: Int = 2                  // Nombre d'instructions recuperees simultanement
def useIMemSeq: Boolean = true            // Ajouter registres pour le acces du fetch
def useIf1Stage: Boolean = false          // Ajouter un etage dans le front-end
def nFetchFifoDepth: Int = 4              // Nombre d'instructions dans le fetch buffer
def useNlp: Boolean = true                // Ajouter NLP (BHT + BTB)
def nBhtSet: Int = 8                      // Nombre d'ensembles dans le BHT
def nBhtSetEntry: Int = 128               // Nombre de compteurs par ensemble du BHT
def nBhtBit: Int = 2                      // Taille des compteurs du BHT
def nBtbLine: Int = 16                    // Nombre de lignes dans le BTB
def nBtbTagBit: Int = 10                  // Taille du tag dans le BTB

// PIPELINE - BACK-END
def nBackPort: Int = 2                    // Largeur du pipeline (Nombre d'instructions par etage)
def nAlu: Int = 2                         // Nombre d'ALU
def nMulDiv: Int = 1                      // Nombre de MULDIV (0 = pas de multiplication/division)
def useFencei: Boolean = true             // Ajouter l'instruction FENCE.I
def useBranchReg: Boolean = true          // Ajouter des registres pour les branchements

// CACHE - L1I
def nL1ISetReadPort: Int = 2              // Nombre de ports de lecture par ensemble
def nL1ISetWritePort: Int = 1             // Nombre de ports d'ecriture par ensemble
def nL1IWordByte: Int = 16                // Octet par mot L1I <-> Memoire
def slctL1IPolicy: String = "BitPLRU"     // Police remplacement
def nL1ISubCache: Int = 1                 // Nombre de sous-caches (groupe d'ensembles)
def nL1ISet: Int = 4                      // Nombre d'ensembles
def nL1ILine: Int = 4                     // Nombre de ligne par ensemble
def nL1IWord: Int = 4                     // Nombre de mots par ligne

// CACHE - L1D
def nL1DSetReadPort: Int = 2              // Nombre de ports de lecture par ensemble
def nL1DSetWritePort: Int = 1             // Nombre de ports d'ecriture par ensemble
def nL1DWordByte: Int = 16                // Octet par mot L1D <-> Memoire
def slctL1DPolicy: String = "BitPLRU"     // Police remplacement
def nL1DSubCache: Int = 1                 // Nombre de sous-caches (groupe d'ensembles)
def nL1DSet: Int = 4                      // Nombre d'ensembles
def nL1DLine: Int = 4                     // Nombre de ligne par ensemble
def nL1DWord: Int = 4                     // Nombre de mots par ligne
```

B. Liste des instructions pour les domes

Format des instructions

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
imm[6 :0]	rs2	rs1[confs1]	func3	rd	opcode	R-type
imm[11 :5]	rs2	rs1[confs1]	func3	imm[4 :0]	opcode	S-type
7	5	5	3	5	7	

Liste des instructions pour les domes

imm[11 :5]	rs2	rs1[confs1]	000	imm[4 :0]	1110111	`dome.load`
imm[11 :5]	rs2	rs1[confs1]	001	imm[4 :0]	1110111	`dome.store`
offset[6 :0]	rs2	rs1[confs1]	010	rd	1110111	`dome.set`
offset[6 :0]	rs2	rs1[confs1]	011	rd	1110111	`dome.clear`
0000000	rs2[confs2]	rs1[confs1]	100	rd	1110111	`dome.mv`
offset[6 :0]	rs2	rs1[confs1]	101	rd	1110111	`dome.cmv`
offset[6 :0]	00000	rs1[confs1]	110	rd	1110111	`dome.imv`
0000001	00000	rs1[confs1]	111	rd	1110111	`dome.check.v`
0000100	00000	rs1[confs1]	111	rd	1110111	`dome.check.u`
0000011	00000	rs1[confs1]	111	rd	1110111	`dome.check.l`
0000000	00000	rs1[confs1]	111	rd	1110111	`dome.check.c`
0000001	00000	rs1[confs1]	000	rd	1111011	`dome.switch.v`
0000011	00000	rs1[confs1]	000	rd	1111011	`dome.switch.l`
0000000	00000	rs1[confs1]	000	rd	1111011	`dome.switch.c`

Cette Table rassemble toutes les nouvelles instructions pour le support des domes. L'encodage de chacune d'entre elles (sur 32 bits) est également détaillé. Elles sont basées sur les formats d'instruction R et S de l'architecture RISC-V. Pour rappel :

- `imm` correspond à une valeur immédiate.
- `rs` correspond à un registre d'entier source.
- `rd` correspond à un registre d'entier destination.
- `rs[confs]` correspond à un registre d'entier source contenant le numéro de la configuration accédée.
- `offset` permet de sélectionner le champ d'une des configurations.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Garantir l'isolation microarchitecturale des processeurs

Mot clés : Sécurité des processeurs, microarchitecture, ISA, attaques par variations temporelles, isolation microarchitecturale

Résumé : Les processeurs sont des composants électroniques omniprésents dans notre quotidien. On en retrouve dans nos téléphones, nos ordinateurs, les serveurs internet *etc.* Ils sont notamment responsables de l'exécution des calculs et ont donc accès aux données du système. Depuis de nombreuses années, des travaux ont montré des problèmes d'isolation entre les utilisateurs d'un même processeur. En exécutant du code, il devient possible pour un attaquant d'influer sur l'exécution ou de retrouver des informations d'un autre utilisateur. Cette menace s'est amplifiée jusqu'à la découverte en 2018 de Spectre et Meltdown, des attaques complexes et affectant une grande majorité des processeurs modernes.

Cette thèse vise à repenser la conception des processeurs pour mettre en place efficacement de nouvelles contraintes de sécurité. Elle s'intéresse plus particulièrement à la mise en place de protections contre les attaques par variations temporelles. Le premier axe exploré est le rôle du jeu d'instructions. Initialement, celui-ci a principalement un rôle fonctionnel : il définit les instructions utilisables par le logiciel et réalisées par le matériel. Il représente l'interface entre ces deux mondes. En le modifiant, il devient donc possible pour le logiciel d'indiquer ses contraintes de sécurité au matériel qui doit alors s'adapter. La stratégie adoptée dans cette thèse est la contextualisation, qui permet d'associer temporellement et spatialement chaque information à un domaine de sécurité.

Le second axe exploré est la modification de la microarchitecture du processeur elle-même. À partir des informations reçues, le matériel doit être capable de respecter les contraintes de sécurité. Des principes de conception génériques ont donc été définis pour modifier les mécanismes internes du processeur, et notamment la gestion des ressources partagées. Ils s'articulent autour de trois méthodes : allocation statique des ressources, partitionnement et effacement des traces.

Pour évaluer ces principes, deux processeurs ont été modifiés. Ils implémentent différents mécanismes susceptibles de générer des fuites d'informations. Les résultats montrent que cette approche est efficace pour isoler des exécutions différentes sur un processeur. Dans le cas du partage simultané du processeur, on observe un compromis au niveau des implémentations possibles entre les performances visées et le coût des mécanismes utilisés. La suite de ces travaux concerne l'étude de ce compromis au niveau matériel, mais aussi l'adaptation du logiciel à ce nouveau jeu d'instructions.

Title: Ensuring microarchitectural isolation in processors

Keywords: Processor security, microarchitecture, ISA, timing attacks, microarchitectural isolation

Abstract: Processors are electronic components omnipresent in our daily lives. They are in our smartphones, computers, web servers *etc.* Processors are responsible for executing the different operations and then have access to system data. For many years, studies have shown the isolation issue between users of the same processor. By executing code, it becomes possible for an attacker to influence the execution or recover information from another user. This threat has grown until the discovery in 2018 of Spectre and Meltdown, complex attacks targeting the most modern processors.

This thesis aims to rethink processor designs to efficiently implement security constraints. It particularly focuses on the implementation of protections against timing attacks. The first explored axis is the role of the instruction set. Initially, it mainly has a functional role: it defines the instructions usable by the software and implemented by the hardware. It represents the interface between these two worlds. Its modification can allow a communication between hardware and software about security. The strategy developed in this thesis is the use of contextualization, allowing each piece of information to be associated temporally and spatially to a security domain.

The second axis is the modification of the processor microarchitecture itself. Based on the information received, the hardware must be able to respect the security constraints. Generic design principles have been defined to modify the processor's internal mechanisms, and particularly shared resources. They are based on three methods: static allocation, partitioning and flush.

To evaluate these principles, two processors are modified. They implement different mechanisms that can generate information leakage. The results show that this approach is efficient to isolate different executions on the same processor. For the simultaneous sharing of the processor, a trade-off must be considered between performances and the cost of the implemented mechanisms. Future works should aim at studying this compromise at the hardware level, but also the adaptation of the software to this new instruction set.