Secure Processors with respect to Micro Architectural Attacks

Valentin Martinoli

▶ **To cite this version:**

Valentin Martinoli. Secure Processors with respect to Micro Architectural Attacks. Micro and nanotechnologies/Microelectronics. Université Grenoble Alpes [2020-..], 2023. English. NNT : 2023GRALT024 . tel-04145576

HAL Id: tel-04145576

https://theses.hal.science/tel-04145576

Submitted on 29 Jun 2023

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.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : EEATS - Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)
Spécialité : Nano électronique et Nano technologies
Unité de recherche : Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés

Processeurs Sécurisés et Attaques de Micro Architecture

Secured Processors with respect to Micro Architectural Attacks

Présentée par :

Valentin MARTINOLI

Direction de thèse :

Régis LEVEUGLE — Directeur de thèse
Professeur des universités, Grenoble INP, Université Grenoble Alpes

Yannick TEGLIA — Co-encadrant de thèse
Ingénieur Docteur, Thales DIS

Rapporteurs :

Guy GOGNIAT
Professeur des universités, Université Bretagne Sud

Jean-Max DUTERTRE
Professeur IMT, École Nationale Supérieure des Mines de Saint-Etienne

Thèse soutenue publiquement le 23 mars 2023 devant le jury composé de :

Régis LEVEUGLE — Directeur de thèse
Professeur des universités, Grenoble INP, Université Grenoble Alpes

Guy GOGNIAT — Rapporteur
Professeur des universités, Université Bretagne Sud

Jean-Max DUTERTRE — Rapporteur
Professeur IMT, École Nationale Supérieure des Mines de Saint-Etienne

Lorena ANGHEL — **Présidente** / Examinatrice
Professeure des universités, Grenoble INP, Université Grenoble Alpes

Sébastien PILLEMENT — Examinateur
Professeur des universités, Nantes Université

Invités :

Yannick TEGLIA
Ingénieur docteur, Thales DIS

Paolo MAISTRI
Chargé de recherche, CNRS

Acknowledgements

Before any further technical development, I would like to start this manuscript by thanking those who guided me, supported me, and taught me a lot during this thesis.

I would like to express my deepest gratitude to my professor and thesis director Régis Leveugle for his invaluable patience and expertise. This endeavor would not have been possible without my thesis co-director Yannick Teglia, who provided intangible support and guidance. I am also extremely grateful to Guy Gogniat and Jean-Max Dutertre who agreed to examine this thesis manuscript. I would like to express my deepest appreciation to my defense committee Lorena Anghel and Sébastien Pillement. Words cannot express my gratitude to Thales DIS and TIMA laboratory for their tangible support. I am deeply indebted to the Université Grenoble Alpes for making this thesis possible.

I am also grateful to Samuel Gröger, Abdellah Bouagoun and Elouan Tourneur who worked in collaboration with me and provided valuable support towards enriching the results of this work. I would like to extend my sincere thanks to my colleague Joseph Gravellier for his advices and his support. Many thanks to Philippe Loubet-Moundi, manager of the Hardware-Lab Team for his feedbacks and suggestions. I would like to acknowledge Jean-Roch Coulon and André Sintzoff for their valuable knowledge and support. I had the pleasure of working with the Hardware Lab Team members who provided me an intangible support throughout my thesis.

Table of contents

Acknowledgements III
Table of contents V
Table of Figures IX
Table of Tables XI
Table of Listings XII
Table of Appendices XIII
Résumé étendu XV
Introduction 25
Chapter 2. Background and State of the Art 29
2.1. Micro architectural mechanisms involved in the presented attacks 29
2.1.1. Architectural and micro architectural worlds 29
2.1.2. Cache memories 30
2.1.3. Speculative execution 33
2.2. Micro architectural covert-channels 36
2.2.1. Prime+Probe 37
2.2.2. Flush+Reload 38
2.2.3. Flush+Flush 39
2.2.4. Evict+Time 39
2.2.5. Prime+Abort 40
2.2.6. Evict+Reload 40
2.2.7. The importance of the encoding technique 41
2.3. Main micro architectural end-to-end attacks 44
2.3.1. Transient-execution attacks 44
2.3.2. Microarchitectural Data Sampling attacks 52
2.4. Derived micro architectural attacks 56
2.5. Classical timing attacks 57
2.6. Software-based Hardware attacks 58
2.7. Existing mitigations 59
2.7.1. User-space mitigations 60
2.7.2. Hardware mitigations 61
2.7.3. System-level mitigations 63
2.8. Discussion and conclusion 66

Chapter 3. The case of Intel SGX: the discovery of critical leakages in all high-end CPUs 67
3.1. Overview of the SGX technology 67
3.2. A first case-study: Applying Foreshadow to a SGX secure enclave 69
3.3. Initial software mitigation ideas and robustness evaluation of the existing countermeasures 71
3.3.1. Intel proposed mitigations 71
3.3.2. Adding random temporal loops 72
3.3.3. Flushing the cache 73
3.3.4. TSX transactions 74
3.4. Putting the mitigations to the test on a real use-case 75
3.5. Conclusions and lessons learned 77
Chapter 4. Micro architectural vulnerability study of the CVA6 RISC-V core 78
4.1. The RISC-V ecosystem 78
4.2. Motivation for CVA6 80
4.3. Overview of the CVA6 80
4.4. Threat Model and Best attack path candidate 81
4.5. In-depth micro architectural study of the CVA6 data cache 84
4.5.1. Dimensioning and behavior 84
4.5.2. How addresses can be decomposed to represent the CVA6's data cache 85
4.5.3. CVA6's writing/eviction policies 86
4.5.4. The data cache memory's structure 88
4.5.5. Storing atomic operations in the AMO buffer 91
4.5.6. Introduction to the MSHR 91
4.5.7. Handling misses and more with the miss unit 92
4.5.8. The write buffer's different usages 93
4.6. Conclusion and outcomes of the study 94
Chapter 5. A first simulated baremetal Proof-of-Concept of the Prime+Probe covert-channel on a CVA6 core 95
5.1. Motivation and objectives of the study 95
5.2. Threat model 96
5.3. Initial version: noiseless and unscheduled 97
5.3.1. Code's structure and chosen encoding technique 97
5.3.2. Data extraction and experimental results 100
5.4. Towards the addition of an OS 103
5.4.1. Impact of a scheduler 103
5.4.2. Noise generation and effects on the attack 105

5.4.3. Combination of scheduling and noise generation 106

5.5. Discussion and contributions 108

Chapter 6. A more realistic use-case: targeting cryptographic implementations with a running Linux OS on an FPGA board 110

6.1. Motivation and objectives of the study 110

6.2. Experimental setup and tools 111

6.3. First experimentations on information transmission and statistical analysis: defining an encoding technique 111

6.4. First version of the attack on a simple victim 115

6.5. AES case-study 118

6.5.1. Implementation of the Prime+Probe covert-channel 119

6.5.2. Results obtained and limitations 121

6.5.3. One more step towards realism: considering a multi-user platform with a noisy environment 123

6.5.4. Summary and conclusions on the AES case study 125

6.6. RSA case-study 125

6.6.1. Implementation similarities and differences with the AES 125

6.6.2. Discussion and conclusions on the RSA 129

6.7. Discussion and limitations 130

Chapter 7. Generalization of the proposed mitigations towards secure cores with respect to micro architectural attacks 132

7.1. Towards the development of mitigations: study the favorable and unfavorable conditions for an attacker 132

7.2. Our proposed micro architectural modifications for a secure handling of sensitive data 137

7.3. Discussion and limitations 142

Conclusion and future works 143

8.1. Summary of the thesis 143

8.2. Future work 145

8.2.1. Research perspectives 145

8.2.2. Industrial perspectives 146

Publications, communications and other contributions 147

References 149

Appendices 166

Résumé 171

Abstract 172

Table of Figures

Figure 1. Representation of the three existing cache organizations and example of placement possibilities for each one ... 31
Figure 2. Schematic representation of the difference between in-order and out-of-order execution ... 34
Figure 3. Typical phases in a transient-execution micro architectural attack ... 45
Figure 4. The different steps of the Foreshadow Attack[1] ... 49
Figure 5. Architectural view of the different elements exploited by the presented vulnerabilities[2] ... 52
Figure 6. Diagram of a typical secure remote computation setup with SGX (yellow) ... 68
Figure 7. SGX-Step single-stepping mechanism for SGX enclaves[3] ... 70
Figure 8. CVA6's Pipeline ... 81
Figure 9. Representation of the data cache in the CVA6 core ... 82
Figure 10. Chosen threat model for the micro architectural covert-channel ... 84
Figure 11. Decomposition of the example address 0x000000008000b010 ... 85
Figure 12. Localization of the Data cache Memory inside the data cache's structure of the CVA6 core ... 88
Figure 13. Implementation of the Data array in the L1 Data cache of CVA6 ... 89
Figure 14. Implementation of the Tag array in the L1 Data cache of CVA6 ... 89
Figure 15. Selection of the tags corresponding to the second set (e.g., index equal to one) . 90
Figure 16. Request of a 64-bit memory block with the address 0x0000008000b010 from the Data cache of the CVA6 core ... 90
Figure 17. Miss unit's Finite-state Machine ... 93
Figure 18. Chosen threat model for the baremetal attack's context ... 97
Figure 19. Representation of the evictions caused by the implemented Trojan's activity inside the CVA6's data cache ... 99
Figure 20. Representation of the simulation workflow ... 101
Figure 21. Representation of the perturbation generator based on an AES algorithm ... 105
Figure 22. Evolution of the success rate (in %) of the two considered encoding solutions compared to the number of evictions caused by the additional process ... 106
Figure 23. Overview of the different cache set indexes where cache misses have been obtained in a "semi-realistic" use-case ... 107
Figure 24. Representation of the state of the CVA6 data cache when the Trojan encodes the secret value ... 114
Figure 25. Experimental results for a simple addition victim ... 117
Figure 26. Zoom on one of the 8 patterns obtained when applying the covert-channel on a simple addition victim ... 118
Figure 27. Trace obtained targeting the "Tiny-AES" with a 128-bit key and using 2000 samples ... 122
Figure 28. Trace obtained when zooming on one of the 8 patterns ... 122
Figure 29. Comparison of the traces obtained in a standard and noisy setup ... 124
Figure 30. Comparison of the zoom on one pattern for a standard and noisy setup ... 124

Figure 31. Synchronization process for the covert-channel when targeting an AES victim in a TEE environment 126
Figure 32. Synchronization process for the covert-channel when targeting an RSA victim .. 127
Figure 33. Illustration of the synchronization issues on an RSA victim 128
Figure 34. Illustration of the synchronization with Prime+Probe timing shorter than the handling of a key bit equal to zero 129
Figure 35. Evolution of the number of samples required to extract a 128-bit AES key varying with the cache size 130
Figure 36. Evolution of the extraction rate with the cache associativity for a 32kB data cache 136
Figure 37. Schematic representation of the micro architectural memory elements of a typical CPU 137
Figure 38. Schematic representation of potential micro architectural information leakages after a regular store operation 138
Figure 39. Schematic representation of a CPU with the proposed micro architectural modification 139
Figure 40. Schematic representation of the STORE_SEC instruction on CPU using the proposed micro architectural modification 140
Figure 41. Schematic representation of the alternative implementation where a dedicated additional secured memory is added to the CPU 141

Table of Tables

Table 1. Summary of the main micro architectural cache covert-channels and their specificities 43
Table 4. Table representing the filling of the data cache when allocating an array of 2048 cells 86
Table 5. Table representing the different states of the bytes contained in the Write Buffer 94

Table of Listings

Listing 1. Example of Spectre attack leveraging conditional branch 45
Listing 2. Structure of a TSX transaction 74
Listing 3. Artificial scheduling pseudocode 98
Listing 4. Pseudocode of an example of the simply scheduled attack.................................. 104
Listing 5. Pseudocode for our Prime+Probe micro architectural covert-channel 113
Listing 6. Pseudocode for the covert-channel targeting the “Tiny-AES” implementation 120
Listing 7. Pseudocode of the randomized skeleton of a scheduler 134

Table of Appendices

Appendix 1. Main characteristics of the existing micro architectural attacks and comparison with related attacks........ 166
Appendix 2. Overview of the existing micro architectural mitigations and their main characteristics 167

Résumé étendu

I. Introduction et contexte

Usuellement, les attaques matérielles sont considérées comme nécessitant un accès physique à la victime. Néanmoins, une classe intermédiaire d'attaques dénommées attaques matérielles à distance est apparue récemment. Ces attaques permettent l'exploitation de failles matérielles via l'utilisation de code, injecté par exemple au travers d'un réseau. Elles ont gagné en importance dans la mesure où les technologies cloud, terreau propice à ce type d'attaques, ne cessent d'évoluer et de se perfectionner. Les attaques matérielles à distance comportent plusieurs sous-catégories, dont les menaces dites micro architecturales. Des publications récentes ont démontré que les ressources matérielles partagées, telles que les caches ou les mémoires tampons internes, peuvent permettre à un attaquant de franchir les limites de sécurité introduites par l'OS ou l'hyperviseur. La première cible de ces attaques dans la littérature a été la technologie SGX d'Intel. L'attaque micro architecturale nommée Foreshadow a permis de franchir l'étanche isolation logicielle introduite par SGX via la création de coffre forts logiciels nommés enclaves. Cette attaque a été le début d'une longue lignée de vulnérabilités. Ces attaques sont basées sur les disparités temporelles observées entre le monde micro architectural et le monde architectural. Le monde micro architectural est interne, invisible, et le programmeur ne peut interagir directement avec ses éléments. Il s'agit d'une implantation spécifique d'un jeu d'instruction donné (exploitant les mémoires caches, les mémoires tampons internes…). Le monde architectural quant à lui correspond à l'ensemble des éléments visibles directement par le développeur par le biais d'un débogueur (les registres banalisés par exemple).

Dans la course aux performances des CPUs, les développeurs n'ont cessé d'ajouter des mécanismes d'optimisation tels que l'exécution dans le désordre ou l'exécution spéculative. Le fonctionnement de ces mécanismes repose sur la présence de ressources matérielles partagées. Cependant, la littérature a démontré que ces ressources peuvent être exploitées afin de récupérer des informations sur les processus qui les partagent. Il s'agit ici du principe fondamental des attaques micro architecturales. Ces attaques sont lancées via du code qui, grâce à une succession judicieuse d'instructions, permet de déclencher les différents mécanismes d'optimisation qui font intervenir les ressources matérielles partagées. Il est ensuite possible de récupérer les données d'un autre processus au sein de ces ressources, puis de les extraire afin de les observer. L'étape d'extraction est une attaque par canal auxiliaire (ou canal caché), et permet de faire transiter l'information du monde micro architectural vers le monde architectural où elle pourra être utilisée. Le canal de communication le plus utilisé dans les attaques micro architecturales est la mémoire cache. Le cache est la ressource matérielle partagée la plus évidente et est présent dans la plupart des CPUs actuels.

Habituellement, les éléments micro architecturaux ne sont pas documentés par les constructeurs. Il s'agit de structures internes que les développeurs n'ont normalement pas besoin de connaitre ; elles sont transparentes pour eux. Cependant, des chercheurs ont réussi à les caractériser en construisant des attaques les ciblant. Intel, ARM, AMD, tous les constructeurs sont vulnérables face à ces menaces. Le modèle de sécurité par l'obscurité qui

semblait suffisant n'est plus efficace, même lorsqu'il s'agit de structures profondément enterrées dans le circuit. L'idée à l'origine de ce projet de thèse est donc d'étudier le jeu d'instruction RISC-V comme alternative à Intel, ARM ou AMD en termes de sécurité face aux attaques micro architecturales. Contrairement aux processeurs propriétaires, RISC-V est une initiative ouverte, menant à de nombreuses conceptions de processeurs également ouvertes. Les processeurs RISC-V sont à priori tout aussi vulnérables que les autres face aux attaques micro architecturales et de plus permettent des analyses plus rapides, les codes source de la microarchitecture étant le plus souvent disponibles. Le but ultime de cette étude est d'établir si un modèle de cœur ouvert et résilient, de par son architecture, serait plus efficace que les approches usuelles pour faire face aux attaques micro architecturales. Cette thèse vise à proposer une étude approfondie de l'origine des attaques micro architecturales tout en répliquant certaines d'entre elles sur le cœur RISC-V CVA6. Ces expériences ont pour but de caractériser plus précisément les mécanismes à l'origine des vulnérabilités et d'estimer les avantages et désavantages des approches ouvertes pour un attaquant, ou un développeur tentant de protéger son système.

II. SGX et la découverte des fuites micro architecturales

Avant d'aborder les processeurs RISC-V, la première étude de cette thèse s'est dirigée vers la technologie SGX d'Intel, et la vulnérabilité Foreshadow. En effet, l'objectif de ces travaux est de constituer une première analyse des attaques de type micro architecturales, ainsi que de déterminer leur dangerosité dans des scénarios d'usages réalistes. L'objectif est également de proposer des premières idées de contre-mesures, notamment au niveau logiciel, afin d'évaluer les possibilités d'endiguer les fuites à moindre coût (et donc en évitant les modifications matérielles). La réplication de l'attaque Foreshadow sur une enclave de test s'est tout d'abord avérée relativement aisée, sur un système de la génération disponible au moment de l'apparition de l'attaque, notamment grâce à l'outil SGX-Step qui permet d'exécuter pas-à-pas des enclaves. A l'issue de cette réplication, il a été décidé d'étudier les possibilités de protections (ou contremesures) au niveau logiciel afin de protéger la victime ciblée par l'attaque Foreshadow telle que répliquée, à savoir un algorithme de chiffrement AES.

Face aux menaces, Intel a proposé plusieurs contremesures. Tout d'abord, la désactivation de l'hyperthreading, bien que coûteuse en termes de performances, a permis de diminuer le risque d'exploitation de ces attaques dans un premier temps. Ensuite, Intel a notamment proposé des mises à jour de micro code, permettant entre autres la vidange des caches au moment des changements de contexte. Ces protections étaient efficaces pour contrer les variantes initiales des attaques Foreshadow et Meltdown. Cependant, elles présentaient d'importants manquements face aux attaques plus récentes, et certaines variantes des attaques existantes.

Face à ce constat, plusieurs propositions de protections supplémentaires au niveau logiciel ont été faites et évaluées dans des contextes semi-réalistes. Nos idées initiales de contremesures comprenaient notamment l'addition de boucles temporelles aléatoires, des vidanges du cache, ainsi que l'utilisation de transactions TSX, permettant l'exécution atomique de sections définies de code. Les vidanges de cache sont basées sur l'idée initiale

d'Intel mais visent à exploiter la présence de l'instruction clflush afin de vidanger le cache après chaque manipulation de données sensibles, et donc plus souvent que la proposition du fondeur. La protection à base de boucles temporelles aléatoires est basée sur des protections traditionnellement utilisées face aux attaques par canaux cachés. Enfin, la contremesure basée sur les transactions TSX permet d'exécuter l'intégralité du contenu d'une enclave de manière atomique. Cela devait endiguer les attaques micro architecturales car toute interruption de la victime, telle qu'une faute de page, interrompt le calcul en cours et remet le système dans son état initial au début de la transaction TSX. Les attaques de type Meltdown ou Foreshadow exploitent les fautes de pages pour déclencher l'utilisation de mécanismes d'optimisation et donc de ressources matérielles partagées. La protection à base de transaction TSX devait donc contrer ce type d'attaques.

Ces contremesures, ainsi que celles proposées par Intel, ont été évaluées sur des algorithmes de chiffrement AES et RSA, sur des machines standards de l'entreprise. Ces ordinateurs contenaient les protections proposées par Intel, à l'exception des défenses matérielles contre les attaques de type Forehsadow. Il a été démontré que l'ensemble des contremesures proposées par Intel ainsi que par les travaux de cette thèse sont efficaces pour empêcher l'exploitation de l'attaque Foreshadow initiale. Cependant, d'autres variantes de l'attaque étaient toujours fonctionnelles, et les attaques les plus récentes telles que LVI (Load Value Injection) passaient au travers des défenses proposées. Tous ces résultats démontrent les limitations des architectures fermées, ainsi que celles des contremesures au niveau logiciel. En effet, pour endiguer les attaques micro architecturales, il est nécessaire d'agir directement à la source de la fuite : au niveau micro architectural.

III. Attaque simulée sans système d'exploitation

L'approche « fermée » proposée par les fondeurs traditionnels ayant atteint ses limites, et l'approche logicielle ayant une efficacité mitigée (notamment face aux attaques récentes), il a été décidé de porter l'étude sur l'initiative RISC-V. En effet, cette initiative étant ouverte, il est possible d'y trouver des projets d'architectures également ouvertes. Dans le cadre de l'étude proposée par ce document, le cœur RISC-V retenu, et qui sera la plateforme dédiée pour toutes les expérimentations décrites, est le cœur CVA6. Ce processeur ouvert a été développé par l'ETH Zurich, pour des objectifs de performance en premier lieu. C'est un cœur simple, sans multi-threading. Il dispose néanmoins de la plupart des mécanismes d'optimisation que l'on trouve sur les cœurs récents tels que la prédiction de branchement ou des caches L1 de données et d'instructions. Le but de l'étude sur CVA6 était dans un premier temps de déterminer son niveau de vulnérabilité face aux attaques micro architecturales.

Il a donc été nécessaire de procéder à une analyse détaillée de la micro-architecture du processeur. Le code du cœur est disponible sur GitHub et est majoritairement écrit en System Verilog. Il a d'abord été nécessaire de recenser les différents éléments qui étaient susceptibles de présenter des fuites de types micro architecturales. Plusieurs candidats paraissaient prometteurs, tels que les mémoires caches, les mémoires tampons utiles à la spéculation, et le scoreboard (structure qui réordonnance les instructions avant finalisation de leur exécution au niveau architectural). D'après la littérature et les résultats de cette

analyse, il a été décidé de se concentrer sur le cache de données. En effet, cette structure avait déjà fait l'objet d'une publication proposant des résultats préliminaires et des bases pour une étude plus approfondie. N. Wistoff et al., de l'ETH Zurich, avaient en effet étudié les possibilités de fuites d'informations au sein du cache de données du CVA6 en utilisant une bibliothèque de vulnérabilités nommée Mastik. Il n'existait toutefois pas à notre connaissance de description fonctionnelle et pratique concernant l'architecture du cache de données du CVA6. Même les documents récemment mis à disposition sur le site du groupe de travail ne détaillent pas cette partie. Une analyse approfondie a donc été réalisée et le résultat de ces efforts de compréhension a été mis sous la forme d'une documentation et partagé avec la communauté.

Pour poursuivre notre étude micro architecturale, il a été décidé de procéder à la réplication de l'attaque par canal caché la plus emblématique sur les caches de donnée, l'attaque Prime+Probe. L'un des avantages des approches ouvertes est la libre disposition du code source des processeurs. Il a donc été choisi d'exploiter cette différence en utilisant des outils de simulation afin de répliquer l'attaque. La chaîne de simulation mise à disposition par l'ETH Zurich sur Github a été utilisée. Le simulateur est le logiciel ouvert Verilator, qui est un simulateur précis au cycle d'horloge près. L'attaque Prime+Probe, bien connue dans la littérature, a donc été reproduite en simulation, sans système d'exploitation, sur le processeur CVA6. L'attaque implémentée est capable de récupérer des informations allant en théorie jusqu'à 256 bits en les faisant transiter par le cache de données. Pour des valeurs secrètes exprimées avec 0 à 50 bits, notre implémentation de l'attaque possède un taux de succès de 90%. Ce dernier diminue à 85% pour des valeurs nécessitant entre 51 et 130 bits, pour finir aux alentours de 70% pour un codage sur 131 à 195 bits. Ces valeurs dépendent fortement du scenario matériel et logiciel choisi. Par ailleurs, contrairement à la théorie, il n'a pas été possible d'extraire des valeurs nécessitant plus de 195 bits. En effet, nous avons été confrontés à ce que nous avons nommé la « zone morte ». Cette zone du cache est constamment balayée par le processeur lui-même, et contient des informations telles que des résultats de calculs d'adresses, des adresses intermédiaires, etc. … Il en résulte que cette zone, comprise entre les sets du cache numéro 196 et 226 (dépendant du scénario logiciel/matériel) n'est pas utilisable pour stocker puis récupérer de l'information dans le cadre d'une attaque micro architecturale. En effet, toute information utile qui serait placée à cet emplacement se verrait expulsée du cache quelques cycles d'horloge plus tard, et serait donc inutilisable.

Cette étude sur environnement de simulation sans système d'exploitation a été conclue par certains enrichissements visant à prédire le comportement d'une telle attaque dans le cas de l'ajout d'un système d'exploitation. Pour ce faire, il a été décidé de simuler un système d'exploitation rudimentaire chargé d'un ordonnancement, codé par nos soins. En plus de cela, plusieurs autres processus ont été ajoutés afin de simuler l'activité générée par la présence de processus multiples dans le cadre d'un système d'exploitation complet. Ces processus supplémentaires sont des générateurs aléatoires, basés sur des algorithmes de chiffrement AES. Leur objectif est de produire des expulsions aléatoires au sein du cache de données, afin de perturber autant que possible le processus de l'attaque. Il résulte de cette expérimentation que l'ajout d'un squelette de système d'exploitation pose des problèmes temporels pour l'attaque Prime+Probe. En effet, il est nécessaire que les différentes phases de l'attaque soient suffisamment rapides (i.e., plus rapides que le temps alloué par le

système à chaque processus) afin qu'elles ne soient pas interrompues en plein déroulement, et ainsi perturbées trop fortement pour rester opérationnelles. De plus, l'ajout de processus supplémentaires en parallèle produit comme prévu des remplacements aléatoires dans le cache. Ces derniers compliquent la tâche de l'attaquant qui doit alors procéder à une analyse plus poussée des résultats de l'attaque afin de discriminer l'information utile du bruit ajouté par les perturbateurs. Pour contrer ces perturbations, il est possible de simplifier l'encodage, ou la manière de cacher l'information à l'intérieur du cache de données.

Toute cette étude a permis de mettre en lumière l'avantage que représentent les initiatives ouvertes pour les attaquants. En effet, disposer librement du code permet de faciliter et d'approfondir la phase de compréhension et d'analyse comportementale de l'architecture ciblée. De plus, cela permet également de recourir aux outils de simulation qui, comme démontré, permettent de développer des attaques et de les rejouer avec différentes variantes, sans avoir besoin d'un accès physique à la cible réelle. Toutefois, un effort considérable de compréhension a été nécessaire, dû au manque de documentation de la partie de la microarchitecture exploitée par l'attaque. Rejouer une attaque connue n'est donc pas si simple, même dans le contexte d'une initiative ouverte.

IV. Attaque sur Linux : vers plus de réalisme

La suite des travaux s'est orientée vers la réplication de la même attaque, Prime+Probe, mais sur un support et accompagnée d'un scénario tous deux plus réalistes. L'objectif est ici de se rapprocher d'un cas d'usage réel du cœur, afin de prouver que l'attaque est dangereuse en pratique. De plus, les éléments méthodologiques sont décrits afin de faciliter la compréhension des différents défis que représente l'implémentation d'une attaque connue sur une nouvelle plateforme et dans un nouveau contexte. Enfin, ce travail permet également une fois abouti d'étudier les possibilités de contremesures lorsqu'un système d'exploitation est présent, et sur une victime plus réaliste. Le cœur CVA6 a donc été instancié sur une carte FPGA Genesys II. Un système d'exploitation Linux a été installé sur le cœur CVA6.

La difficulté principale a dans un premier temps été de définir la méthode d'encodage de l'information. En effet, chaque cache possède une structure ainsi qu'un comportement différents, et nécessite donc la prise en compte de ses caractéristiques pour adapter l'attaque. Ici, le cache de données possède une politique d'éviction pseudo-aléatoire, rendant le travail sur les lignes de caches complexe car les remplacements sont difficiles à prévoir. Il a donc été décidé de travailler sur les sets (ensembles cohérents de lignes de cache). L'idée générale de la méthode employée est de faire correspondre à chacun des sets une valeur binaire. Pour cela, la victime est supposée couplée à un cheval de Troie (Trojan) qui a été inséré par un utilisateur malveillant sans avoir été détecté. Ce Trojan cause des évictions dans les sets du cache d'une manière cohérente au secret à récupérer. Si un bit de la valeur secrète à récupérer est égal à 1, alors le Trojan va causer des évictions à l'intérieur du set correspondant à la position du bit secret à récupérer. Si le bit considéré et valant 1 est le 3ème, alors le Trojan causera des évictions au sein du 3ème set du cache. Si le 4ème bit du secret vaut 0, alors le Trojan ne causera pas d'évictions supplémentaires sur le 4ème set du

cache. Ainsi, il est possible de relier la quantité de contention au sein des sets du cache à la valeur du secret à récupérer. Ici, nous cherchons à déterminer la valeur de la clé secrète de la victime AES en employant une attaque par canal caché de type Prime+Probe. En utilisant l'encodage décrit ci-dessus, il est possible de reconstruire le secret en observant à posteriori la contention sur chacun des sets du cache. Si un set contient beaucoup de données qui ont été remplacées pendant l'attaque, dans ce cas le cheval de Troie cherche à transmettre un bit égal à 1. Sinon, il s'agit d'un 0. On obtient en sortie des traces dont l'allure est illustrée en figure 1 avec un seul processus exécuté correspondant à une victime simple.

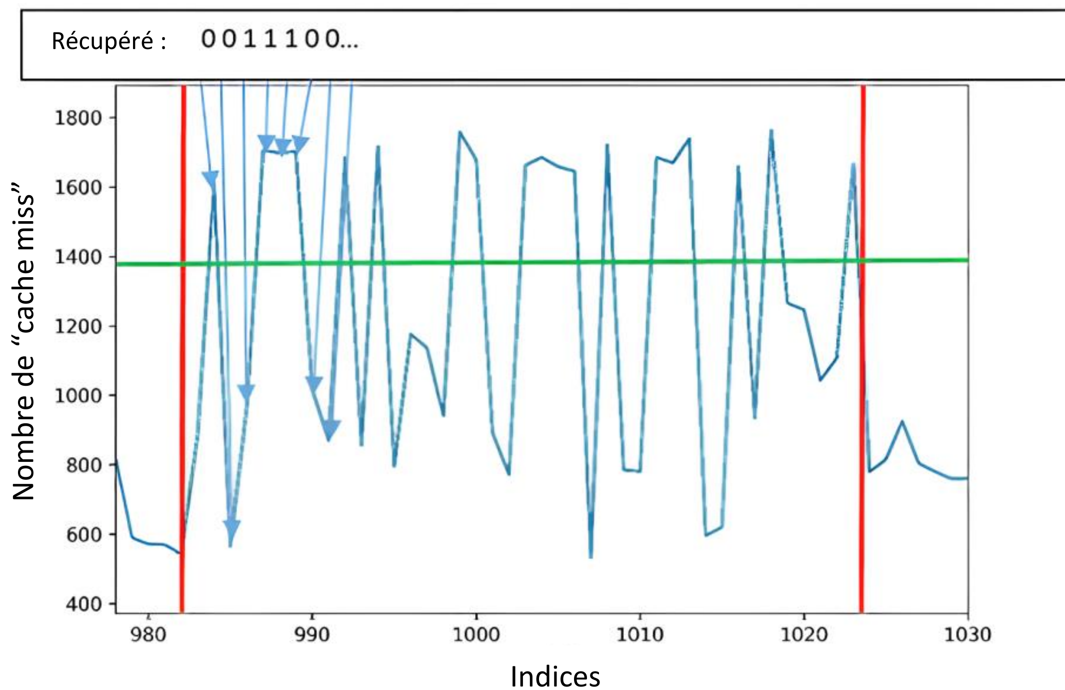


Figure 1. Motif de trace obtenu en résultat de l'attaque Prime+Probe sur une victime simple basée sur une addition dont le résultat constitue le secret.

Sur la figure 1, 2000 échantillons ont été collectés. Les barres rouges délimitent la trace utile, tandis que la verte représente la valeur seuil permettant de discriminer les bits valant 1 de ceux valant 0. Cette expérience a été réalisée à titre de vérification de l'encodage proposé, et permet ainsi d'obtenir des courbes en facilitant la visualisation.

La technique d'encodage proposée ayant été validée par une preuve de concept sur une victime simple de type addition, elle a été appliquée sur une victime plus complexe, un AES. Les traces obtenues lors de l'attaque d'un algorithme de chiffrement AES sont similaires à celles présentées en figure 1, et sont visibles sur la figure 2.

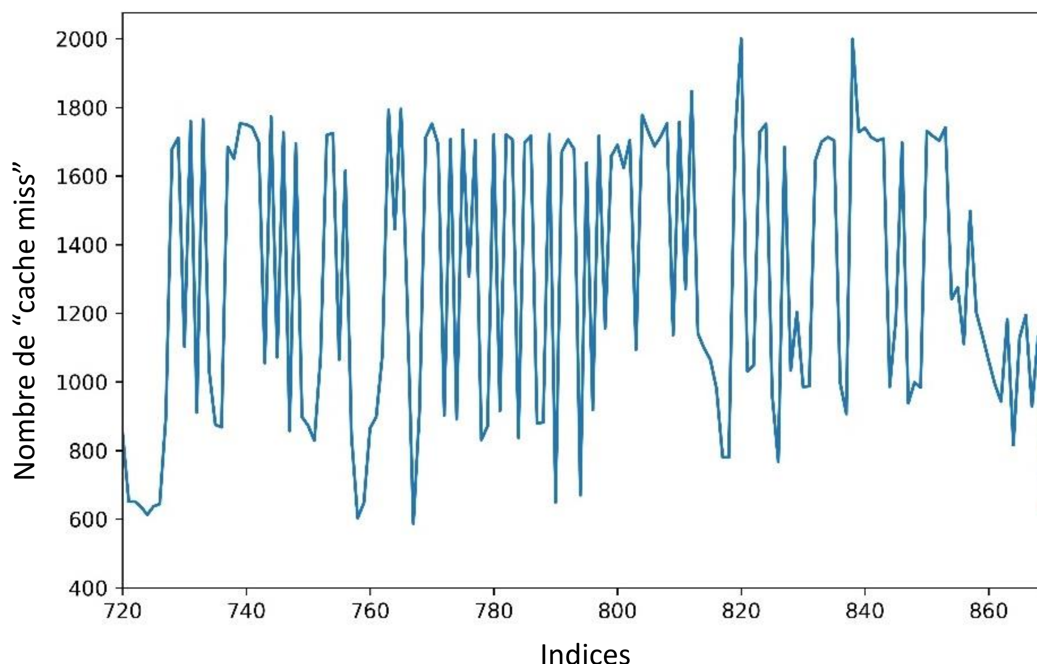


Figure 2. Trace obtenue lors de l'application de l'attaque Prime+Probe sur une victime AES avec 2000 échantillons.

Les résultats de l'implémentation sont satisfaisants ; l'attaque proposée permet de récupérer l'intégralité des 128 bits de clé de l'AES en moins de deux secondes avec un taux d'extraction de 97,6% pour 2000 échantillons. Cela signifie qu'en moyenne 2,4% des bits récupérés sont erronés. De plus, il n'est pas possible d'identifier la position des bits faussés. Cependant, il existe des techniques d'attaques cryptographiques qui permettent de déduire les bits restant d'une clé à partir de bits aléatoires. Il est donc tout à fait envisageable de recourir à ces techniques afin de récupérer les bits restants de la clé. De plus, lorsque l'on considère un nombre d'échantillons supérieur à 3500, on obtient un taux d'extraction égal à 99%, suffisamment élevé pour considérer que l'on récupère l'intégralité de la clé correctement à chaque itération.

De manière analogue à l'étude précédente sans système d'exploitation, des ajouts visant à augmenter le réalisme du scénario ont été proposés. En effet, en parallèle du processus attaquant et du processus victime, ont été implémentés des clients bénins, qui réalisent des chiffrements AES en boucle afin de perturber le processus de l'attaque. Il en résulte que le taux d'extraction de l'attaque diminue. Par exemple, pour 2 clients bénins, le taux d'extraction diminue à 94% pour 2000 échantillons. D'une manière générale, pour contrebalancer la présence de bruit, il est nécessaire de recourir à l'utilisation de plus d'échantillons. Là où l'attaque dans un environnement non bruité pouvait récupérer 95% de la clé en 6 échantillons, il en faut désormais 100 avec deux clients supplémentaires pour conserver le taux de succès à 95%.

V. Développement et propositions de protections

Les derniers travaux réalisés dans le cadre de la thèse présentée dans ce document concernent les efforts de protection dérivés des études réalisées et décrites dans les sections précédentes.

Une première étude a été menée afin d'analyser l’influence de l’architecture d’un cache spécifique sur la capacité d’un attaquant à en extraire des données utiles dans le cadre d’une attaque par canal caché. Il a été établi que la taille du cache et la valeur de son associativité ont une influence non négligeable sur la capacité d’une attaque à extraire des informations. La taille d’un cache conditionne directement la quantité d’information qu’il sera possible de faire transiter au travers de ce dernier à chaque itération de l’attaque. Cependant, plus le cache sera grand, plus il sera difficile pour l’attaquant et la victime de partager des sets du cache, rendant certaines variantes d’attaques par canal caché plus complexes. D’un autre côté, l’associativité contraint également les attaques. En effet, plus elle est élevée, plus il sera probable que d’autres processus en plus de la victime et de l’attaquant partagent un set du cache. Ainsi, les risques que l’attaque soit perturbée seront plus grands. Par ailleurs, une associativité trop petite signifie également que le nombre de sets dans le cache diminue proportionnellement, réduisant ainsi la capacité de l’attaquant à encoder beaucoup d’information à chaque itération de l’attaque. Il existe donc des structures de cache qui sont moins vulnérables aux attaques par canaux cachés du fait de leur disposition. Les valeurs de la taille ainsi que de l’associativité du cache permettent une première estimation grossière du niveau de fuite d’un cache spécifique avec l'encodage utilisé, et donc conséquemment, de l’importance des efforts qu’il sera nécessaire de déployer afin de les colmater.

Ensuite, la contremesure proposée par N. Wistoff et al. a également été étudiée. En effet, ils implémentent une instruction supplémentaire, nommée fence.t, dont le but est de purger le cache et de provoquer un changement de contexte. Lorsqu’employée correctement par le développeur, cette instruction permet de bloquer la plupart des attaques par canaux cachés exploitant les caches. Elle a été utilisée sur l’implémentation d’attaque évaluée précédemment et s’est avérée efficace, empêchant l’attaque de récupérer la clé secrète de l’AES. L’attaque ne récupère plus que des bits égaux à 0 (correspondant à un manque de contention sur les sets du cache) et son taux de succès est donc de 50% car lié au nombre moyen de bits égaux à 0 dans les clés secrètes aléatoirement générées puis extraites.

Enfin, plusieurs contremesures ont été proposées et étudiées. La protection sous forme de boucles temporelles aléatoires a été implémentée et mise à l’épreuve face à la même attaque que précédemment. Il s’avère que la protection est efficace et perturbe grandement l’attaque en modifiant le comportement temporel de la victime de manière aléatoire. Le taux de récupération de l’attaque est également réduit aux alentours de 50%, prouvant son efficacité. Cependant, cette protection n’est pas viable car trop coûteuse : en moyenne, elle double le temps d’exécution de la victime AES. Il serait donc intéressant de procéder à des études supplémentaires afin de raffiner cette proposition. Une idée similaire serait de modifier le comportement temporel du système d’exploitation. En effet, si chaque processus se voyait attribué un temps d’exécution variable, et généré aléatoirement, cela augmenterait grandement la difficulté de procéder à une attaque par canal caché sur le cache. Aussi bien le comportement de la victime, que celui de l’attaquant, seraient modifiés

constamment et de manière aléatoire. Cette proposition a été étudiée dans le cadre de l’attaque simulée sans système d'exploitation, en modifiant le squelette d'ordonnanceur qui avait été implémenté. Ce dernier attribue alors des temps d’exécution variables et aléatoires à chaque processus au lieu d’attribuer un temps fixe et égal. Cela s’est avéré très efficace pour contrer l’attaque dont le taux de succès a été réduit à 51% dans un environnement sans bruit, puis à 10% lors de l’ajout d’un processus perturbateur générant des évictions pseudo-aléatoires à l’aide d’un AES.

La dernière proposition effectuée dans les travaux de thèse présentés dans ce document concerne une modification de la micro architecture elle-même. Cette proposition concerne le cache de données. Afin de contrer les fuites présentes dans les différents éléments micro architecturaux, l’ajout d’un second bus de données vise à contourner le bus usuel. Lors des calculs qui utilisent des données sensibles, ces dernières transiteraient par le bus sécurisé, qui ne fait pas appel aux différentes mémoires micro architecturales partagées. Ainsi, les données sensibles ne peuvent pas être extraites en utilisant des attaques par canaux cachés sur le cache, ou toute autre mémoire tampon au niveau micro architectural. Afin de déclencher l’utilisation de ce bus de données supplémentaire, il a été envisagé d’ajouter deux instructions au jeu d’instruction ; LOAD_SEC et STORE_SEC. Cette contremesure n’a pas été implémentée dans le cadre des travaux réalisés, et sa caractérisation pratique est laissée pour des travaux futurs.

Chapter 1

Introduction

With the fast growth of industries such as embedded systems, Internet of Things, or Cloud Computing, more systems are being connected to networks while cybersecurity is an increasing concern due to the wider range of application domains. To protect these devices, it is necessary to study the different types of attacks they may be confronted with. Historically, attacks have been separated into two main categories: software attacks and hardware attacks. The former require no physical access to the victim device and are leveraged remotely only by running some malicious code on the target [1-3]. The latter often require physical access to the target in order to recover sensitive information or modify the device's behavior [4-6]. Some techniques for hardware attacks enable an attacker to recover information without needing physical contact with the target, but rather being relatively close to it (a few centimeters up to a few meters) by exploiting emissions such as acoustic or electromagnetic signals [7, 8]. Hardware attacks were therefore considered threatening only in use-cases where an attacker could potentially get access to the device, or get close enough to it. However, hardware attacks are not solely limited to attacks requiring physical access to the target. Some techniques that can be carried out remotely have indeed been discovered by P. C. Kocher et al. in 1996 [9]. These attacks lead to considering of a new category of attacks: the remote hardware attacks. These new techniques use software code to exploit hardware leakages remotely, when connections to the networks are available. Since their initial discovery, several types of remote hardware attacks have been disclosed, including micro architectural attacks.

Recent publications have demonstrated that shared hardware resources, such as caches or internal buffers can be leveraged by an attacker to bypass regular security boundaries like memory protection units for instance. Intel's SGX technology [10] has been a recent target of these attacks. The Foreshadow [11] micro architectural attack enables an attacker to bypass the strong software isolation introduced by SGX enclaves. These enclaves can be described as vaults that protect any piece of code or data that is placed within it. No other entity, even with higher privileges like the OS or the hypervisor is capable of accessing the content of an enclave. However, Foreshadow allows bypassing the security barriers and has led to the development of many other micro architectural attacks. These offensive techniques are based on temporal disparities existing between the micro architecture and the architecture of a CPU. The micro architecture is concealed, almost transparent for the programmer. It cannot be interacted with directly. It is the specific hardware implementation of a given instruction set architecture (ISA), typically composed of cache memories, inner buffers, etc. It is opposed to the architecture that can be observed and directly manipulated through a debugger. The architecture contains the standard registers and memories for example.

In the race towards CPU performance, many optimization mechanisms have been added such as out-of-order execution and speculative execution. These mechanisms require the addition of shared hardware mechanisms that have proven to be exploitable by micro architectural attacks. These vulnerabilities can be leveraged by an attacker to recover information about the processes that share these hardware resources. Micro architectural attacks are triggered through software code and do not require a physical access to the target. They take advantage of certain instruction sequences to force the CPU using optimization mechanisms based on shared resources. The attacker can then recover data belonging to another process sharing these hardware memories. The specific action of extracting data is called a covert-channel. Covert-channels transfer the information from the micro architecture (where it cannot be directly retrieved) to the architecture where it can be observed and exploited. Cache memories are the preferred extraction medium, being the largest and most studied shared hardware resources. They are also present in most high-end CPUs.

Traditionally, micro architectural elements are not documented. They are generally transparent for the software developers and thus do not require to be described in details. However, researchers demonstrated that even without documentation, it is possible to characterize these elements and their behavior, while also carrying out an attack targeting them [12]. All the providers are vulnerable to these micro architectural attacks, including Intel, ARM and AMD. This raises the question about the effectiveness of the security model based on obscurity. With the rise of remote hardware attacks, some additional security mechanisms have to be introduced and the development of open cores also raises new questions about the level of risk. The main goal of this thesis work is therefore to study the security level of a CPU core implementing the RISC-V ISA [13], with a focus on micro architectural attacks. RISC-V is an open initiative thus providing the mold for many open CPU architectures. RISC-V cores are as vulnerable as other ones. However, thanks to the availability of their source code, they allow a faster analysis of the micro architectural vulnerabilities. The goal of this work is therefore to study to what extent open architectures are more or less resilient than closed approaches when facing micro architectural threats. This thesis aims at providing an extensive study of the origin of micro architectural attacks while replicating some of them on the CVA6 RISC-V core [14, 15]. These experimentations aim at determining the advantages and disadvantages of open approaches in the case of an attacker, or a developer willing to protect his system.

To reach the proposed objectives, the CVA6 micro architecture has been studied in-depth and the results have paved the way for the implementation works done afterwards. While replicating the attacks, this thesis work proposes a focus on the Prime+Probe cache covert-channel [16] that is one of the most widely used and well-known techniques. This covert-channel has been replicated on the CVA6 core by leveraging one of the advantages of open initiative for an attacker: the possibility to simulate the internal behavior of the core. To study the implications of such a new possibility, the Prime+Probe covert-channel has been implemented on a simulated baremetal version of the CVA6 core. Some mitigation ideas were proposed, and the impact of the addition of mechanisms used by an operating system (OS) has been studied to anticipate the next implementation work. To propose more realistic mitigations and experimental results, the same attack has been developed on an FPGA-instantiated CVA6 with a running Linux OS. This experimentation aimed at providing a more

realistic attack, as well as studying mitigations on a use-case close to real-life. Throughout both implementations, this work also explored the different requirements and considerations to adapt such an attack to a new hardware/software scenario. Moreover, it also studied how specific architectural designs and choices may affect the development of the chosen covert-channel. Finally, once the attacks were functional and characterized in-depth, the work focused more on the mitigation aspects. Several different approaches were studied at different levels (user-mode, system, hardware) and new mitigations were proposed and explored.

This thesis' contributions are the following:

- A state-of-the-art of micro architectural and related attacks and of the existing mitigations to hinder them. Their core principles are explained as to provide a synthetic approach of the global micro architectural panel.
- A demonstration of the limitations of Intel SGX enclave system in real-life scenarios, with the application of micro architectural attacks in realistic use-cases.
- An in-depth study of the CVA6 data cache, its structure, behaviors, and specificities to consider when trying to leverage or mitigate a micro architectural attack on this core.
- The development of a simulated baremetal micro architectural covert-channel on a RISC-V core, enabling infinite replaying possibilities for an attacker. The whole methodology to implement such an attack is detailed and the inner mechanisms are studied in depth.
- The development of a more realistic micro architectural covert-channel on a real-life application use-case. This attack targets a cryptographic algorithm running on an FPGA-emulated RISC-V core. Different attack conceptions are detailed and the focus is set on the methodology and its impact on the resulting implementation.
- A proposal of a novel micro architectural structure leading to a stronger resilience with respect to micro architectural attacks by adding a secure bus to the architecture.

The rest of this document is organized as follows:

Section 2 provides a detailed background on the main micro architectural concepts to understand the rest of the document, as well as a state-of-the-art related to micro architectural and related threats, and the existing mitigations against them.

Section 3 describes the initial study of the micro architectural vulnerabilities on Intel SGX and concludes on the limitations of "closed" architectures for mitigating these leakages.

Section 4 consists in an extensive analysis of the CVA6 RISC-V core, which is the main support for most of the experimentations carried out in this document. This section emphasizes on the description of the data cache of CVA6 and the chosen attack path.

Section 5 focuses on a first simulated baremetal proof-of-concept implementation of a Prime+Probe micro architectural covert-channel on this RISC-V CPU. This section also elaborates on the effect of perturbations on such an attack, and the methodology to reproduce it with a running OS.

Section 6 proposes a more realistic implementation of the Prime+Probe micro architectural covert-channel on an FPGA-emulated CVA6 core running a Linux OS. This implementation is put to the test by targeting the attack of AES-128 and RSA cryptographic algorithms. This section then draws conclusions on the methodology to protect such algorithms against micro architectural attacks, and potential countermeasures.

Section 7 focuses on the protections analyzed or proposed in the work both at the software and hardware levels.

Section 8 provides conclusive remarks on the work described in this document and research perspectives.

Chapter 2

Background and State of the Art

This chapter provides the background knowledge required for understanding the micro architectural information leakages in modern processors. It first provides essential concepts about micro architectural mechanisms that are exploited by the attacks described later in this document. A classification and state-of-the-art on micro architectural covert-channels is then provided. This chapter then details the state-of-the-art and classification on existing micro architectural end-to-end attacks before presenting mitigation techniques. It ends with a general discussion on the conclusions that can be drawn from the existing works in the literature.

2.1. Micro architectural mechanisms involved in the presented attacks

Before being able to describe the different information leakage methods that exist at the micro architectural level, it is mandatory to detail several micro architectural notions that are involved in the process of these attacks. The proposed study mainly focuses on cache memories, the behavior of which will be detailed in the next section. Then, an overview of Speculative and Out-of-Order execution will be given, as they are the two most important micro architectural optimization mechanisms involved in the attacks.

2.1.1. Architectural and micro architectural worlds

Every recent CPU contains two different worlds that constantly coexist. The first one is the architectural world. It consists in every element that can be seen and explicitly used by the software developer (and/or the development tools) who can modify the data processed in these elements. More specifically, the architectural world consists in the registers, the memories, and every element that can be monitored or modified with a debugger.

The second world is called the micro architecture. This second part of the CPU is more hidden, deeper inside the CPU’s construction. The micro architectural world cannot be observed by the developer if the source code of the CPU is not available and it is usually not documented in details by the manufacturer. It is in general not possible to interact with this part of the CPU with a debugger neither to have a deep understanding of its behavior. The micro architectural world is composed of the cache memories, the logical cores, all the internal buffers used for optimization purposes, etc. All of these elements are transparent for the programmer. The micro architecture of a CPU is a specific hardware implementation of computations required to execute the elementary actions defined in a given instruction

set (ISA). Each processor, even based on the same ISA as others, will have a unique micro architecture that will reflect the different technical choices made by the developers.

Architectural and micro architectural views of a CPU may present mismatches. Indeed, the state values and their evolution differ from one world to another. The result of the computation of a given instruction will reside inside micro-architectural elements such as caches and inner buffers of the pipeline before being eventually committed and appear at the architectural level. For this reason, there is an inherent timing discrepancy between the micro architecture and the architecture. Moreover, with some optimization mechanisms detailed below, it is even possible that some content existing inside the micro architecture never gets committed. This content is discarded without having ever existed at the architectural level. The existence of these divergences is the base principle of the micro architectural attacks.

2.1.2. Cache memories

In the race towards performance that exists in CPU designs, memory latency is a main limiting factor. In practice, memory accesses are as slow as several hundreds of CPU clock cycles [17] or even more. One solution to this issue was the introduction of cache memories inside the CPU architectures. Cache memories are smaller and faster memories compared to the main memory. Even if they contain a smaller amount of data, these cache memories can be accessed in a smaller amount of clock cycles. Cache memories can store either data, instructions, or both. Their main objective is to save some data and/or instructions that are most likely to be soon reused so that they can be accessed rapidly. When requesting access to a data or instruction, two possibilities arise: a cache hit or a cache miss. The former consists in requesting information that is already stored in a memory block inside the cache and is therefore accessed rapidly. The latter experiences a slower access time.

Caches have a specific organization. They are divided into smaller individual memory elements called “cache lines”. Each cache line contains several independent elements, either pieces of data or instructions. Some caches are also composed of another subdivision named “sets”. These “cache sets” are containing one or more cache lines. The number of cache lines contained in each cache set varies with the cache organization used, as detailed below. Three cache organizations exist:

1. In a direct-mapped cache, the element to store is placed inside a definite cache line, based on its address, leaving only a single possibility. In this organization, there are as many cache sets as cache lines. Each set is therefore composed of a single cache line.

2. For a fully-associative cache, the element selected can be placed in any cache line. There are no specific restrictions, unlike for the direct-mapped cache. In this organization, there is only a single cache set. This set contains all of the cache lines and represents the whole cache.

3. The last cache organization possible is the set-associative cache. In practice, it is a mix of the two previous organizations. When an element to store inside the cache is chosen, its address defines which cache set will be used, similarly to the direct-mapped cache. More specifically, a part of the address is used to calculate an index that refers directly to a given cache set inside the cache organization. Once the index

and thus a cache set are selected, there are still several cache lines to choose from (as many as the cache's associativity). For the cache line, the choice is made by cache policies. The selected element can therefore be stored in any cache line of the selected set. To remember which line has been chosen, a part of the address of the element to store is used as a tag. This tag can be composed of a variable amount of bits (depending on the cache architecture and technical choices) and is stored in a specific array structure, named the tag array. This tag array enables to make a correspondence between an address and the corresponding cache line where it has been stored.

Figure 1 represents each cache organization that has been presented, and provides a simple example on how a given element is placed within each organization.

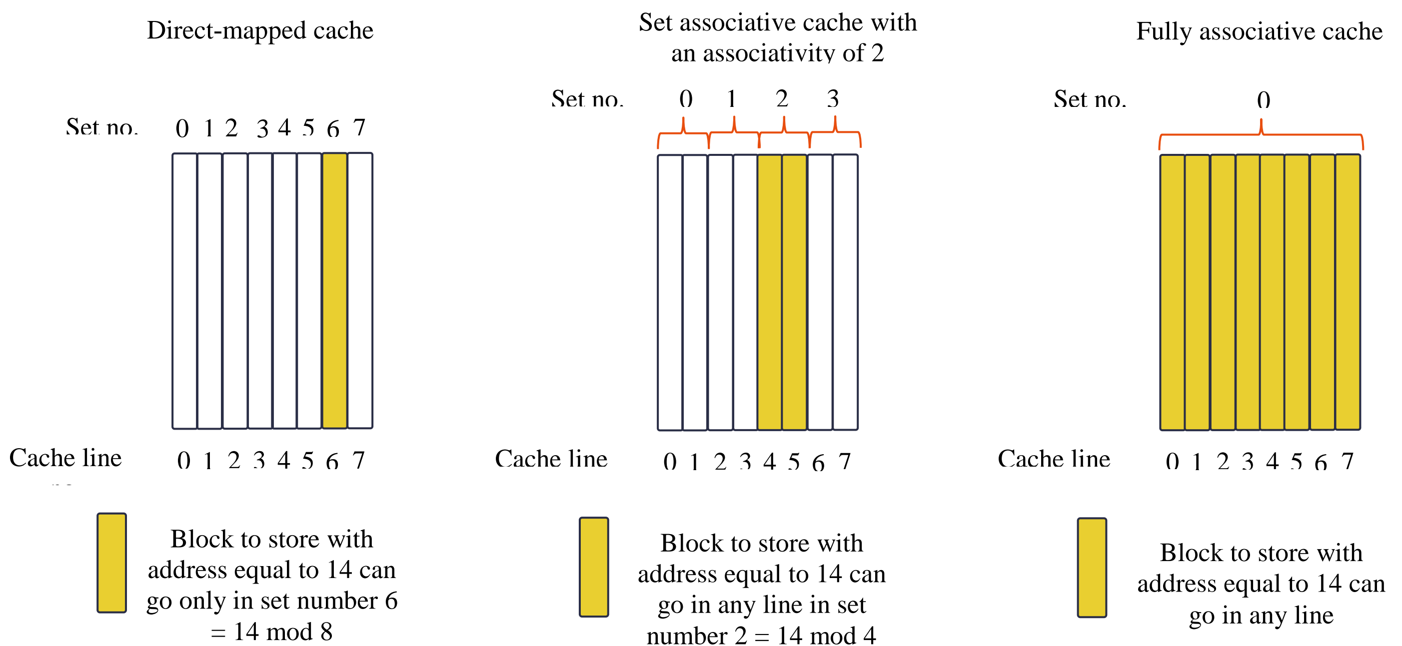


Figure 1. Representation of the three existing cache organizations and example of placement possibilities for each one

As cache memories have a limited size, it is required to have an eviction policy. This policy determines which cache line to remove when an element needs to be stored in a filled cache. For eviction policies, there are many possible solutions. This type of policy is not needed in a direct-mapped cache and therefore only applies to fully-associative and set-associative organizations. The main policies used to choose which cache line to evict are the following:

- Least Recently Used (LRU) evicts the cache line that has been used the least recently. This implies that each cache line contains a supplementary field to store a value reflecting the duration since its last use. This value is incremented for every cache line when there is a cache access, and it is reset to zero for the cache line that is being accessed. Therefore, the cache line with the highest value will be selected for eviction, as it is the least recently accessed one. This policy can be costly especially when there are many cache lines. Indeed, there has to be one counter per line. For this reason, some Pseudo Least-Recently-Used policies tend to simplify the complexity of the LRU policy while having a similar behavior. The main goal of these

policies is to maintain an approximate measurement of each cache line’s age rather than their exact value. This implies that the value to be stored inside the register is smaller. Thus, the policy is less costly to implement due to the reduced amount of memory required.

- Random policies can also be used for eviction. They rely on a specific software or hardware element to generate randomness and evict cache lines based on it. There are two types of potential generators: pseudo-random and random generators. The former has a low cost, as it does not require adding some dedicated hardware and solely relies on software algorithms. The resulting pseudo-randomness can be predicted by a motivated attacker however. The latter is based on a hardware random number generator and therefore comes at a much higher cost. The resulting randomness is almost impossible to predict.

- First-in-First-Out (FIFO) policy implies that the cache line to be evicted is the one that has been filled the longest time ago. As opposed to the LRU, this policy does consider the accesses made to the cache lines. Analogously to the case of the LRU, a supplementary field is required in every cache line to keep track of the “age” of each cache line.

Moreover, a policy has to be introduced to determine when to update the main memory when updating an element that already exists inside the cache. This is a writing or coherence policy. These policies are only required in the case of data caches (instruction caches do not need them). There are two main possible writing policies:

- The CPU updates the cache and the main memory synchronously when there is a request to modify the content of the accessed memory block. This is a write through policy.

- The CPU only updates the cache; it writes the data to the main memory only when there is another request to overwrite an attributed cache way that is "dirty", i.e. where values have been modified by the processor. This is done to avoid losing the content previously stored in the data cache, while avoiding too many main memory accesses that are useless for temporary data quickly changed by the processor. It is called a write back policy.

There are also two possible approaches to handle write operations when the data is not already allocated in the data cache:

- Write-allocate consists in writing the data in the cache and then into the main memory

- No-write-allocate consists in writing the memory block only in the main memory and not in the cache. Data is only loaded into the cache on read misses.

In modern processor architectures, high-end CPUs generally embed several cache memories with a specific hierarchy. These cache memories have different sizes, and the smaller is the first one to be used, as it is the fastest. The smallest and fastest cache memory is named the L1 (for level 1) cache. If a request to the L1 cache is a cache miss (i.e. the element requested does not reside inside the L1 cache), then the request is sent to the L2 cache, then to the L3

cache and so on until the main memory is reached. The main memory is always the last one to be used as it is the slowest (but also the biggest). L1 caches are the smallest and fastest caches, and are generally separated into two distinct caches: one for instructions and one for data. L2 caches are bigger and slower and are designed to reduce the impact of L1 cache misses. They are generally unified, meaning that they both contain instructions and data. This depends on the specific application they will be used for however. L1 and L2 caches are generally core-specific, meaning that each physical core has its own L1 and L2 caches. They are still shared among several threads on the same core. This is also application-dependent as in some specific cases, the L1 and L2 caches can be shared among several physical cores. In some architectures, there are also higher level L3 and L4 caches. These caches are even bigger and slower than the L2, but still faster than the main memory. Analogously to the L2 cache, the goal of the L3 and L4 caches is to reduce further the timing penalties of cache misses. These caches, contrarily to the L1 and L2 caches are generally shared among several different physical cores. The combination of all the caches and the main memory composes the memory hierarchy.

2.1.3. Speculative execution

Speculative execution is an optimization mechanism aiming at improving CPU performances. The general idea behind this mechanism is to anticipate part of the future computations and perform them in advance and in an optimized way in order to save some precious time. Speculative execution regroups two main inner CPU mechanisms that work together: branch prediction and Out-of-Order execution.

2.1.3.1. Out-of-Order execution

Processor pipelines are composed of several stages (frontend, instruction decode, issue stage, execution stage…). When instructions are issued, the actual computation happens at the execution stage of the pipeline. The execution stage is composed of several execution units, for example a load-store unit (LSU), a multiplier, an arithmetic-logic unit (ALU), etc. These resources are limited and can only be used by a single instruction at a time. The basic method for computing instructions is called “in-order”. This means that the instructions are computed and calculated in the same order as they are issued by the programmer’s code. Thus, a given instruction cannot be computed before all of the previous ones have been. This has several drawbacks, especially when consecutive instructions require the same execution unit to be computed. This leads to a pipeline stall until the specific execution unit is freed. In order to avoid this type of pipeline stall, and to gain some precious execution time, another method for computing instructions has been developed. “Out-of-Order” execution, as opposed to “in-order” execution, is an optimization mechanism that tends to optimize the usage of the available execution resources. To proceed, it disorganizes the instructions compared to the program semantic. The objective is to compute all the instructions as fast as possible by maximizing the usage of every execution unit available in the pipeline. This results in a much better parallelization of the execution units’ usage. However, the instructions are not computed in the order requested by the programmer, as visible in Figure 2. In this figure, the considered CPU is a simple “superscalar” processor. It can read up to two instructions per cycle. We consider three execution units: integer calculations (two stages EI and WI), floating-point calculations (four stages E1F, E2F, E3F and

WF) and memory accesses (two stages EM and WM). We also suppose that there is a memorization element in the completion unit, and that there are two writeback entries. We then consider the following simplified assembly pseudo-code:

```
Load R3
Load R8
Add R3, R8, R3
AddF RF2, RF4, RF6
```

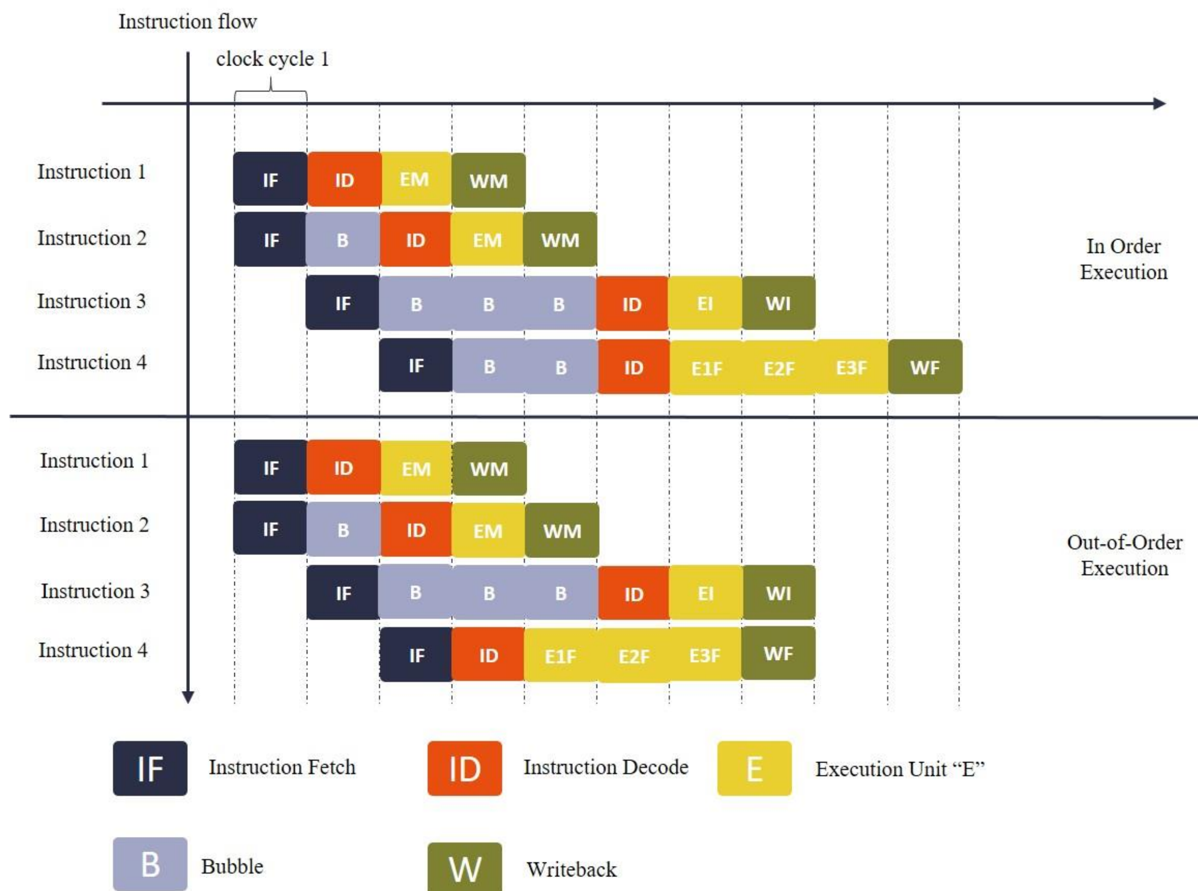


Figure 2. Schematic representation of the difference between in-order and out-of-order execution

Some instructions that are supposed to be computed after can be computed before another instruction that uses a busy or slow execution unit. To restore the correct flow of instructions, and issue it as requested by the programmer, a specific structure is added to the pipeline. The reorder buffer stores the computed instructions and reorganizes them in the correct order before transmitting them to the commit stage, thus returning the appropriate architectural state requested by the programmer. The reorder buffer is not the only supplementary hardware element that needs to be added in the case of an Out-of-Order CPU. Several tables to memorize the initial program order, intermediary states in case or error and many other elements have to be further implemented. Therefore, Out-of-Order execution is very costly and only added in high-end processors that have strong performance requirements and are not too limited in terms of space or cost.

2.1.3.2. Branch Prediction

Along with the Out-of-Order execution, Branch Prediction is another micro architectural mechanism that aims at speeding up instruction executions at the cost of potentially useless pipeline calculations. Conditional jumps and branching instructions are a major cause of slowdowns inside pipelines. Indeed, they break the linearity of the instruction processing. Generally, pipelines compute instructions that are located at consecutive addresses in memory. The jump and conditional instructions cause the computation of some specific instructions, located elsewhere in the memory. However, the specific instructions to be computed are known only when the jump and conditional instructions reach the third stage of the pipeline generally (this depends on the specific CPU architecture considered). This implies that all the computations done during this time have to be discarded if the branch is taken, causing a considerable time loss. To avoid this type of situation, the branch prediction is a hardware mechanism that tries to predict where these instructions will redirect the instruction flow. The objective is to anticipate the result of these jumps and conditional instructions, as the computation of the instructions following those ones is slow. This enables the CPU to compute the anticipated result of these instructions while the actual result is also being computed meanwhile, thanks to parallel and Out-of-Order execution. The predictions are based on the behavior of the previous jump and conditional instructions. Therefore, several hardware structures are required here to keep track of where previous jumps and conditions redirected the execution flow. These structures are Branch Target Buffer (BTB), Branch History Table and Pattern History Table (BHT and PHT) and the Return Stack Buffer (RSB). What makes branch prediction interesting is that, in the case of a correct prediction, the instructions resulting from a jump or a condition are already being computed, and some precious execution time has been gained. In the case of an incorrect prediction, only a minimal amount of computation time (depending on the performance of the selected prediction algorithm) has been lost since not using branch prediction would result in a greater time loss. In this case, the results of the speculative computations are discarded before commitment at the architectural level and the computation resumes normally using the outcome of the jump or condition computation.

2.2. Micro architectural covert-channels

Covert channels are specific methods used to create a capability to transfer information between entities (e.g., processes) that are not allowed to communicate according to the security policy. Moreover, they do not use the legitimate data transfer mechanisms. Therefore, they are very hard to detect by the operating system. However, they suffer from several downsides. First, depending on the quantity of data that is transmitted and the protocol used, several variants can be detected by monitoring system performance as they imply a visible overhead. Then, they generally suffer from a low signal-to-noise ratio and low data rates (typically a few bits per second).

The addition in the most recent high-end CPUs of many optimization mechanisms has led to the complexification of the underlying micro architecture. Consequently, several covert-channel possibilities have been uncovered relying on various micro architectural elements (e.g., cache memories, internal buffers…). Most of the end-to-end micro architectural attacks that will be presented in a later section are concluded with a micro architectural covert-channel to extract the data and transmit it from the micro architectural world, to the architectural world, where it can be observed.

Covert-channels must not be mistaken with micro architectural side-channels that are passive information leakages that occur based on the way a computer algorithm or protocol has been implemented and the resulting physical characteristics (e.g., power consumption, electromagnetic emissions, …) induced by the hardware element activations. Side-channels can be exploited to carry out an attack, whereas covert-channels are active communication channels that bypass security boundaries to transmit information. The best candidates, when considering the micro architecture to create a covert-channel, are the cache memories. These memories are shared between several processes or even several cores, and can contain all types of information about the programs that are being run.

Cache side-channel attacks are not a new concept [18]. They emerged in cryptanalysis as a practical way of extracting information from a side-channel leakage of some well-known cryptographic algorithms. This led to some classification efforts as proposed by Onur Aciicmez in [19]. This classification introduces three different categories: time-based covert-channels, trace-based covert-channels and access-based covert-channels. Even if relevant in the context of cryptanalysis, a new classification has been introduced a few months after the main micro architectural attacks have been published. Qian Ge, Yuval Yarom, David Cock and Gernot Heiser [20] and Maria Mushtaq [21] proposed a new classification based on the specific method the covert-channel leverages to transmit information through the cache, rather than the type of exploited side-channel. This document will focus on this more recent classification, as it is more relevant in the context of the proposed study and has been introduced specifically in the context of micro architectural leakages. The next section will cover the different types of cache covert-channels that can be built at the micro architectural level according to the classification proposed by [20] and [21]. It is important to note that cache covert-channels are not attacks only by themselves. They only provide a specific way of transmitting data outside of the conventional communication channels. They need to be combined with a preliminary attack that gathers information before being useful in an attacking context.

2.2.1. Prime+Probe

A first and widely used cache covert-channel technique in micro architectural leakages is the Prime+Probe [16] covert channel. It only requires that the victim and attacker program time-share a core and therefore also the cache memory. Time-sharing consists in the sharing of a limited computing resource among many users at the same time. To achieve this, multiprogramming and multi-tasking are used to attribute each user a given portion of the computing resource. The Prime+Probe technique enables an attacker to observe the cache contention caused by the victim's activity. More specifically, it gives information about the precise cache sets that were accessed by the victim. The Prime+Probe technique can be divided into three successive steps:

1) The attacker places the exploited hardware element such as the L1 data cache in a state where the cache's status is known. This can be translated to the attacker filling it with its own data. After this prime phase, if the attacker accesses its data again, it will result in a faster response compared to a potentially evicted data.
2) The attacker's execution stops and the victim is then run. Running the victim code leads to the eviction of a part of the attacker's data. This is the normal behavior of the data cache in case it is completely filled. At this step, the data cache contains a part of the attacker's data, and some data belonging to the victim.
3) The victim execution ends. The attacker can now probe the time it takes to access the cache by reading its own data and using a timing source (e.g., a hardware timer as commonly found on every CPU). As the victim's execution replaced some of the attacker's data, these elements will have a longer access time compared to the others, due to the occurrence of cache misses. The attacker can therefore infer which part of the cache has been used by the victim, provided the attacker knows the structure of the cache he is targeting. This enables him to recover information about the victim's behavior, depending on the target.

This technique has been applied to most types of caches that can be found inside a high-end processor: data caches [18, 22], instruction caches [19, 23] and last-level caches [24]. This implies that the Prime+Probe technique can be used when the victim and attacker are located on the same physical core, or on different cores (e.g., leveraging the Last Level Cache (LLC) Prime+Probe variant presented in [24]). However, in practice, this technique is easier to implement and more effective when targeting the L1 cache rather than a LLC. The more applications that share the targeted cache, the harder it gets to leverage a Prime+Probe covert-channel.

2.2.2. Flush+Reload

Flush+Reload [25, 26] is another covert-channel technique that differs from the previous Prime+Probe technique. It has been widely used in the context of micro-architectural attacks, beginning with the Foreshadow attack [11]. Contrary to the Prime+Probe covert-channel that gathers information about the contention on the cache sets, Flush+Reload has a more precise granularity. Indeed, this technique enables an attacker to get the precise address of the cache lines that were accessed by a victim program. However, this comes at the cost of stronger base hypothesis: the attacker needs a cache flushing instruction, the victim and the attacker need to share their address spaces and the attacker and victim need to be in continuous synchronization. This covert-channel can be applied in single-core, cross-core and through VMs setups.

Flush+Reload is strongly correlated to the micro architectural attacks. The Flush+Reload technique can be decomposed into three steps:

1. The attacker flushes the cache region he is targeting. The flushing depends on the extraction structure used. The extraction structure is the data structure allocated by the attacker that will receive the secret information retrieved. The main idea here is to allocate a structure that will have exactly the same dimensions and subdivisions as the shared region of the cache. Typically, the extraction structure will be an array/buffer. The attacker therefore flushes the cache lines relating to this buffer using their addresses (e.g., a specific instruction is required for this purpose, as available in the x86 architecture).

2. In the second step, the attacker stops running and lets the victim operate. This causes the cache to be filled by victim-owned information.

3. The attacker causes the CPU to reload the cache lines of the extraction structure. There are several solutions to cause such a reload. The idea is to force the CPU into a corner-case scenario, such as a misprediction, that leads to the rollback of the micro architecture. During this reloading phase, the attacker can use timing measurement tools to gather information about each slot of the extraction structure previously used as the CPU reload them to their former content. If the reloading of a given slot is slow, then it has not been modified during the victim's execution. It does not need its content to be updated. Otherwise, the slots that have been modified during the victim's operation will experience a significantly longer reload time as their content needs to be modified and rolled-back. This enables an attacker to access the specific addresses of the cache lines that have been modified by the victim program.

The Flush+Reload technique is therefore very powerful when a flushing instruction is available, which is not the case for every ISA/implementation. Its level of granularity enables an attacker to determine precisely the cache lines that are of interest. This technique has been applied on all types of caches (data and instruction) and all levels of caches (L1 to LLC).

2.2.3. Flush+Flush

The Flush+Flush [27] technique works with the same logic as the Flush+Reload technique. However, it abuses the presence of a flushing instruction even further. Indeed, this technique is used to measure the time it takes to flush a certain cache region before and after the victim's activity to determine which cache lines have been accessed. The required hypotheses are the same as for the Flush+Reload technique: the attacker and the victim must share their address spaces, it requires a flush instruction, and the victim and attacker need to be constantly synchronized. This technique has the advantage of being stealthier as it does not imply any modification of the amount of cache misses (unlike Flush+Reload or Prime+Probe). It is also faster, but less accurate (higher error rates). The three steps of the Flush+Flush technique are:

1. The attacker uses an extraction structure similarly to the previous ones, typically a buffer (or an array) in the shared cache memory space with the same dimensions as the shared cache memory region. Then the attacker flushes each slot of the extraction structure and measures the time it takes for each related cache line to be flushed. As the structure is initially empty, these first flushing operations are fast.

2. The attacker goes idle and lets the victim run. The victim's activity causes some lines inside the cache to be filled with victim-owned data.

3. The attacker flushes the same area of the cache again, corresponding to the extraction structure. He measures the time it takes for each line to be flushed. Cache lines (e.g., extraction buffer slots) that experience a longer flushing time have been modified by the victim and are therefore of interest.

This technique therefore only relies on the presence of a cache-flushing instruction to work. However, when available, it enables an attacker to have a stealthier approach as compared to the other cache covert-channels, as it cannot be detected by monitoring the amount of cache misses occurring.

2.2.4. Evict+Time

Evict+Time [18] is a unique cache covert-channel technique as it reverses the roles of the attacker and the victim as compared to all the other techniques presented above. In this covert-channel, the victim is run twice and the attacker evicts cache lines in-between the two executions. This main goal of this technique is to identify the timing variations implied by the eviction of given cache lines. This enables the attacker to gain knowledge on the influence that each cache line can have on the victim's execution. The only other requirement is that the attacker and the victim share their address spaces, as usual. This covert-channel will typically be used in use-cases where the attacker has the control over the victim's execution. The best example would be an encryption service that the attacker can run as desired with chosen inputs which gives the output in return. In such cases, the covert-channel is composed of three usual steps but with inverted roles:

1. The trigger phase: the attacker runs the victim with an untouched cache and measures the time it takes. This measurement will be used as the reference timing.

2. The evict phase: instead of running the victim to evict some cache lines, this time the attacker himself will be run in this second step. The goal of the attacker here is to evict some chosen cache lines to affect the victim.
3. The time phase: the victim is run again and the execution time is measured one more time. The attacker can then compare it with the reference measurement and deduce if the cache lines he previously evicted affected the victim or not, and to what extent.

By repeating these three steps several times, it is possible for an attacker to characterize the behavior of a victim program in terms of cache usage.

2.2.5. Prime+Abort

The Prime+Abort [28] technique is very specific as it only applies to Intel processors that have the Intel TSX (Transactional Synchronization eXtension) extension [29]. This new set of instructions for x86 adds support for hardware transactional memory. This extension aimed at speeding up the execution of multi-threaded software. Among other functionalities, it enables a programmer to create TSX transactions. These transactions are pieces of code that are seen as atomic by the processor, meaning that they either successfully run completely, or fail and abort inflight which causes a rollback to the state at the beginning of the transaction.

However, this extension also opened new attack paths as demonstrated by the Prime+Abort covert-channel. This technique, unlike all the previously presented ones, does not rely on any type of timer. It has been developed to bypass the defenses introduced to mitigate the covert-channels with strong timer dependencies by leveraging the Intel TSX extension. This technique also proved to be better in terms of efficiency and accuracy as compared to the Prime+Probe technique on Intel processors. Prime+Abort begins with a prime step, where the attacker opens a TSX transaction and fills the exploited cache memory region with attacker-owned data. Then, the victim is run and when it accesses an address in the targeted cache set, the attacker’s TSX transaction will abort as the whole attacker code cannot be executed atomically (as it has suffered from perturbations by another process trying to access memory regions related to the TSX transaction). The attacker can then conclude that the victim process tried to access a cache line within the targeted cache set, and thus gets access to the same type of information as the Prime+Probe technique.

2.2.6. Evict+Reload

The Evict+Reload [30] technique is an alternative to the flush-based covert-channels that does not require any flushing capabilities. This covert-channel uses exactly the same concept as the Flush+Reload technique and only replaces the flushing step with an attacker-controlled eviction of the cache lines (e.g., as done in the second step of the Evict+Time technique). Therefore, this covert-channel does not require the ISA of the targeted core to include a cache flushing instruction. This technique has been introduced in order to qualify the effectiveness of the deactivation of the cache-flushing instructions as a countermeasure to cache covert-channels.

2.2.7. The importance of the encoding technique

Before continuing towards the description of the complete micro architectural attacks that leverage these covert-channels as a final step for data extraction, it is necessary to develop some aspects of methodology regarding the usage of micro architectural covert-channels. The covert-channels are only a method for recovering information while bypassing security boundaries. This means that to carry out a covert-channel, a preliminary step is required to gather information to be transmitted using the covert-channel. This preliminary step is also responsible for the encoding of the data. The notion of encoding is very important when considering micro architectural covert-channels or even covert-channels in general. Encoding refers to the specific method used to hide the information inside the exploited resource. Practically, encoding is the method that an attacker uses to make the information obtained from the communication channel useful. Based on the communication channel's size and the type of data it can transmit, the attacker will use a specific encoding technique to place the data obtained from the preliminary attack in a specific disposition that fits the current communication channel he leverages. The goal of an encoding technique is to obtain exploitable information at the end of the covert-channel. This is caused by the fact that the information obtained as the output of the communication channel has been directly correlated with the input by the attacker before proceeding to the data extraction. The type of data, its size, the disposition and organization of the exploited micro architectural resources are as many parameters that need to be taken into consideration when choosing the type of encoding and thus the type of covert-channel that best suits the situation. For all the previously presented covert-channels, their effectiveness and utility are polarized by the encoding technique used. A single covert-channel technique can also be used in several different ways, depending on the secret to extract and the chosen method to hide it within the cache.

It is important to note that the content of the caches can never be accessed. Caches are micro architectural elements that cannot be observed by a programmer. The developer can only guess what happens from the architectural level. This means that the micro architectural cache covert-channels enable an attacker to recover an information without reading it directly inside the data cache. All the methods presented in the previous sections are solutions to overcome the difficulty of not being able to read the cache directly. Let us consider two examples to clarify these statements and show the diversity of the presented covert-channels when it comes to encoding techniques.

The victim considered uses a secret value between 0 and 256 named *S*, that an attacker wishes to recover and extract. The targeted cache memory is the L1D. A preliminary micro architectural attack manages to leverage an information leakage and accesses the secret value at the micro architectural level. From this point, there are several possibilities for encoding the secret value inside the L1D cache before extracting it. If one wants to carry out a Flush+Reload covert-channel, then the secret value must correspond to the address of the cache line that will be evicted. This means that the preliminary attack needs to evict exactly one cache line: the cache line number *S.* This is an example of an encoding technique. Another possibility would be to consider that the secret value would correspond to the amount of cache misses caused during the attack's execution. This means that the preliminary micro architectural attack has to evict as many cache lines as the secret value *S.*

Using this second encoding technique, it is possible to recover the secret using a Prime+Probe covert-channel as it can also enable an attacker to count how many cache lines have been evicted.

These are only two examples of the many possible methods to encode a secret value inside of a cache memory. The choice of the encoding technique depends on many factors: the type and size of the secret to extract, the organization and size of the exploited cache memory, the behavior of the victim application, the capacities of the preliminary micro architectural attack (e.g., it influences the capabilities for the attacker to place complex information inside the cache), etc… The different existing covert-channel techniques and their related specificities, such as the main encoding technique for each one, are summarized in Table 1.

Table 1. Summary of the main micro architectural cache covert-channels and their specificities

Covert-channel variant / Characteristics	Prime + Probe	Flush + Reload	Flush + Flush	Evict + Time	Prime + Abort	Evict + Reload
Initial hypotheses on the victim and attacker	The victim and attacker program time-share a core meaning they also share the cache memory	The victim and the attacker need to share their address spaces The attacker and victim need to be in continuous synchro.	The victim and the attacker need to share their address spaces The attacker and victim need to be in continuous synchro.	The attacker and the victim share their address spaces The attacker has control over the victim's execution	The victim and attacker program time-share a core meaning they also share the cache memory	The victim and the attacker need to share their address spaces The attacker and victim need to be in continuous synchro.
Type of secret transmitted (encoding) and comm. channel	The secret is transmitted as the amount of contention created inside the cache sets	The secret is transmitted by addressing a cache line	The secret is transmitted by the amount of cache lines modified by the victim execution	The secret is transmitted by the timing differences caused in the victim execution (cryptographic algorithm)	The secret is transmitted by the cache line that the victim code tries to access during a TSX transaction	The secret is transmitted by addressing a cache line
Maximum extractible secret size in a single iteration	Secret size = number of cache sets	Secret size = number of cache lines	Secret size = number of cache lines	Secret size = number of cache lines	Secret size = number of cache lines	Secret size = number of cache lines
Requires a flush instruction	**No**	**Yes**	**Yes**	**No**	**No**	**No**
Stealth level (ranging from 1 = very stealthy to 3 = can be detected using performance counters)	2	3	1 (no modification in the amount of cache misses observed)	3	3	3
Type and level of caches that can be targeted	L1D, L1I, L2, L3, LLC	L1D, L1I, L2, L3, LLC	L1D, L1I, L2, L3, LLC	L1D, L1I, L2, L3, LLC	L1D, L1I, L2, L3, LLC	L1D, L1I, L2, L3, LLC
Scenarios the covert-channel can be applied into	Single core, Cross core (physical & logical)	Single core, Cross core (physical & logical), Cross VM	Single core, Cross core (physical & logical)	Single core, Cross Core (physical & logical)	Single core, Cross Core (physical & logical)	Single core, Cross core (physical & logical)
Types of architectures it can be applied to	Intel, ARM, AMD, RISC-V	Intel, ARM, AMD, RISC-V (if flush available)	Intel, ARM, AMD, RISC-V (if flush available)	Intel, Arm, AMD, RISC-V	Specific to Intel	Intel, ARM, AMD, RISC-V

2.3. Main micro architectural end-to-end attacks

Now that the micro architectural covert-channels have been presented, the next section will focus on the different micro architectural end-to-end attacks that can be carried out as preliminary attacks. These attacks are classified according to the micro architectural element(s) that they take advantage of and are the core attacks that the study proposed in this document aims at mitigating.

2.3.1. Transient-execution attacks

Transient-execution attacks are specific micro architectural attacks that leverage the presence of speculative and/or out-of-order execution to extract secret information through the security boundaries of a system. The "transient" window refers to the moment when the CPU is speculatively executing some code that will eventually be rolled-back afterwards because the initial prediction was wrong. During this "transient" window, a potential attacker has the possibility to access secret data using different leakages exploitation that will be detailed in the next sections. Most transient-execution micro architectural attacks follow the same pattern for carrying out the attack as described in Figure 3:

1. A preface stage, where the attacker prepares the attack by placing the micro architecture in an advantageous state for the attack. He also prepares the data structures required to recover the information via the chosen covert-channel.
2. The attacker leverages a specific instruction leading to a corner-case scenario where optimization mechanisms, typically out-of-order execution, are required. This trigger instruction has to be used wisely so that the targeted secret is involved in the transient computations done afterwards.
3. The transient calculations happen and the secret is computed upon, as well as the attacker's code that accesses the secret value and places it correctly for the covert-channel extraction.
4. The CPU realizes the transient computations should not have happened and rolls back the changes.
5. During the rollback, the attacker leverages a covert-channel to recover the secret value before it is discarded by the CPU.

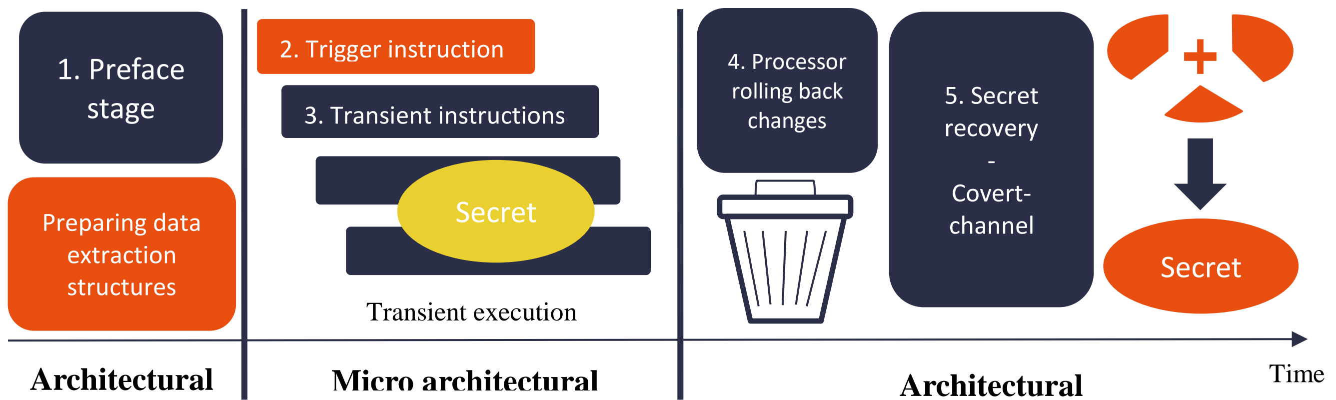


Figure 3. Typical phases in a transient-execution micro architectural attack

2.3.1.1. Spectre

The Spectre vulnerability was disclosed in January 2018 [31]. This vulnerability tricks a victim program into accessing arbitrary locations in the program's memory space. Therefore, an attacker can read the content of the program's memory and potentially get access to sensitive data. Spectre is not the name of a single attack, but rather a generic name representing several types of attacks, that all exploit the side effects of speculative execution. More specifically, the Spectre vulnerability focuses on the branch predictor. It requires that the targeted processor possesses a branch prediction mechanism, and an L1 data cache used for the final data extraction (e.g., the covert-channel). Let us consider the function proposed in

Listing 1 that has access to two arrays, *array1* of size *array1_size*, and *array2* of size 1MB.

Listing 1. Example of Spectre attack leveraging conditional branch

```
IF (chosen_index < array1_size)
     y = array2[array1[chosen_index] * 4096]
```

It is assumed that the attacker and the victim share the memory space. Moreover, the attacker already knows an address that points to a secret byte from the victim address space. Then, the attacker picks *chosen_index = secret_byte_address – array1_base_address*. The secret address known to the attacker is assumed to verify that *array1[chosen_index]* resolves to a secret byte in the victim's memory. Under these conditions, the general attack pattern consists in three essential steps:

1. The branch prediction logic is "trained" to miss predict the outcome of a conditional branch, so that the CPU will speculatively execute the attacker's code which should not be computed. The branch prediction mechanism works with statistics. For example, let us consider a conditional branch featuring two outcomes, X and Y. If the CPU has been through the outcome X more times than the outcome Y, the branch predictor will speculatively execute the outcome X. Analogously, the attacker can run several times the program with modified

variables so that the outcome is always Y during what is called the training phase. Afterwards, when executing the victim code, the branch predictor will execute the outcome Y because statistically, it has been computed more times than the outcome X. Here, the branch prediction is mistrained to predict that *chosen_index < array1_size* is true even if *chosen_index* is bigger than *array1_size*.

2. Once the branch predictor has been poisoned, the attacker causes a bound check on *array1* for an index purposefully and maliciously chosen out of the bounds (named *chosen_index*). The chosen index has to be related to the secret address. It is typically equal to the secret byte's address minus the base address of *array1*. The CPU will then assume the check is successful (even though it is not), because it has been poisoned. Therefore, the attacker can execute whatever piece of code he wishes during the transient speculative execution phase after the bound checking, thus requiring access to a cell from the other genuine array named *array2*. The attacker requests a cell that has an index related to the secret byte to recover, such as *array2[array1[chosen_index]*4096]*. *Array1[chosen_index]* resolves to a secret byte residing in the victim's memory space (starting hypothesis). The secret byte is therefore stored in the cache, as it has been requested by the attacker's transient code.

3. The attacker carries out a covert channel to recover the data stored inside the L1 cache and accesses the secret byte using a Flush+Reload or Evict+Reload covert-channel.

To conclude, one could say that Spectre can be used to manipulate a process into revealing its own data through poisoning of the branch predictor. The attack described previously is the general behavior of the initial conditional branch Spectre attack. Several variations of the attack have been implemented and published. A first classification [32] categorizes the Spectre attacks based on the micro architectural root cause that triggers the misprediction. This categorization introduces four variants:

- Spectre PHT [31, 33] that exploits the Pattern History Table (or PHT). The PHT is the main memory-like element that keeps track of the results of conditional branches and serves as a basis for the prediction for future condition branches.
- Spectre BTB [31] takes advantage of the Branch Target Buffer (BTB). This micro architectural structure is used to predict the addresses of the destination branches.
- Spectre RSB [34, 35]. This variant exploits the Return Stack Buffer (RSB) that is used to predict the return addresses.
- Spectre STL [36] that takes advantage of memory disambiguation to predict the "store to load" data dependencies.

Several other micro architectural attacks use Spectre variants as a part of their implementation. These attacks leverage Spectre as a basis and then build a more specific and complex attack based on it. NetSpectre [37] and SgxPectre [38] are part of the Spectre-based micro architectural attacks targeting specific victims.

2.3.1.2. Meltdown

Meltdown is another type of attack on Intel's CPUs that was disclosed in January 2018 [39]. Meltdown exploits the way the different security features introduced in this work are interacting. This exploit bypasses the CPU's fundamental privilege boundaries and enables an attacker to access privileged and sensitive data from the operating system, or other processes. The attack can be carried out in three steps:

1. The CPU attempts to execute an instruction, accessing to a memory location (a read instruction for example). Computing this instruction requires to calculate the targeted memory location address. This calculation uses sensitive values that are regulated by a privilege check before the memory is accessed, to ensure the entity trying to read the values in memory is authorized to do so.

2. The privilege check eventually fails and states that this memory location should not be accessed by the attacker. The read instruction should therefore fail. However, before the permission check, the address of the memory location and the data contained have been computed, and stored inside the cache. Even if the CPU rolls back all the calculations at the architectural level, the modifications done in the cache (at the micro architectural level) cannot be suppressed. This constitutes the information leak that Meltdown will take advantage of.

3. As usual in all the L1 Terminal fault attacks (Meltdown, Foreshadow…), a covert channel is used to recover the secret data. Flush+Reload is the most widely used for carrying out a Meltdown-type attack. The attacker can now recover an information related to the data stored inside the memory location he initially tried to access. This information can either be proportional to the secret value to recover, or it can be the secret value itself depending on the encoding technique used during the Flush+Reload covert-channel.

The Meltdown technique can be used in sequence to read every address of interest at high speed. It is possible to recover all types of secret information, such as passwords, encryption data, etc. Meltdown can target any process that exists in its memory map.

Contrarily to Spectre-type attacks that are taking advantage of branch mispredictions, Meltdown type attacks abuse the transient instructions that are created after a CPU exception such as a page fault. The work done in [32] also introduces a classification for Meltdown-type attacks. The first level consists in segregating the variants based on the exception that causes the transient execution. The second-level classification further discriminates the attacks based on which memory location can be reached and if the variant crosses privilege boundaries or not. This second-level classification is for classifying Meltdown variants with a thinner level of granularity. As studying Meltdown-like attacks is not the primary scope of this document, only the first-level classification will be introduced for the purpose of the state-of-the-art. The first-level classification introduces eight different Meltdown variants:

1. Meltdown-US (Supervisor-only Bypass): this variant contains the original Meltdown attack [39]. The attacker is capable of reading a cache line that is based on the privileged data red during a page fault triggered by dereferencing an unauthorized

kernel address. The data is reconstructed at the end of the attack using a micro architectural covert-channel, typically Flush+Reload.

2. Meltdown-P (Virtual Translation Bypass): this category of Meltdown-type attacks consists in the Foreshadow attack [11] that will be studied more in details in a later section of this document. This variant raises a page fault by accessing an unauthorized memory page. The data is then extracted using a Flush+Reload micro architectural covert-channel at the end of the attack.

3. Meltdown-GP (System Register Bypass) [40]: allows an attacker to read privileged system registers. Indeed, trying to access an unauthorized system register raises a specific fault named General Protection fault (*#GP*). This fault leads to transient execution that can then be leveraged to carry out a Meltdown-type attack. Again, this variant is concluded by a micro architectural cache covert-channel, preferably Flush+Reload.

4. Meltdown-NM (FPU Register Bypass): Stecklina and Prescher [41] demonstrated a fourth variant of Meltdown taking advantage of the so-called lazy state switch. Indeed, when switching contexts, saving the Floating-Point Unit (FPU) and Single Instruction Multiple Data (SIMD) registers is very costly. Thus, instead of saving these memory elements, the FPU is simply marked as not available. Issuing an instruction to the FPU once it is marked as unavailable causes a device-not-available exception (*#NM*). This exception then causes transient instruction that enables an attacker to leverage a Meltdown-type attack.

5. Meltdown-RW (Read-only Bypass): Kiriansky and Waldspurger [33] demonstrated that it is possible to carry out a Meltdown-type attack within the current privilege level. They managed to overwrite read-only data within their privilege level.

6. Meltdown-PK (Protection Key Bypass): [32] presents a Meltdown variant targeting specifically Intel’s Skylake-SP CPUs. These core use memory-Protection Keys for User space (PKU). The presented Meltdown variant is capable of bypassing the hardware-enforced read and write isolation provided by PKU.

7. Meltdown-BR (Bounds Check Bypass): x86 CPUs are equipped with specific hardware instructions that raise an exception when array indices are out of bound leading to a bound range exceeded exception (*#BR*). Similarly to the all of the Meltdown variants, this errors leads to creating a transient execution window that can then be leveraged to carry out a Meltdown-type micro architectural attack.

2.3.1.3. Foreshadow

Foreshadow is one of the most iconic micro architectural attacks as it is the first one that has been able to break the isolation introduced by the SGX technology from Intel. It also started a trend for developing micro architectural attacks exploiting undocumented internal CPU buffers. Having a precise understanding of this initial attack is therefore mandatory to get the general methodology and mindset that are required to perform micro architectural attacks.

In this section, as in the article describing the initial attack [11], it is assumed that the attacker knows the address where the secret is located in the victim's memory. This means that in a realistic situation, before being able to carry out the Foreshadow attack, prior knowledge about the victim has to be acquired. The following will describe the basic and initial version of the Foreshadow attack, which extracts one byte from enclave memory. This byte will be referred to as secret or secret value. The Foreshadow attack consists in six steps, as represented in Figure 4.

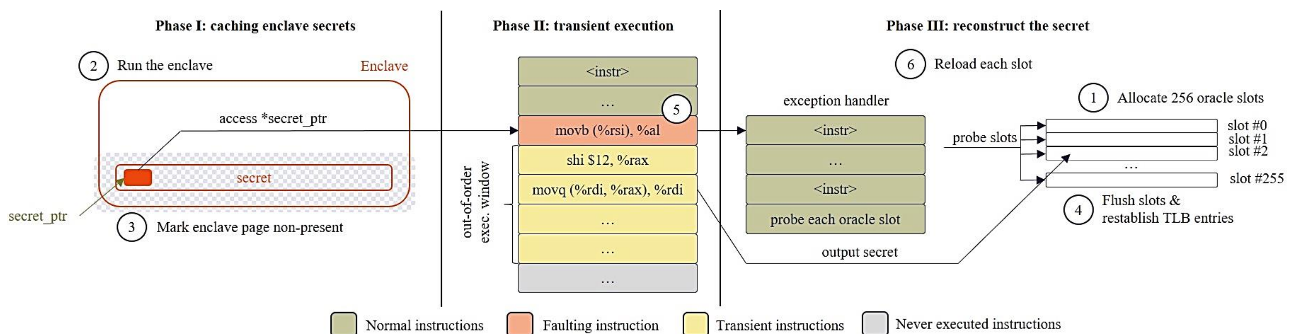


Figure 4. The different steps of the Foreshadow Attack[1]

Each of these steps will now be detailed and explained:

1. The first step consists in the attacker allocating an oracle buffer (e.g., an array structure that will be used for secret extraction) containing 256 entries. This buffer is a standard data structure that the attacker can freely manipulate: it is possible to allocate values in it, to address it, etc. The oracle buffer has to be allocated according to the dimensions of the targeted secret. Here, the attacker is willing to extract a single byte, which values range from 0 to 255. The reason why an entry is needed for every possible secret value will be explained in phase 5.

2. The second step, which is crucial for the attack to be successful, consists in executing the victim enclave's code. By doing so, the attacker ensures that the secret handled by the victim will reside in the L1 cache, as it was computed by the enclave.

3. Then, the attacker will use the mprotect instruction to suppress its own rights to access the enclave memory page where the secret resides. This is possible here because we assumed that the attacker knew the pointer related to the secret

[1] By J. V. Bulck et al., available at: https://foreshadowattack.eu/foreshadow.pdf, [accessed 23 Feb 2021]

memory page. Also, the mprotect instruction enables a user to change his access rights to any processor memory, no matter how it was allocated. Therefore, the attacker can use this instruction to remove all his rights to access the secret page, as he knows a pointer to that page. Doing so ensures the attacker that the access to the secret memory location will result in a page fault, and not in a page abort semantic (which would replace the secret data with only ’11….11’, and would not return any error).

4. As the buffer was allocated in a previous manipulation, it now resides in the L1 cache. In this stage, the attacker will flush the cache lines where the oracle buffer is located. This cache flush is done to ensure that the oracle buffer was not already inside the L1 cache. By doing so the attacker keeps the oracle buffer “fresh”, so that it does not get evicted from the cache by the time of the attack. If another process uses the cache, the oracle buffer might get evicted from L1D if it has not been accessed recently enough. For this purpose, the attacker uses the clflush instruction. Also, the clflush instruction furthermore enables him to restore the Translation Lookaside Buffer (TLB) entries. Upon clflushing the oracle buffer’s cache entries, the CPU will access the physical memory location of the oracle buffer to ensure it does not have to make any update to the value. Therefore, the CPU will make a virtual to physical memory translation which will be stored in the TLB (buffer to speed up memory translations). Restoring the TLB entries is mandatory, because the TLB is used to speed up recurrent memory accesses. Upon enclave entry/exit (which happened in phase 2 where the enclave has been executed), the TLB is entirely flushed. Therefore, accessing the oracle buffer after executing the enclave results in a very expensive page table walk, and the access time would be longer than the execution window that the attacker possesses in the later phases of the attack. Using the clflush instruction thus restores the TLB entries related to the oracle buffer and enables the attacker to access it faster. This double benefit (flushing the cache and restoring the TLB) from the clflush instruction is crucial for the attack to be successful. The TLB’s role will consist in storing the most recent virtual to physical translations that the CPU will do, while the L1 cache will store the data accessed by the CPU when doing the address translation (= accessing physical memory).

5. Once the oracle slots are removed from the cache and brought back to the TLB, the attacker will try to access the enclave memory page containing the secret. It has been stated that the attacker already knew the address of the secret page before carrying out the attack. Upon accessing the unauthorized enclave memory page, the attacker will trigger a complex routine that will check which type of error the CPU has to return. While this check is ongoing, the CPU will still execute the attacker’s code speculatively: this is the transient execution window. The attacker will be able to execute some piece of code that will access the forbidden memory page. This page can temporarily be accessed because the error has still not been returned and the processor speculatively executes the attacker’s code, assuming that the access to the memory page will eventually be granted. This is part of the processor optimization named “Speculative execution” aiming at increasing performance. In case the access to the page is not granted, all the changes will be discarded, so in the end, it doesn’t really matter if the code accesses forbidden data as it won’t be able to recover and

read it. However, this assumption is false. Even though the changes will be discarded, a skilled attacker can still recover the secret data through the micro architecture. The attacker's code will transiently access the page containing the secret, and fetch the said secret (which will be named Sc). Then it will allocate the oracle slot oracle[Sc]. This slot indexed with the secret value will therefore be brought to the L1 cache. The attacker has the time to fetch the secret and allocate the corresponding oracle slot because the secret has been brought to the L1 cache at stage 2 where the enclave has been executed. Also, the TLB entries corresponding to the oracle buffer have been restored, so it is possible to access the oracle buffer and the secret within the small temporal window that the attacker has. Moreover, one can now understand why it was required to allocate a 256-entries oracle buffer. The secret has to be used as an index of the corresponding entry, therefore as the value of the extracted byte can range from 0 to 255, it is mandatory to have 256 entries.

6. Once this allocation of oracle[secret] is done, a few more useless instructions are speculatively executed before the CPU finally issues the error. When issuing the error, it realizes that it has speculatively computed values that it should not have. Therefore, it will discard all the changes made during this speculative execution phase. Doing so means that it will also reload the oracle buffer's slots because they were modified during the speculative execution (and indexed with a secret value). When the error is eventually issued, and the changes are about to be discarded (also discarding the secret value recovered by the attacker), it is possible for the attacker to call an exception handler, which will probe the time it takes the CPU to access each oracle slot. The CPU will access all the slots to discard their content because the oracle buffer is considered as a single entity which has been modified during speculative execution. Therefore, the only solution to recover the secret before it is destroyed, is to use the exception handler to regain control during this phase. However, none of the oracle slots were modified during the previous operations, except one: oracle[Sc]. This slot is actually residing in the L1 cache while the other slots are not. Therefore, this secret-indexed slot will be accessed much faster by the CPU. As the attacker can probe the time it takes the CPU to access each slot using the exception handler, he is able to deduce which of the oracle slots is in the L1 cache, and therefore its index is the secret value he recovered. The attacker can then read the secret value by reading the index of the slot with the shortest access time. This probing of the CPU access time, while it tries to discard the changes made during speculative execution, corresponds to a Flush+Reload [26] covert channel as described in section 2.2.2, using an encoding technique where the secret value corresponds to the address of the only cache line evicted by the attacker during the transient phase.

On top of the original Forshadow attack that has just been described, there are also several variants of the attack. The two most representative ones are Foreshadow-NG [42] and Foreshadow-L3 [43]. The former consists in two very close variants to the initial Foreshadow attack, namely Foreshadow-OS and Foreshadow-VMM. Foreshadow-OS can be used to read any cached contents located at the physical address pointed by a (Page Table Entry) PTE. To do so, the attacker waits for the OS to clear the PTE present bit in some PTE entry when it swaps a page out of memory to disk. Foreshadow-VMM on the other hand leverages a page

fault when a guest malicious virtual machine clears its own access rights to its own memory page to create a transient window. By exploiting Foreshadow, it is then possible to read any cache physical memory on the system, including data belonging to other VMs or the hypervisor. Lastly, Foreshadow-L3 widens the application of the initial Foreshadow attack, that was limited to the L1 cache, and extends it to target the L3 cache as well, thus negating the Foreshadow-L1, Meltdown and Spectre in-silicon mitigations.

2.3.2. Microarchitectural Data Sampling attacks

In 2019, some new attacks have been disclosed by researchers. These attacks are known as the Microarchitectural Data Sampling vulnerabilities (or MDS). Originally, there were three MDS vulnerabilities: ZombieLoad [44], RIDL [12] and Fallout [45]. These MDS attacks were developed after the Foreshadow attack and its variants. The MDS attacks exploit the presence of information in the internal buffers of the CPU micro architecture that are represented in Figure 5. Typically, the victim will be run first, thus leaving information about the instructions executed inside the inner micro architectural buffers. Then the attacker's code will be run. This code will use instructions that cause the usage of the same buffers used by the victim before. As the information from the previous execution (the victim code) is still present in the internal buffers, the CPU will speculatively consider this data as belonging to the attacker. As a result, an incorrect transient value forwarding happens. This is the basic principle of the Micro architectural Data Sampling attacks.

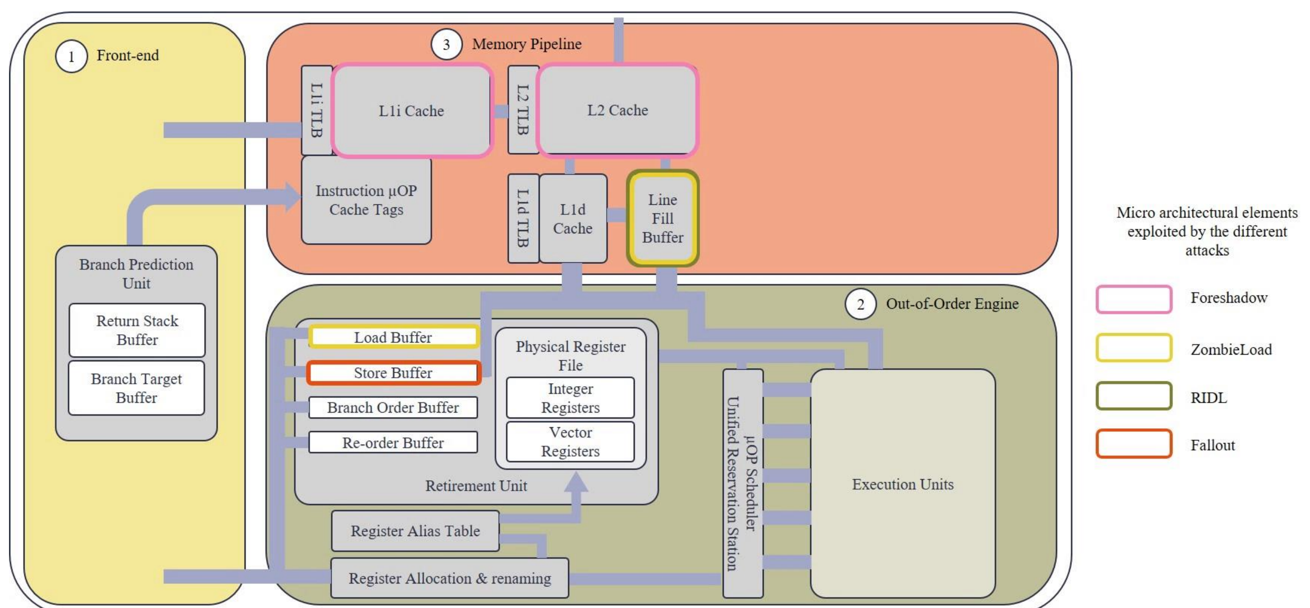


Figure 5. Architectural view of the different elements exploited by the presented vulnerabilities[2]

[2] Figure concept by @themadsetphan, available at: https://www.vusec.net/projects/crosstalk/, [accessed 25 Apr 2022]

2.3.2.1. ZombieLoad

The very first disclosure of this attack was a demonstration. It showed a user, browsing the Internet normally, and a terminal where the ZombieLoad attack was running. The attack was able to recover every action the user made on the web-browser, every website visited, every element clicked. This attack targets user space applications, kernel data and SGX enclaves. The victim has to be on the same physical core as the attacker, but on a different logical core (typical simultaneous multithreading scenario). What was interesting by the time this vulnerability was released is that it worked on fully mitigated systems (mitigated against Foreshadow and Meltdown). Thus, it showed the limitations of these mitigations.

ZombieLoad takes advantage of the load buffer (this has not been proven by the authors yet and will therefore not be detailed in the mechanisms of the attack) and the line fill buffer. The load buffer (see Figure 5) is a small micro architectural element, which is used whenever there is a load operation that is carried out. The CPU will reserve an entry for that specific load operation inside the load buffer. The attack also exploits the behavior of the line fill buffer (see Figure 5). This buffer is working closely with the load buffer. When a load operation is issued, and the target address is not in the L1 cache, then the CPU will reserve an entry for that load in the line fill buffer. This buffer thus acts as a queue for load operations that cannot be completed quickly and directly using the fastest memory area. These L1 cache miss load operations will be pending in the Line Fill Buffer until they are later serviced by the CPU, the latter having to go through a long process of memory checking.

These two buffers involve a speculative behavior, which is exploited by the ZombieLoad attack. When there is a complex situation, a fault on a load operation for example, the CPU has to perform a “microcode assist”. It is a small piece of code that is run on x86 CPUs to deal with corner-case situations that cannot be handled natively. The microcode execution will then require the pipeline to be flushed first. However, some instructions might still undergo computation, and have to be processed as quickly as possible. To do so, the load operations remaining in the pipeline are optimistically matched with entries inside the line fill buffer as long as part of the operation’s target address partly matches the address in the entry.

Therefore, a load operation can be carried out with the wrong data, obtained through a missed speculation, in order to accelerate the processing. The load operation issued by the attacker now contains incorrect data, which belongs to a previous load operation inside the line fill buffer. This data can belong to any other security domain from the sibling logical core. It can be any type of data, from any other entity. Of course, all these changes will be rolled back once the computation is over, but as usual, the data transited through the L1 cache and can be recovered using a covert channel, such as Flush+Reload. The data recovered is, totally random however. The attacker cannot precisely leak some particular memory locations. Instead, ZombieLoad can recover any data that has been recently loaded on the sibling hyper thread or even enclave data that is supposed to be inaccessible. ZombieLoad can observe the load operations executed inside an enclave.

2.3.2.2. RIDL

RIDL is another MDS attack. It can target SGX enclaves. The first demonstration of the attack consisted in leaking the root password hash from an unprivileged user, leaking sensitive data from the Linux OS Kernel, and leaking from some Javascript code. The main difference between RIDL and all the MDS and Foreshadow-like attacks, is that RIDL requires no page fault, and even no fault at all. The attack only requires legit usage of the different CPU features. As in the previously presented ZombieLoad attack, RIDL also exploits the line fill buffer's behavior. However, it takes advantage of a different speculative mechanism.

Let us consider a victim code, running normally. During its execution, it performs load and store operations on sensitive and secret data. This data will transit through the internal buffers, including the line fill buffer. The attacker will then allocate a new arbitrary memory page. This process is legitimate and the attacker has the authorization to do it. However, this process also involves a speculative behavior in order to potentially speed up the allocation process. When the attacker creates this new memory page, the CPU will speculatively load a value from memory, in the hope that it belongs to the newly allocated page and thus speeding up the computation. Therefore, the CPU will take one the latest entries in the line fill buffer, which contains the latest cache miss operations. Unfortunately, the data that the CPU will forward to the new memory page does not belong to it, but is rather arbitrary in-flight data coming from the line fill buffer. This data belongs to an arbitrary different security domain. As usual, the CPU will detect that it speculatively forwarded incorrect data, and will rollback all the architectural changes. However, the data transited through the micro architecture (L1 cache) and that cannot be undone. The attacker therefore carries out a Flush+Reload covert channel to recover the secret data. The researchers who developed the attack found out a reliable way to target precise data within the victim program. Thanks to it, they are capable of recovering sensitive information in very diverse scenarios. Using RIDL, it is possible to build a cross-process attack, a cross-VM attack, to carry out a page table disclosure, to attack Javascript code, to leak Kernel data, and ultimately to leak the values of enclave registers in the SGX context.

2.3.2.3. Fallout

The Fallout attack targets another micro architectural buffer inside the CPU: the store buffer. The store buffer is a small structure that stores copies of recent stores made by programs to memory. Every time an application writes to a data variable, that write goes through the store buffer. The store buffer is central to several features that were introduced in the previous sections. Firstly, it enables speculation and out-of-order execution by allowing stores to memory to be speculative. Thus, the store operations only exist in the store buffer until they are confirmed at the architectural level. Then, the store buffer also enables store-to-load forwarding thanks to which load operations in a program can re-use older store operations that transited through the store buffer. This aims to make the load operations to a recently stored memory area faster.

In the store buffer design, entries are tagged as containing valid addresses and/or valid data. In some particular situations, it happens that entries containing data from previous stores may speculatively be forwarded as matching with more recent loads that are falsely seen to depend upon them. For this reason, it is possible for an attacker to monitor all the recent

stores that were performed by other program, the operating system, and virtual machines running on the same thread… Moreover, store buffers are partitioned between 2 sibling hyper-threads of the same core which means Fallout can be carried out across two different hyper-threads (the attacker is in one thread, and the victim is on another one). The Fallout attack is composed of three distinct steps:

1. The attacker performs a faulty load instruction. The fault can be caused by an access to a forbidden memory area for example.
2. This fault will cause a pipeline flush, because of the IT routine that is being called in order to deal with the corner case situation. Some sets of instructions have to be injected in the pipeline to solve the faulty situation, so all the pending instructions have to finish execution and the pipeline has to be flushed. As the execution of the pending instructions has to be as fast as possible, there are some speculative behaviors that exist inside the store buffer. The in-flight instructions currently being processed are matched optimistically by the processor. In the case of the store buffer, if there is still a pending load operation in the pipeline, the processor will match it with older entries contained in the store buffer (store-to-load forwarding) as long as only a part of the destination address matches.
3. The load operation injected by the attacker got speculatively matched with a previous store operation, thus forwarding an incorrect data value to the load operation. This will of course be discarded later by the CPU at the architectural level, however, the data transited through the cache. Therefore, the attacker ultimately carries out a Flush+Reload covert channel to recover the sensitive data.

Fallout can therefore recover secrets through all the usual security boundaries.

2.3.2.4. Load Value Injection (LVI)

LVI [46] has been developed following the initial MDS attacks. It novelty resides in its capacity to inject a transient value inside a victim code, instead of recovering it. The objective of the LVI attack is mainly to hijack the result of a « load » type operation. One of the initial hypotheses is that the attacker is able to cause an arbitrary software fault when the victim uses a load operation. The objective of the attack is therefore forcing the victim to compute attacker-controlled data transiently. The attack's behavior will now be described. One can consider a victim application that issues load operations during its normal computations. If a fault, such as a page fault (accessing an unauthorized resource) for example, is to happen during one of these load operations, the CPU will cause a pipeline flush to handle this corner-case situation.

Meanwhile, it may also transiently and erroneously forward a value to the faulted load operation. The value is generally coming from some micro architectural inner caches and forwarded during the pipeline's flush to make it as fast as possible.

Therefore, if the attacker fills the micro architectural resources with its own data, then the CPU will incorrectly forward attacker-controlled data to the faulted load operation. This causes the victim to transiently compute on incorrect data chosen by the attacker. Analogously to the Foreshadow attack, the transient computation's results will be discarded

eventually. However, it is possible for an attacker to encode secret values during the transient execution to recover them, or to redirect the execution flow towards an attacker-controlled gadget to carry on the attack.

2.4. Derived micro architectural attacks

The main micro architectural attacks and their variants have been presented in the previous sections. Since the release of the initial attacks, many more attacks have been published in the literature that all have their own specificities. These specificities can be either the targeted victim, the applicative scenario, a slight modification of the initial attack, or a combination of several different techniques. This section presents a few examples of other micro architectural attacks that leverage effects discovered in the initial attacks but with a slightly different approach. This section is not to be considered as an exhaustive listing of all the existing micro architectural attacks in the literature, but rather as an example of what micro architectural attacks can do when combined with other techniques, and to what extent they are dangerous in real-world settings.

The most exploited micro architectural elements in the literature is speculative execution. GhostKnight [47] is a good example of combinative attacks, as it leverages a variant of the Spectre [31] attack. However, they use the speculative execution to cause illegitimate accesses in the DRAM memory. The objective is to provoke an effect similar to the Rowhammer [48] attack. If the DRAM is “hammered” enough during speculative execution, then a bit flip can happen inside the DRAM. By combining this effect and previous micro architectural attacks, the authors have been capable of crossing the boundaries of a trusted execution environment. They also managed to obtain a controllable signature for the targeted 1024-bit RSA algorithm. Similarly, the Spoiler [49] attack leverages speculative execution to improve Rowhammer and cache attacks.

Works like CacheOut [50] show a more recent evolution of the micro architectural data sampling attacks. By refining the techniques detailed in the initial attacks, the authors managed to improve the attack protocol and cause more severe leakages. The MDS attacks are based on the recovery of information inside the line fill buffer (LFB) when there is a fault or an assisted load operation. These initial attacks could not control the information leaked and were rather opportunistically leaking what transited through the buffer. CacheOut proposes a technique to force evictions inside of a given L1-D cache set to control what information will reside inside the LFB when carrying out the attack. Using this technique, it is possible to leak specific bytes of interest inside the victim’s address space, to read entire victim memory pages, to de-randomize the kernel’s address space layout (e.g., defeating Kernel Address Space Layout Randomization or KASLR), and even to attack Intel’s SGX and cross its security boundaries to recover the memory content of an enclave. This work has also been improved further with SGAxe [51] that proposes to leverage CacheOut to break the enclave’s integrity and confidentiality by recovering the secret attestation key. This enables the attacker to create malicious enclaves and pass them as genuine and attested enclaves.

Another interesting work is CrossTalk [52]. This work shows how reverse-engineering efforts can lead to the discovery of new leakages. The authors created a test-bench to inspect the detailed behavior of the most complex instructions of the x86 ISA. By doing so, they discovered the existence of a buffer, namely the staging buffer, which is used for storing sensitive data such as the output of the hardware digital random number generator (DRNG). This buffer is shared among all the physical cores present on the CPU. By leveraging a RIDL-like attack, it is possible to recover the information contained in this buffer and compromise the system's security. This attack can be used for example to recover the private key of an SGX enclave that runs on a separate physical core.

Many existing attacks abuse other mechanisms such as the directional branch predictor in BranchScope [53] and Bluethunder [54] or the return stack buffers in the ret2spec [35] attack. This shows that there are potentially as many micro architectural attack paths as there are shared micro architectural shared resources inside CPUs.

The examples above show how diverse and potent micro architectural attacks are when combined and refined. This leads to the development of a multitude of specific attacks that leverage the basic variants presented in the previous sections. These examples also show the importance of methodology and reverse-engineering efforts in order to discover previously over-looked vulnerability, as the micro architecture can be compared as a black-bock system in proprietary architectures like in Intel, ARM, or AMD CPUs.

2.5. Classical timing attacks

Because of performance optimizations and other factors, cryptographic algorithms generally perform computations in non-constant time. This is especially true for algorithms that use low-level arithmetic operations that tend to yield different timing behaviors based on the input. Non-constant timing means that there is a possibility to discriminate different scenarios, and thus infer information about the operations being run based on timing measurements. Indeed, if secrets are involved in such non-constant timing operations, it is possible to gather sensitive information that can then be exploited using a statistical analysis. In the case of cryptographic algorithm, it can even lead to the total recovery of the secret key, as demonstrated by Kocher [9] in 1996. This is considered the first work mentioning timing side-channel attacks. Kocher's initial results proved to be practical as demonstrated two years later by Dhem et al. [55] by carrying out a timing side-channel attack against an RSA algorithm implemented on a smart card.

Based on Kocher's initial findings, timing attacks have been developed and improved over time. To carry out such attacks, the target algorithm must perform operations that have a run time somehow depending on the secret (or a part of it). Several mitigations are based on this idea. These defenses try to remove any timing dependencies on critical or sensitive data to make timing attacks harder or even unfeasible. This is the case of noise injection mitigations for example. Earlier defenses were based on constant-time execution due to careful balancing of instruction paths in the program. However, such an approach is no more efficient when micro-architectural optimizations such as speculative execution are used.

Many other examples of timing attacks followed Kocher's discovery. A wide range of cryptographic algorithms and hardware systems has been targeted and compromised using this technique. The RSA has been a primary target. Works such as Schindler's [56] present timing attacks on specific RSA implementations that use the Chinese Remainder Theorem. Brumley and Boneh [57] demonstrated that the famous OpenSSL crypto library's RSA implementation was also vulnerable. The threat of timing attacks is not limited to the RSA however. Hevia et al. [58] demonstrated that timing attacks could be leveraged to recover the hamming weight of the secret key against a data encryption standard (DES) [59] algorithm. Timing attacks have also been demonstrated on SSH protocol [60]. There are many more examples of timing attacks in the literature showing the seriousness of the threat that these attacks represent. They have evolved and improved ever since the initial article and led, among others, to the development of Spectre and Foreshadow. The micro architectural attacks and covert-channels studied in this document are therefore an evolution of the timing attacks initially reported by Kocher [9] that use the same idea and timing measurement principle to infer secret information.

2.6. Software-based Hardware attacks

Micro architectural attacks can be considered as hardware attacks as they exploit hardware behaviors from the micro architectural hardware elements. Moreover, as they are conducted remotely without requiring any physical access to the target, they belong to the category of remote-hardware attacks. Software-based hardware attacks are traditional side-channel and fault injection attacks conducted remotely and leveraged using software code [61]. They do not require any physical access to the target either. For this reason, software-based hardware attacks also belong to the remote hardware attacks category, alongside the micro architectural attacks. Software-based hardware attacks can be divided into two main categories: software-based fault injection attacks, and software-based side-channel analysis attacks.

Software-based fault injection attacks can be categorized based on the location of the fault created. Rowhammer exploits are used to cause bit flips inside of DRAM memory cells by repeatedly accessing a given DRAM location. This causes a "hammering" effect that causes the surrounding DRAM locations to be disturbed and even corrupted. Several works have used Rowhammer exploits to carry out a wide variety of attacks. Generating random bit flips [48, 62] or cause targeted bit flips in cryptographic algorithms [63], privilege escalations [64, 65], or carrying out a full side-channel attack on an Open-SSH RSA algorithm [66] are some examples. FPGA-based power glitches are also part of the software-based fault injection category and can be leveraged to cause denial of services [67, 68] or bit flips [69]. It is also possible to carry out power or clock glitch injections to perform remote side-channel analyzes on cryptographic algorithms. ClkSCREW [70], VOLTpwn [71], Voltjockey [72] and Plundervolt [73] leverage power/clock glitches to compromise AES and RSA cryptographic information using side-channel analyzes. Finally, the delay-line in some processors is vulnerable to glitch injection and can lead to cryptographic key extraction as shown in [74].

Software-based side-channel analysis consists in finding and subverting on-chip or onboard sensors to act as a side-channel bench to collect information on the system's power

consumption. With this information, it then becomes possible to carry out a traditional side-channel attack. The sensors collect power traces remotely using software code. These attacks therefore require the injection of a malware code on the targeted platform. Software-based side-channel analysis is mainly composed of FPGAs-based power side-channel attacks. These attacks leverage sensors embedded on FGPAs to collect information on a victim running on the same FPGA. This method has been implemented to carry out side-channel attacks on cryptographic algorithms in works such as [75], [76] or [77]. FPGA-based power side-channel attacks can also be used as covert-channels as demonstrated in [78] and [79]. The analog-to-digital converters contained in some chips are also valuable targets for these attacks as they can be used as power sensors to carry out side-channel analyzes. The works done in [80] and [81] have demonstrated that these attacks are practical on IOT devices. Finally, Platypus [82] and SideLine [83] have shown that it is possible to find some inner elements within CPUs that can be subverted into sensors to gain information about the system's power consumption even if the elements exploited were not primarily designed as sensors.

2.7. Existing mitigations

There are three levels where mitigations against micro architectural attacks can be implemented: the user-space level, the hardware level, and the system level. System and software level are useful mainly for protecting already deployed vulnerable hardware platforms, and reacting rapidly to newly found micro architectural vulnerabilities. However, their cost in terms of performance is generally high. In comparison, hardware defenses are less costly in terms of area and performance, but they can only be deployed in new products or on reconfigurable platforms and take time to develop and implement.

Intuitively, one could think that the best way to prevent micro architectural attacks would be to remove resource sharing as much as possible. However, this comes at a large cost in terms of performance. Moreover, for some applications, such as cloud environments, the primary objective is to share resources. The case of personal computers is similar to this issue: eliminating resource sharing would not remove the threat as users willingly run third-party software and code.

Another mitigation idea would be to modify the process scheduling. Indeed, it is possible, for example, to run processes that belong to different security domains only on different CPU cores, as demonstrated in [22] and [84]. This is effective for micro architectural attacks targeting victims located on the same core as the attacker. For the case of cross-core vulnerabilities, running processes on a different CPU would remove the vulnerability. However, these solutions are not consistent. First, for the solution of running processes on a different core or CPU, there need to be the available hardware resources. This leaves simpler and smaller systems vulnerable. Then, it again comes at the cost of reduced performance, as it makes the optimization of resource usage harder. Finally, in some threat models, especially when the system owner is not trusted, these solutions might fall short. The system owner can choose on which hardware each process will run, and can even target a specific process and isolate it on a single core where only the victim and an attacker program would run.

For these reasons, removing resource sharing and working on scheduling are not persistent solutions. It is therefore mandatory to investigate other mitigation possibilities. This section presents the state-of-the-art of mitigations against micro architectural attacks and covert-channels classified according to the layer they are implemented in (user-space, hardware, and system).

2.7.1. User-space mitigations

When trying to protect against side-channel attacks on cryptographic algorithms, the standard protections consist in removing the timing differences and the data dependencies. Indeed, several works propose mitigations based on constant-time implementations of cryptographic algorithms [85-88]. Another idea that has been proposed in the literature is to eliminate secret-dependent computations. Works like [89-91] propose RSA implementations that do not perform any secret-dependent operation. These implementations are therefore resilient to cache side-channel attacks. Another mitigation that has been proposed consists in preloading all data into the cache before running the algorithm. As demonstrated in [92], this defeats some single core micro architectural attack variants exploiting cache misses. However, D. Gruss demonstrated that this is ineffective against Flush+Reload or Prime+Probe cross-core cache covert-channels [93]. Moreover, writing a constant-time code by hand is tedious and complex. Some works like Constantine [94] propose automated solutions to obtain a side-channel resilient code. Constantine is a compiler-based system that linearizes secret-dependent data and control flow. Even if this work makes constant-time programming easier to achieve, some micro architectural covert-channel variants can still be applied.

The effort on constant-time programming has been carried over by Cauligi et al. In [95], a semantic is proposed for defining a constant-time extension to speculative execution. They also implement Pitchfork based on this semantic, which is a symbolic analysis tool capable of detecting Spectre-type attacks. Intel also proposed Retpoline [96], a method to bypass the indirect branch predictor. This method enables to mitigate Spectre variants that rely on indirect jumps and calls to redirect the speculative execution to a targeted branch. Similarly, recent efforts have been put on mitigating micro architectural threats on RISC-V cores. Bălucea et al. [97] took inspiration on Retpoline to propose software mitigations against two variants of the Spectre attack by modifying the behaviors of indirect jumps and indirect calls to remove any speculative attack while maintaining the same functionalities. They proved the efficiency of this method on the Berkeley Out-of-Order Machine (BOOM) [98], a RISC-V core created by the University of California at Berkeley.

Mitigations at the user-space level remain very limited and not completely effective. Even if they do not entirely solve the vulnerability, they need however to be considered and applied as attacking a constant-time and data-oblivious algorithm is more complex and costly for the attacker. This also demonstrates why communication and sensitization about micro architectural vulnerabilities is important, as it is the programmer's responsibility to implement a robust code and avoid unnecessary leakages.

2.7.2. Hardware mitigations

2.7.2.1. Modifying leaky instructions

The hardware layer being the core of the issue, it will lead to the most effective mitigations. Multiple micro architectural cache covert-channels rely on the timing differences introduced by some instructions. This is the case for Intel processors where some cache covert-channels leverage the *clflush* instruction. *Cluflush* produces different timing behaviors for cached and uncached regions. This is also the case for prefetching instructions as they leak through timing differences. More generally, a first hardware mitigation consists in removing the timing differences introduced by the instructions related to optimization mechanisms. Micro architectural attacks run software that places the CPU in a state where optimization mechanisms are called. This causes the use of leaky instructions that produce timing differences based on the state of the hardware elements. Therefore, proposing constant-time instructions for optimization mechanisms is an effective solution. Gruss et al. studied the case of the Intel *clflush* instruction in [99, 100] and proved that the performance cost is small. Removing timing differences in instruction sets is therefore an effective mitigation idea.

Moreover, it is also possible to introduce new mitigation-oriented instructions. N. Wistoff et al. proposed the addition of the fence.t instruction in the RISC-V ISA in [101, 102] as a mitigation to cache side-channel attacks. This instruction introduces the possibility to cause a flush of all the leaky micro architectural resources and leads to a context switch. When used correctly by the programmer, this instruction can mitigate many micro architectural cache covert-channels. However, it does not prevent all the existing variants from being carried out, and relies solely on the programmer's correct usage of the instruction. Moreover, it does not provide protection against potentially new covert-channels that have not been discovered yet. Finally, this addition only works for mitigating single core micro architectural attacks as a multi-core attack could extract the secret information before the fence instruction is executed, during the victim's execution.

2.7.2.2. Proposing resilient micro architectural memory designs

Many micro architectural attacks are concluded by a cache covert-channel to extract the stolen secret data. The more recent MDS attacks exploit the leakages coming from the small cache-like memories related to optimization mechanisms. Therefore, working on the design of the leaky micro architectural memory elements that are exploited by the attacks provides effective mitigation. As the cache is the most well-known and exploited micro architectural memory element, it is also the most studied in the literature. RPcache and PLcache [103] are two cache architectures that intend to defeat cache side-channel threats. Wang et al. propose a cache design that is partition-locked (PLcache) and proceeds to random permutations (RPcache). PLcache introduces a locking of the cache lines that makes them impossible to evict. This causes the attacker to be unable to carry out any cache covert-channel based on the observation of the cache misses obtained (such as Prime+Probe and Flush+Reload). RPcache adds a randomized mapping between cache sets and physical addresses for every process. This causes the attacker to be unable to carry out the priming phase as each process has its own cache sets. However, these two designs have been proven

to be insufficient as Kong et al. [104] demonstrated how they could be defeated using specific instructions named informed loads. Another cache design has been proposed by Liu et al. [105] that improves the RP cache's concept further. Indeed, the mapping between cache sets and addresses is randomized at runtime dynamically. It does not mitigate all the leakages, but prohibits many cache covert-channels from being carried out.

More recent approaches like CEASER [106] propose a specific mapping of the cache to mitigate eviction-based attacks. The main idea of this mapping is to translate the physical line-addresses into encrypted line-addresses. The cache is then only accessed using the encrypted addresses, making harder for an attacker to distinguish the cache mapping correctly. Another recent cache design approach is IE-Cache (for Indirect Eviction Cache) [107]. It aims at increasing the cost for an attacker to detect an eviction set. The main idea is to modify the replacement policy based on cached memory addresses and a secure-indexing function. The cache line that has been replaced last is also evicted. IE-Cache introduces a random memory-to-cache mapping. This causes perturbation in the usual determination of eviction sets as it introduces cache lines that are never evicted. This cache architecture has been further refined to optimize its energy cost [108].

Purnal et al. [109] showed that randomized caches however are not immune to micro architectural cache covert-channels. Indeed, they develop Prime+Prune+Probe, a variant of the Prime+Probe covert-channel that defeats randomized caches such as CEASER or ScatterCache [110]. This novel technique uses probabilistic models to build a reliable eviction set, even if this was assumed impossible because of randomization. In response, Thoma et al. [111] developed ClepsydraCache. It proposes the introduction of a cache decay concept, linking every cache entry to a Time-To-Live (TTL) value. By a combination of a dynamic scheduling of the TTL and index randomization, they manage to defeat micro architectural attacks that previously bypassed randomized cache mitigations.

Another mitigation possibility when working on cache-like memories is to reduce or even remove the resource sharing directly at the hardware level. Cache covert-channels such as Prime+Probe take advantage of the fact that the victim and the attacker are sharing cache sets. Works like [112] and [113] propose cache architectures that prohibit different processes from sharing a cache set using dynamic mappings. The work of Sanchez et al. [114] proposes a cache design that decouples cache ways and cache associativity by providing associativity by modifying the number of replacement candidates instead of the number of cache ways in each set. This design impedes eviction-based covert-channels such as Prime+Probe to be carried out. The non-monopolizable cache proposed by Domnister et al. [115] consists in preventing any process from allocating enough cache lines to observe collision with other processes. This prohibits cache covert-channels based on cache contention introduced by the attacker.

The Last Level Cache (LLC) has been studied more in details in the recent years and several mitigations have been proposed to hinder micro architectural cache covert-channels on it. SwiftDir [116] and CacheBar [117] are two approaches aiming at securing the LLC. SwiftDir enforces protection by introducing a constant-time latency when serving requests from the LLC. Additionally, write-protected data are forced to be severed from the LLC directly. These modifications efficiently secure cache coherence against cover-channel attacks. CacheBar is a memory-management subsystem implemented in the Linux kernel that focuses on

disabling the line-sharing in the LLC to prevent Flush+Reload covert-channels. The mitigation dynamically manages physical memory pages that are shared among different security domains in order to ensure that no cache lines are shared between there pages. This technique is also effective to prevent cross-tenant Prime+Probe covert-channels on the LLC.

The cache is not the only vulnerable micro architectural shared memory element. Spectre attacks also take advantage of small buffers that are cache-like memories to create a leakage. Tan et al. [118] introduced a new design for the branch target buffer that detects suspicious activities and prohibits resource sharing in case of a positive detection.

In an effort towards globally mitigating the micro architectural threats, Escouteloup et al. [119] proposed to study the build blocks to construct a core that would be immune to micro architectural timing threats. They provide two examples of implementation using the RISC-V ISA with an added extension providing several supplementary instructions to support the addition of the Dome. It also requires the addition of multiple hardware elements.

2.7.3. System-level mitigations

2.7.3.1. Hinder the exploitation of timing elements

Analogously to the hardware and software layers, the main idea for a micro architectural mitigation is to introduce constant timings and remove data dependencies. Some works like [120] propose frameworks that transform a given code into a more side-channel resilient version using conditional CPU operations. It is also possible to develop libraries that enable constant time computations, such as [121] proposing such a library for fixed-point numeric operations.

As detailed in the previous sections, most of the micro architectural attacks and covert channels rely on timing measurement. The ideal scenario is to have access to a hardware timer that provides accurate measures. Several works [122-124] in the literature focused on rendering these timers less accurate by simulating them to prohibit micro architectural attacks. However, this solution proved to be ineffective according to [125, 126]. More generally, working on mitigations at the timing measurement source is not ideal. Even when the attacker is left without any timer, it is always possible to find another solution such as building its own timer [127, 128] or rely on counting threads [129, 130]. Some other works proposed to make algorithms always having the worst timing behavior [131-134]. This mitigates Evict+Time covert-channels, but fails to protect against Prime+Probe or Flush+Reload as they do not rely on overall differences in execution times but work at finer granularity.

2.7.3.2. Reduce the amount of memory and data shared

When Foreshadow was initially published, Intel’s first-response mitigation consisted in disabling Hyperthreading. Reducing or even removing completely memory and data sharing is an interesting idea as most micro architectural attacks rely on shared hardware resources to operate. However, disabling or removing all the shared resources is not a satisfying option as it drastically reduces the CPU’s performances. In [26], Yarom et al. proposed to remove

shared memory, thus disabling cache-line sharing. However, this is not an interesting solution when evaluating the performance implications.

Another solution would be to remove cache-set sharing. Indeed, some micro architectural cache covert-channels entirely rely on cache-set sharing to be carried out. Shi et al. [135] therefore proposed cache-coloring. This technique consists in the OS using some bits in the physical address to encode the color of the memory page. The main idea behind cache coloring is to map as many memory pages to different colors as possible. This has the benefit of reducing in-application conflicts on the cache. This results in a mapping between physical addresses and cache colors. This mapping can further be controlled by the OS to perform cache reconfiguration. Cache coloring defeats micro architectural covert-channels such as Prime+Probe as it relies on cache set sharing. Indeed, the OS will attribute different colors to different applications to enforce cache isolation. Moreover, some types of caches such as L1 virtually indexed physically tagged caches cannot use cache coloring because each page would have the same color (because the index bits are part of the page offset).

Multiple works implemented and evaluated cache-coloring protections on several platforms: [136] worked on Intel CPUs, [134] implemented cache coloring on ARM CPUs. In response to Intel SGX, Costan et al. developed Sanctum [137]. Sanctum leverages cache-coloring to propose a strong software isolation that also provides protection against micro architectural cache covert-channels on enclaves.

Most micro architectural attacks and covert-channels exploit the presence of data that belong to other processes inside the shared micro architectural resources. A solution to mitigate such vulnerabilities is therefore to remove any sensitive data, or data depending on another process from the leaky hardware resources. One solution is to flush the cache and the other shared micro architectural resources upon context switches. Works such as [138, 139] develop mitigations for cloud applications using cache flushing. Intel's response to the Foreshadow attack also consisted in microcode updates that lead to a cache flush upon context switched [140].

Another mitigation solution is to work at the compiler level. Braun et al. [141] introduced a compiler that makes any marked function constant-time, data-oblivious, and not accessing any shared cache sets. Haehyun et al. [142] leveraged the same idea and developed SmokeBomb. SmokeBomb is a collection of applications that are added during compilation time to find and patch vulnerable code, and detect sensitive data that will be protected by the SmokeBomb's protection features: creating a private space in the L1 cache for each process. The applications proceed to prefetching any sensitive data before the vulnerable code is run. This impedes an attacker from identifying an eviction set. Then during the code's execution, SmokeBomb ensures that the sensitive data remains in the private L1 cache reserved to the victim process currently being run. Finally, it flushes the cache to remove any secret data after the code's execution is over.

2.7.3.3. Detection mechanisms

One last type of countermeasure against micro architectural attacks and covert-channels that has been developed recently are the detection mechanisms. The work done by M. Mushtaq [21] or Akram et al. [143] propose extensive studies of the state-of-the-art of cache side-channel detection mechanisms. Several works propose solutions for static detection of vulnerabilities in software such as [30] or [144]. Several other frameworks introduce the detection of vulnerabilities for cryptographic implementations [145, 146]. [147-149] and provide the possibility to quantify cache leakages. However, detecting vulnerabilities at the software level is insufficient to mitigate the micro architectural threat. Still, the information given by these different approaches are valuable for the programmer in order to correct its code and remove potential leakages. Yuan et al. [150] proposed CacheQL, a detector that quantifies and localizes information leaks on binary code. The framework analyzes the cache side-channel traces produced by production software to pinpoint the leakage sources and enable the developer to correct them.

A more promising type of attack detection technique consists in detecting and preventing ongoing attacks, at runtime. The approaches continuously scan the system for abnormal or potentially dangerous behaviors, and stop the processes implied. Several approaches rely on performance counters to detect unusual behaviors (such as unusual performance variations) that could lead to a micro architectural vulnerability being exploited [151-153]. Some similar approaches chose to monitor cache misses and cache hits to detect threats. [154, 155] leverage this idea to detect micro architectural attacks such as Flush+Reload or Rowhammer. However, [27] showed that monitoring performance counters is an insufficient mitigation. Indeed, these counters cannot be used to detect all micro architectural attack variants, especially the stealthier ones: Flush+Flush and some variations of Flush+Reload. A more recent approach consists in adding machine learning in the automated detection mechanisms to hinder cache side-channel attacks. Approaches like [156], [157] or [158] leverage machine learning for mitigation purposes.

Another solution that has emerged is the use of performance-monitor interrupts [159]. The main idea is to detect the usage of sensitive instructions that are generally used to carry out micro architectural attacks (typically the clflush instruction on Intel CPUs). HexPADS [160] is a system that can detect cache attacks and Rowhammer exploits during runtime execution. It monitors cache misses, cache events, and page faults to detect behaviors that are related to a micro architectural attack. For Intel SGX, [161] proposed a framework to detect covert-channels at runtime. They leverage the TSX extension to build counting threads inside enclaves that detect unusual behaviors such as interrupts.

2.8. Discussion and conclusion

This section presented the different micro architectural concepts that are required in order to understand most micro architectural attacks. The main attacks and covert-channels are then studied and their related variants are detailed and summarized in Appendix 1. The different approaches to mitigate micro architectural attacks that are available in the literature are listed and studied. This section provides an insight of the many attack paths and attack variants that exist when it comes to micro architectural attacks. In response to these vulnerabilities, many mitigation proposition have been made, at different levels, including user-space, system, and hardware levels as shown in Appendix 2 that summarizes the different existing protections and their main characteristics.

Nowadays, all the published micro architectural attacks have one or more countermeasures that have been developed specifically to counter them. Multiple cache architectures and mitigation solutions have been proposed in order to hinder cache covert-channels. Despite the multiplicity of defensive solutions, there is still an existing need for a generalized and durable mitigation proposition that would provide a strong defense against existing and upcoming micro architectural vulnerabilities. Even if there is a large variety of mitigations available, it is not realistic to plan to implement them all on a single CPU in order to ensure the best protection possible. The next section will provide an analysis of the attacks targeting the SGX technology in real-life contexts and aims at providing insight as to whether the attacks themselves are realistic or not. Moreover, this section will experiment on some of the proposed mitigations at the software level, such as Intel's mitigations, to verify their capacity to protect a victim application efficiently in a specific realistic use-case without drastically reducing the CPU performance.

Chapter 3

The case of Intel SGX: the discovery of critical leakages in all high-end CPUs

Now that the state-of-the-art has been established, the first step of the study proposed in this document will be detailed. This section presents the preliminary study of the first micro architectural vulnerabilities that arose on the Intel SGX technology with the publication of the Foreshadow attack. This initial study aimed at showing if the technology was secure in the specific context of an untrusted Cloud Service Provider (CSP). To achieve this, the SGX technology will briefly be introduced first. Then several use cases and proofs-of-concept targeting real-life applications will be detailed. Moreover, the mitigations proposed by Intel as well as some initial software mitigation ideas will be explored. Finally, this section concludes on the viability of the SGX technology in the untrusted CSP model and on the robustness of the proposed mitigations in the "closed" architecture approach proposed by Intel.

3.1. Overview of the SGX technology

Secure remote computation is the problem of executing software on a remote computer owned and maintained by an untrusted party. The service user wishes to have some integrity and confidentiality guarantees when he utilizes the remote computer, as he uploads his personal code and data. The service provider could get access to this data and utilize it without the consent of the data owner. Nowadays, secure remote computations is an unsolved issue. In response to this problem, Intel developed SGX (or Software Guard eXtensions). It aims solving the secure remote computation problem by leveraging trusted hardware in the remote computer, as shown in yellow in Figure 6. During runtime, the SGX hardware protects the data inside the container, thus providing confidentiality and integrity while the computation is being remotely performed.

In SGX, the "secure containers" are named enclaves. These enclaves only contain the private data, and the code that operates on it. One could take the example of a cloud service performing image processing on confidential medical images. This service requires the users to upload encrypted images. The users would send the encryption keys to the software running inside an enclave. The enclave would therefore contain a code capable of decrypting the images, processing the decrypted pictures, and encrypting the results. Another code, placed outside of the enclave, would manage the reception of encrypted images and emission of encrypted results. This technology has a wide variety of industrial applications, including cloud services, the internet of things (IoT), many-party applications… It is therefore

interesting for security and cybersecurity companies, as it would enable them to develop secure software for all remote computation applications.

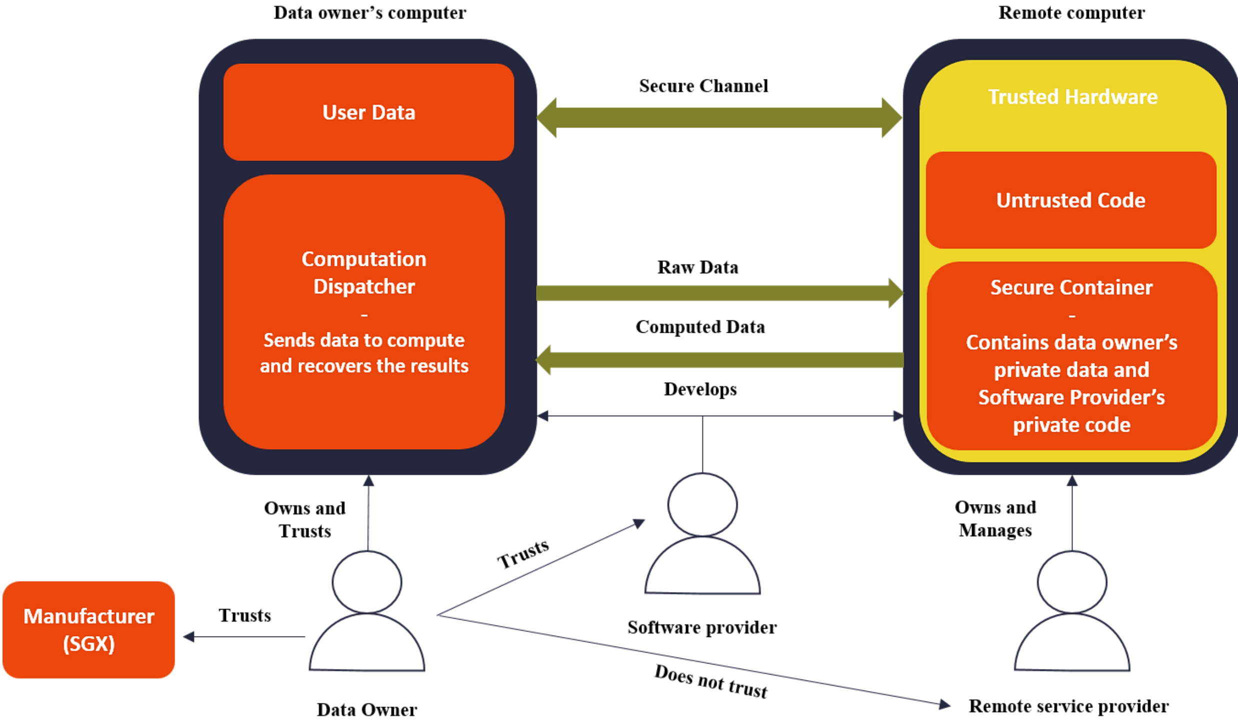


Figure 6. Diagram of a typical secure remote computation setup with SGX (yellow)

An application using the SGX technology is divided into two parts: a trusted part and an untrusted part. The untrusted part is responsible for the communications between the enclave and the rest of the system. This part also creates the enclave. The SGX enclave is considered as the trusted part of the application. All the sensitive data will be managed by the enclave. All the enclave instances are isolated from each other. However, they can interact using several SGX functionalities such as local and remote attestation. The untrusted part will then act as an intermediate. Even if the access commands come from an untrusted memory address, it is impossible for malicious programs to get into the enclave, and manipulate the code or the data stored in it.

The security of the data and code stored inside the enclaves can be improved by using some SGX primitives in the enclave code. SGX provides instructions that allow encrypting anything that goes out of the trusted part using a secret key that is unique to the platform and to its identity. Comparatively to enclave data and code, this key cannot be accessed by any software. Additionally, SGX natively provides integrity and confidentiality protection between the enclave and its DRAM memory area. This is done by a dedicated autonomous hardware unit called the Memory Encryption Engine (MEE) [162] whose role is to protect the enclave memory.

The main advantage of SGX enclaves is that they reduce the attack surface, i.e. the total sum of vulnerabilities that can be exploited to carry out a security attack. Attack surfaces can be physical or digital. The term attack surface is often confused with the term attack vector. The surface is what is being attacked; the vector is the mean by which an intruder gains access. When considering a system not using the SGX technology, there is a large attack surface. A potentially malicious user could either exploit vulnerabilities contained in the OS, in the

VMM, in the hardware or in the application itself to carry out a security attack and recover the application's secret data. The addition of SGX enclaves greatly reduces the attack surface, as it can be seen in Figure 6. When SGX enclaves are used, only the application and the processors might present exploitable vulnerabilities to access application data. Thanks to secure enclaves, malicious code that would subvert the OS, the VMM, the BIOS or the drivers cannot recover the secrets from the application. Indeed, these entities do not have access to the enclave's data and code, even when functioning normally.

The security model of SGX can be beneficial for several reasons. For example, if all claims hold, even a naïve user can safely manage valuable financial accounts thanks to the SGX technology. For companies, employees can handle corporate documents on unmanaged platforms; these documents are protected from theft even if the employee's computer has malwares in it and has multiple security breaches. In the context of cloud services, SGX enables customers to execute workloads in the cloud with the assurance that their data and code are safe from any ill-intentioned party located outside of their workload. Finally, SGX could also benefit application developers that would be capable of deploying protected applications much more easily. This technology could also be used by the industry to develop new products and services, for example: trusted web browser, secure document-sharing application, secure video conferencing application…

Intel announced several security guarantees regarding its SGX technology. However, in January 2018, the Foreshadow attack [11], broke SGX's security boundary and raised many concerns regarding potential micro-architectural leakages present in all high-end CPUs. Indeed, this attack is derived from Meltdown and Spectre that were the first of a long series targeting the shared optimization mechanisms contained in all high-end CPUs. SGX marked the beginning of the generalization of micro architectural attacks to all types of architecture, and to crossing many security boundaries. Studying the case of the SGX technology is therefore crucial in order to understand how the initial Meltdown attack has been adapted to cross the strong isolation provided by SGX. The main idea behind the leakage exploited by the Foreshadow and Meltdown attacks is the starting point of the entire micro architectural trend that has been observed in the literature, and therefore of the study proposed in this document.

3.2. A first case-study: Applying Foreshadow to a SGX secure enclave

A first experimentation has been carried out in order to test the Foreshadow attack and understand its different mechanisms by modifying and replicating it. The idea was to adapt the initial attack to target a use-case that would be similar to a real attacker/defender scenario. Every software manipulation and code development in this section were made using the SGX-Step framework. SGX-Step is an open-source framework created to facilitate side-channel attack research on Intel SGX platforms. It has been created by the authors and creators of the Foreshadow attack. SGX-Step consists of an adversarial Linux kernel driver and user space library that allows to run SGX-enclaves one instruction at a time (single-stepping). The framework is available on GitHub [163], and Figure 7 details its behavior.

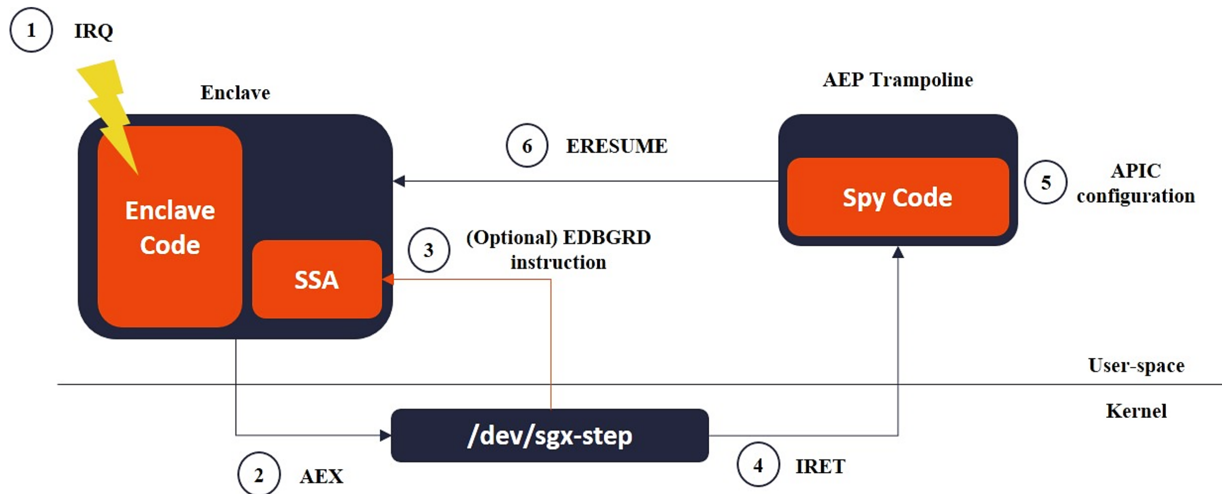


Figure 7. SGX-Step single-stepping mechanism for SGX enclaves[3]

The framework uses the Local APIC Timer, as shown in Figure 7. This timer can trigger an interruption while an enclave is executed. The generation of an interrupt ①, including the APIC interrupt, while inside an enclave causes an Asynchronous exit (AEX) ② of the enclave. This AEX cause the SSA (Save State Area) to be saved. It therefore contains all the values of the different registers within the enclave at the moment the interruption happened. In addition to the APIC Timer interrupt, there is an "APIC event callback list" which defines what piece of code will be executed once there is an interrupt. Using this list, it is possible to execute a custom loadable kernel module every time the APIC timer triggers an interrupt. This module is located in /dev/sgx-step. Within this module, it is possible to insert "attack" code, to dump memory pointers of the enclave for example. On the Figure 7 above, the attacker uses the kernel module to call the EDBGRD instruction ③ and recover the pointer on the enclave's SSA region. Once the kernel module execution is ended, the code returns to user space using an AEP trampoline ④. In ⑤, the attacker can insert spy code, to probe execution or reload timings, and he also configures the APIC timer for the next interruption. This configuration requires setting the value of the "initial-count MMIO register" which defines the amount of time the APIC timer will wait before the next interrupt is triggered. Once the timer is initialized, the ERESUME instruction ⑥ is called, and enclave execution is resumed. This mechanism enables SGX-Step to single-step SGX enclaves.

The very first step of the implementation work consisted in choosing the appropriate RSA algorithm. For simplicity purposes and to speed the process up, it was chosen to use a text book Rivest–Shamir–Adleman (RSA) algorithm [164]. The RSA is an asymmetric cryptographic algorithm. The chosen implementation of the algorithm did not contain any protection; it consisted only in arithmetic operations and used a 16-byte key. All the RSA's code was placed inside the enclave. The main (untrusted) application only called RSA_code (plain) and RSA_decode (ciphered) enclave functions. The Foreshadow attack was then applied to its new target: the 16-byte private key. The experimental setup consisted in a laptop equipped with an Intel® Core™ i5-6440HQ CPU with microcode version 0xa6 (early

[3] Figure concept by J. V. Bulck, available at: https://github.com/jovanbulck/sgx-step, [accessed 23 Jan 2022]

microcode version not containing the mitigations proposed by Intel) running an Ubuntu 18.04 OS.

As a result, the attack was capable of recovering the 16-byte key on every attempt. The RSA was also modified to a 64-bytes version and the attack was still capable of recovering the secret key. This experiment enabled to prove that the Foreshadow attack works on settings close to real-life ones. It is capable of recovering the secret keys of a security algorithm coded inside an SGX enclave, and thus to break its confidentiality and integrity.

After adding some of the state-of-the-art protection for RSA algorithms (padding [165], square and multiply always [166]…), the Foreshadow attack was still capable of recovering the secret keys. As long as the keys are allocated inside the enclave, and that the attacker knows where in memory these keys are located (starting assumption for the Foreshadow attack), then the attack can reliably extract them, whatever the rest of the targeted algorithm is composed of. All the arithmetic protections seemed useless here as the keys are extracted as soon as they are allocated inside the enclave.

This experimentation led to the conclusion that the Foreshadow attack was indeed practical, and could potentially target real-life applications. Moreover, it showed that traditional side-channel protections are useless against this new kind of threats. This means that new protections need to be designed to face the micro architectural threat. The next section details Intel's efforts to mitigate the leakage and our first mitigation ideas at the software level.

3.3. Initial software mitigation ideas and robustness evaluation of the existing countermeasures

3.3.1. Intel proposed mitigations

This section is studying the existing mitigations proposed by Intel, to protect enclaves against the Foreshadow and related attacks.

Initially, Intel proposed three main solutions to address the L1TF and MDS vulnerabilities. The first solution is an update to Intel's microcode, a microcode patch. This patch adds a new microinstruction to the SGX set, causing a flush of the L1 cache each time an application enters/exits an enclave. This causes the Foreshadow and some related attacks to mostly fail because the secret is erased from the L1 cache before the attacker can recover it using the Flush+Reload covert channel.

The second protection proposed by Intel consists in deactivating the Simultaneous Multi-Threading (SMT) optimization. This is effective because almost all the L1TF and MDS attacks are working for the typical multithreading scenario (the attacker is one thread, and the victim another one). Hyperthreading (Intel's proprietary implementation of the SMT) is the root cause of most of the current attacks on SGX. Thus, deactivating it solves most of the problems Intel CPUs are facing. However, it drastically decreases the CPU's performances, which might not be acceptable for some cloud providers or other companies. Moreover, some attacks and variants of the Foreshadow attack can still work while the SMT is disabled. Therefore, even if this solution is very effective, it does not solve all the issues at all.

Moreover, this disabling is under the control of the BIOS and the enclave has no control over it.

Intel's last countermeasure consists in adapting its CPU designs to impede these attacks. These "in-silicon" mitigations are very effective and totally block the attacks they aim to protect against. However, these upgrades cannot be deployed on already released CPUs. Moreover, some new attacks are even exploiting these in-silicon defenses (one of RDIL's variants for example) to create new vulnerabilities and security flaws. This solution is very useful to prevent today's attacks from affecting tomorrow's CPUs. However, it creates new attacks and vulnerabilities, and does not help companies with CPUs that do not possess these upgrades. Therefore, this solution alone is far from being sufficient.

Depending on the applicative scenario considered, the effectiveness of these proposed mitigations can be qualified. Let us take the example of an attacker who is mastering the system as for instance a malicious or compromised CSP. The victim places an encryption service in an enclave, running in the cloud. Even assuming that the hardware has been patched, and runs the latest mitigations, the victim is still unsafe as the CSP masters all the aspects of the software and the hardware alike. For example, it is possible to roll back the microcode update, making the system vulnerable again. Moreover, there is currently no way for an enclave to ensure that SMT is activated or not. Therefore, even if it is disabled for security purposes, while the enclave runs, the malicious CSP can still activate it again, and the enclave would be unaware about it. Even worse, considering the provider is mastering the OS and/or the hypervisor, it is possible to reduce the signal to noise ratio drastically, thus making the attack more reliable and dangerous for the victim. For example, it is possible to pin the victim's execution on a precise core, where nothing else runs apart from the attacker's code and the targeted application. There are also some variants to the Foreshadow attack that were developed later requiring administrator privileges but being more dangerous depending on the scenario. One of the variants, for example, could recover enclave secrets without even executing the victim enclave [42].

3.3.2. Adding random temporal loops

In an attempt to hinder the vulnerability at the software level, we developed several mitigation ideas and propose an analysis or their potential effectiveness in the following sections. Our first mitigation developed was the "random temporal loops" mitigation. This idea originated from the behavior of the Foreshadow attack. It uses a very small transient window to carry out its attack. Therefore, it was expected that, if random temporal loops are inserted at critical sections of the code, the Foreshadow attack would miss the transient execution window and fail. The idea was to create some "for" loops with an incrementing counter which would stop at a randomly generated value. The optimal placement for these loops was the most determinant factor to determine their effectiveness. Several different positioning have been tried out. The best option seems to place two random temporal loops, around critical sections where secret data is being computed, and/or stored in memory.

Several experiments have been carried out and proved that random temporal loops are capable of reducing the attack's precision and reliability. While implemented both on the first proof of concept, and on the RSA use case, random temporal loops enabled to prohibit the Foreshadow attack from recovering all the secret bytes. For example, when the random

number was high enough, the Foreshadow attack would reliably fail in recovering 8 of the 64 secret bytes. For testing purposes, the same experiments were carried out using fixed values for the loops (instead of random values). This showed that the “randomness” was not necessary to prohibit the Foreshadow attack from recovering secret bytes, but that it was more efficient. Using fixed temporal loops, it was possible to prove that the Foreshadow attack failed in recovering 8 bytes out of 64 for 1 000 000 000 iterations in the loop (which takes about one second to complete).

This study enabled to conclude that the random loop mitigation idea was partially effective as it does not stop the attack completely, but can impede it from recovering complete secrets. However, it also proved that the performance impact of such a mitigation is huge, as the amount of iteration required to be effective is very large. Also, as the experiments were carried out inside the SGX-Step framework, it is almost impossible to tell whether the mitigation prohibits the Foreshadow attack itself, or impedes the framework. Therefore, temporal loops have been considered an interesting solution because timing disruptions seem effective against the Foreshadow attack, even if in its current form, it cannot be used as a functional mitigation.

3.3.3. Flushing the cache

The second mitigation idea developed is based on Intel’s microcode patch. This patch has the effect of flushing the L1 cache every time an application enters/exits an enclave. Therefore, it has been decided to develop a mitigation based on this assumption to prove whether it is truly effective, and if it is possible to understand precisely why it prohibits the Foreshadow attack. To mimic the microcode patch, an assembler instruction, INVLPG has been used. This instruction causes a flush of the entire L1 cache when used. The INVLPG instruction was called using inline assembler in the enclave’s C code.

The INVLPG instruction has been used to carry out several tests, both on the proof of concept, and on the RSA use-case. Similarly to the random temporal loop mitigation, the position of the INVLPG instruction is the only factor determining its effectiveness. Different positions have been tested inside the enclave codes. This experiment showed that the only position where the INVPLG instruction is effective is when the enclave code calls EENTER or EEXIT instructions. However, during asynchronous exits (AEX) happening when the enclave is interrupted, flushing the cache would also have been useful; still it is not possible to impose such a behavior to the user’s code. Only the microcode patch can do it.

By placing the INVLPG after every instruction causing an entry/exit of the enclave, one can prohibit the Foreshadow attack from recovering any secret. In addition to more reliably prevent the attack, it is possible to flush the L1 cache every time secret data is computed. By doing so, one ensures that secret data is never residing in the L1 when the application is outside of the enclave, and thus, nothing can be recovered. This works particularly well both on the proof of concept and on the RSA use case. The attack is totally impeded by the INVLPG instruction provided it intervenes with the correct timing.

3.3.4. TSX transactions

In an article published in the 2018 [167], researchers introduced Transactional Synchronization Extensions (TSX) transactions as a way for defending against side channel attacks. As the Foreshadow and related attacks are considered side-channel attacks, it has been thought that TSX transactions could also be used in order to protect against this new class of attacks.

TSX is an extension to the x86 ISA that adds hardware transactional memory support. It is a hardware functionality, and has to be used with inline assembly code in order to implement a TSX transaction inside a C code. This functionality enables the developer to define arbitrary sections of code that will be considered atomically by the CPU. A TSX transaction is defined by a beginning line and an ending line (see Listing 2). Every line of code between these two assembly instructions will be considered as a single instruction by the CPU, at least at the architectural level. Either all the code within the TSX transaction is successfully computed and the results are committed, or any part of it fails, and everything is rolled back and set to the state it used to be before the computation of the TSX transaction.

In the former article, the authors used TSX transactions as a way to defend against side-channel attacks. Indeed, they managed to "blur" the Flush+Reload covert channel. By placing small TSX transactions around several critical pieces of code, they managed to perturb attacks and to disable the attacker's ability to probe correctly the reloading time of the slots. Thus, the attacker became unable to recover any secret from the cache. Some of these experiments were also done on SGX enclaves. Therefore, it has been decided to carry out some tests and to use TSX transactions as a mitigation against the Foreshadow attack.

Listing 2. Structure of a TSX transaction

```
asm("xbegin NAME");  <---------- Begin TSX Transaction
   CRITICAL INSTRUCTION 1
   CRITICAL INSTRUCTION 2
   CRITICAL INSTRUCTION 3
   …
asm("xend");  ┐
asm("NAME");  ┘---------------- End TSX Transaction
```

Several tests were carried out, by placing TSX transactions at multiple different positions inside the enclave code. Both the proof of concept and the RSA use-case were tested. It proved that the best position for the TSX transaction is at the very beginning of the enclave, when the secrets are allocated. Wrapping the allocation inside a TSX transaction causes the attack to fail completely.

However, it was impossible to determine whether this was because the covert channel failed, or because the SGX-Step framework was hindered. By placing a TSX transaction inside the enclave code, one would probably perturb the framework that would be unable to single-step the enclave's execution, thus missing the calculations to predict reliably how long an instruction takes. Therefore, the framework might be unable to single-step correctly the enclave and therefore to recover the secret. As the previous mitigation idea, the TSX transactions seemed to be interesting, even if one cannot reliably conclude that it impedes

the Foreshadow attack. Being able to conclude would require redeveloping the Foreshadow attack from scratch outside of the SGX-Step framework.

3.4. Putting the mitigations to the test on a real use-case

The previous sections showed that it is possible to mitigate the micro architectural threat, at least partly, at the software level. The next sections present an attempt at developing the initial Foreshadow attack on a CPU containing the latest Foreshadow defenses, and a second one with a more recent MDS attack. This experimentation aims at demonstrating to what extent the defenses proposed by Intel combined with the previously studied software and system mitigations are effective at mitigating the initial and the most recent micro architectural attacks.

As in the previously presented use-case attack on an enclaved RSA algorithm, the Foreshadow attack has been applied to a new victim containing the Intel-proposed mitigations. The targeted RSA implementation was the same as presented in section 3.2. The victim was running inside an SGX enclave and the main untrusted application could only interact with it through *encrypt()* and *decrypt()* functions. The main goal was to recover the RSA' secret key using the Forehsadow attack, as previously but on a protected target. The victim computer contained a patched CPU: an Intel® Core™ i5-6440HQ CPU @2.60GHz with microcode version 0xe2 (microcode version containing the updates aimed at mitigating the Foreshadow attack and the early micro architectural vulnerabilities). Again, the SGX-Step framework has been used to carry out the implementation process.

Despite our best efforts, the Foreshadow attack could not be replicated in this specific use-case. After implementing the attack again and adapting it to the new target, the results showed that the attack failed. The proposed implementation has a 0% success rate and cannot recover any secret bit of the key. The only difference in this scenario compared to the one presented in section 3.2. being the micro code update, the reason the attack now fails resides in the effects caused by the update. Hence, the updated micro code causes the sensitive information inside the data cache to be removed consistently. This means that when the Foreshadow attack is run, after the execution of the victim enclave, the secret key residing in the data cache is removed thanks to a flushing operation. The proof-of-concept contained in the package of the SGX-Step framework was also unsuccessful. This shows that the micro code updates are effective and can deter a standard attacker (who can also be qualified in the case of a compromised or untrusted CSP).

In order to complement the previously presented experiments, and to point out Intel mitigations' limitations, an LVI-type attack has also been implemented in order to compromise a symmetric-key cryptographic algorithm. Symmetric-key algorithm means that the same key is used for both encrypting and decrypting the data. We have chosen the most widely used one, the Advanced Encryption Standard (AES) [168], which is a symmetrical block cipher algorithm. It has been decided to change the targeted algorithm type because we believed that the mechanisms involved in LVI would be more adapted to performing fault attacks on an AES algorithm. The initial objective was to inject an attacker-controlled value during one of the many load operations of the AES' computation in order to obtain faulted cipher texts. The exploitation of these faulted ciphers may enable an attacker to recover the

key. We used an Intel® i5-6440HQ core, which was previously proved resilient to Foreshadow-type attacks. The first experimentations rapidly showed that the core was vulnerable to the LVI attack. Indeed, the proof of concept contained in the SGX-Step framework, showing if the computer is vulnerable of not, worked perfectly. This result indicates that the efficiency of Intel's proposed mitigations can be relatively limited when facing the most recent micro architectural attacks. They might not be sufficient to protect an average computer completely.

However, the implementation of an LVI-type attack targeting an AES algorithm proved to be harder than expected. Indeed, the fault caused by the LVI attack is transient, and thus, it does not have any repercussion on the cipher's value. Therefore, it was not possible to implement the attack as initially planned. Using LVI to carry out a "traditional" fault attack does not seem possible, as the computation of the whole victim algorithm is long, thus it is not totally computed during the transient execution. This means that we could not get faulted cipher texts, as it is not possible to cause a fault on the entry of the AES and transiently observe the resulting ciphers that requires too much time.

Implementing a "simple" LVI-type fault attack is therefore not as easy as it seems on a realistic victim, as it is necessary to rely on another attack code. For example, a solution is to hijack and redirect the execution flow towards an attacker-controlled gadget in order to get a persistent fault and thus access the secret key. Another technique that we implemented to recover the secret key consists in encoding the key directly into a micro architectural resource (such as the L1 data cache) once the LVI attack is done. This technique requires faulting a load operation depending on the key and raises synchronization issues that were solved using the SGX-Step framework, but would require more work in a realistic scenario. With this technique, we could extract the key's value with a high success rate, close to 90%. This shows that the existing mitigations on a quite recent computer fail to protect secret information against micro architectural threats if no specific actions are taken to enforce the security of the application at the software and hardware levels.

Moreover, similarly to the previous study where we applied the Foreshadow attack on an RSA algorithm, we added some usual side-channel protections (masking and shuffling) to the AES algorithm targeted by LVI. Even if redirecting the execution flow or encoding the secret directly cannot be considered as traditional fault attacks, one could expect these mitigations to hinder the micro architectural attacks such as LVI. However, it is not the case. The attack's success rate barely dropped to 85% when adding the said defenses. Moreover, this drop in the success rate might be explained by a difference in the random perturbations caused by the CPU and the other processes running in parallel rather than the mitigations themselves. This shows that the results of this experiment can also add up to the previous statement that traditional side-channel defenses are not effective against micro architectural attacks. This is caused by the fact that micro architectural attacks and traditional side-channel attacks are not exploiting the same source of information leak and rely on different CPU mechanisms.

3.5. Conclusions and lessons learned

The different studies proposed in the previous sections show that the micro architectural attacks are a real threat. Mitigations do exist, and take place at several levels: software, hardware and at the system level. Mitigations at the software level seem highly unreliable whereas hardware mitigations are very effective. At the system level, depending on the applicative scenario, there are several reliable solutions to hinder the micro architectural threat. Moreover, traditional side-channel defenses proved to be inefficient at impeding the micro architectural attacks from being carried out. These observations emerge from the fact the root cause of these vulnerabilities lays at the hardware level: the shared resources. In a never-ending race for performance, CPUs are getting more complex and include optimization mechanisms that drastically increase performances, such as Out-of-Order execution, speculative execution… These different mechanisms require the presence of shared hardware resources to work correctly while not increasing the cost too much: caches, speculative buffers, load ports… All these micro architectural shared buffers are potential sources of critical information for an attacker that is now capable of leveraging an information leakage by exploiting these shared resources with a micro architectural attack. Therefore, the best potential solution would be to work at the hardware level in order to reduce the leakages from shared resources.

The different micro architectural elements that are exploited by the attacks presented in this document were previously thought to be inaccessible because they were not documented. However, this assumption fell short with the publication of the Foreshadow attack, as the authors managed to reverse-engineer several deep micro architectural structures and their respective behaviors in their work. This questions the model of security by obscurity. Moreover, the evolution of the effectiveness of mitigations on Intel platform shows this model is limited when considering some specific use-cases where the system manager cannot be trusted. Additionally, the root cause here is related to the hardware. Mitigations therefore imply hardware modifications to be truly effective. Such modifications are complex in the case of closed architectures such as Intel, ARM or AMD as compared to open architectures because of the lack of documentation and heavy architectural legacies. Hence, the interest of open-initiatives arises. The ideal mitigation would be to develop a processor that would be immune, by its design, to the leakages introduced by micro architectural attacks. For this purpose, the RISC-V–based core are the best candidates, as they provide the largest open-source effort for hardware architectures.

Chapter 4

Micro architectural vulnerability study of the CVA6 RISC-V core

---

This section presents an in-depth micro architectural analysis of the chosen RISC-V core for the rest of the document: the CVA6 core. It presents the RISC-V ecosystem before detailing the motivation for this study. Then it presents an analysis of the micro architecture of the CVA6 and the potential vulnerabilities related to it. It concludes on the threat model and best attack path that have been chosen for implementing the micro architectural attacks presented later in the next chapters of this document.

4.1. The RISC-V ecosystem

The RISC-V initiative [13] is an open standard Instruction Set Architecture (ISA). An ISA is a specification of the interactions between the hardware and the software. It defines how the software controls the CPU. The ISA only details the architectural level of the CPU. It mainly defines the registers, the data types, the main memory management, and the different instructions that can be run on a microprocessor. The RISC-V ISA is based on the design principles introduced by the Reduced Instruction Set Computer (RISC) [169]. This means that the RISC-V ISA defines how the architectural level of a RISC-V CPU should behave. A designer can implement his own micro architecture as long as it produces the results described in the RISC-V specifications. Let us note that Thales is a strategic member of the RISC-V foundation.

The RISC-V foundation only drives the evolution and adoption of the RISC-V ISA. However, a real ecosystem is revolving around the foundation. This ecosystem is extensive and includes open-source and commercial software and cores as well as design suites and verification tools. A complete software ecosystem has been developed around RISC-V from low-level software and bootloaders up to operating systems and applications. For the hardware ecosystem, there are now more than 10 billion cores on the market [170] and a wide array of different core designs.

The RISC-V foundation is the initial layer for the development of many other initiatives and developments. The PULP (Parallel Ultra Low Power) [171] platform is one of them. PULP is a joint project between the ETH Zürich [172] and the University of Bologna [173]. It aims at developing open architectures for ultra-low-power applications. The resulting architectures are typically suited for IoT applications and include:

- Three cores: CV32E40P [174], CVA6 (formerly Ariane) [14, 15] and Ibex [175]
- Systems-on-chip: PULPissimo [176], PULPino [177], OpenPULP [178], Hero [179]

The CVA32 and CVA6 core families are now part of the OpenHW Group "core-v-cores" project [180]. The OpenHW Group [181] is a not-for-profit organization that aims at providing open-source cores, related IP, tools and software.

The CHIPS (Common Hardware for Interfaces, Processors and Systems) Alliance [182] is another example of initiative revolving around the RISC-V foundation. This alliance aims at providing open-source hardware codes, interconnect IP and open-source software development tools for design verification. The CHIPS alliance hosts the SweRV [183] core family developed by Western Digital [184]. The RTL of these architectures are freely available online [185]. The SewRV family includes a superscalar processor (EH1), a dual threaded superscalar processor built off the EH1 (EH2) and a smaller core (EL2). These architectures have mainly been designed for embedded computing.

For high-performance computing (HPC) applications, the MEEP (MareNostrum Exascale Emulation Platform) [186] has been developed. Its goal is to provide a flexible FPGA-based environment to work with hardware-software co-designs. It aims at proposing an improved RISC-V toolchain for HPC-specific applications.

Concerning security considerations, LowRISC [187] is a not-for-profit company with a full stack engineering team based in Cambridge, UK. They use collaborative engineering to develop and maintain open source silicon designs and tools. Open Titan [188] is the first transparent silicon root of trust (RoT) created by LowRISC in partnership with Google and other commercial and academic partners. It aims at providing a safer hardware environment by making the silicon RoT design and implementation more transparent, trustworthy, and secure for enterprises, platform providers, and chip manufacturers. A GitHub repository [189] is available. It contains the RTL, helper scripts, technical documentation, and other software necessary to produce Open Titan's hardware designs.

Moreover, strong actors of the CPU architecture market are also investing in the RISC-V initiative and developing RISC-V cores. Alibaba [190] has open-sourced four of its high performance cores: OpenE902, OpenE906, OpenC906, and OpenC910 [191]. Intel has also built partnerships with RISC-V actors to take part in the initiative. The NIOS V processor [192], developed by Intel is based on the RISC-V RV32IA specifications. Several companies also propose proprietary RISC-V core implementations such as SiFive [193] and Esperanto technologies [194]. SiFive proposes a family of RISC-V development boards under the HiFive family [195]. Esperanto proposes a small energy-efficient Minion core and a more complex Maxion core both developed for AI applications [196].

RISC-V is therefore an interesting open initiative with a growing extensive ecosystem including both software and hardware components. For this reason, it has been chosen as a good candidate to work on the mitigation of micro architectural leakages. Indeed, RISC-V cores are as vulnerable to micro architectural leakages as any other core. Indeed, these leakages depend on the specific micro architectural implementation of the core and not on the ISA's specification. Moreover, the open approach proposed by RISC-V has several advantages for the studies proposed in this document. Micro architectural attacks are exploiting leakages at the micro architectural level. Thus, they require modifications of the low-level hardware. RISC-V proposes a wide variety of open-sourced cores. Being able to access the RTL enables both an easier comprehension of the mechanisms involved in the

leakages, as well as more content for mitigation development and smoother integrations/modifications. The section 4.5 introduces an in-depth micro architectural study of the CVA6 data cache using the openly available RTL resources. This preliminary reverse engineering effort is valuable for replicating micro architectural attacks and mitigating them, as shown in the following sections.

4.2. Motivation for CVA6

The CVA6 has been chosen as the preferred target for the study detailed in this document. This choice has been motivated by several factors. As Thales is involved in the study, the chosen core had to present an industrial interest for the company. Moreover, it was required to select a processor that was not subject to the American International Traffic in Arms Regulations (ITAR). Cybersecurity products are considered as “weapons” in the United-States and therefore the ITAR regulation applies. ITAR is a major constraint as it slows down the development and sales of a product. CVA6 is not subject to these regulations as it is developed in Europe. Moreover, Thales is interested in this core. It was therefore selected as the preferred target for the rest of the work.

Before carrying out micro architectural attacks or mitigating them, it was necessary to understand the CVA6 micro architecture. The first step consisted in analyzing the micro architectural mechanisms that are embedded in the core, and their potential leakages due to their specific implementation. This is mandatory in order to replicate any micro architectural attack as it helps determining the most favorable path for a successful attack. The next section presents the CVA6 core and its general characteristics.

4.3. Overview of the CVA6

The CVA6 is an application-class 64-bit processor. It has been developed in System Verilog, and is available online[4]. This core is Linux-capable and can run the M, S and U privilege modes. It contains a Translation Lookaside Buffer (TLB), tightly integrated Data and Instruction caches and a hardware Page Table Walker (PTW). CVA6 has been optimized for performance and not for security. It is therefore an ideal candidate for replicating and mitigating micro architectural attacks.

With GlobalFoundries 22FDX technology, the core achieves a frequency up to 1.7GHz and up to 40Gop/sW peak efficiency. CVA6’s pipeline (shown in Figure 8) is made of six stages. The issue and commit of instructions are both in-order. It is worth noting that the CVA6 core is not fully in order. Indeed, the execution stage is partly out-of-order, specifically, the write back happens out of order. The scoreboard structure serves as a reordering buffer for the in-order architectural commit to work properly. This part of the CPU is complex and has been added in the perspective of CVA6 becoming a superscalar processor in a near future. This particularity has several effects on the current core’s structure. Indeed, the scoreboard is a unique feature compared to classical CPU pipelines. CVA6 also contains a branch-prediction

---

[4] Available at: https://github.com/openhwgroup/cva6, [accessed on: 17 Apr 2022]

unit enabling speculative execution. The rest of the pipeline is classical and does not differ from other processors. The CPU's documentation [197] is available on the OpenHW Group's website and details the behavior and characteristics of the pipeline and its different stages in details.

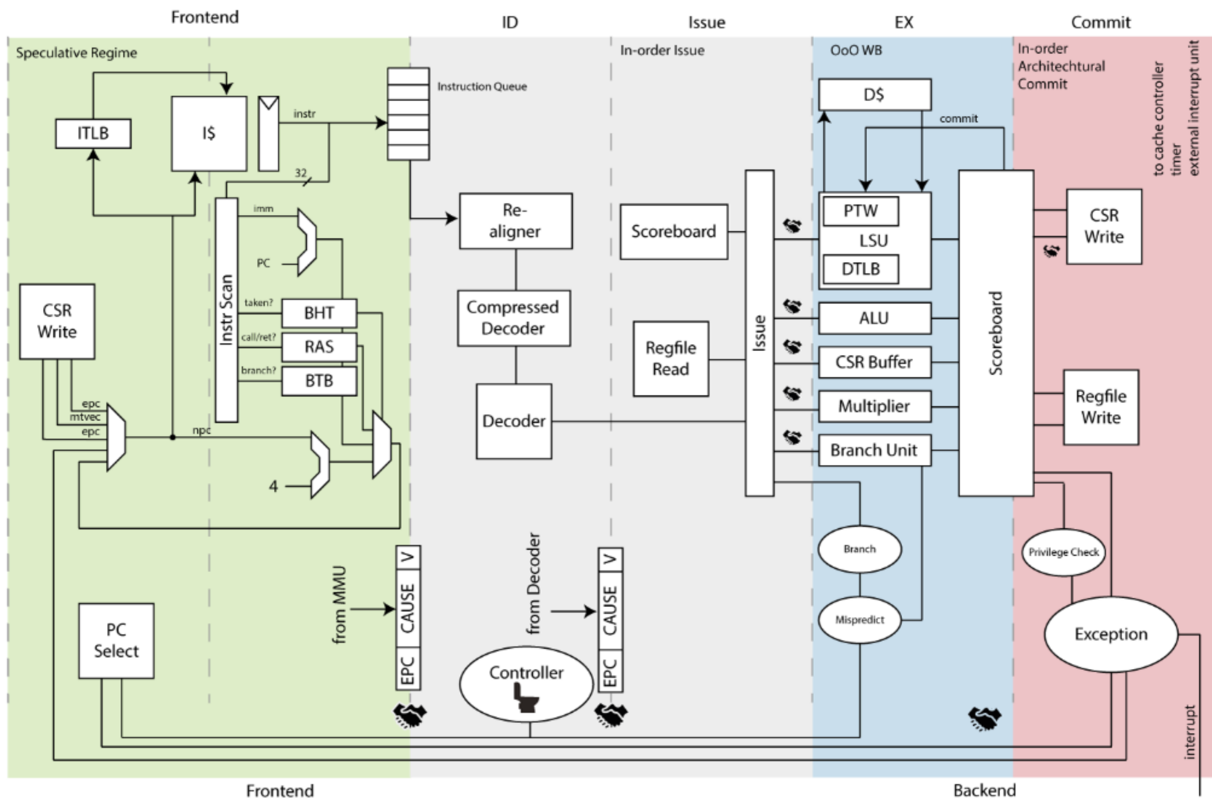


Figure 8. CVA6's Pipeline[5]

4.4. Threat Model and Best attack path candidate

First, it was necessary to select the best attack path candidate to carry out micro architectural attacks. Upon studying the CVA6's pipeline more in details, several attack paths were found to be promising. Three micro architectural elements were believed to present potential leakages at the micro architectural level: the cache memories, the branch prediction mechanism and the scoreboard. All of these elements are micro architectural optimization mechanisms that contain hardware resources shared between different processes. Indeed, the branch prediction mechanism contains a Branch History Table (BHT) and a Branch Target Buffer (BTB). These two elements are shared hardware resources that are typically exploited by variants of the Spectre attack. The scoreboard itself is a hardware resource that handles information from different processes as it reorders the instructions before committing them at the architectural level. As it is a specificity of CVA6, there are no known micro architectural attacks targeting this structure. However, it is also an interesting attack path for extracting information. Finally, the cache memories are a classical vector for carrying out micro architectural covert-channels as presented in section 0.

[5] Figure by OpenHW Group, Available at: https://github.com/openhwgroup/cva6, [accessed 20 Feb 2021].

Spectre attacks are well established, have many variants and mitigations. It was therefore not chosen as the preferred attack path. Despite being an ideal candidate, the scoreboard has also not been selected. Indeed, the vocation of this work is to propose mitigations that can be applied to more CPUs than only CVA6. The scoreboard is too specific for any mitigation developed for it to be applied to another CPU. The chosen attack path was therefore the cache memories. CVA6 has two caches: a data cache and an instruction cache. The instruction cache is rather small and has a classical structure. The data cache on the other hand is highly parametric, possesses many different configurations, and is a major point of discussion within CVA6's development groups. For these reasons, the data cache has been selected as the preferred attack path for the rest of the study.

The data cache inside the CVA6 CPU is a complex subsystem, composed of several functional blocks interacting continuously with each other. Figure 9 represents the whole data cache's structure and the components it cooperates with directly. The data cache mainly interacts with the CPU, and the main memory. The study presented in this document focuses on one of the configurations of the core as available by default directly from the Open Hardware group. The data cache and its default configuration will be presented more in details in section 4.5.

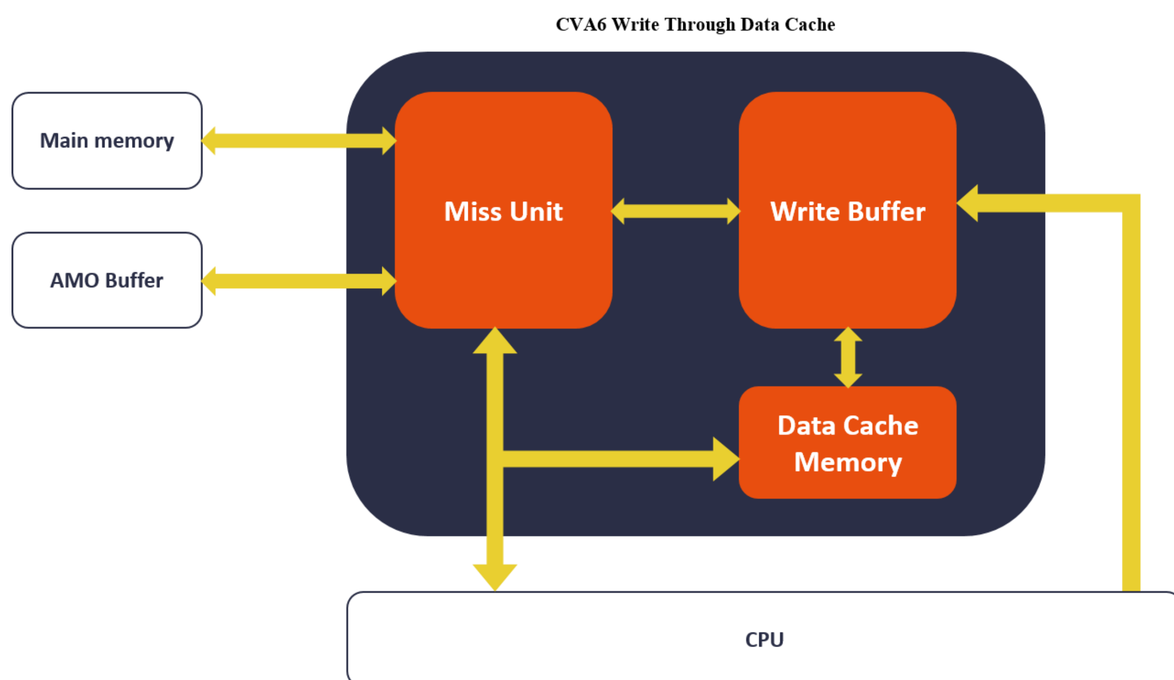


Figure 9. Representation of the data cache in the CVA6 core

Now that the data cache has been selected as the best attack path for the study, the chosen applicative scenario can be detailed. The targeted victim application runs computations implying a secret value. We consider that it is isolated from other processes for security purposes, preventing any eavesdropping by software means. We consider this victim application to be compromised either by a library containing a Trojan or a buggy implementation causing leakages at the micro architectural level. This strong hypothesis is legitimate as the recent Ripple20 series of vulnerabilities [198] has shown it: a series of critical vulnerability has been discovered in a widely used TCP/IP library deployed in a vast range of applications. The goal of this hypothesis is to simplify the attack scenario. Indeed, it has been chosen to focus on the cache memories, and therefore on covert-channel micro

architectural attacks. A preliminary source of information is required in order for it to be extracted by the implemented covert-channel. This information can be gathered either by an embedded Trojan or a leaky library, or by a preliminary micro architectural attack such as Foreshadow, Spectre, or RIDL. The presence of the Trojan in the threat model represents this preliminary micro architectural attack as a source of information emanating from a micro architectural leakage and giving information that can be extracted afterwards using a cache covert-channel.

We consider an attacker that is aware of the preliminary leakages (Trojan, bug, or micro architectural attack) and willing to recover the secret used by the victim process. The technical target consists in a mono-core and mono thread scenario. The mono-core scenario first adds some difficulty for a potential attacker, as he cannot take advantage of high-end optimization mechanisms (out-of-order execution, Simultaneous Multi-Threading …) to gather information. These mechanisms are widely used in micro architectural attacks to recover data from a neighboring process located on another logical core. More generally, a simpler core design implies a restricted number of attack paths available. It also reduces the available covert channel techniques that are applicable, as several of them require the use of the mechanisms named above. However, even if a multi-core scenario causes a greater attack surface, the exploited hardware resources are shared among several cores. This implies a significantly higher amount of perturbation for the attacker trying to leak data from these resources. In the specific case of caches, the previous logic still applies with a variation induced by the cache’s size. Even if there is more contention in a multi-core system, if the cache’s size is sufficient, there might be no additional contention compared to a mono-core system.

In a mono threaded use-case, the execution time is shared only by a single processing thread. It results in more time for the attacker to leverage a covert channel and less chances of being interrupted during the attack.

The victim is running on a CVA6 that runs an operating system (OS) on top of which runs the victim within a first domain, considered to be trusted. This victim program is assumed to be compromised by a Trojan trying actively to leak the secret information from the victim application.

The attacker itself is contained in a second untrusted security domain. He time-shares the core with the victim application and runs a spy program that tries to recover the information leaked by the Trojan. The threat model is summarized in Figure 10. Both the victim and the attacker applications run in the user space. Indeed, CVA6 does enable the use of timestamp instructions in user mode. It might not be the case for other CPUs where developing kernel modules would therefore be required to access those necessary instructions in order to carry out a time or access-driven cache covert channel.

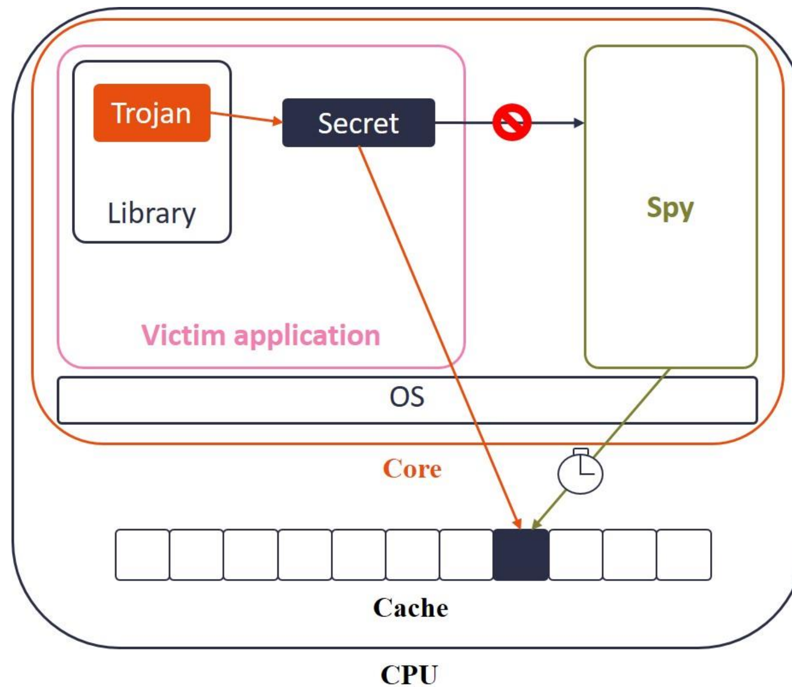


Figure 10. Chosen threat model for the micro architectural covert-channel

4.5. In-depth micro architectural study of the CVA6 data cache

Now that the threat model and attack path have been established, the preliminary reverse engineering study of the targeted data cache can be developed. This study is mandatory as it enables to understand how the cache is built and how it behaves. This knowledge will enable the implementation of the desired covert-channel as it gives insights on where and how to hide the data inside the cache so it can be extracted reliably.

Moreover, this reverse engineering effort represents a contribution to the CVA6 community, as there is no existing study or documentation concerning the data cache. The work detailed in the following sections completes the CVA6 documentation proposed by the OpenHW Group [181] and proposes an in-depth analysis of the cache's behavior that cannot be found elsewhere in the literature by the time the work was being carried out.

4.5.1. Dimensioning and behavior

The default cache is configured in a write through no write-allocate configuration that will be further explained later in the document. The different other configuration possibilities for the data cache will be mentioned and explained but will not be detailed further.

The data cache is composed of 256 sets. Each set is composed of eight cache lines also called "ways". For the rest of the document the term "way" will be used to designate the subdivisions of the cache sets. The value of the associativity for the default configuration of the data cache is therefore 8 (thus the 8-way set-associative denomination). Each way contains 16 bytes according to the standard configuration of the data cache. Based on this information, one can conclude that the size of the L1 data cache is *256 (sets) × 8 (ways) × 16 (bytes in one way) = 32KB*. The addresses in the CVA6 CPU are 64-bit long, and the data handled is the size of a single way (i.e. 16 bytes). The cache addresses are virtually indexed

and physically tagged. Therefore, the addresses' sizes in the data cache vary with the cache's size. The data and instruction caches are both linked to an AXI adapter module in order to connect them to a 64-bit AXI bus.

The cache's associativity directly determines the number of ways that each set is divided into. In the CVA6's code, it is possible to modify its value, in the ariane_pkg.sv file to a power of two (4, 8, 16, 32…). This results in a lot of flexibility for the data cache's dimensioning and structure. Two other parameters can also be modified in the ariane_pkg.sv file: the total data cache's size, and the size of the cache ways. These parameters can be modified separately, but the programmer is responsible for choosing values that result in a data cache configuration that is coherent, e.g., verifying the previous calculation *total_cache_size = number_sets x number_ways x bytes_in_each_way.* For example, for a given cache size of 32KB, changing the associativity from 8 to 16 will cause each cache set to be subdivided into 16 ways, and each way needs to be composed of 8 bytes as a result.

4.5.2. How addresses can be decomposed to represent the CVA6's data cache

Considering the previously presented architecture, it is now possible to specify how to decompose a specific address in order to find which area in the cache it represents. For the CVA6's data cache, the physical addresses are 64-bit long and are composed of the following fields:

- 0 to 3: Offset => it specifies the location of a byte inside the way and thus enables to choose the appropriate data bank
- 4 to 11: Index => these bits allow to distinguish between the 256 available Sets
- 12 to 63: Tag => these 52 bits are used to represent the requested memory location

For clarification purposes, an example will now be detailed in order to put into practice the previously presented decomposition. In this example, the following address will be considered: 0x0000008000b010. By applying the decomposition previously introduced, we obtain the following result as shown in Figure 11:

Dcache address

	Tag	Index	Offset
Hexadecimal	000000008000b (52bits)	01	0
Binary		00000001	0000

Figure 11. Decomposition of the example address 0x000000008000b010

4.5.3. CVA6's writing/eviction policies

The current implementation of the CVA6 core embeds several cache configurations. This includes a write back data cache configuration and another one where the data cache is bypassed (e.g., similar to removing the data cache from the processor). However, the write back cache that was formerly used as default has now been replaced by a write through data cache as it proved to have better performances. It has therefore been selected as the new default configuration thus receiving more updates and support.

The default configuration of the CVA6's data cache that is considered in this document is a write through, no-write-allocate data cache. This means that upon writing a new data that is not already allocated inside the cache, it will first be stored only in the main memory (write through). Then on the next read of this data, it will be uploaded from the main memory and written inside the data cache (no-write-allocate).

To understand more specifically how the data cache is filled, one can carry out the following experimentation: allocate an array that is exactly coinciding with the cache. More specifically, in the case of the default cache configuration, the array would contain 2048 cells, each corresponding to a unique cache way (*256 set x 8 ways = 2048* ways in total). Each cell of the array contains 16 bytes. By looking closely at the cache's behavior in simulation and through precise timing measurements, we can determine in which cache set every cell inside the array will be allocated. The results are summed up in the Table 2 below.

Table 2. Table representing the filling of the data cache when allocating an array of 2048 cells

Array index	Cache set number
0	0
1	1
…	…
255	255
256	0
257	1
…	…
511	255
512	0
513	1
…	…
2046	254
2047	255

We did not focus on the ways but it is possible through simulation to determine precisely which way is chosen every time by the pseudo-random LFSR. The previous table shows the data cache fills every cache set from 0 to 255 with an information as big as one cache way. Every set appears 8 times, corresponding to the 8 cache ways composing each cache set. The cache does not fill a single set completely before filling the next one but rather fills one cache way inside every cache set before filling a second way in any set.

The eviction policy is necessary in case all the potential cache ways for a required data are already filled and "valid". In that case, a choice must be made about which line will be evicted. In the CVA6 core the index of the cache way to be evicted is the output on a LFSR (Linear-Feedback Shift Register). The CVA6, in the default configuration considered here, uses a parametric LFSR with precomputed coefficients for LFSR lengths from 4 to 64 bits. However, only the 3 lowest bits are used. This depends on the cache's parameters that can be changed by the developer. The parametric aspect of the LFSR permits to quickly adapt the eviction policy. The bigger the cache, the more bits needed to be requested from the LFSR. Overall, the number of bits for the LFSR has to be changed only when modifying the cache's structure (and accordingly to this modification). Otherwise, there is no effect on the cache's behavior. This LFSR serves as a pseudo-random number generator for defining which cache way has to be evicted. Upon a cache miss, and if all the cache ways are valid, then the LFSR's output obtained is used as the index of the cache way to be evicted. Otherwise, if there are some empty cache ways (i.e. with their valid bit equal to 0), the first line with a non-valid bit (i.e. the line with the lowest index having its validity bit set to 0) is selected and used to store the new data. Therefore, the CVA6 core uses a pseudo-random eviction policy when the cache is filled. The LFSR uses Galois LFSR feedback masks that were taken online[6]. These coefficients are hard coded in the sources and a XOR operation is applied between the input of the LFSR (i.e. the output of the previous round) and these masks. Moreover, additional block cipher layers can be instantiated to non-linearly transform the pseudo-random LFSR sequence at the output, and hence break the shifting patterns. The implementation contains the code for these cipher layers that can be used to transform the original linear transformation into a non-linear one that is therefore less predictable. The additional cipher layers can only be used for an LFSR width of 64 bits, since the block cipher has been designed for that block length. All of this eviction process is handled by the "miss unit" component, which will be detailed further in the next part.

Moreover, during some of our experimentations, we came across a noticeable behavior regarding the evictions happening inside the CVA6 data cache. When running several applications in concurrently and applying considerable contention on the cache (filling it completely and constantly), we noticed that some cache sets were permanently evicted despite any action from the running applications. We analyzed this behavior and came to the conclusion that the CPU stores some of its intermediate data (addresses where it has to store data, intermediate computation values…) in a specific area that we named the "CPU deadzone". This area of the cache is constantly swept by the CPU and the data it contains is therefore evicted very rapidly, compared to other regions. Both the size and location of the deadzone are highly code dependent, and will change from one cache configuration to

[6] Available at: https://users.ece.cmu.edu/~koopman/lfsr/, [accessed 10 Dec 2022]

another. For example, a longer applicative code means a bigger CPU deadzone. In our specific experimentation where two applications were running in parallel in baremetal simulation, we observed that the deadzone started from the cache set number 196 up to the cache set number 200.

4.5.4. The data cache memory's structure

The main component inside the Data cache's structure is the Data cache Memory. It contains all the data stored inside the Data cache at a given time. It is represented in green in Figure 12 for easier visualization.

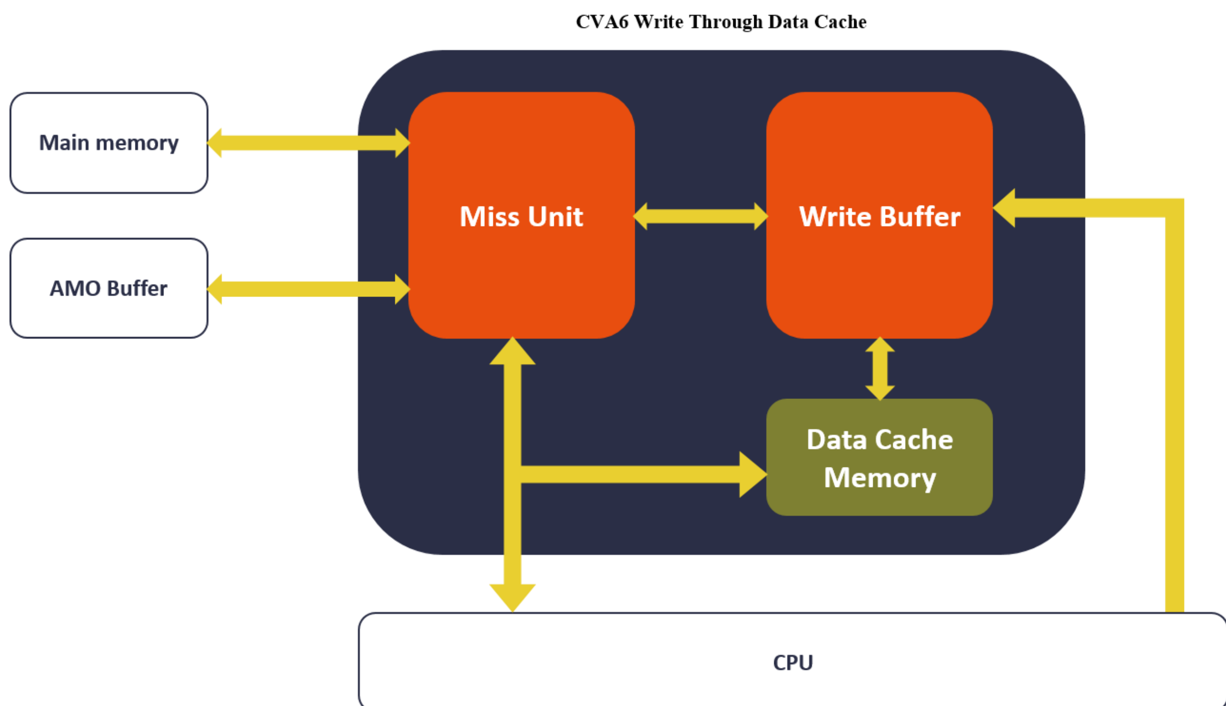


Figure 12. Localization of the Data cache Memory inside the data cache's structure of the CVA6 core

The Data cache Memory is composed of two main structures: the data array, and the tag array. The former is used to store the raw data while the latter stores a part of the address used to access a specific region inside the Data cache. The next sections will successively present the structures of the data array and of the tag array.

The Data array is used to store the data inside the Data cache. It has a specific organization and structure that will now be developed.

The CVA6's data array consists of 256 Sets, indexed from 0 to 255 as shown in orange in Figure 13. As for the specific implementation of this structure, two memory matrices (SRAM matrices for example), called banks are used. They are represented in green in Figure 13. Each bank is a block of 256 entries (half sets, composed of 8 half ways) and each half set contains 8×64 bits (eight half ways). Using this structure, it is possible to dedicate bank 0 to storing the 64 Least Significant Bits (LSB) of each way (represented as the w0 part of each way in Figure 13) for the 256 sets, and bank one for the 64 Most Significant Bits (MSB) of all the ways (represented as the w1 part of each WAY in Figure 13).

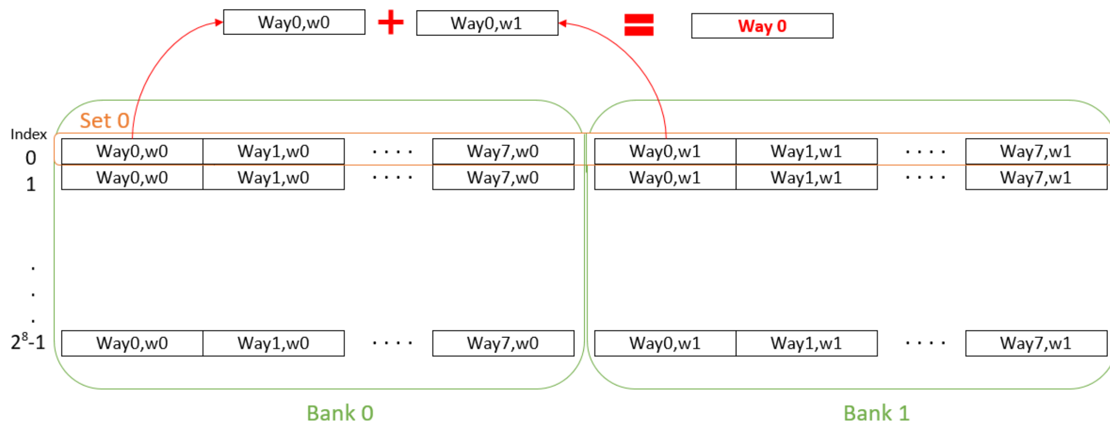


Figure 13. Implementation of the Data array in the L1 Data cache of CVA6

A given address enables to access a cache set. However, a single set may contain several different memory locations. Therefore, tags are required to specify the rest of the address bits in order to be able to distinguish between the different memory locations mapping to the same set. Let us consider two data cache addresses: 0x0000008000b010 and 0x00000081234010. These two addresses map to the set number 1 in the bank 0. If we remove the information carried by the tag, this leads to considering both addresses as 0x010 (only considering the index and the offset). These two addresses map to the same set in the same bank as their indexes and offsets are equal. In such a situation, the tag is required to specify exactly which way is considered and to enable addressing at the way granularity level.

In the CVA6 core, the tag array is implemented using eight memory matrices, as represented in Figure 14. Each matrix, numbered from zero to seven, is composed of 256 slots of 53 bits: 52 bits for the tag itself and one validity bit. Therefore, the memory matrix number “n” will contain the tags dedicated to the WAY number “n”. For example, the tag for the 5th WAY of the set number 125 will be the entry number 125 inside the 5th matrix.

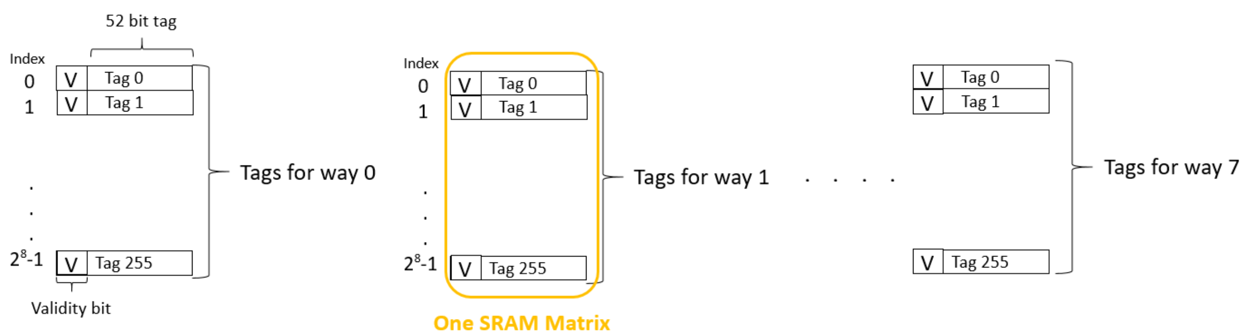


Figure 14. Implementation of the Tag array in the L1 Data cache of CVA6

For clarification purposes, we will now reuse the previous addressing example in order to put into practice the previously presented structures and elements. In this example, the address considered was 0x0000008000b010. The decomposition previously introduced is depicted in Figure 11. Here, the offset is equal to 0x0, or 0000 in binary. The MSB of the offset is used to choose which bank to target in the Data array:

- For offsets between zero and seven, the MSB is equal to zero, which means that the bank 0 will be selected.
- For the remaining values (8 to 15), the MSB is equal to one, meaning that the bank 1 will be selected.

For the address 0x0000008000b010, the bank number zero will be used. The index's value is 0x01, or 00000001. This index refers to the second set in the data array. This leads to the selection of the eight tags that refer to the second set, with an index equal to one. The selection for this example is represented in Figure 15.

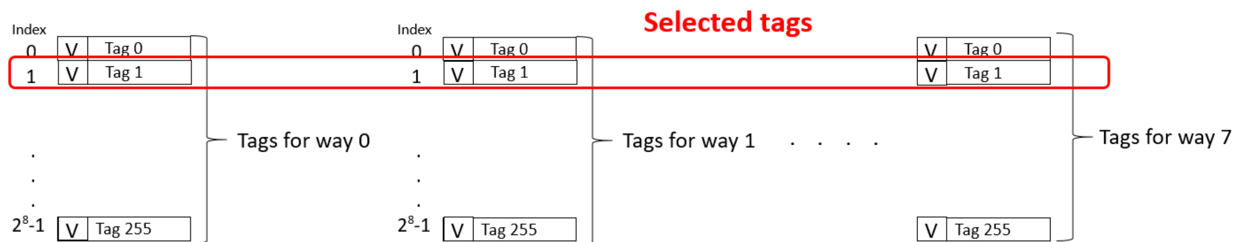


Figure 15. Selection of the tags corresponding to the second set (e.g., index equal to one)

The tag is equal to 0x0000008000b. Among the previously selected tags, the number of the SRAM matrix of the corresponding entry will give out the way targeted by the address considered. The entry from the eighth SRAM matrix is also equal to 0x0000008000b, meaning that the tag refers to the way number seven.

As a conclusion, the requested 64-bit memory block with the address 0x0000008000b010 is in bank 0, 2nd Set (i.e., the set with the index equal to 1), 8th WAY. The results developed above are represented in Figure 16.

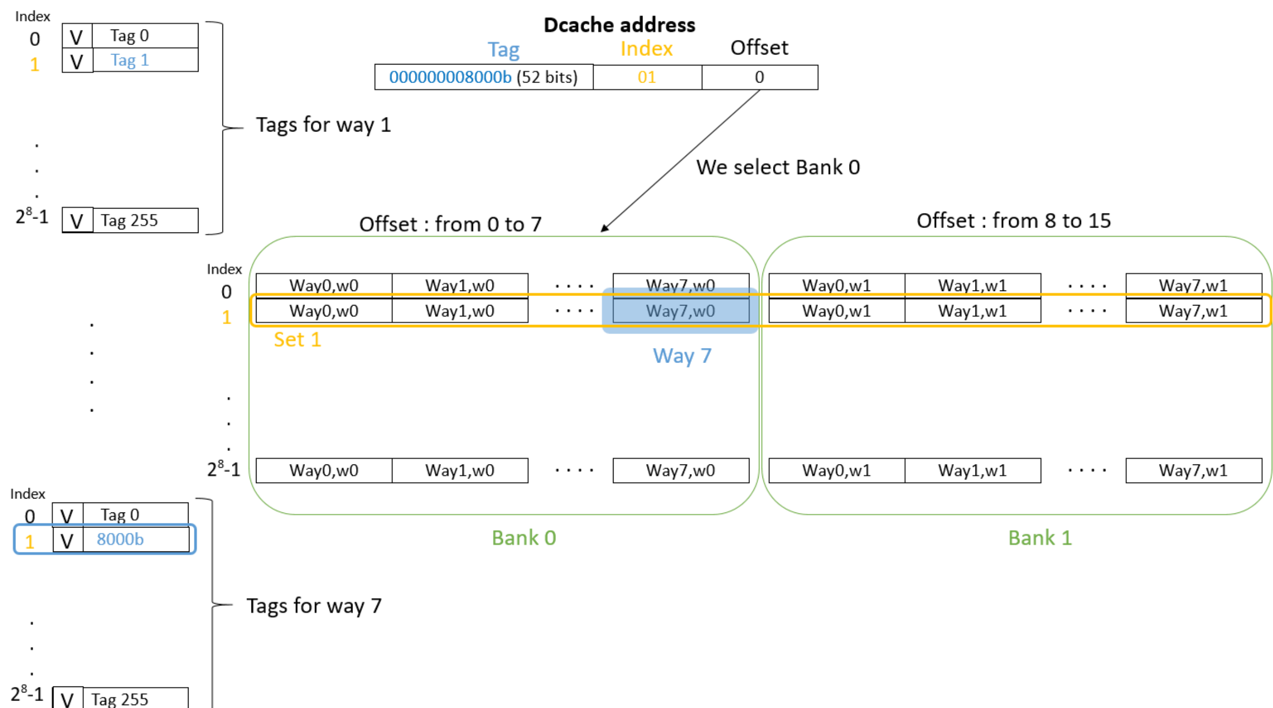


Figure 16. Request of a 64-bit memory block with the address 0x0000008000b010 from the Data cache of the CVA6 core

4.5.5. Storing atomic operations in the AMO buffer

The Atomic Memory Operations (AMO) buffer is located outside of the Data cache's structure. It interacts with the Miss unit and is used to buffer an atomic memory operation for the cache subsystem. It also handles interfacing with the commit stage, acting as an intermediate between the Data cache subsystem and the pipeline's commit stage.

The AMO buffer issues requests to the cache subsystem by providing a physical address and a data for the requested store in the Data cache. The information contained in these requests comes from the commit stage, which also issues the requests to the AMO buffer. Once all the stores have drained, and the AMO buffer is in the commit stage, the request is issued to the Data cache. The buffer then receives an acknowledgement response from the cache subsystem once the request has been served. This buffer is also being flushed when its content is not being committed.

4.5.6. Introduction to the MSHR

This section will now detail two important notions: the Miss Status Holding Register (MSHR) and read/write stalls.

The MSHR is a structure used when a read-miss happens in the cache. It permits to store some information about a pending read-miss operation for later computation. The CVA6 data cache's MSHR contains the physical address of the involved miss, the size of the data, and the validity of the corresponding MSHR entry. The MSHR can only contain one entry. The ID is used and passed to other components inside the data cache and enables them to select and compute a specific cache miss.

A stall inside the cache is a specific corner-case situation where the processing of a request in the cache cannot continue because inflight requests are in contradiction or overlapping each other or with another request (a request stored in the AMO buffer or the MSHR for example). In such a situation, the request's processing cannot continue and the pending requests have to be completed before being able to pursue the calculations. Otherwise, some requests might read incorrect data or overwrite entries that were not supposed to be deleted. For example, if a read-miss is pending on a given address and a write-miss is issued to the same address, there is a stall because if the pending read-miss is not completed first, then it might read the wrong data because the previous one was overwritten by the write-miss to the same address. All of this is possible in the CVA6 core because it is not fully an in-order processor. The execution stage is out-of-order and tries to optimize the usage of the execution units. Except the execution stage, which interacts a lot with the caches, the rest of the CPU is in order. Therefore, there are two possible stall situations:

1. A write-miss collides with the MSHR (read-miss) address.
2. A read-miss cache way address overlaps with a write contained in the AMO buffer that is in flight.

4.5.7. Handling misses and more with the miss unit

Now that the MSHR and stalls have been defined, it is possible to study the miss unit's role. The miss unit is a very important part of the data cache subsystem. It is responsible for the handling of data requests when there is a cache miss. It plays several key roles:

- Checking for read/write stalls
- Managing the cache ways' replacement for read operations (pseudo-random policy previously detailed)
- Flushing the cache
- Communicating with the main memory
- Communicating with the write buffer
- Writing to cache memory

The miss unit is the only component of the data cache subsystem that communicates with the CPU's main memory (located outside of the Data cache). It issues outgoing memory requests. The main memory also sends information to the miss unit. The different signals received from the main memory enable it to keep track of the pending stores, and to get responses to the issued requests: acknowledgements, validity of the requests, pass or fail status. In case of cache misses, the miss unit writes all the requests to memory that are not colliding with MSHR addresses back to the cache once the data is received from it. Additionally, the miss unit transmits the IDs (contained in the MSHR) of the cache misses back to the write buffer. The write buffer is in charge of checking the data cache memory and notifies the miss unit when a cache miss happens and passes all the required information to handle the miss correctly.

The miss unit's behavior is represented in Figure 17. It has seven possible states: Idle, Drain, AMO, Flush, Store_Wait, Load_Wait, AMO_Wait. Idle is the "waiting state" where the miss unit is inactive and waiting for a trigger signal. In the drain state, the miss unit will wait for the write buffer and the MSHR to be cleared. The AMO state corresponds to the miss unit sending out an AMO operation request to the AMO buffer. In the flush state, the miss unit will trigger a cache flush. Store_Wait and Load_Wait correspond to a waiting state where the miss unit is idle until the corresponding request (load or store) is acknowledged by the cache memory. In the AMO_Wait state, the miss unit is stalled and waits until the pending AMO operation returns.

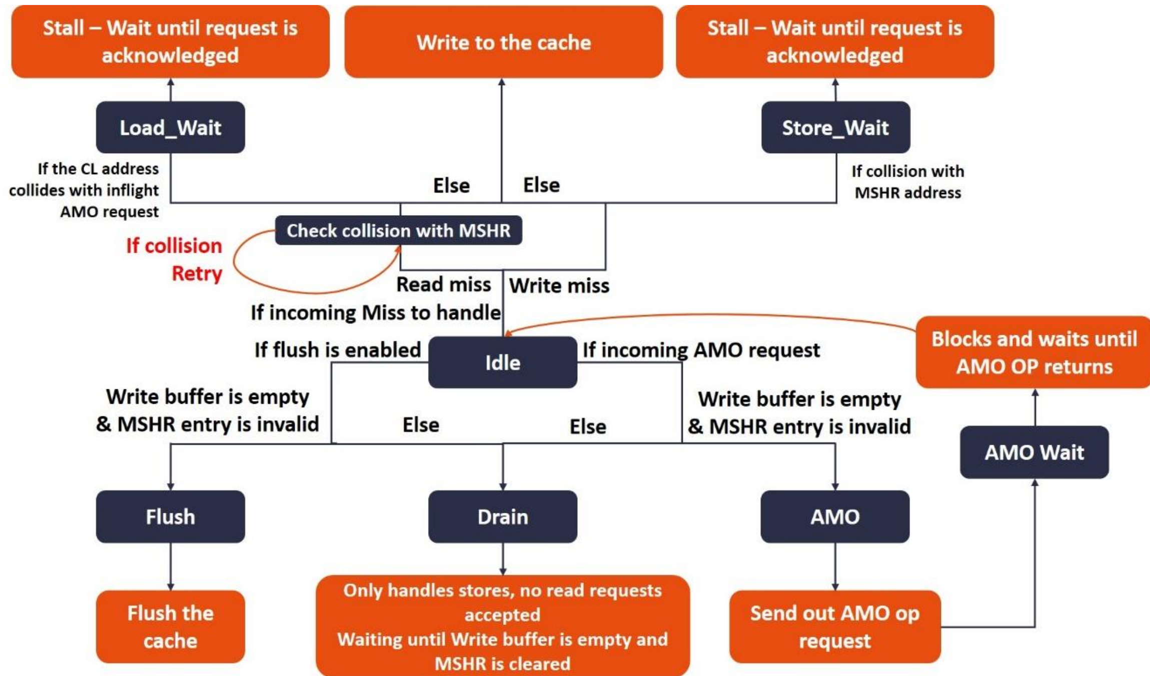


Figure 17. Miss unit's Finite-state Machine

4.5.8. The write buffer's different usages

This buffer behaves as a fully associative cache. This also means that this cache is coalescing. This means that two adjacent free blocks of cache memory are merged. The write buffer is a central component for the Data cache as it communicates with the Cache's memory, the CPU and the Miss unit.

The write buffer is connected to the CPU through the Load and Store Unit (LSU) that is contained in the "CPU" block in Figure 12. More specifically, the load unit, which is a component of the LSU, is directly connected to the data cache (D$) through the Write buffer as shown in Figure 8. The role of the write buffer is to forward data to the load unit for further processing in the pipeline. The Data cache memory containing the raw data stored in the cache is not directly connected to the rest of the CPU. Therefore, the Write Buffer is in charge of transmitting the requested data from the Data cache to the rest of the CVA6's components in case of a cache hit. This means that the Write Buffer is in charge of updating the already allocated cache ways. In case of a cache miss, it is directly connected to the Miss unit and passes the request to this component for further handling. The write buffer can write either a whole cache way or a single word. It can also store several pending transactions, up to DATA CACHE_MAX_TX which can be defined accordingly to one's needs. Overall, the Write Buffer is a small cache used to store read requests issued by the CPU to the data cache and is responsible for reading entries in the Data cache memory, updating them when possible, and returning the read data back to the CVA6's execution stage of the pipeline.

Each byte in the Write Buffer can have several states. Each state is defined by a combination of three bits representing the byte's status: valid, dirty and txblock. The "txblock" bit is set to one in case a byte is currently involved in a memory transaction. It enables to identify inflight bytes. Table 3 shows the different possible states for a byte in the Write Buffer.

Table 3. Table representing the different states of the bytes contained in the Write Buffer

Valid	Dirty	Txblock	Status	Conclusion
0	0	0	Invalid	The entry in the buffer needs to be freed
1	1	0	Valid and dirty	Byte is not part of an inflight transaction
1	0	1	Valid and not dirty	Byte is part of an inflight transaction
1	1	1	Valid and dirty and part of a transaction	The byte has been overwritten while inflight and needs to be retransmitted once that byte's write returns
1	0	0	Clean state	This state is never reached in the current implementation of the Write Buffer

The previously introduced states are mostly used to distinguish between bytes that have been written and not sent to the memory subsystem yet, and bytes that are part of an ongoing transaction. When considering a byte, its current status is checked, and it is then computed accordingly.

4.6. Conclusion and outcomes of the study

This section studied the different characteristics of the CVA6 core and more specifically of its data cache architecture. An initial study of the core enabled to choose the most appropriate attack path among several potential vulnerabilities. The data cache has been selected as the best candidate. An in-depth description of the data cache and its different components has then been proposed. Several specificities of the data cache have been highlighted such as its specific policies, and the observed CPU deadzone. All the aspects mentioned in this chapter have been regrouped and published as an open-archive document [199]. To the best of our knowledge, this paper is the first one to propose documentation aspects for the CVA6 data cache, thus being a significant contribution to the community.

The results of this in-depth study of the CVA6 core have been exploited during the implementation work presented in the next chapters. Reverse-engineering works similar to the one presented here are mandatory for carrying micro architectural attacks. Indeed, each core has its own cache architecture with a different set of policies and specificities. All of these aspects need to be studied thoroughly in order to be taken into consideration in the related attack's implementation afterwards. The next chapter presents the implementation of a Prime+Probe micro architectural covert-channel on a simulated baremetal CVA6, taking advantage of the prior reverse-engineering efforts detailed in this chapter.

Chapter 5

A first simulated baremetal Proof-of-Concept of the Prime+Probe covert-channel on a CVA6 core

---

5.1. Motivation and objectives of the study

Now that the targeted core, the threat model and the attack path have been established, the next step consists in choosing the covert-channel to implement. To exploit leakages in a data cache, several covert-channel candidates are possible, as detailed in section 2.2. The simplest and most common techniques being Prime+Probe and Flush+Reload, these were the two main candidates. Indeed, using well-known techniques that are detailed in the literature was believed to ease the replication and adaptation phases. The work done by N. Wistoff on the CVA6 [101, 102] studies the existence of covert-channel leakages with the Prime+Probe technique and proposes the fence.t instruction to mitigate these leakages. The Prime+Probe technique therefore seemed to be the appropriate candidate for our replication work.

As opposed to N. Wistoff's work, it has been decided not to use the Mastik toolkit [200] that proposes attack implementations but rather to redevelop it from scratch and tailor it to the CVA6 core. This approach enabled us to focus more on the micro architectural causes of the leakage and to study more in-depth the origins of these covert-channels before mitigating their root causes directly.

The first stage of replication consisted in carrying out the Prime+Probe covert-channel in baremetal, as a proof-of-concept of the attack's capabilities to extract data. This proof-of-concept has been implemented in simulation. Indeed, there were no available silicon products based on the CVA6 core by the time of this work. The only possibility to get a physical CVA6 core is therefore to emulate it on an FPGA board. That would be the next implementation step provided the simulated proof-of-concept was successful. There were several objectives for this first-step implementation:

3. Handle CVA6 and start interacting and implementing with it
4. Assess the vulnerability of the CVA6 core to cache covert-channels
5. Pinpoint the exact micro architectural primitives inside the CVA6 for triggering a Prime+Probe covert-channel
6. Provide a first proof-of-concept that the covert-channel is practical before going further into implementation and mitigations

7. Take advantage of the benefits of open-source cores to carry out a simulated attack. Indeed, having access to the core's RTL enables us to simulate its behavior. From an attacker's standpoint, this is beneficial as it is possible to carry out an attack without even having access to the physical product. Moreover, it also implies the capability to replicate and replay the attacks indefinitely in order to characterize it and perfect it. Implementing a micro architectural covert-channel in a simulation environment also represents an interesting contribution to the literature.
8. Provide a background work to back up the implementation process when an OS will be added and guide the mitigation studies thereafter.

The next sections detail the implementation process of the simulated baremetal proof-of-concept of our Prime+Probe micro architectural covert-channel on the CVA6 data cache.

5.2. Threat model

As presented in section 4.4, the applicative scenario we have chosen is based on a victim application that runs computations implying a secret value, and is isolated from other processes. This application is supposed to be compromised by either a Trojan, or a buggy implementation that causes leakages at the micro architectural level. We consider an attacker that is aware of this leakage, and willing to recover the secret used by the victim process. Even if the attacker and the victim are logically isolated, the former will use the micro architectural information leakage to recover the secret information. The threat model for this section differs slightly from the initial threat model, as it does not contain an OS. This threat model depicts a baremetal setup.

As for the technical setup, let us consider first a mono-core and mono-thread baremetal scenario. The victim is run on the CVA-6 core and we suppose that its implementation prohibits it from forwarding directly the secret data to another application, due to the configuration of the Physical Memory Protection (PMP) for instance.

Assume the victim program is compromised by a Trojan that is actively trying to leak the secret information from the victim application. The attacker itself is contained in a second baremetal application. He time-shares the core with the victim application. This attacker runs a spy program that tries to recover the information leaked by the Trojan. The threat model is summarized in Figure 18.

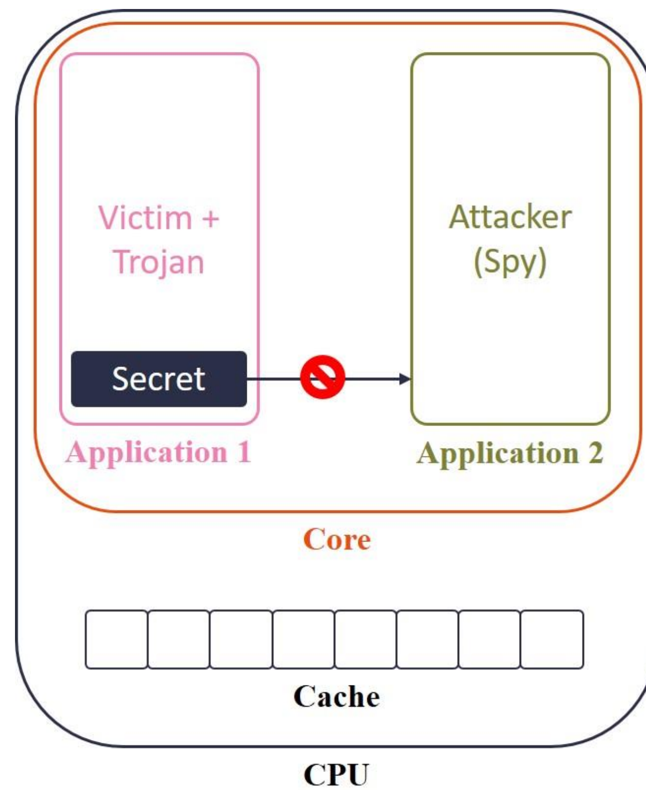


Figure 18. Chosen threat model for the baremetal attack's context

To carry out our experimentations, we chose the open-source toolchain and processor description provided by the Open-Hardware group [181]. CVA6 was used in the default version, with a 32 KB 8-way associative data cache configuration and 16-byte lines. It is in a write-through no-write allocate configuration [199]. The toolchain comes with several simulation and development tools, including the cycle accurate simulator Verilator [201]. We slightly adapted a test suite from the toolchain to include our attack codes, written in C. We also used the Spike Instruction Set Architecture [202] simulator for coherency checks.

5.3. Initial version: noiseless and unscheduled

5.3.1. Code's structure and chosen encoding technique

Our implementation of the attack contains some inline assembly commands adapted to the baremetal setup. The attacker and the victim are in this context two different functions running concurrently, and being "scheduled artificially" by ourselves inside the code itself as shown in Listing 3.

Listing 3. Artificial scheduling pseudocode

```
Prime+Probe_artificial_scheduling(void)

    Run priming_phase()
    Run victim_process(secret_value)
    Run probing_phase()
    Return(secret_value)
```

We chose to work on cache sets rather than on cache lines because of the pseudo-random eviction policy used by CVA6. Indeed, it causes the cache lines to be evicted pseudo-randomly thus making it more difficult to predict the cache's state during the attack. Cache sets on the other hand are always filled with the same logic. Even if working on the sets reduces the amount of values that the attack can extract, it also makes it easier as it reduces its granularity, and the impact of the eviction policy. We used while loops affecting values to data structures corresponding to the size of a single zone inside the cache to proceed to the priming phase, from set 0 to set 255. The data cache is totally filled with attacker data after this step.

Then the victim and the Trojan are run. Our victim simply consisted in an addition operation whose result was our secret value. As for the Trojan, we chose a basic encoding strategy, for simplification purposes. An encoding strategy consists in placing the secret inside the cache with respect to the capacity of the communication to extract data. Here, our most reliable communication channel is the filling and contention of the cache sets, as we know it is definite and it has been studied beforehand. The secret is thus encoded within the cache sets, by applying contention on well-chosen cache sets as to transmit a value. In our case, the Trojan encodes the secret value *S* by completely filling sets of data inside the cache. The victim in this first experimentation is a simple addition process, the result of which is the secret. Moreover, *S* was chosen within the range of the extractible values with a single attack iteration (i.e. between 0 and the maximum value represented with 256 bits). As a first approach, and to test the validity of the proposed encoding, the Trojan simply fills the sets numbered from 0 to *S-1* to cause contention in a given amount of sets. Each set representing a binary value based on its contention state, this means we can theoretically extract values on up to 255 bits.

The spy code is then run again and it measures the access time to its data. We used the privileged machine mode instruction mcycle that accesses a control register (CSR) storing the CPU's clock cycle to measure the amount of cycles it takes to access a given cache set. By calibrating our threshold through several tests, we were able to distinguish cache hits from cache misses using this instruction, and thus to recover the secret value in the spy code.

Considering the specific configuration and design of the CVA6's data cache presented earlier, we had to adapt the Prime+Probe general attack pattern. The main modification is related to the data cache's filling policy: write-through/no-write-allocate. This causes the data to be brought into the cache only on read misses. Therefore, for the priming step, we need to ensure that the attacker's data we want to fill the cache with is both written and read by the spy code in order to ensure a read miss. Failure to comply with the previous statement

would cause the data to be written only in main memory. For this reason, our attack code's "priming_phase()" shown in Listing 3 differs from the classical implementation.

Moreover, as detailed in section 844.5, the data cache is composed of two separate physical memory cells called "banks". This implies that every cache line is physically split into two parts. This specific design choice has an impact on the implemented attack. For the priming phase to work optimally, the attacker needs the cache to be completely filled, therefore taking into consideration such design specificities is mandatory.

This specific structure has also implications on the attack itself. Indeed, when access to a specific cache line is requested, both blocks are loaded, even if only one is required resulting in 128 bits (the size of a cache line) being handled. For simplification purposes, and to avoid redundancy, we chose to focus on the content of the bank 0 in our experimentations. As a result, the evictions caused by the Trojan's activity are only located inside the bank 0. Figure 19 shows the positions of the cache replacements obtained with our implementation of the Trojan.

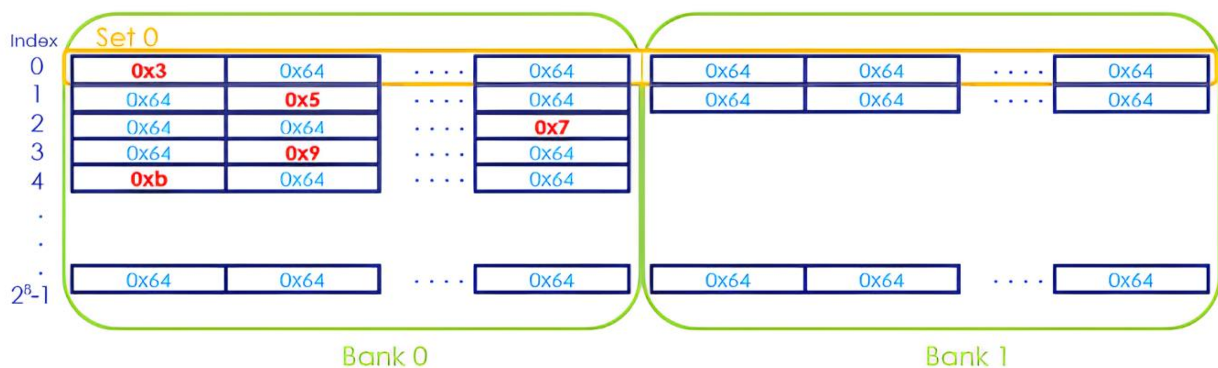


Figure 19. Representation of the evictions caused by the implemented Trojan's activity inside the CVA6's data cache

These observations further enforce the choice of working on cache sets instead of cache lines, as it is more in agreement with the cache's filling policy: it is easier to predict in which cache set the eviction will occur than to choose or predict precisely which cache line will be evicted, even if it is not impossible (the cache line replacement policy is pseudo-random and highly predictable as coming from a simple LFSR).

Our main idea was to propose an encoding technique that would resist to unwanted evictions more than considering the classical increase in total cache access latency. The initial idea was to be able to recover the secret information even if we had unplanned perturbations caused by another process running. We also chose to consider the worst-case scenario: a process that can cause evictions between the priming and the probing phase, thus altering the cache's state before the information is extracted.

The chosen encoding technique consists in causing as much contention as possible on some cache sets. The cache sets targeted are chosen based on the binary decomposition of the value to be extracted. For example, if the secret value to extract is 0101, then the Trojan will create contention on the cache sets number 2 and 4, and ignore the cache sets number 1

and 3. We are capable of measuring which cache sets contain many cache lines that have been evicted, and which cache sets do not contain evicted cache lines. The technical details behind this statement will be explained in the next section. These observations can be directly correlated with the Trojan's activity, as it is the one causing the evictions.

It is then possible to extract binary values based on the presence of contention within a given set or not: if the Trojan has evicted cache lines in a given cache set, it implies that this set will suffer from contention, and we chose that this situation represents a value to be extracted equal to 1. Else, if the Trojan did not evict any cache lines in a given cache set, we chose to consider that the extracted value is a zero as there should be no contention.

We expected that this encoding technique would be more resilient than the classical attack considering the increase in the total latency as the source of information. Indeed, such a worst-case perturbation would probably cause the classical attack to fail in a mono-core mono-thread scenario, as the total latency would be increased by the processing time of the unwanted code thus modifying the transmitted value. This expected noise resistance will be discussed further in section 0.

5.3.2. Data extraction and experimental results

To illustrate the code's implementation detailed in section 5.3.1, this section will describe the method for extracting and recovering data based on a practical example. In this example, the data is filled with attacker-controlled data during the prime step. The attacker arbitrarily chooses to fill the cache with the data 0x64, by filling an array of 64-bit integers with a size of 4096. The number 4096 corresponds to the total number of cache lines. The array obtained is therefore coinciding with the data cache (same size, same subdivision). An array's cell corresponds to a cache line.

The victim application is a simple addition process. Here, the secret value is equal to 95 (chosen arbitrarily). The secret value is the result of the addition. This result is read by the Trojan code and hidden into the data cache using the encoding technique described above. The Trojan also processes to filling a 64-bit integer array composed of 4096 cells in order to cause the desired cache evictions. The Trojan only fills this table partly by modifying only the first 95 lines of the array and filling them with arbitrary values (different from 0x64 in our example so that the cache evictions can be located easily in the simulation environment). According to our observations in the simulation environment, this corresponds to filling one single cache line (chosen pseudo-randomly by the LFSR) inside the first 95 cache sets.

Observing the different signals and specifically the signals giving the content of the different cache lines confirmed this observation. When filling an array that coincides with the data cache (array composed of 4096 64-bit integers), filling the 256 first lines of the array results in filling a single cache line in every cache set. Furthermore, when filling the 257th line in the array, a second cache line in the first cache set is filled. Filling the lines 257 to 512 in the array causes the filling of a second cache line (different from the previous one) in every cache set.

According to these observations, the Trojan code will proceed to causing the eviction of a single cache line inside 95 different cache sets. As we are working in baremetal, modifying a single cache line in each cache set is sufficient to cause a significant difference in the total access time of a whole cache set, as there are no exterior perturbations.

The spy code then proceeds to the probing phase by reading every cell of its initial array (coinciding with the data cache and filled with 0x64). Meanwhile, it also measures the time taken to access every cell by using the MCYCLE instruction, specifically designed to be used in machine mode. The probing phase returns a list of array indexes that experienced a longer access time and thus were cache misses. For this example, and more generally for every manipulation on the simulated CVA6 core, we obtained 16 cycles for a cache hit, and 27 cycles for cache misses. These measurements include the time taken by the CPU to access the register containing the actual amount of clock cycles since the CPU boot, as well as the measured operation itself, i.e. accessing the specific array cell and reading its content. Figure 20 gives the list of array indexes obtained for the example developed in this section.

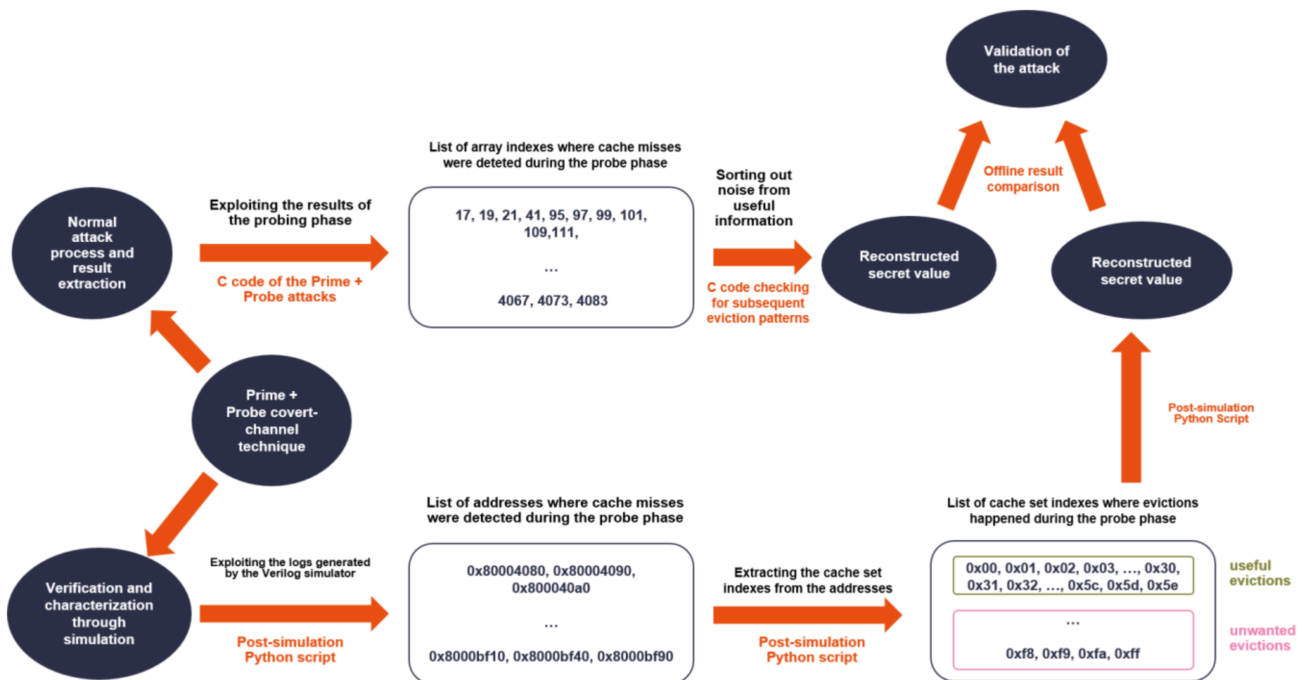


Figure 20. Representation of the simulation workflow

The list of indexes obtained is composed of more than 95 values, for example 123 here. The Trojan causes exactly 95 cache evictions, therefore the expected result was to obtain exactly 95 values as the output of the probing phase. However, it is not the case, even when repeating the experiment multiple times. We always obtain a resulting number that is higher than the expected amount of cache misses. Using the advantages of the simulation environment, we analyzed the signals transmitting the addresses where cache misses happened during the execution of the attack. The results of this analysis are given on the bottom part in Figure 20.

The addresses obtained are then sorted and simplified to ease the analysis. We extracted the 8-bit index from the addresses of the cache misses. This implies that the only information kept here is the index of the cache set where the eviction occurred. We kept a single occurrence of each set index, as some appeared several times. Finally, we sorted these indexes in ascending order. Indeed, this enables us to distinguish the sets where cache misses occurred because of the Trojan code from the others. The resulting list is given in the bottom-right part in Figure 20.

On this figure, we can observe that the cache sets from index 0x00 to 0x5e (or 94 in decimal) suffered from a cache miss during the execution of the attack. This implies that the Trojan code caused the desired evictions. However, there are other evictions that were not caused by the Trojan code itself. These supplementary cache evictions are caused by the CPU processes when handling intermediate data and addresses during the computation of the different instructions composing the Prime+Probe attack. These evictions are not caused by our code and can be observed at every iteration of the attack. These evictions correspond to the CPU deadzone introduced in previous sections.

By taking into consideration that the Trojan causes subsequent eviction inside the data cache starting from set number 0x00 (as implemented in the code), we can discriminate the evictions based on their origin. Indeed, the evictions that occurred in cache sets number 0x00 up to 0x5e are caused by the Trojan code. These evictions represent the secret value. The other cache evictions are caused by exterior sources and are therefore considered as perturbations. The spy code can therefore be implemented according to these observations to extract secret values. Using this technique, we were capable of extracting secret values between 1 and 196 because of the observed perturbations.

Our proposed implementation, even simple, could extract secret values between 0 and 50 with more than 90% of success rate. For higher values, the success rate decreases slightly: between 51 and 130, the success rate is approximately 85%; and for values between 131 and 195, the success rate is around 70%. However, these values are highly code and hardware dependent and change from one software implementation to another and from one cache configuration to another. We could not extract values higher than 195 on our setup because the CPU keeps evicting the sets starting from the 196th. We chose to call this cache area the “CPU deadzone” as the core’s activity prohibits us from placing any information there as it gets evicted a few cycles after being written. Depending on the code’s size, the CPU “deadzone” might change: the longer the code, the bigger the zone.

This effect had to be taken into consideration in order to successfully carry out the initial attack. According to our observations, and the way we implemented our Trojan code, the evictions caused by the Trojan’s activity in the data cache are successive at the level of the cache sets. This means that there is a recognizable pattern we are able to exploit to separate the “useful cache misses” from the ones related to the “CPU deadzone”. Once our spy code recovered all of the cache misses that occurred, we only keep the sets where evictions happened that are adjacent to each other.

The fact that we were able to extract data from a given process to another process, transiting through the cache, shows that the CVA6 core is vulnerable to micro architectural threats. These observations motivated the second type of experimentations presented in section 5.4.

For instance, we wondered to what extent we could predict how noise, either because of the CPU's activity or another untargeted process, would affect the previous attack, and how to modify it accordingly. In addition, we wondered if scheduling could also generate noise affecting the attack's behavior.

5.4. Towards the addition of an OS

Let us now introduce a new threat model, slightly modified compared to the initial one. The goal of this new threat model is to present a use-case situation closer to a real-life scenario.

In the following, the term "noise" refers to an unwanted (for the attacker) software activity affecting the cache and thus potentially reducing the attack's effectiveness. Scheduling is one possibility to introduce such noise and supposedly reproduce an OS' behavior. The main objective is to characterize the presented micro architectural attack in a noisy and "scheduled" environment in order to anticipate how it would behave when run on an Operating System (OS), and how to take these observations into consideration for an attacker. In this second threat model, we now have a running skeleton of a minimalist OS that encapsulates both the victim and the attacker processes and places them in two separate domains: a confined and an unconfined one. By design, the two domains cannot share information directly, but they do share the same micro architectural structures.

5.4.1. Impact of a scheduler

The first modification consisted in adding a scheduling process to the experimental setup, as shown in Listing 4. The idea here is to mimic the scheduling implied by the use of a potential OS in the threat model. The purely sequential process execution shown in Listing 3 has therefore been replaced by a very simple simulation of scheduling. The percentage values are arbitrary and can be modified, like the order of execution, enabling to test any and every possible scenario.

Listing 4. Pseudocode of an example of the simply scheduled attack

```
Prime+Probe_simple_scheduling(void)
      Run 60% of priming_phase()
      Run 50% of victim_process(secret_value)
      Run the 40% remaining of priming_phase()
      Run the 50% remaining of victim_process(secret_value)
      IF noise_generation == 1
            THEN run 100% of eviction_perturbation(quatity_of_noise)
      Run 100% of the probing_phase()
      Return(secret_value)
```

With our scheduler, we manually choose the allocated execution time for each function on a mono threaded mono core processor. Here, only the victim and the attacker processes are running. We will discuss the introduction of new processes in the next section.

Scheduling introduces new issues to carry out the Prime+Probe attack. We needed the attack to be triggered at the appropriate timing. The ideal timing is just after the Trojan's execution. If the attack is executed before the Trojan's encoding, the attacker is unable to extract anything, as the secret information has not been brought to the cache yet. If it is executed too late, data in the cache is likely to be overwritten quickly. Moreover, the computation time of the victim and the scheduling process needed to be known and the attack had to be adapted to it. More specifically, the execution of the priming, the victim and the probing had to be adapted not to exceed their given timeframe. Otherwise, the scheduler switches context, and the CPU runs another process thus evicting the secret data to extract.

The main idea to account for this issue is to implement strategically our attack code in order to make it as fast as possible. Encoding by filling completely or not a single cache set (instead of 256) is an appropriate solution. It reduces the possible values to extract from 196 to only two (0 if the set is empty or 1 if it is filled) but makes each iteration much faster. It implies several iterations to extract a secret value bigger than 1 bit. This encoding can be used practically in real use-cases to extract a single bit of the nonce in an ECDSA cryptographic algorithm which is enough to perform an attack [203]. One of the easiest methods to speed up the attack's computation is thus to choose the most efficient cache-encoding technique, with respect to the capability of extracting the whole secret.

5.4.2. Noise generation and effects on the attack

One last important effect of introducing an OS to our threat model is the addition of perturbations. Indeed, in the context of an OS, several different processes are being constantly computed concurrently. This will generate what can be considered as “noise” to the attack scenario. This noise will in fact cause unknown and unwanted evictions inside the cache that might hinder the attack’s capacities. We studied the impact of the presence of an additional process generating noise to our implementation. We propose here an analysis of the effects of this noise on the attack code and recommendations to avoid being impacted by it.

When adding our additional noise-generating process, we chose to consider the worst-case scenario: a process that would cause as many random cache evictions as possible during its allocated timeframe. To this end, we implemented a pseudo-random generator using an AES [168] algorithm as represented in Figure 21. We used those pseudo-random numbers as addresses to evict from the cache. The contents at these addresses are replaced with data unrelated to the attack. The additional process is run between the Trojan’s execution and the spy’s probing phase, otherwise it would not affect the attack.

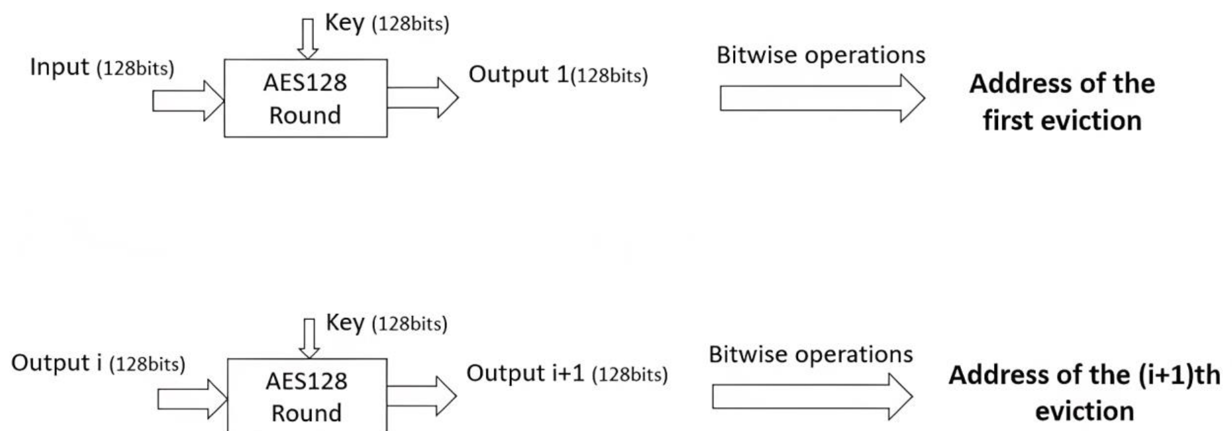


Figure 21. Representation of the perturbation generator based on an AES algorithm

When considering the first encoding technique used in this work (extracting values from 0 to 255 based on the cache sets), such a random-evicting process has very significant effects. Figure 22 gives the evolution of the success rate of the attack in function of the number of evictions caused by the additional process.

The second encoding considered earlier (with only 2 values possibly extracted, 0 or 1) is much more robust facing noise as it only uses a limited amount of space inside the data cache compared to the first one. These results show that the attack’s resistance to noise is depending on the encoding technique used. Moreover, to make a micro architectural cache-based covert-channel more resilient towards noise, it is recommended to study, when possible, the CPU’s and the victim’s behavior in terms of cache usage in order to target specific areas that are the least used ones, so that it maximizes the attack’s success rate.

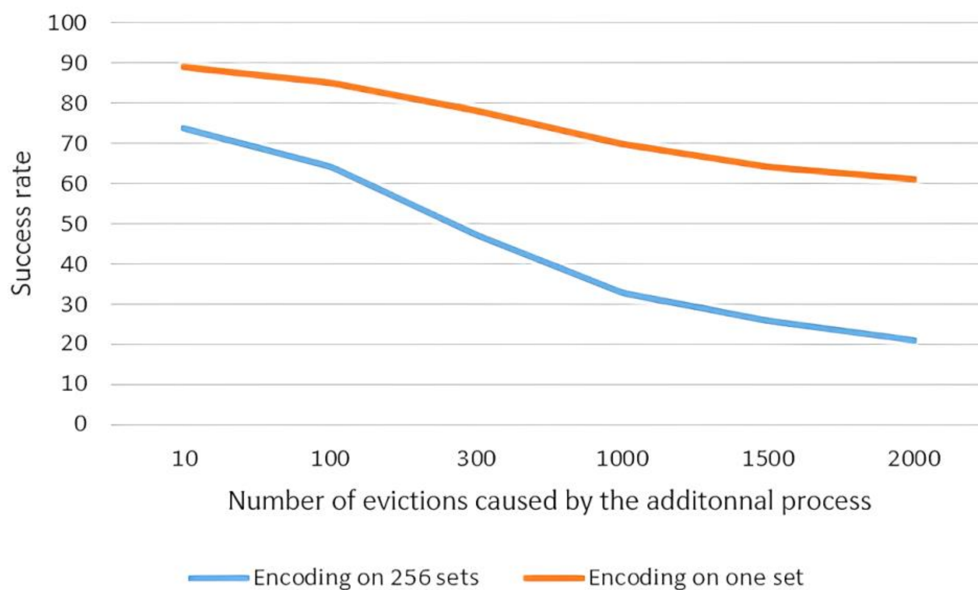


Figure 22. Evolution of the success rate (in %) of the two considered encoding solutions compared to the number of evictions caused by the additional process

5.4.3. Combination of scheduling and noise generation

Our last experimentation consisted in combining a process that is causing random cache evictions based on a process performing an AES encryption similarly to the experimentation presented earlier, and the scheduling process presented in the previous section. The results obtained give a clear vision of the overall quantity of perturbations an attacker has to take into consideration when trying to run a cache-related attack in a noisy environment, without any privileges to reduce it.

Out of the 256 cache sets available on the CVA6 standard data cache configuration, 111 sets were modified between the priming phase and the probing phase of the attack, as represented in . The useful information placed by the Trojan here is 30 as we have a total of 30 cache sets that were modified by the Trojan in a chosen range. This leaves 81 cache sets that were modified during the attack by processes other than the victim or the Trojan. The superposition of scheduling and random noise generation causes the implementation of the attack to be even more challenging.

It is therefore crucial to be able to differentiate the evictions that were caused willingly by the attacker from the rest that is related to the activity of other processes. Some reverse engineering efforts enabled us to identify and characterize the “CPU deadzone” in this precise use-case. This zone is always in the same range of cache sets, and is only modified when one of the codes is modified, making its identification easier than for the other eviction sources.

The most difficult step consists in sorting out the useful information from the noise generated by the presence of the process causing random evictions. Its activity can be divided into 2 categories: the effective evictions caused by the code we implemented, these are pseudo-random; and the evictions caused by the presence of an encryption table when it

is accessed. We will now detail how we proceeded in our specific scenario, considering we know exactly what happens during our attack.

The former can be identified, provided it does not “blend in” with the useful information. Indeed, the sets that are impacted by these pseudo random processes stand out when considering all of the sets that were impacted during the time of the attack, as they are isolated from the rest. There is still a possibility for these evictions to occur in a set that is successive to the “useful sets” that were modified by the Trojan. In that case, the attack fails as we recover a value that is higher or lower by one to the expected value.

The latter can be characterized through some reverse engineering. To this end, we carried out several attacks without running the victim code, meaning that the Trojan does not cause any evictions inside the cache. We were then in capacity to determine the cache sets that were modified by the operations on the encryption tables. These sets are the same from an experiment to another, and only change when the tables are modified.

By combining all of the previous observations, we were able to recognize the pattern caused by the Trojan amidst the rest. It consists in a successive pattern, that is located at a range outside of the “CPU deadzone” and the encryption table’s activity that are both definite and not changing from an experiment to another. There is still a small chance for the process generating random evictions to make the attack fail, and we are not able to detect it. This chance is relatively small as the noise-generating process is limited and can only proceed to a few evictions in the timeframe given by the scheduler. Generally, this process generates between 5 and 10 evictions during a single timeframe according to our implementation of the scheduler and noise-generating process. Figure 23 summarizes the obtained cache set indexes where evictions happened during the attack and details the sorting of the results.

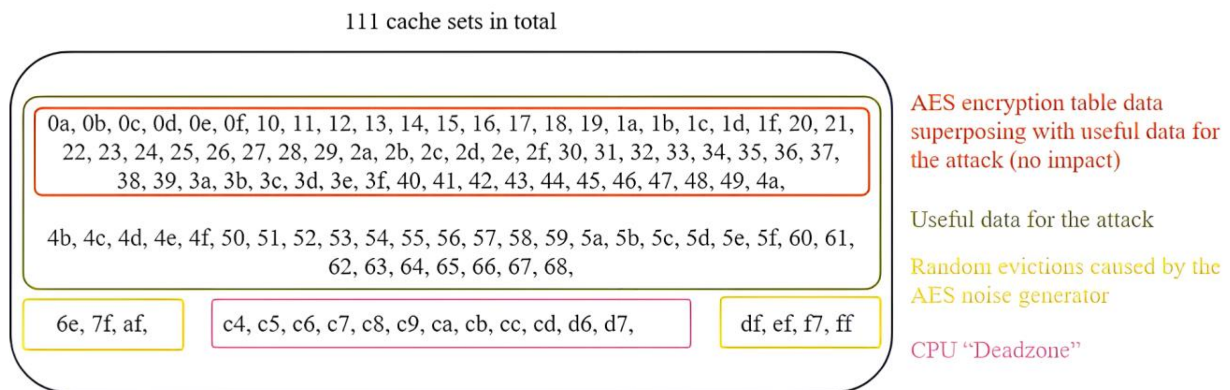


Figure 23. Overview of the different cache set indexes where cache misses have been obtained in a “semi-realistic” use-case

In Figure 23 the sets circled in orange have been filled with data related to the AES’ encryption table. The sets circled in green are filled with useful information for the attack, placed by the Trojan. The sets circled in yellow are the sets filled by our random noise generators. The sets circled in pink are the sets in the CPU deadzone containing CPU operations and various other intermediate operations. A total of 111 sets were modified after the priming phase in this example.

When attacking in a more realistic scenario, the attacker does not know about the structure of the other processes that are running and causing perturbations. Therefore, sorting out the different sources of evictions can prove to be challenging, depending on the other processes' behaviors.

Similarly to the method that we applied during our experimentations, reverse engineering efforts are required. Proceeding to several "calibration measurements" will yield interesting outputs about the behavior of the processes running concurrently with the victim and the victim itself. This information can then be used as an input for the attack. This only applies to an unprivileged attacker, as a privileged one can prohibit unwanted evictions caused by other processes while the victim is being run. This can either be done by moving other processes to another core (when available), or by modifying the priority levels of the different processes.

5.5. Discussion and contributions

Every attack scenario (combination of a targeted hardware and a software victim code) is different and implies a different CPU behavior. More specifically, the way the micro architectural elements react to the code's execution will also differ based on the hardware/software combination as shown by the previous experimentations.

The previous implementation work proved that both methodology and prior knowledge are mandatory to successfully reproduce published micro architectural covert-channels on a completely new target. In order to carry out an attack at the micro architectural level, there are several key points to take into consideration in order to maximize its success rate. First, the knowledge about the targeted victim application is very useful. Carrying out micro architectural attacks without any prior knowledge is still possible however, but requires more efforts. Most micro architectural attacks need to be "triggered" at a precise timing (when a specific instruction is issued by the victim code, computing on secret data for example). Moreover, this also implies that it is beneficial to know the nature and the structure of the secret one wants to extract. This information will drive the choice for a pertinent encoding technique in order to maximize the information extraction.

Then, a precise knowledge of the targeted micro architecture is also necessary. It does not need to be exact or at a very-low granularity level. Sometimes, only a general intuition of how it works is enough, as it was the case in the Branchscope attack [53]. This knowledge is generally gained thanks to reverse-engineering effort, because most micro architectural elements are "black box" and undocumented. However, in the case of an open-sourced CPU such as CVA6, this knowledge is easier and faster to get, which can be considered as a leverage. We did put this into practice with our presented attack as we reverse engineered the CVA6's data cache using both the available source code (which sped up the process) and our observations. Knowing how the exploited cache is being filled and emptied is part of the required knowledge to build a successful micro architectural attack. As such, for an attacker targeting an unknown cache implementation without any implementation details, the preliminary reverse engineering study would require much more efforts. An open-source

implementation helps the attacker, but even in such favorable conditions, the reverse engineering effort proved to be quite time consuming.

Chapter 6

A more realistic use-case: targeting cryptographic implementations with a running Linux OS on an FPGA board

---

6.1. Motivation and objectives of the study

The first replication in the simulation environment yielded interesting results and details about the behavior of CVA6 and its data cache. These results can now be exploited in a second step implementation consisting in a more realistic setup: an FPGA-emulated CVA6 running a Linux OS. This section will therefore focus on the implementation of the Prime+Probe covert-channel on this new platform. These experimentations have several objectives:

1. Provide a more realistic scenario that would be closer to potential real-life applications
2. Target a widely deployed cryptographic implementation and show the threat that micro architectural covert-channels represent
3. Reuse and confirm the observations done during the baremetal simulations and confront them with an FPGA implementation containing a running OS
4. Study the favorable and unfavorable conditions for an attacker to carry out a micro architectural cache covert-channel and the impact of the hardware/software environment on the attacker's capability to extract information
5. Propose more realistic mitigations and evaluate their effectiveness and impacts on the core
6. Reassess the advantages and disadvantages of an open-source context for an attacker.

The next sections will detail the different experimentations carried out to implement a Prime+Probe covert-channel on an FPGA-instantiated CVA6 core running a Linux OS.

6.2. Experimental setup and tools

We will now detail the setup we used during our experimentations. For the hardware configuration, we chose to work with the default version of CVA6, as directly available from the Open Hardware Group's Github repository [15]. More specifically, this means that the core we worked on uses 64-bit addresses, and has the cache configuration corresponding to the one introduced in section 4.5.

We instantiated this CVA6 on a Genesys 2 FPGA platform from Digilent [204], as there are a lot of tools and configurations available to work with. The Digilent Genesys 2 board is based on the latest Kintex-7™ Field Programmable Gate Array (FPGA) from Xilinx that is a high-speed FPGA. We communicate with the FPGA platform using a Linux computer through SSH and using the Ethernet port available on the Genesys 2 board.

For the software configuration, we relied on the toolchain contained in the Open Hardware Group's Github repository to compile and synthetize the RTL into an exploitable bitstream and memory configuration file. We then used the Xilinx's Vivado Design Suite [205] to instantiate the results of the synthesis on the FPGA board. We then selected a Linux OS, generating our own Linux image using the toolchain available on CVA6-SDK Github repository [206]. It is based on a pre-built image directly available on the same repository, but we slightly modified it to fit our specific needs, adding support for SSH communication. The Linux image is lightweight, and we chose not to add any tool or process that could hinder our experimentations. The Linux image is then embedded in a standard 64Gb SD Card and inserted inside the Genesys 2. The OS is loaded at each reset of the FPGA board.

6.3. First experimentations on information transmission and statistical analysis: defining an encoding technique

We began by carrying out several preliminary experiments in order to identify a potential encoding technique e.g., a specific method to place the secret information inside the data cache in order to transmit it. Our initial idea was to propose an implementation of the Prime+Probe covert channel, therefore we began by observing the cache misses. These tests aimed at achieving a first level of information transmission that would be a building block for a more complex transmission and attack process.

Several tests have been made in order to establish a correlation between the activity we caused with a given code, and the resulting evictions. The understanding of the hardware implementation of the data cache was a valuable input for these first experiments. We started out by running a skeleton of Prime+Probe covert-channel consisting of an attacker process (or spy process) filling the cache with its own data (prime step) and a second process (or victim process, containing the Trojan) trying to cause evictions equal to a given secret value that was hardcoded inside it. We then proceeded to timing measurements inside the spy process (that carried out the prime step) using the unprivileged *RDCYCLE* RISC-V instruction in order to observe the different cache evictions caused by the victim process and its Trojan.

The spy process also generated a log file containing all of its measurements that we statistically analysed thoroughly afterwards using Python scripts. This statistical analysis over a large amount of experimentations is a mandatory step compared to a baremetal context as the OS causes multiple unwanted cache evictions because of the concurrent processes that are running. Considering multiple experimentations and subtracting the results of the experimentation without the victim process running (normal activity only with the OS running) enabled us to reduce the impact of this unwanted "noise" in our measurements, and observe only the useful evictions caused by the Trojan contained in the victim process.

After these experiments, we were able to observe a difference between the cases in which the victim process creates evictions and the other cases. Recognizable patterns caused by the victim process' activity (generating cache evictions) showed that it was possible to transmit information, even if the OS adds some "noise" that could hinder the transmission. It also enabled us to create a set of analysis tools to minimize the impact of the OS on the observed traces and visualize the results of our minimal Prime+Probe covert-channel. The next step consisted in establishing a covert-channel, enabling us to transmit a given value through the data cache.

To build a micro architectural cache covert-channel, it is mandatory to be able to recover a precise value hidden by the Trojan inside the data cache. According to the previously presented observations, and the understanding of the data cache's structure, we built a covert-channel on the CVA6 capable of transmitting up to 256 bits of information with the default data cache configuration.

Applying the same code structure as in the previous section inspired by the Prime+Probe technique, Listing 5 gives the pseudocode of the improved version of our initial covert-channel. The Spy() function serves as the main function for our proposed implementation.

Considering the characteristics of the data cache, the spy process builds an extraction array fitting the cache's structure. This array contains 2048 cells of 16 bytes, analogously to the CVA6's data cache. This array is used to fill the cache during the prime step, made by the spy process. As the cache uses a no-write-allocate policy, it is necessary to cause a read-miss on every element of the extraction array in order to bring them inside the data cache. To this end, we sequentially allocate and then read back each cell of the array.

Once the cache has been filled, the spy process creates a thread running the victim and the Trojan it contains. We will discuss the realism and the applicability in a real-life context of this scenario in a later section. For this iteration of the experimentations, the victim simply consists in an addition and the secret is the result of the operation, or even just a hard coded value acting as a secret to be extracted. The rest of the victim code is composed of the Trojan that will cause targeted evictions inside the data cache. In order for the evictions to be effective and transmit the desired value, it is required to take into consideration the structure and behavior of the data cache.

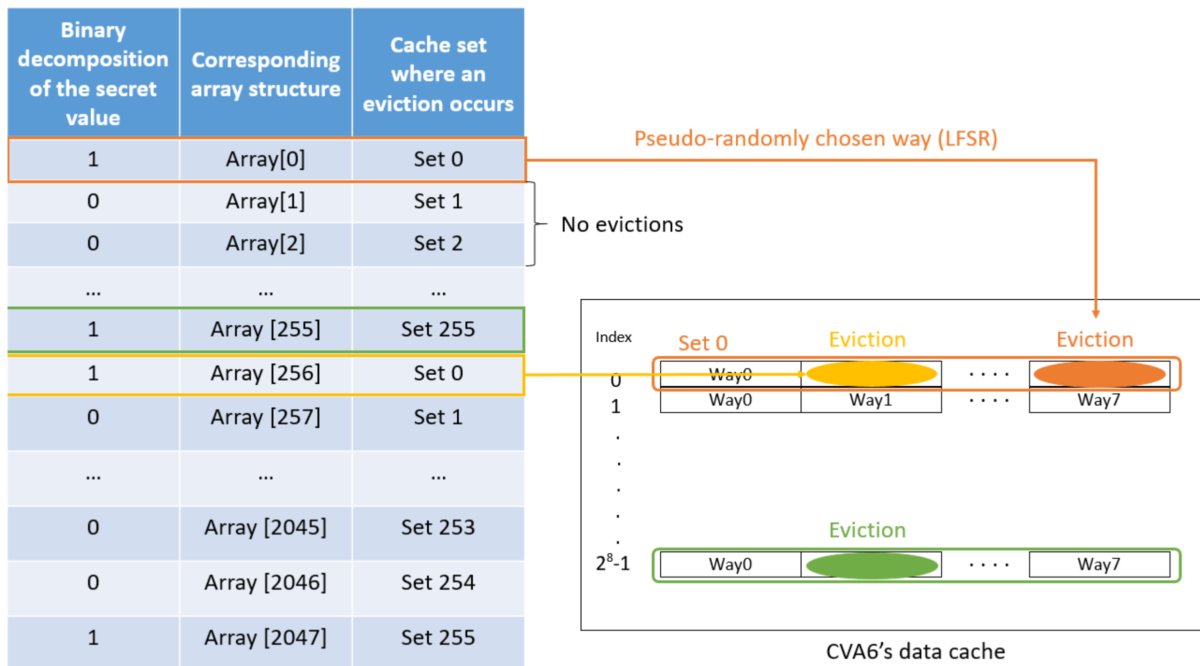
We noticed that working with the cache sets produced the expected results, and was easier to implement than working at the cache way granularity, mainly because of the pseudo-random eviction policy. The secret value is transmitted bit by bit, meaning that a cache miss

will be interpreted as a value of 1 whereas a cache hit will be interpreted as a value of 0 (because the spy's value has not been evicted).

Listing 5. Pseudocode for our Prime+Probe micro architectural covert-channel

```
Prime(table[])
      FOR i in range(0, size(table)-1)
            filling_table[i] = 400
            temporary_variable = table[i]
      ENDFOR

Probe(table[])
      FOR i in range(0, size(table)-1)
            t1 = RDCYCLE
            temporary_variable = table[i]
            t2 = RDCYCLE
            IF (t2-t1 > threshold)
                  miss_count[i] += 1
            ENDIF
      ENDFOR

Trojan(secret)
      Trojan_table[cache_size]
      FOR i in range(0, size(Trojan_table))
            IF (i% NbSets == 0)
                  FOR j in range(0, size_binary_secret)
                        IF (binary_secret[j] == 1)
                              Trojan_table[i+j] = 400
                              temporary_variable = Trojan_table[i+j]
                        ENDIF
                  ENDFOR
            ENDIF
      ENDFOR


Victim(secret)
      …
      Trojan(secret)
      …

Spy()
      FOR i in range(0, NB_iteration)
            FOR j in range(0, NB_samples)
                  Prime(spy_table)
                  Victim()
                  Probe(spy_table)
            ENDFOR
            Generate_logs()
      ENDFOR
```

The Trojan will proceed similarly to the spy process during the prime step and allocate then read again cells in another array (called "Trojan's extraction array") with the exact same dimensions as the cache. However, the Trojan will not fill every cell of the array, but only the cells with indexes that correspond to a value to be extracted equal to 1. It will ignore cells with indexes corresponding to a value to be extracted equal to 0. The main idea is that filling specific regions (bit of the secret equal to 1) of the Trojan's extraction array directly translates in a cache miss for the spy inside the corresponding cache sets inside the data cache while for the untouched regions (bit of the secret equal to 0) it will translate in a cache hit. Figure 24 represents the situation and shows the corresponding evictions caused by the Trojan's action.



Binary decomposition of the secret value	Corresponding array structure	Cache set where an eviction occurs
1	Array[0]	Set 0
0	Array[1]	Set 1
0	Array[2]	Set 2
...	...	...
1	Array [255]	Set 255
1	Array [256]	Set 0
0	Array [257]	Set 1
...	...	...
0	Array [2045]	Set 253
0	Array [2046]	Set 254
1	Array [2047]	Set 255

Figure 24. Representation of the state of the CVA6 data cache when the Trojan encodes the secret value

In practice, the Trojan will therefore fills its extraction array at specific locations to cause evictions in the corresponding cache set. Given we want to extract a value equal to 1 as the first bit in the secret's value decomposition, the Trojan will therefore fill its array cells having indexes that verify:

$$Index \equiv Targeted_set \mod(Total_number_of_sets)$$

Hence, in the cache configuration we are studying throughout this work, and for the first bit of the binary decomposition (thus we target the set number 0) this translates to:

$$Index \equiv 0 \mod(256)$$

The value written inside the cells does not matter at all, as only the cache eviction is useful for the covert-channel. Each time the Trojan will fill one of the cells, it will cause a corresponding eviction inside the set having an index equal to the second term of the previous equation. For our example, the eviction will be caused in the set 0. There are 8 possible ways to be replaced inside each set, and we cannot easily predict, or choose which one will be evicted. This is the reason for choosing a cache set encoding technique instead of working at cache line granularity that proved to be harder since the choice is made pseudo-

randomly by the LFSR. We have no guarantee however that the evictions caused inside a given set will occur on different cache ways. It might happen that the way 0 inside the set 0 is evicted twice for example (as an example, when filling *Array[0* Total_number_of_sets]* and *Array[2* Total_number_of_sets]*). Overall, and on several repetitions of the covert-channel, we are capable of distinguishing data sets where “a lot” (will be quantified in the next section) of evictions occurred, caused by the Trojan, compared to data sets where no evictions were caused and the spy’s data are still intact, leading to the reconstruction of the secret value.

This reconstruction is made possible by the spy process’ measurements during the probe step. Once the Trojan has filled its extraction array according to the secret’s binary decomposition, we run the spy process again. It measures the time it takes to access again to all of its data. We are then capable to distinguish data that have been evicted from data that have not been evicted, using the *RDCYCLE* instruction. Indeed, some cells inside the spy’s array will experience a higher access time compared to the others, thus they have been evicted from the cache. Using these measurements and our Python script that processes the logs generated by the spy process, we can reconstruct the secret value to be extracted. Experimental results proving that our attack implementation is practical on a simple applicative example will be given in the next section.

To summarize, the main aspect of this covert-channel consists in translating the binary decomposition of the secret value to a corresponding contention inside the matching data sets. Having a value of 1 to extract at a given index, called *p*, in the binary decomposition, means we want to create contention inside the cache set number *p*. For that purpose, the Trojan fills all of the cells in *Trojan_table* that will cause evictions inside the set number *p*. By experimenting, we found out that these cells are the ones verifying:

Index≡p mod(Total_number_of_sets)

Filling these cells with Trojan’s data will cause an eviction of the Spy’s data inside the set number *p*. This data was previously allocated inside the cache during the prime phase. Therefore, when the Spy probes its data again to verify the time it takes to access it, it can conclude about the sets that contain contention or not by looking at the amount of cache misses for every cache set. According to the evictions caused by the Trojan’s activity, a high number of cache misses for the set number p is identified. This means that the Spy process will experience a higher access time for all the cells of *Spy_table* verifying:

Index≡p mod(Total_number_of_sets)

6.4. First version of the attack on a simple victim

For this section, let us take a schoolbook victim to apply the previously presented covert-channel. Our considered victim will therefore only perform an addition on 40 bits to simplify the understanding of this toy example. The secret to be extracted is the result of this addition. We now apply the code presented in Listing 5 to this new victim. For demonstration purposes, we gather 2000 samples for an easier visualization (in practice we need much less samples to extract a secret value). Let us consider that the result of the

addition is 672726424737. This corresponds to the binary decomposition: 1001110010100001100111101001110010100001 (40 bits in total). As detailed in the previous section, our covert-channel will proceed as follow:

1. The spy code Primes the data cache with any data, as the value itself does not matter. This is agnostic from the victim's behavior and will therefore hold for whatever process involving a secret value. Changing the cache's characteristics (number of sets, cache's size, associativity) and the cache's filling policy would require to only change this part of the covert-channel.
2. The victim makes its addition. The Trojan is run as it is contained in the victim. It decomposes the result of the addition in binary. For each bit, it will, or not, replace some values inside its array (named *Trojan_table* in Listing 5). The leftmost bit (i.e. at position *p=0*) being a 1, the Trojan will fill its array at all the following indexes: 0, 256, 512, 768, 1024, 1280, 1536, 1792. All these 8 indexes verify *Index≡0 mod 256*, where 256 is the total number of sets and 0 the position of the value to leak in the binary decomposition. This operation then causes 8 cache misses in the set number 0. The next 1 in the binary decomposition is the bit at position *p=3*. The Trojan will therefore replace its array at all indexes verifying *Index≡3 mod 256*. Therefore, it fills the following indexes: 3, 259, 515, 771, 1027, 1283, 1539 and 1795. This causes 8 evictions inside the set number 3 (starting from set number 0). The Trojan will repeat these steps for the whole binary decomposition of the secret value to extract.
3. The spy Probes its array and measures the time it takes to access every cell. It also generates the logs.

This experiment produces the results given in Figure 25, where one of the eight patterns caused by the Trojan's activity has been circled for illustration purposes. On the figure, we represented the number of cache misses at every index in *Spy_table* cumulated over the 2000 iterations of the covert channel. We can clearly see 8 patterns corresponding to the 8 ways of the targeted set that represent the Trojan's activity. Those patterns highlight the associativity of the data in the default cache configuration we are studying. The patterns are very similar and carry the same information about the secret to be extracted. In fact, these patterns directly correspond to the evictions done by the Trojan at the cells verifying:

Index≡p mod Total_number_of_sets

Each pattern contains one "peak" corresponding to one of the eight possible values verifying the equation for a given value of *p*. If we consider the previous example, the Trojan replaced the indexes 3, 259, 515, 771, 1027, 1283, 1539 and 1795 (for *p=3*). This means that one of the 8 patterns has a peak corresponding to the eviction caused on the index number 3, another pattern has the peak for the index 259… What is important to note here, is that these peaks however have the same position inside all of the 8 patterns: the third position. This is because they are all related to an eviction caused in the cache set number 3 corresponding to the 4th bit (i.e. at index number 3) in the binary decomposition of the secret value.

Here we wanted to extract 40 bits of secret data, therefore we targeted the sets number 0 to 39. We can notice the presence of an “offset” at the beginning of the trace, but it does not hinder the analysis in any way. On this figure, every pattern corresponds to the activity of one way inside the cache sets from 0 to 39. For example, the pattern number p represents the amount of cache misses on the way number p inside every cache set from 0 to 39. As we have 8 patterns, we cover all the possible way placements on the figure, as there are 8 ways in total (corresponding to the value of the associativity) in the configuration we study. The data between each pattern corresponds to the amount of cache misses inside the rest of the sets, from 40 to 255. In the sets number 40 to 255, the Trojan did not cause any eviction because we only wanted to extract 40 bits therefore we needed only the first 40 cache sets to encode our secret. One can note that we are capable of distinguishing the activity at the cache way granularity here. However, we are forced to cause the Trojan to act at the cache set granularity because of the LFSR. We cannot choose easily and precisely which way the eviction will occur inside a given cache set, as the choice is pseudo-random. Predicting the outcome of the LFSR is possible. However, it implies a longer computation time thus reducing the overall stealth of the attack.

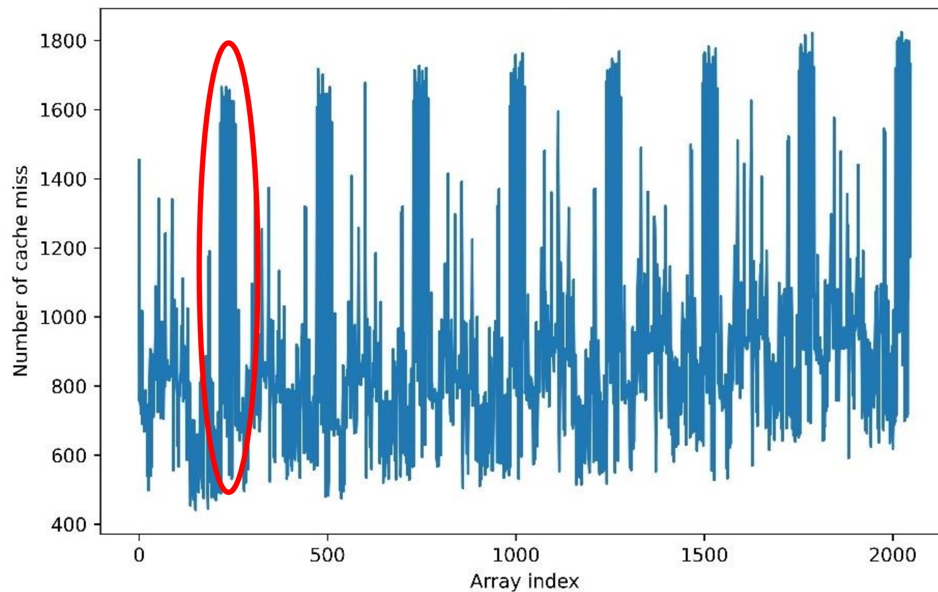


Figure 25. Experimental results for a simple addition victim

When we zoom on one of the patterns (the circled one in Figure 25), we obtain what is represented in Figure 26. In the figure, the red bars are delimitating the pattern itself from the rest of the data surrounding it. The green bar is the threshold for considering the resulting peak as a 0 (under the line) or as a 1 (above the line). A few examples of the bit values recovered are given in the upper part of the figure. Each pattern of the 8 patterns is made of 40 values, corresponding to the 40 bits we want to extract. A threshold is required to differentiate what we consider as 0 on the trace from what we consider as a 1. We observed that the arithmetic mean of the extreme values of the whole trace (number of cache misses), or *(Max_value+Min_value)/2*, worked perfectly for that purpose. Considering the peaks going beyond this value as ones, and the others as zeros, we recover a value of 1001110010100001100111101001110010100001. This corresponds to the result of the victim’s addition, therefore we recovered the secret value through the CVA6’s data cache with a running Linux OS.

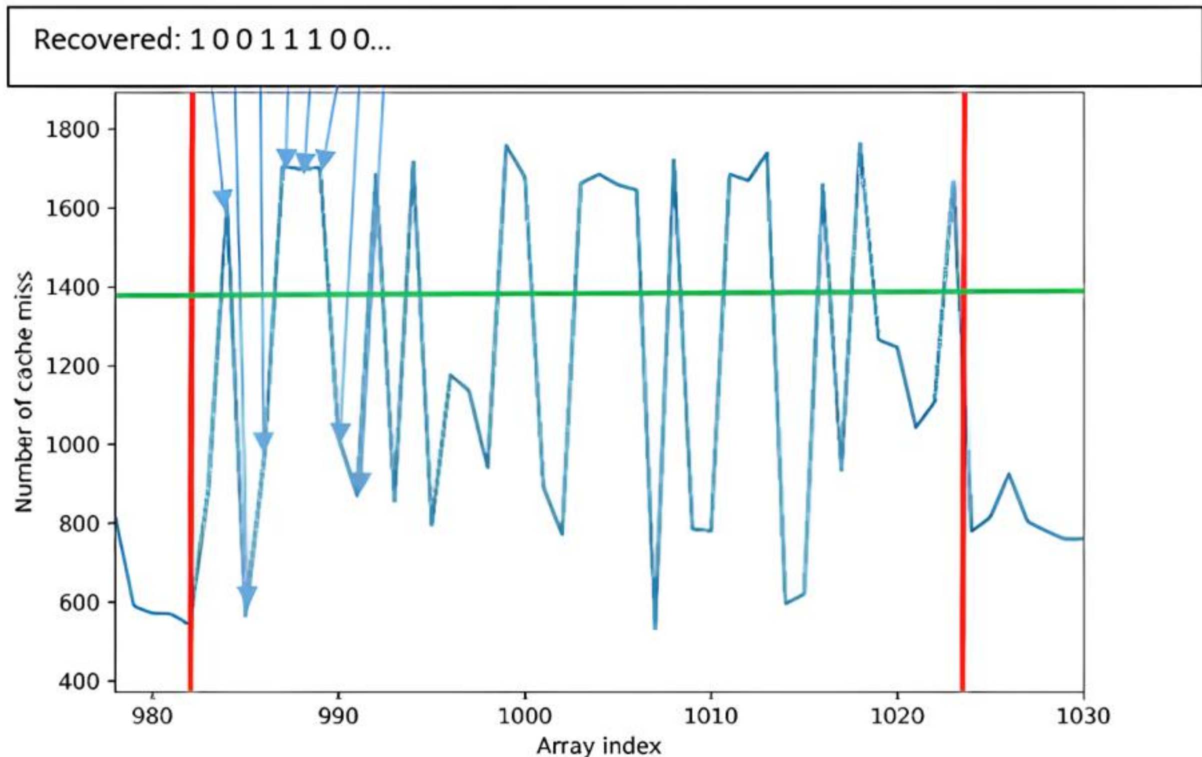


Figure 26. Zoom on one of the 8 patterns obtained when applying the covert-channel on a simple addition victim

6.5. AES case-study

Now that we studied the covert-channel's mechanisms, and a practical example on a very simple victim, let us study a more realistic use case. We propose to apply our implemented covert-channel on an encryption service. Let us consider a victim that is running an AES-based encryption service, in a use-case similar to the one of a Trusted Execution Environment (TEE). Here, the victim proceeds to AES encryptions and decryptions, as requested per the user. These encryptions use a secret key that will be the targeted secret value. As per the previously introduced threat model, the victim can only transmit this key to trusted entities. This means that the user cannot access the secret key, as it is contained in an untrusted domain. Moreover, the user can only interact with the encryption service through defined functions that only give limited outputs: the cipher text for the encryption function, and the plain text for the decryption function. Moreover, we consider that the AES' implementation is based on an open source code. This means that a Trojan might compromise the AES itself, and its future updates. However, as the compromised AES will run in a given security domain it cannot directly interact outside this domain. Instead of considering a legitimate user, we will consider a malicious attacker that is aware of the bug or the Trojan's presence and tries to leak the secret key using our implementation of a micro architectural cache covert-channel.

6.5.1. Implementation of the Prime+Probe covert-channel

This section describes our setup for this experimentation. The implementation of the covert-channel presented in this section is available online on GitHub[7] for reproduction purposes. We chose to work with the "Tiny-AES" [207] open-source implementation of the AES encryption algorithm for simplicity purposes. The "Tiny-AES" implementation is fast and easy to modify while also not embedding T-tables that might cause an important amount of cache evictions when running the victim. We chose a key size of 128 bits. The new victim code in this scenario calls some of the "Tiny-AES" functions when required. We consider that the library itself is compromised and contains the Trojan. For the application use-case, we still consider the application in a TEE environment. Therefore, the attacker (spy) still calls the victim process to request for an encryption. The victim then uses the library containing the Trojan. It is still possible to carry out the covert-channel in the case that the attacker cannot directly choose the moment the victim will be run. However, this requires more work for synchronizing the victim and the attacker. As it is not the primary goal of this section, these aspects will not be detailed further.

Changing the victim implies several changes on the covert-channel itself. Most of the changes occur for the Trojan as it is directly related to the type of victim targeted. Each victim requires a different Trojan, tailored for it. More specifically, the Trojan's code does not change itself. The changes occur in the interfacing between the Trojan and the target victim code. Compared to our previous experimentations, the Trojan now resides in the "Tiny-AES" library instead of being inserted directly inside the victim, as the secret key we want to extract is computed inside the library. More precisely, we inserted the Trojan inside the "Key-expansion" function, as it uses the key directly in every AES implementation. In our case the implementation of the "Tiny-AES" requires the victim to manipulate the key directly as it is an input of the library's functions. However, placing the Trojan inside the Key-expansion function should work for any AES implementation as it always uses the key directly. The new pseudo code for the covert-channel is presented in Listing 6. The *Prime(Table[])*, *Probe(Table[])* functions do not change at all from the previous pseudo-code presented in Listing 5 and are therefore not presented again. The Trojan(secret) function only embeds a supplementary module that will decompose the AES' key in a binary format.

The main challenges that arose from the victim change are: the handling of the Trojan's activation, the handling of the current part of the key to be extracted in the case it is bigger than the maximum extractible size. Extracting a secret bigger than the maximum chosen extraction size (here we chose 128 bits) requires more work but is possible, as detailed later in this section. It is important to note that pattern recognition is harder when approaching of the maximum extraction size for the considered cache configuration (e.g., 256 bits). When extracting secrets of 250 bits and above, the patterns are not separated by enough values to be distinguished by our analysis algorithm. For simplification purposes, we chose to stay with a 128-bit extraction size where the patterns are easier to distinguish.

[7] Available at: https://github.com/CCALK-work/CCALK, [accessed 06 Jan 2023]

Listing 6. Pseudocode for the covert-channel targeting the "Tiny-AES" implementation

```
KeyExpansion(roundKey, Key)
      …
      FirstRound()
      …
      IF (sequence != 0)
            trojan(secret)
      ENDIF
      …
      OtherRound()
      …

AES_INITIALIZATION(message, key)
      …
      FOR i in range(0, 10)
            IF (message[i] != i*4)
                  sequence = 0
            ENDIF
      ENDFOR
      iteration = meassge[11]
      …
      KeyExpansion(roundKey, key)
      …

Victim_encryption(message)
      …
      AES_INITIALIZATION(message, key)
      …

Spy()
      FOR i in range(0, NB_iteration)
            FOR k in range(0, 10)
                  message[k] = k*4
            message[11] = i
            ENDFOR
            FOR j in range(0, NB_samples)
                  Prime(Spy_table)
                  Victim_encryption(message);
                  Probe(spy_table)
            ENDFOR
            Generate_logs()
      ENDFOR
```

For the Trojan's activation, we wanted to avoid triggering the Trojan in case a legitimate user (therefore not a malicious attacker) uses the library. This would cause an increase in the computation time and therefore it would make the Trojan easier to spot. To this end, we chose to use a determined sequence inside the input message. For demonstration purposes, we chose that a specific sequence of values for the first 10 8-bit integers (equivalent to the first 80 bits) of the message would activate the Trojan. This sequence is set in the message at the lines 31 to 33 in Listing 6. The probability that a legitimate user triggers the Trojan

unwillingly is then 2^{-80}. It is still possible to consider a longer activation sequence to reduce this risk further.

For secrets bigger than 128 bits we included a selection mechanism. This mechanism is also based on the input message. We chose the 11th 8-bit integer of the message as an indicator of the part of the secret to be extracted by the Trojan. To extract a bigger secret, we split it into 128-bit parts (for example, as the patterns are easily seen when encoding 128 bits) that we can extract sequentially. For example, if we take a 512-bit secret, we split it into 4 pieces of 128 bits. The Trojan will then look at the 11th 8-bit integer to know which part of the secret it has to extract at the current iteration. If this integer is equal to two, the Trojan will then extract the second piece of 128 bits, starting from the bit number 129 of the secret key. From one iteration to another, the spy code increments these bits in the plaintext. With this technique, we can extract a key up to *128*256 = 32768* bits by pieces of 128 bits. For even bigger secret values, it is possible to extend the mechanism to two 8-bit integers (the 11th and 12th) or more if required.

In practice, our implementation of the Trojan code represents 40 lines of code inserted inside the “Tiny-AES” library that is composed of approximately 570 lines of code. This means that our Trojan increases the size of the library’s code by around 7%. The Trojan’s code should remain the same for bigger libraries, encompassing several other algorithms for instance. As the “Tiny-AES” has a small code size, the Trojan would probably be stealthier when placed in bigger libraries. Moreover, it is possible to improve the stealth of our Trojan’s implementation, and to reduce its number of lines. However, it was not the goal of our experimentations, and we did not go any further regarding stealth. This part is left for future work.

6.5.2. Results obtained and limitations

When we apply our implementation of a micro architectural covert channel on a “Tiny-AES” implementation having a 128-bit key, we obtain the results given in Figure 27. We chose to run the covert-channel with 2000 samples to improve the accuracy of the key’s retrieval. The trace is generated by our Python script using the logs produced by the spy code. The logs contain the time it took the spy to access each cell of *Spy_table*.

As on the traces obtained for the simple addition victim, we can observe the same type of patterns on this trace. As we are still working with the default cache configuration, we still have 8 distinct patterns that are looking very similar, as expected. Here we can also clearly see the offset’s presence that split one pattern over the figure. We chose not to correct this offset, as it does not influence the success of the attack at all. The result of zooming on one of the patterns is given in Figure 28.

Once we have generated the trace given in Figure 27, our Python script can recognize the patterns. As for the simple victim, the script picks the first complete pattern it finds and sets the threshold. It then recovers a binary value. Here the value recovered is composed of 128 bits just like the secret key. We then compare the value recovered to the binary decomposition of the key to check the attack’s success.

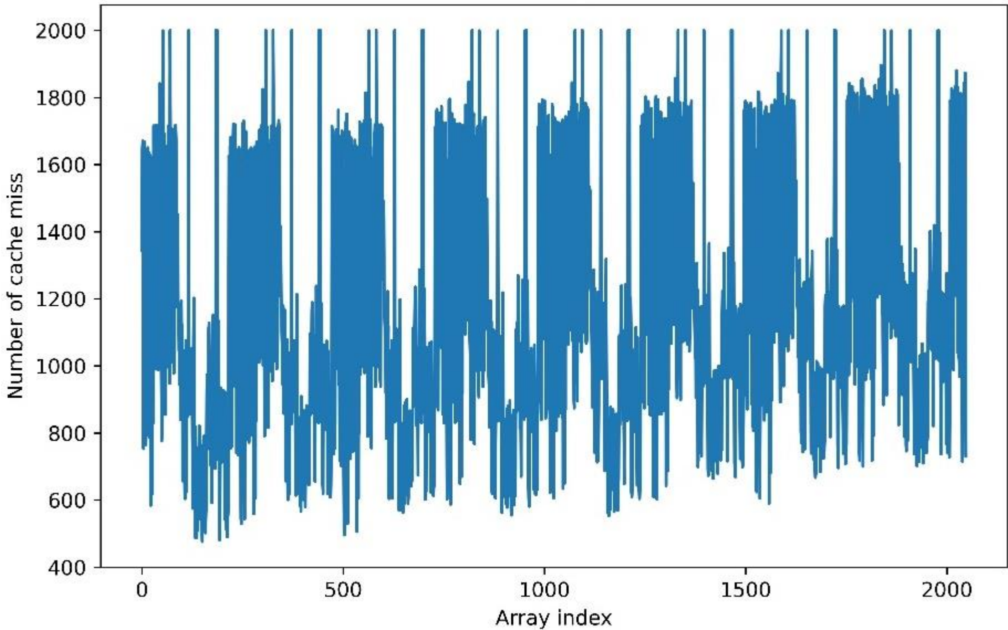


Figure 27. Trace obtained targeting the "Tiny-AES" with a 128-bit key and using 2000 samples

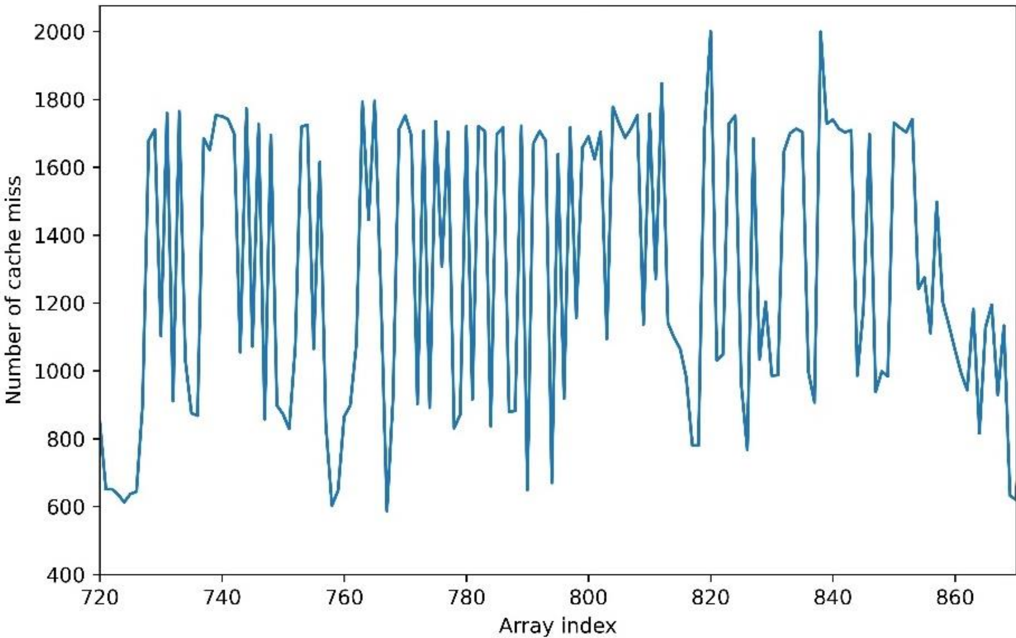


Figure 28. Trace obtained when zooming on one of the 8 patterns

We achieve to recover the 128-bit key with a success rate of 97.6%. More precisely, we are able to recover 97.6% of the secret key for every covert-channel we carry out. The 2.4% of error are due to the presence of ways in the cache where there is an important amount of cache misses, independently of the Trojan's activity. This is visible with the presence of some very high peaks outside of any recognizable pattern (some are also located inside the patterns). This means that there is some activity that is not caused by the Trojan nor the spy that evicts the spy's data before it can access it again. This effect is probably due to the presence of the Linux OS.

We interpreted these peaks as the processes related to the OS that are running while we carry out our attack. We cannot ensure that our covert-channel (victim + Trojan + spy) can be run within the timeframe allocated by the OS. Indeed, some other processes are run in

between, after the prime step, but before the probe step. This causes some of the spy's data to be evicted. These unwanted peaks are located at some indexes inside the patterns, causing 2.4% of the key being recovered incorrectly upon every extraction. Increasing the sample count reduces the impact of these unwanted peaks, and thus increases the success rate of the covert-channel. To illustrate, the covert-channel can recover 95% of the key correctly using 10 samples only. This considerably reduces the computation time for a slight decrease in the success rate.

Moreover, the covert-channel takes approximately 1 second when using 10 samples. We measured that we could run about 40 AES per second on our FPGA board without any covert channel. When using 10 samples, this corresponds to 10 AES encryptions per second, resulting in a noticeable slowdown. Again, we did not focus on making the covert-channel stealthy, and future work could focus on this aspect.

6.5.3. One more step towards realism: considering a multi-user platform with a noisy environment

We carried out some other experimentations focusing on the impacts of the "environment" on the covert-channel efficiency. We consider that the "environment" is mainly composed of the data cache's architecture, and the activity of the OS. These two elements are the ones having the highest influence on our covert-channel.

For the cache architecture, we carried out our covert-channel with several changes in the cache's parameters. We tried to modify the LFSR to see if it influenced the outcomes of the covert-channel. We alternatively changed the polynomial used while keeping the same degree, and changed the degree. In both cases, we did not notice any modification in the attack's behaviour. The success rate did not evolve significantly. However, the cache associativity has a very high impact on the covert-channel's behaviour. Changing the associativity means changing the amount of values we can extract. The higher the associativity, the more ways each set will contain. For a fixed cache size, this means that increasing the associativity decreases the number of sets, and thus the amount of bits a Trojan can transmit. We have an increased number of patterns on our traces, but each pattern will be composed of less values. The opposite is also true: decreasing associativity with a fixed cache size increases the amount of sets and thus the number of bits that can be extracted. We could still adapt the attack in several different scenarios. For associativity values of 4, 8, 16 and 32, the covert-channel is still working, provided it is adapted to fit the new environment. The success rate in these cases does not evolve significantly. We could not try some extreme cases, such as changing to a direct-mapped cache, or a fully associative cache (meaning that we have only one set composed of as many ways as the cache size requires) because these configurations are not supported by the CVA6's data cache. As a conclusion, we can say that the cache structure has an impact on the covert-channel. However, a simple adaptation inside the code is sufficient. Cache associativity has more impact on the attack as it directly modifies the amount of information transmitted. Intuitively, and considering the previous results, a fully associative cache would be more resilient to our proposed covert-channel, as we would only be able to extract one bit at a

time (as there is only one cache set). This would not make the attack impossible, but less practical and slower.

In an effort to propose an even more realistic approach to the application of our covert-channel, we carried it out in a “noisy” environment. We added some genuine clients (e.g., also embedding the Trojan but not activating it) running in parallel with a malicious user. We placed two other clients using the same encryption service as the attacker, with the same execution priority. These clients are doing AES encryptions in tight and infinite loops, in order to maximize the perturbations caused. These perturbations were created to amplify the effect limiting the success rate because of the un-wanted peaks appearing in the patterns. The effects caused by these two genuine clients were observable as represented in Figure 29 and Figure 30.

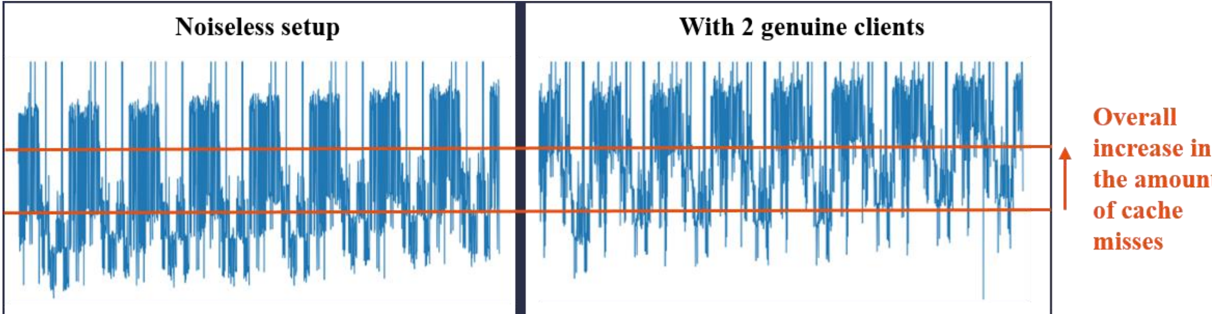


Figure 29. Comparison of the traces obtained in a standard and noisy setup

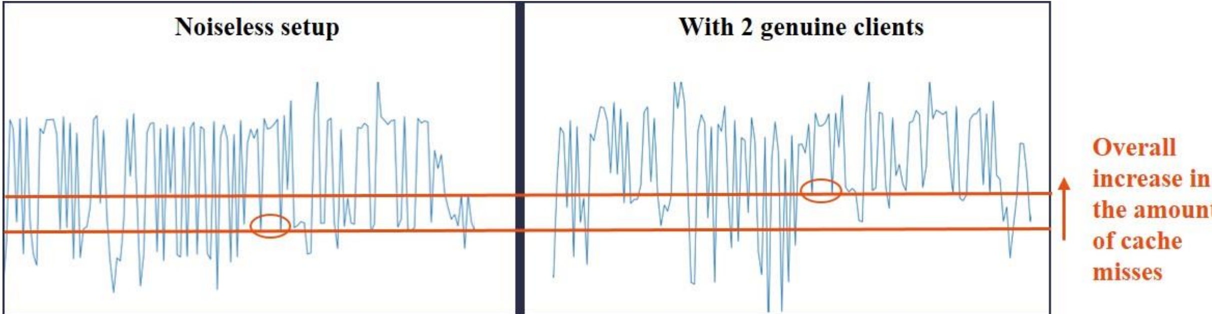


Figure 30. Comparison of the zoom on one pattern for a standard and noisy setup

The added perturbation caused a significant drop in the success rate so we had to increase the amount of samples to achieve a success rate of 95%. For example, the covert-channel without perturbations can achieve 95% of success rate with only six samples. To achieve 95% of success rate with the perturbations, we needed to use 100 samples instead. To conclude, if some processes are running concurrently with our covert-channel, the perturbations degrade the quality of the extraction but it is still possible to account for it by increasing the amount of samples considered.

6.5.4. Summary and conclusions on the AES case study

We presented an implementation of an access-driven cache-based micro architectural covert channel on the RISCV CVA6 core, in a running OS context. This attack is practical and has been applied to a simple victim consisting of an addition process then to a more realistic use-case, targeting an implementation of the AES encryption algorithm, with a 128-bit key. The attack's success rate is around 95% for 10 samples.

We also studied the challenges that the presence of an OS implies when carrying out such a covert-channel. Moreover, we also showed that the cache's architecture and dimensioning has an impact on the ability of these covert-channels to extract information, and therefore on the cache's level of vulnerability to such threats.

With more knowledge about the target and the victim comes more power for secret extraction. Open-source implementations of both hardware and software modules come at the price of new challenges for the designer to propose a secure platform.

A future work is to propose a stealthier version of the Trojan's implementation to make the covert-channel even more realistic and practical. It is also important to work on the potential mitigations to such covert-channels, as none of the observed perturbations could completely prohibit the extraction.

6.6. RSA case-study

The RSA [164] is an asymmetric cryptographic algorithm created in 1977. The following sections describe the implementation and experimental results obtained when applying the developed Prime+Probe covert-channel on an RSA algorithm instead of the former AES algorithm. The chosen RSA implementation can be found online [208]. The choice for carrying out this experimentation on an RSA algorithm has been motivated by the algorithm's structure: it uses the key bits sequentially. It was therefore believed to be an ideal candidate to apply the proposed implementation, as the sequential behavior would enable the attack to extract one bit at a time.

6.6.1. Implementation similarities and differences with the AES

The previous sections detailed the implementation of the Trojan program. This implementation has been conceived so that it can be easily applied to any victim. Indeed, the Trojan's code is very generic and does not rely on the victim's specificities to function. The core of the Trojan's code is therefore not modified when applied to the new RSA victim. It consists in the filling of the Trojan's array structure to cause the desired evictions inside the data cache. The part of the Trojan's code where the victim's secret value is accessed and recovered highly depends on the targeted victim and will therefore need to be adapted every time the Trojan targets a new victim. As a result, the Trojan code does not change in itself. Its integration inside the victim's code changes according to the target. In the specific case of the RSA algorithm considered here, the key is used sequentially. We implemented

the Trojan so that it could take advantage of this specificity by extracting the secret key one bit at a time.

Considering the performance of the FPGA instantiated CVA6, the execution of an RSA encryption is rather long (approximately five minutes). Considering this observation, it has been decided to recover the secret key completely during a single execution in order to get the result as quickly as possible. The other option would have been to recover a single bit of the secret per execution of the algorithm, resulting in a much slower overall key extraction.

Extracting completely the secret key at every execution of the RSA algorithm introduces important synchronization issues. Indeed, in the case of the AES algorithm, the synchronization was mastered by the attacker. In the considered scenario reproducing a TEE environment, the attacker can decide when to start the AES encryption and can therefore use this as a synchronization point to carry out the attack correctly. Figure 31 shows the synchronization process for the covert-channel targeting the AES victim.

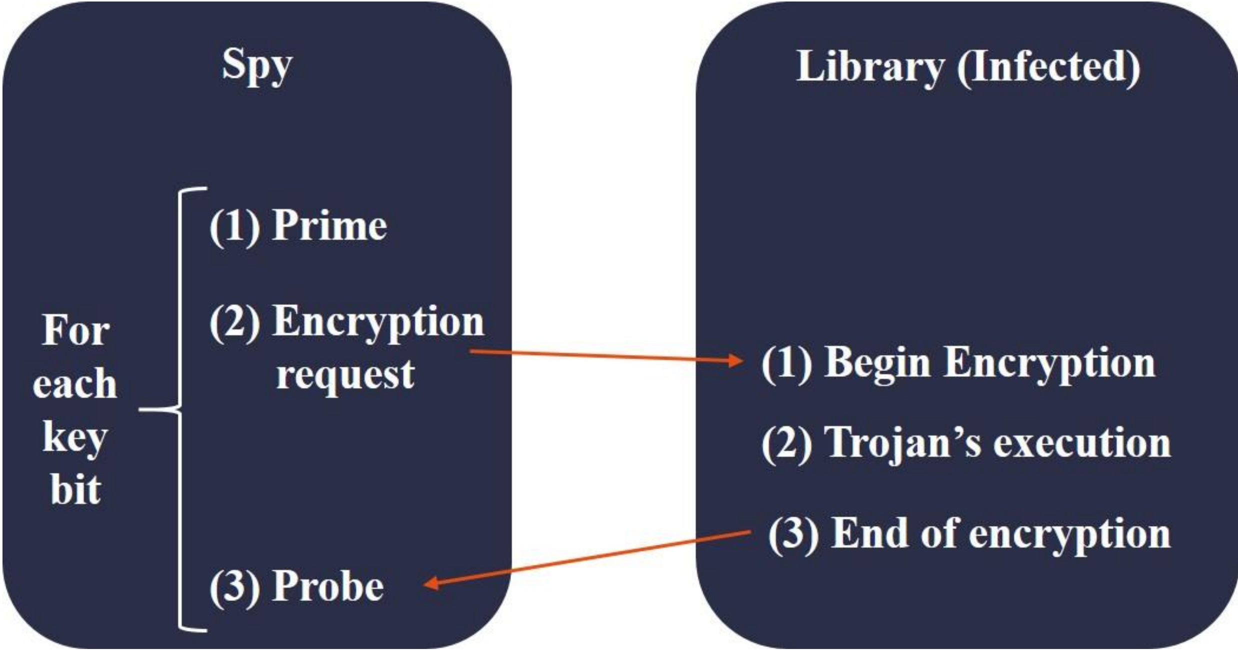


Figure 31. Synchronization process for the covert-channel when targeting an AES victim in a TEE environment

In this situation, the attacker can easily trigger the AES encryption at the end of the priming step and run the probing code once the victim has returned the result of the encryption. The case of the RSA algorithm is more complex however, as represented in Figure 32. Indeed, in this situation, the spy program needs to run two threads. The first one will run the prime and probe functions required to carry out the covert-channel. The second thread is responsible for triggering the RSA encryption service. The difficulty relies in the fact that once the second thread begins the encryption process, there is no possibility to synchronize the first thread (running the covert-channel) with it.

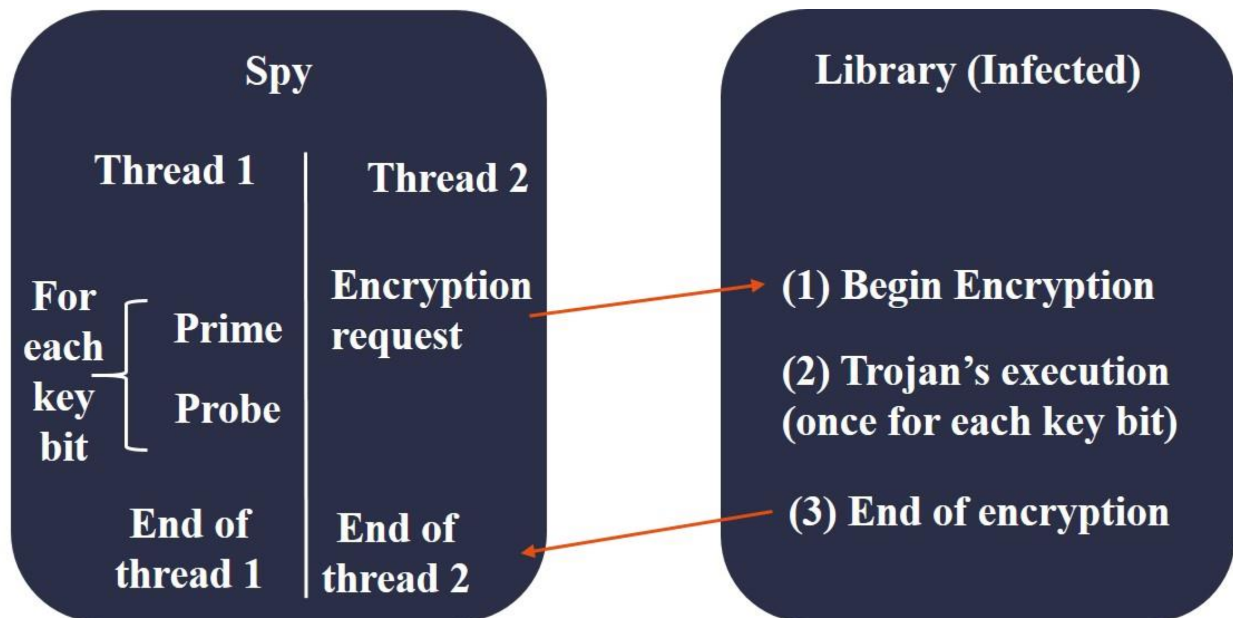


Figure 32. Synchronization process for the covert-channel when targeting an RSA victim

For the attack to be executed ideally, it is necessary that the priming plus the probing time is equal to the time the RSA algorithm needs to handle a single bit. However, according to our observations, it is never the case. We ran the RSA algorithm alone and concluded that it takes approximately 130ms to handle a key bit equal to one and 70ms in the case of a zero when we considered the key sequence equal to 00001.

On the other hand, the priming plus the probing steps take approximately 100ms. These observations imply that the desynchronization gets worse as the covert-channel and the encryption proceed. Additionally, the covert-channel is slower than the computing time for a key bit equal to zero. There is no efficient possibility to stop the victim encryption process. More specifically, there are some possibilities to stop the victim and re synchronize the attacker (using interruptions and other intrusive techniques) but it would result in a different and less stealthy attack protocol. The desynchronization between the attacker and the victim is bound to getting worse as the encryption progresses. These observations are summed up in Figure 33.

Moreover, as we are considering a mono-core scenario, a single thread can be run at any given time. This causes numerous context switches making the covert-channel more difficult to carry out, as it implies more perturbations and unwanted cache evictions caused by the switching of contexts.

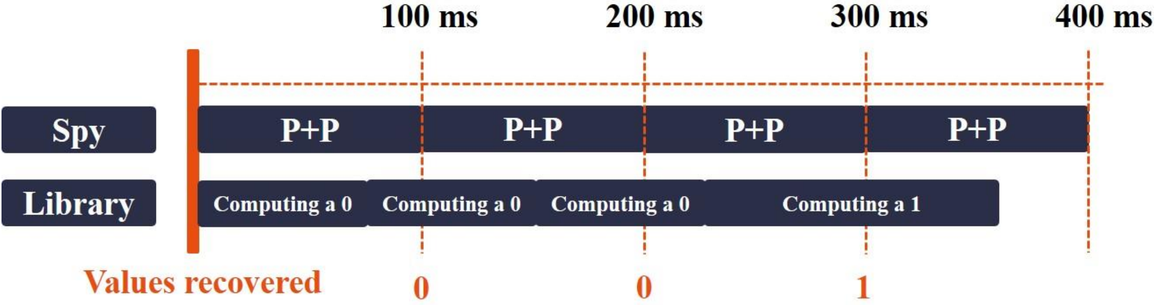


Figure 33. Illustration of the synchronization issues on an RSA victim

By looking at Figure 33, it appears that a small timing discrepancy is introduced between the two threads (attacker and victim threads). In the sequence shown when the third Prime+Probe covert-channel is carried out, the spy code recovers a one. However, it was supposed to recover a zero (as the third bit of the secret is equal to zero). Because of the previous observation, it is not possible to reproduce the same attack as on the AES victim. Indeed, for the case of the AES, the synchronization between the attacker and the victim thread was easier and led to the potential extraction of the complete key in only a single execution (even though the extraction rate is not ideal with only one sample). In the case of the RSA, it is required to make several assumptions and to proceed with multiple samples to get a chance to recover a partial key.

By assuming that there is on an average as many bits equal to one as there are bits equal to zero in the RSA key, statistically, the timing discrepancies should be evened out over a single encryption, and the effect should be even lowered further over an important amount of samples. To test this assumption, we proceeded to carry out the recovery of the 1024-bit private key as provided in the Github repository of the considered RSA algorithm. We applied the same method as in the case of the AES victim: causing cache evictions by filling data arrays coinciding with the data cache and translating the secret key into a binary decomposition so that it can be transmitted bit by bit.

We applied this technique with 2000 samples. It previously provided an extraction rate of over 97% for an AES victim. In the case of the RSA, we made a probabilistic estimation of the most likely value for each key bit recovered. If over 2000 samples a given key bit has been guessed more as a one than as a zero, we then assume that the correct guess is a one. By proceeding according to this method for all of the key bits, the extraction rate we obtained was not satisfying and revolved around 45%. This very low rate is mainly observed because we could not use more than 2000 samples reliably. Indeed, running 2000 RSA encryptions on the FPGA setup is very lengthy and trying to proceed with more samples led to lags and crashes. Moreover, the obtained extraction rate is about 50% questioning whether the successful key bit extracted are indeed guessed correctly by the spy code, or rather sheer luck over a high amount of tests. We chose not to investigate this issue further as the results produced were not encouraging enough.

6.6.2. Discussion and conclusions on the RSA

We carried out the same attack protocol on two targets: an AES and an RSA algorithm. The latter produced very unconvincing results. This can be explained by several synchronization and timing issues that did not exist in the case of the AES algorithm. Overall, the private key extraction in the case of the RSA algorithm could not produce an extraction rate greater than 50%. Several factors explain this result. First, the setup was a mono-core mono-thread scenario. For this reason, the synchronization for an RSA algorithm is harder to achieve due to its sequential behavior. A multi-threaded scenario would be more favorable to carrying out a micro architectural covert-channel on an RSA algorithm as a first logical core could spy on the second logical core. In this specific scenario, synchronization would be easier as demonstrated in the multiple micro architectural attacks that rely on Simultaneous Multi-Threading (or SMT) to be carried out [11, 44].

Then, the second issue is related to the attack protocol itself. Indeed, the threat model and the implementation of the Prime+Probe covert-channel have been developed to target an AES algorithm. This has several implications on the code itself that cannot be applied as such on an RSA algorithm. Indeed, the timing structure of the code has to be thought differently. To achieve better results, it would be necessary for priming and probing steps to be faster. The main condition is that the time it takes to prime and probe should be shorter than the time it takes the RSA algorithm to handle a zero (because it is the fastest case compared to a bit equal to one). In this specific setup, we would obtain the situation represented in Figure 34.

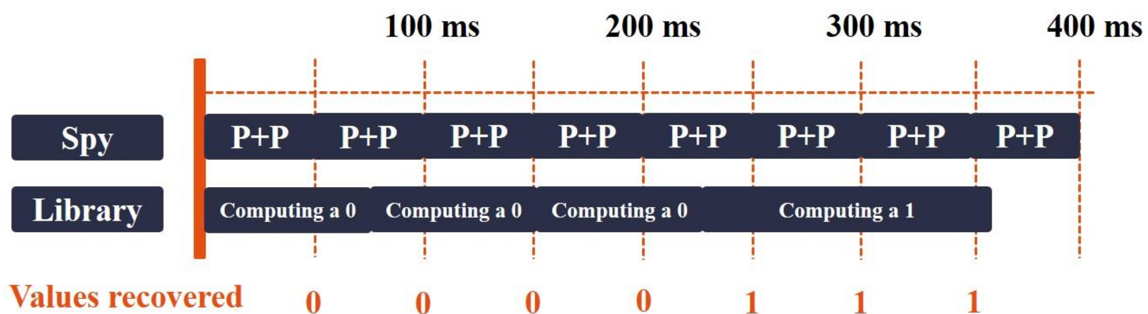


Figure 34. Illustration of the synchronization with Prime+Probe timing shorter than the handling of a key bit equal to zero

In this situation, the covert-channel will read at least once every value of the key correctly. It is then possible to sort out the values that have been read several times by measuring the timings of the attack and of the handling of the different key bits. With these timings, a python script can proceed to a post-process analysis in order to sort out the bits and remove the unwanted occurrences. Achieving a Prime+Probe covert-channel that runs in under 70ms would require rethinking the whole structure of the attack’s code. Moreover, it might also be more interesting to leverage another type of covert-channel such as a Flush+Reload, in a multi-threaded environment.

6.7. Discussion and limitations

The proposed attack is not a new technique in itself. Prime+Probe is a well-known covert-channel that has been implemented and leveraged in multiple works in the literature. However, to the best of our knowledge, it is the first time that the Prime+Probe covert channel has been implemented from scratch specifically on the CVA6 platform.

The proposed attack is practical and working for an AES and a simple addition victim. The extraction rate is satisfying as we are capable of recovering 95% of the 128-bit AES key after 10 samples reliably in a noiseless environment, and 99% of the key with 3000 samples still without perturbation. When adding noise such as generated by other processes, we were capable of extracting 99% of the key by using 4500 samples. Tormer et al. [209] proposed an implementation of the Prime+Probe covert-channel also targeting an AES algorithm on two single-core platforms: an Athlon 64 [210] and a Pentium 4E [211]. They manage to recover the full 128-bit AES key with 300 samples on the Athlon 64 and 16 000 samples on Pentium 4E. Figure 35 shows the results of [209] combined with our results for an easier visualization.

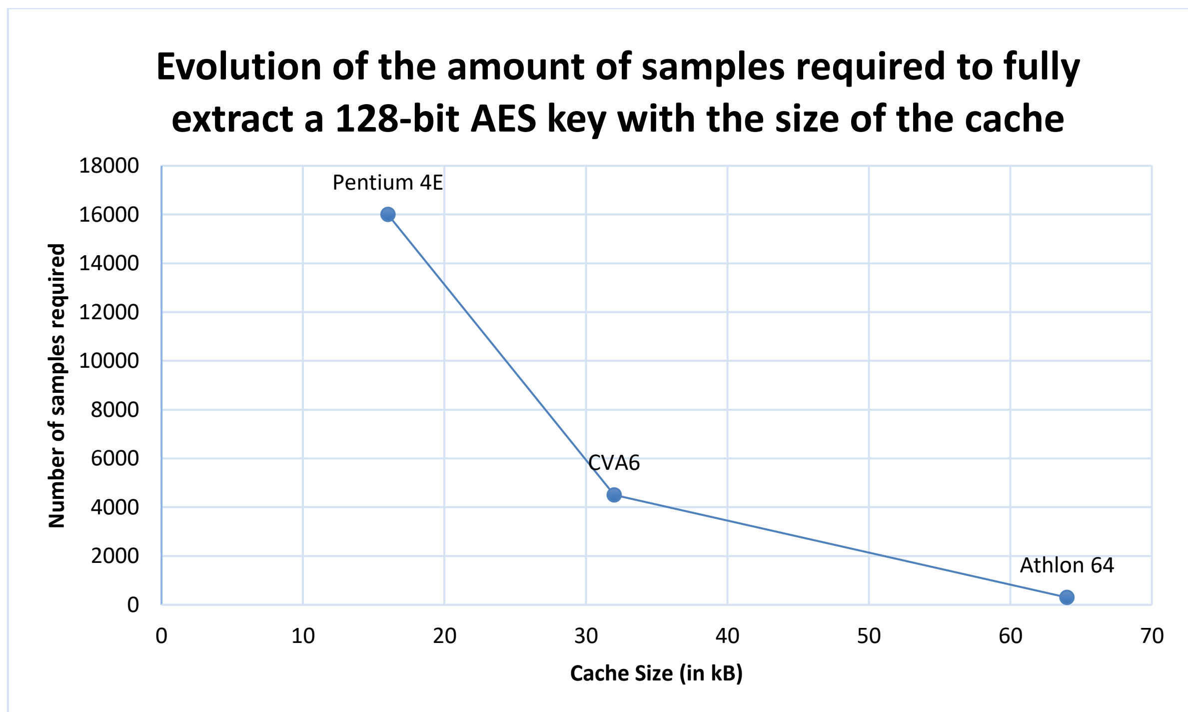


Figure 35. Evolution of the number of samples required to extract a 128-bit AES key varying with the cache size

This comparison shows that our results are coherent with the literature. Moreover, the Athlon 64 possesses a 2-way associative 64kB L1 cache while the Pentium 4E has an 8-way associative 16kB L1 cache. The CVA6 having an 8-way associative L1 cache of 32kB is located in between the Athlon 64 and the Pentium 4E, as our experimental results confirm. This shows that the bigger the cache, the faster an attacker can extract a given secret because the leakage is more important. This will be discussed more in details in section 7.1.

In addition to completing the literature by providing results on a new platform, our experimentations also provide useful methodology and fine-grained implementation details for reproducibility purposes as well as a better understanding of the root cause of the attack and of the difficulties that might arise when applying a well-known attack on a different platform or victim. Our studies show that implementing from scratch an attack that has already been studied in the literature in a new context is not straightforward and requires proceeding to additional steps (e.g., reverse-engineering of the cache architecture, observation of the victim's behavior inside the data cache…).

We also demonstrated that these additional steps are easier in an open-source context as the codes are available and can therefore be studied in details. Our work also shows that open-source contexts provide the unique capability to simulate a working core completely, thus giving the opportunity to attackers and mitigation developers alike to replay a given scenario indefinitely without needing to implement the core physically (on an FPGA platform for example). The main limitation of the simulated cores is the need for a powerful computer to run more complex simulations.

Finally, the obtained results and contributions have some limitations. Indeed, our experimentations were limited to single-core scenario, on an FPGA platform. This implies that a cross-core attack on a real core (and not an FPGA-emulated CPU) might produce different results and behaviors. Moreover, we had limited performance and realism due to the presence of the FPGA platform. In addition to that, the running OS on the FPGA platform was only a minimalistic embedded version of the Linux OS. Our experimental results were also limited by the fact that not all cache configurations were reachable on the FPGA instantiated CVA6, thus limiting our testing possibilities. Finally, our proposed attack proved to be effective against an AES victim, but could not be adapted satisfyingly to extract an RSA private key reliably, thus showing the limitations of our proposed implementation and attack protocol. Modifying the victim implies that our implementation has to be adapted likewise.

Chapter 7

Generalization of the proposed mitigations towards secure cores with respect to micro architectural attacks

---

7.1. Towards the development of mitigations: study the favorable and unfavorable conditions for an attacker

The previous experimentations on the AES and RSA victims were both aiming at the development of mitigations against the micro architectural covert-channel threat. These tests yielded interesting results from an attacker's perspective as described in the previous sections. This section will focus on the different results and conclusions that can be drawn in the case of mitigations and defenses. It will take back what has been introduced in the previous section and summarize the overall defenses contributions brought by this study. As in section 2.6, the mitigation approaches will be studied and classified as user-space mitigations, then system-level mitigations and finally hardware mitigations. All the mitigations studied in this section are applicable for single-core micro architectural covert-channels in the first place. Some can be derived and applied to cross-core scenarios, but further studies would be required to reinforce these assumptions.

For user-space level, the main defense source comes from the implementation of the victim algorithm considered. The programmer can implement it specifically to disrupt the attacking process and make any data recovery more complex. Works like [85-88] propose approaches to implement constant timing and data oblivious cryptographic algorithms. This technique would be effective in the proposed scenario as we are in a single-core scenario. Instead of aiming at a perfectly constant-timing behavior for our cryptographic algorithm, which has been demonstrated insufficient against certain attacks, we studied the possibilities to introduce variability in the victim's timing behavior. When applying the mitigation proposed in 3.3.2 on the AES victim, we observed interesting results. By adding several random temporal loops within the victim code, we produced a very different timing behavior for the same initial code. Moreover, this behavior changes between each execution. By adding two loops in the middle of the AES code that proceed to a random number of iterations between 100 000 and 1 000 000, we severely crippled the attack process. This mitigation modified the synchronization between the two processes and the covert-channel recovered an incorrect value on certain iterations. Overall, when using two random temporal loops in the implementation of the AES algorithm, the extraction rate dropped down to 47% on average. The drawback of this simple mitigation idea is the cost in performance, as the random temporal loops are just lost execution time as the computations done are useless for the AES

algorithm. We measured an increase of the AES execution time of around 100% on average for two random temporal loops. The proposed mitigation may not be ideal or optimized, but the idea relying behind it is worth being considered: changing the timing behavior of the victim program in an unexpected way for the attacker is an effective user-space mitigation possibility that has to be experimented and studied further.

For the system-level mitigations, the comparison between the RSA and the AES victim algorithms showed that changing the timing behavior and the secret-dependent operations induce difficulties for an attacker. One aspect that has not been studied in the literature in terms of mitigations and that has been pointed out during our comparative experimentations (especially visible during the RSA experimentations) is synchronization. Whether we consider a single-core or a cross-core attack, the issue remains the same: the attacker needs to know exactly when the victim is running and trigger his attack accordingly. Therefore, a potential mitigation proposition would be to reinforce the scheduling itself. Considering a scheduling process that induces variations such as randomly allocated timeframes for each process would highly hinder any attacker from carrying out a micro architectural cache covert-channel attack. This would make the preliminary observation and setting phases, where the attacker observes the victim's behavior and determines an eviction set, much more complex. Moreover, in a multi-processing environment, such a mitigation would be even more useful as the other running processes would then introduce a random amount of perturbation and cache evictions, based on the scheduler. This assumption is backed up by the experimentations we have detailed in this document where we introduced random-based perturbation such as the AES random number generation-based perturbation. Indeed, it proved that random cache evictions can be highly disruptive for the attacker and reduced the success rate. Providing such random evictions at the system level through a randomized OS-attributed time slice appears like an interesting idea to develop a mitigation. According to this idea, we have adapted the skeleton of a scheduler proposed in the baremetal study in section 5.4.1 Instead of determining the timeframes statically in advance, we experimented with allocating the timeframes randomly. This produces the behavior represented in Listing 7.

Listing 7. Pseudocode of the randomized skeleton of a scheduler

```
Prime+Probe_random_scheduling(void)
     Run rand(25, 75)% of priming_phase()
     Run rand(25, 75)%  of victim_process(secret_value)
     Run the remaining % of priming_phase()
     Run the remaining % of victim_process(secret_value)
     IF noise_generation == 1
          THEN run rand(25, 75)% of  eviction_perturbation
          (quatity_of_noise)
          Run rand(25, 75)% of probing_phase()
          Run the remaining % of eviction_perturbation
          (quantity_of_noise)
          Run the remaining % of probing_phase()
     ELSE
          Run 100% of probing_phase()
     Return(secret_value)
```

The timeframe was allocated randomly. It is represented here in percentage of execution for each function for visualization purposes. In this scenario, each function would be run partially. The partial amount of each function being run is chosen randomly however. This implementation of the scheduler disturbed the attack process. Similarly to the case of random temporal fences, the synchronization has been modified. Moreover, it happens that the priming or probing functions get interrupted midflight because the allocated timeframe was too short. With this modification alone (no other perturbations), the success rate of the attacked dropped significantly to 51% on average for the simple addition victim. On top of that, when adding our AES-based random number generator disruptor processes, we also observed random cache evictions at random timings because of the concurrent processes running in parallel. The success rate dropped to 10% in that case. Overall, providing non-linear timing behaviors that randomly change over the course of time at the scheduler level introduces a high level of variability that disrupts the attacker's capacities to synchronize, and to predict the victim's timing behavior. This approach could result in an interesting mitigation idea should it be studied further. It is unknown however if works like [109] proposing probabilistic cache covert-channel attacks could defeat such system-level randomness despite being able to counter cache-level randomness.

Wistoff et al. [101, 102] proposed the addition of a fence.t instruction in RISC-V ISA as a mitigation to micro architectural cache covert-channels. This instruction causes a flush of the data cache thus removing any potential sensitive information remaining in it. We tested the effectiveness of this proposition on our implemented attack scenario by adding fence.t instructions before and after the execution of the AES victim code. By first adding a fence at

the beginning of the AES computations, we ensure that the priming phase fails and that every data that the attacker placed inside the data cache previously will be evicted and experience a cache miss. Adding another fence instruction at the end of the AES computations ensures that no secret data remains within the data cache after the victim is being run. This mitigation technique, when applied to our proposed Prime+Probe implementation produces an extraction rate of 47% on average. Indeed, the attack only recovers key bits equal to zero (e.g., when obtaining mainly cache misses on a given cache set), which on average correspond to half of the different secret keys studied as they almost contain as many zeros and ones. We can conclude that the mitigation is effective and causes the attack to extract correct bits only based on luck, thus hindering the covert-channel. There are some works that propose methods to recover complete secret keys based only on a few correct bits [212] using arithmetic calculations, however in this specific case, it might not be possible as we are unable to determine which bit of the recovered key is correct.

For the hardware level, we have studied the impact of the cache's structure on the capacity of the attacker to extract information. Cache architecture and disposition has been thoroughly studied in the literature as shown in section 2.7.2.2. Our experimentations propose a complementary analysis of the cache architecture impact on the level of vulnerability of a given cache implementation. By studying the applicability of our proposed attack on an AES algorithm on an FPGA board and several cache configurations, we proved that the size and associativity of a given cache could reduce an attacker's capacity to recover information. Indeed, we showed that the larger the cache, the more information the attacker can extract, and the more space it gives a malicious application to work with within the cache. If the cache is sufficiently big, the attacker can conduct a micro architectural cache covert-channel without being bothered by unwanted cache evictions. In complement to cache size, the associativity also plays in important role. Indeed, it determines the cache set size and thus it needs to be taken into consideration by the attacker when determining eviction sets. Our experimentations showed that changing the associativity also leads to modifying the potential effectiveness of the Prime+Probe covert-channel. Indeed, the higher the associativity, the more cache lines are available in every cache set, thus increasing the probability of set sharing between the victim and the attacker. However, this increase of the amount of cache lines available in each cache set comes at the cost of an increased probability of another unwanted process sharing a given cache set. Increasing associativity therefore enables an attacker to extract more data, more efficiently, but also raises the probability of suffering from unwanted cache evictions on the targeted set. Cache size and associativity are therefore two parameters that will determine the attacker's capability to extract information on a given cache configuration. Depending on their values, carrying out a micro architectural cache covert-channel can be either complex or very effective. Figure 36 shows the evolution of our Prime+Probe covert-channel when changing cache associativity for a given cache size.

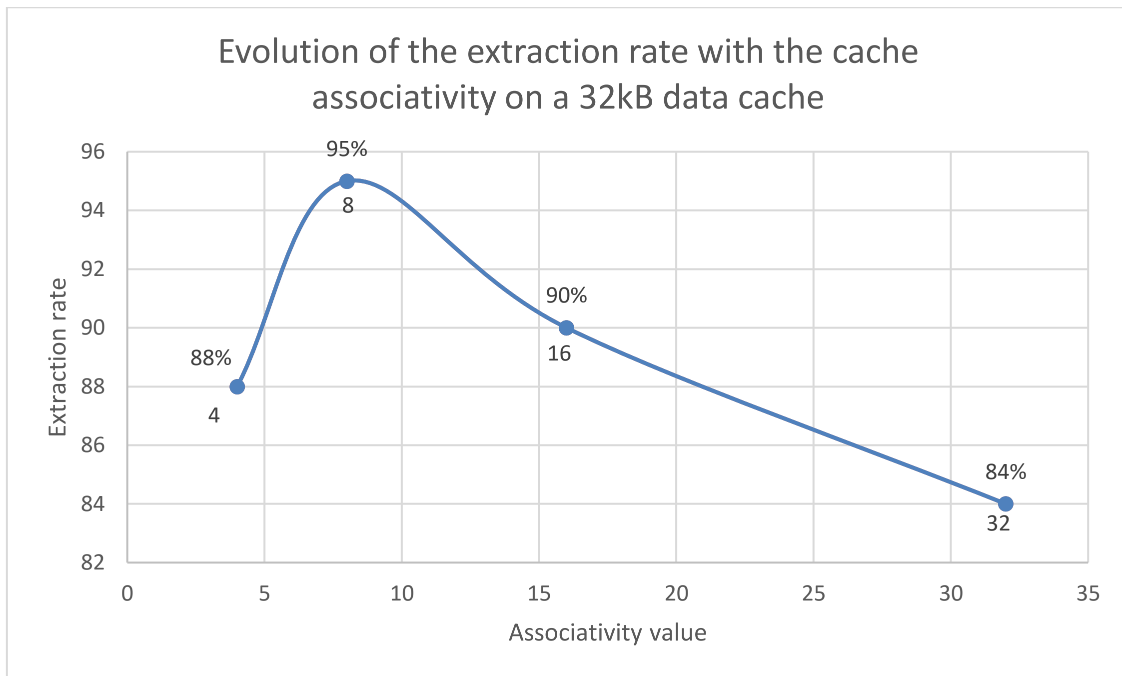


Figure 36. Evolution of the extraction rate with the cache associativity for a 32kB data cache

When developing CPUs, developers study the best combination of cache size and cache associativity to maximize their core's performance. Similarly, it is possible to study the evolution of this combination to find the best values that minimize the attacker's capacity to extract information. Having a cache architecture that is, by design, more complex to attack (because the attacker cannot transmit much data on each iteration, or has a high probability of being disrupted) could deter some attackers from carrying out cache covert-channels. Moreover, such a cache disposition would also be easier to protect, as the severity of its leakages is lower. The impact on performances would have to be evaluated as well as the possibility to achieve efficient trade-offs.

According to our experimentations, and our assumptions based on the configurations we could not achieve with our experimental setup, we concluded that the optimal disposition for an attacker is to target a cache with an associativity of eight, as shown in Figure 36. From a mitigation perspective, the caches that present the lowest leakages should be direct-mapped caches and fully associative caches. Direct-mapped caches totally remove the chances for the attacker to share a cache set with the victim, as each cache set is composed of only a single cache line. Fully associative caches are also not ideal for an attacker as they are composed of a single set containing all the cache lines. The attacker shares a cache set with the victim, but also with any other running process, increasing the chances of facing unwanted cache evictions. Figure 36 proposes an analysis of the results we obtained during our study and could be completed by carrying out further experimentations on different platforms that propose cache configurations that could not be implemented with the FPGA-instantiated CVA6. It can be used to evaluate the theoretical level of vulnerability of a given cache architecture (associativity and size combination) when facing timing micro architectural cache covert-channels.

7.2. Our proposed micro architectural modifications for a secure handling of sensitive data

As a complement to the proposals summarized in the previous section, this section will focus on a modified CPU micro architecture. Our goal with this proposal is similar to the on presented by Escouteloup et al. in [119]: to propose building blocks and design guidelines to achieve micro architecturally resilient core designs. During this work, we came across design modifications that could help mitigate part of the cache micro architectural covert-channel threats. In summary, the proposed architecture contains an additional secured bus to handle the transfer of sensitive data between the CPU and the main memory (or between the CPU and a dedicated additional memory). The goal of this additional bus is to circumvent the micro architectural resources that present leakage possibilities including, but not limited to, the load ports, store buffers, line fill buffers and cache memory. For this addition to work properly, we propose that the ISA of the related CPU contains two supplementary instructions for triggering the secure transfer through the additional bus. The first operation would result in a secure loading of the sensitive data in the CPU from the primary memory by transmitting it through the secured bus. The second operation would consist in securely storing the sensitive data from the CPU to the main memory transiting through the secure bus. These additional instructions would be similar to adding a *secure_load* and a *secure_store* instruction to the ISA.

Our proposed mitigation is aimed at being implemented inside modern CPUs for applications where micro architectural threats have to be mitigated. The typical CPU architecture before implementing our micro architectural modification is given in Figure 37.

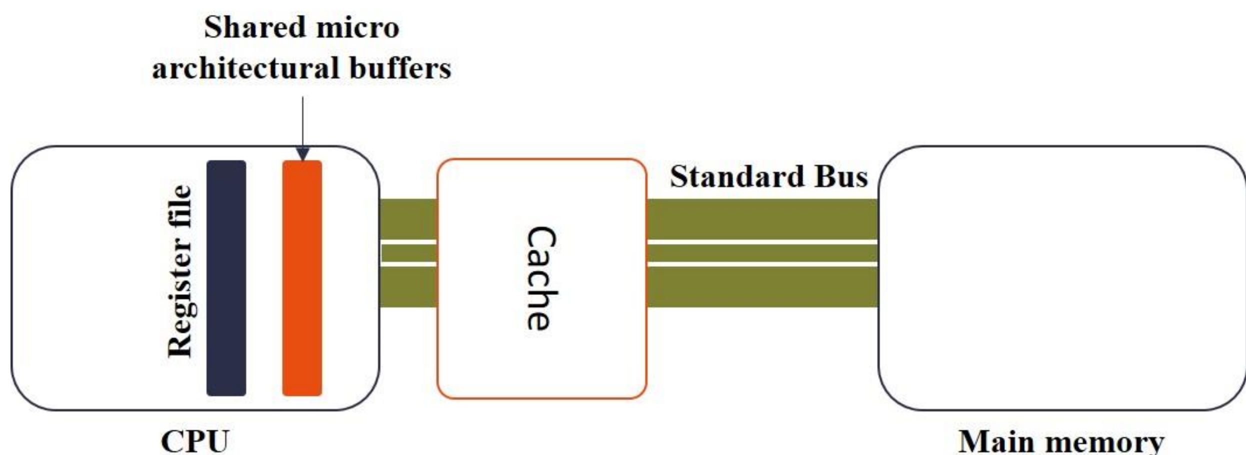


Figure 37. Schematic representation of the micro architectural memory elements of a typical CPU

All the micro architectural memory elements mentioned above are intended to speed up the memory load and store requests and thus to increase the CPU's computational efficiency and save some precious time. They act similarly to cache memories and contain different types of data. These inner micro architectural buffers have been exploited by the authors of the MDS attacks that reverse engineered their behavior will carrying out a micro architectural attack. We will now briefly remind what these three micro architectural cache-like memories are. A load port is a small buffer activated when data is loaded into registers.

It keeps track of the previous load operations and enables subsequent loads to the same address to be computed faster by forwarding its entries to the CPU. A store buffer holds temporary data about the recent store operations. Its entries may be forwarded to the CPU on subsequent load operation with addresses that are similar to the ones contained in the store buffer's entries. The line fill buffer is a small buffer between the L1 cache and the lower-level memories. This buffer stores load requests that resulted in an L1 miss before handing them to the cache.

Upon a typical load or store operation, all the micro architectural optimization mechanisms related to these memory units are triggered. The CPU checks the presence of the requested memory element inside the different cache-like structures. Upon usage, these hardware elements will be modified and therefore they will contain a footprint, i.e. some information about the previous operations that were computed. This footprint enables subsequent load or store operations to the same addresses or on the same data to be faster.

However, as shown in the literature as well as in the different experimentations presented in this document, the information stored in these micro architectural memories can leak. The cache memories are known for being subject to cache covert-channel attacks, and the other micro architectural buffers have been exploited by the different MDS attacks even if they were not documented initially. Figure 38 represents the potential locations of the micro architectural leakages.

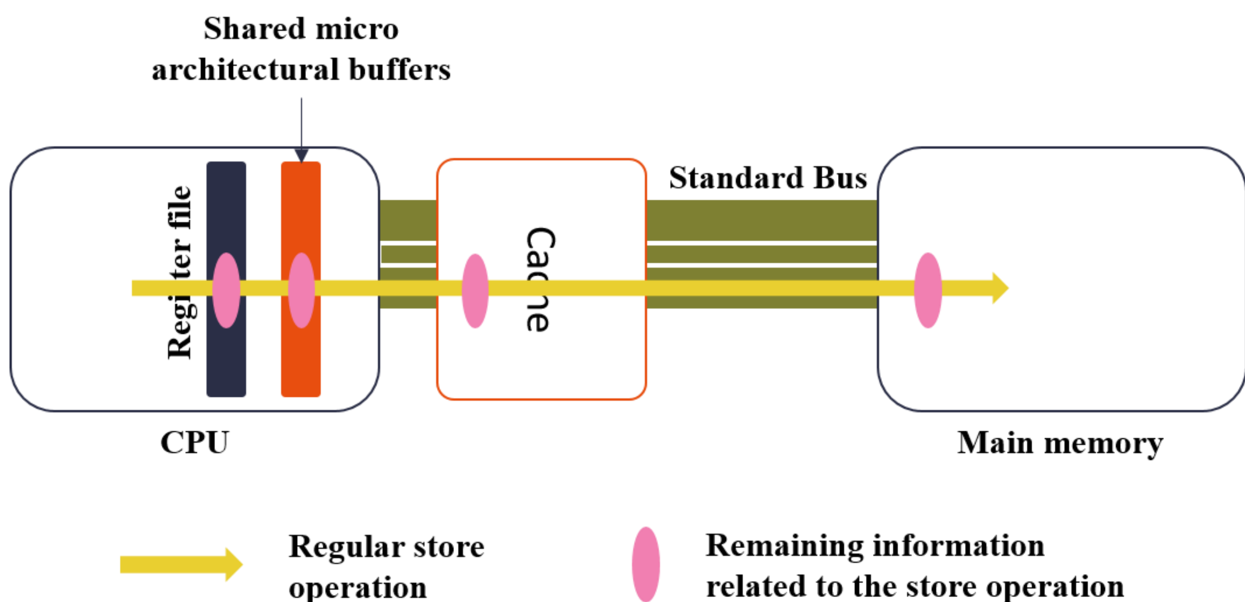


Figure 38. Schematic representation of potential micro architectural information leakages after a regular store operation

These leakages mainly come in the form of timing discrepancies. Depending on the presence or not of data inside these buffers and memories, it produces timing differences that can be observed and exploited by an attacker leveraging micro architectural attacks. The main goal of the proposed micro architectural modification is to add an alternative path for handling sensitive information inside the micro architecture that would not lead to leaving such timing information. Figure 39 proposes a schematic representation of our mitigation.

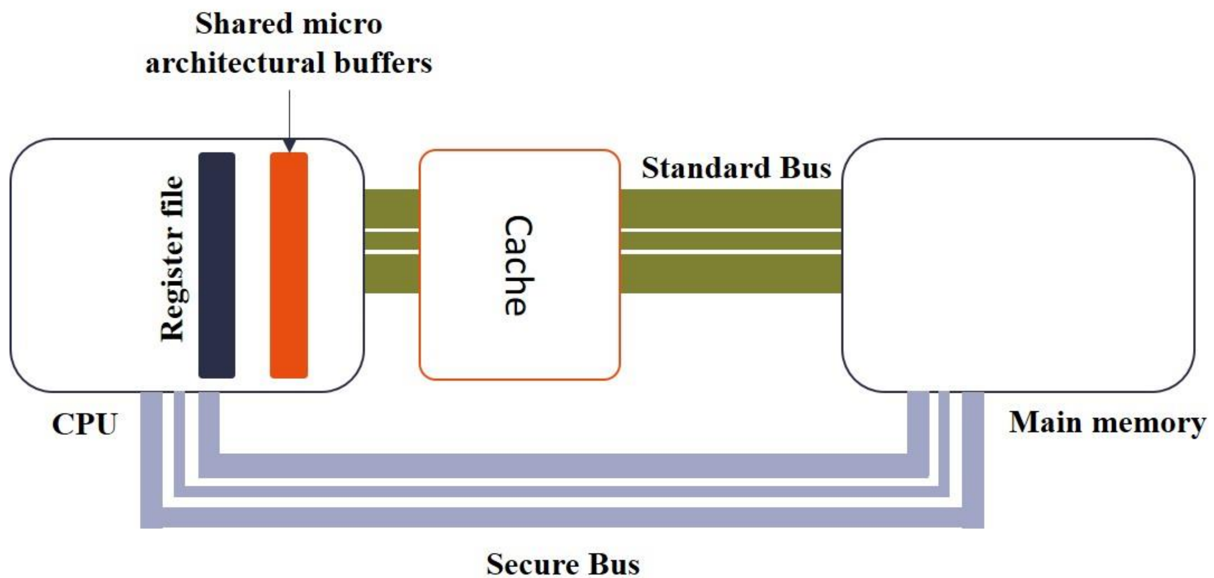


Figure 39. Schematic representation of a CPU with the proposed micro architectural modification

The main proposition of our modification consists in adding a secure bus to the micro architectural design of the core. This bus is used for secure transfer of data from/to the CPU to/from the main memory. Moreover, this secure bus is designed to bypass the micro architectural buffers and memories that present data leakages such as the line fill buffer, load ports, store buffer and cache memories. This should prevent several variants of micro architectural attacks such as the MDS attacks, and micro architectural cache covert-channels. Indeed, these attacks would not recover any data, as the sensitive information would transit through the secure bus directly to the main memory, without leveraging the optimization mechanisms. Therefore, the usual footprint that is exploited by attackers during micro architectural attacks would not exist, rendering micro architectural attacks relying on this type of information ineffective. As to not reduce drastically the CPU's overall performance by proposing a solution close to removing the caches and micro architectural buffers, our proposition includes the addition of two additional instructions: *LOAD_SEC* and *STORE_SEC*. These instructions are designed to complement the regular *LOAD* and *STORE* operations. These *LOAD_SEC* and *STORE_SEC* are designed to be used by the programmer when handling sensitive data such as secret cryptographic keys or passwords. When encountered, the system is aware of the sensitive nature of the handled data and can then trigger the appropriate operations. The *LOAD_SEC* and *STORE_SEC* instructions take effect upon being decoded. The concerned data will not transit through the regular bus and thus through the regular micro architectural buffers, but will instead be transmitted through the secured bus. This process should leave no micro architectural traces that could be exploited by an attacker. The usage of the *STORE_SEC* operation is represented in Figure 40.

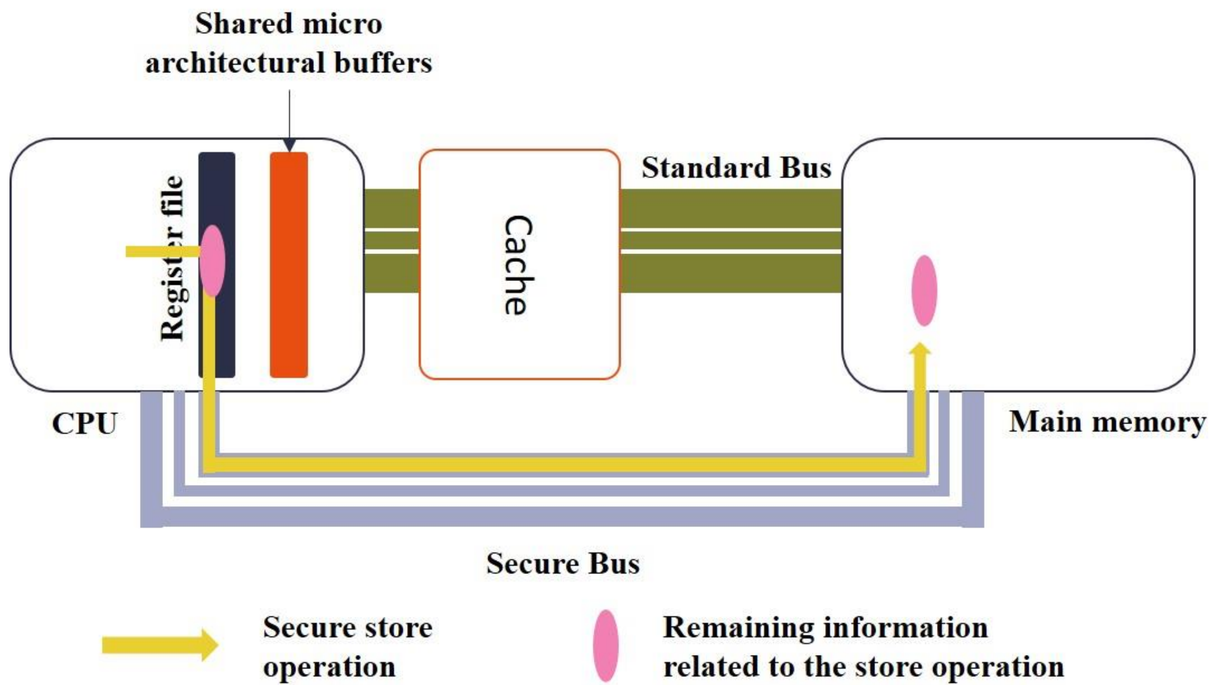


Figure 40. Schematic representation of the STORE_SEC instruction on CPU using the proposed micro architectural modification

These instructions will be made available to the developer so that he can tag sensitive data during the implementation that will result in the use of the secure bus only when handling the most critical data. This enables not to decrease the CPU's performances too radically even if the secured part of the code will experience a longer computation time compared to the rest of the code. The overall cost of our proposed mitigation would therefore reside in adding a bus, and slightly modifying the ISA to add two instructions. The addition of these two instructions combined with the proposed hardware modifications enables to reduce the potential sources of micro architectural leakages as depicted in Figure 40. Indeed, the remaining elements that could present leakages are now limited to the register file, and the main memory.

We also established three alternative embodiments of our micro architectural modification. A first alternative embodiment implements a virtual version of the *LOAD_SEC* and *STORE_SEC* instructions. This has been thought useful in cases where modifying the ISA would be cumbersome. The main difference of this alternative is that the secured version of the *LOAD* and *STORE* operations are triggered when encountering a store or load on a specifically predefined memory location. This could also work if a specific address of the operand is located within a specific and protected memory area. Similarly to the initial idea, the developer would then have to declare and place his sensitive variables in these specified locations to protect them. Therefore, each load or store operation would trigger an address check to determine whether it belongs to the secure area or not. This solution therefore only induces modifications in the hardware design itself, but not in the ISA.

A second alternative implementation would consist in adding a supplementary dedicated memory. This additional memory would coexist with the standard main memory. However, the additional secure bus would then connect the CPU to this secured memory directly.

More specifically, in this design, the *LOAD_SEC* and *STORE_SEC* operations would cause a load and store through the secure bus directly from/to the secured memory, which is dedicated to holding sensitive information. This proposition is represented in Figure 41. The advantage of this solution is that the addition of the secured memory does not add extra complexity to the design, while removing the potential leakages that exist in the main memory. However, this solution causes the most important increase in the overall area of the chip. It is worth to note that this implementation has no significant impact on the execution time of regular unsecured operations.

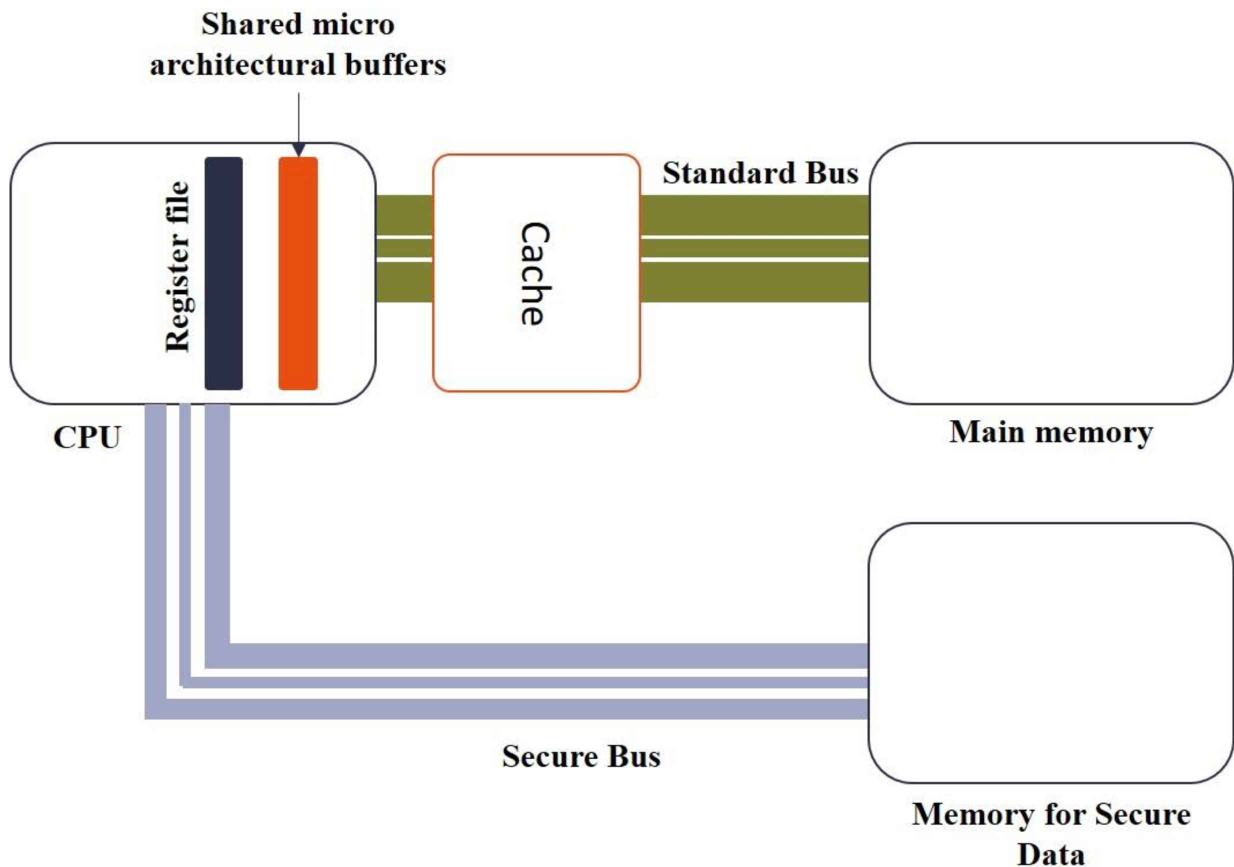


Figure 41. Schematic representation of the alternative implementation where a dedicated additional secured memory is added to the CPU

Finally, our last proposed alternative implementation consists in adding the secure bus as a partially virtual additional bus. The additional secure bus would therefore coexist within the standard initial bus. However, upon triggering the secured load and store operation, it would cause a flush of all the leaky micro architectural resources. Additionally, all the memories and buffers that have been flushed would remain temporarily deactivated until the secure processing is over. A technical solution to implement this idea would be to use multiplexers that would be triggered when decoding the secure instructions. This alternative implementation presents minimal hardware modifications compared to the others, and it does not cause additional computation time for regular operations.

7.3. Discussion and limitations

The work presented in this document lead to several mitigation ideas and propositions. Overall, we proposed some complementary results and studies to reinforce and complete what is already available in the literature, as presented in section 7.1. The general idea of section 7.2 was to propose guidelines, and design ideas towards a CPU architecture that would be, by design, immune to most micro architectural threats.

Some mitigation propositions specifically designed for RISC-V platforms have been studied in the literature, such as the fence.t instruction [101]. This document proposes mitigation guidelines at all the possible levels: user-space, system and hardware. Overall, it appears that the best approach is to develop mitigations at the hardware level, as the root cause is located within the micro architecture itself. Our different experimentations confirmed that the hardware design is the core issue to be considered when developing a resilient CPU with respect to micro architectural threats. Section 7.1 described our experimentations when testing the different types of mitigations available at different levels as to determine the most promising of them. This study lead to considering the micro architectural hardware modifications as the most effective and generalizable ones, hence the proposed micro architectural modification presented in section 7.2.

Section 7.2 presented a novel mitigation consisting in the addition of a supplementary secure bus for data transfer. This design led to patent application number 22305710.0 at the European Patent Office [213]. Even though this proposal appears promising, it has not been implemented yet. We could not test its effectiveness against our proposed Prime+Probe covert-channel, and leave the evaluation of this mitigation solution for future work.

Conclusion and future works

8.1. Summary of the thesis

The recent multiplication of micro architectural threats such as Foreshadow and the MDS attacks shows that this is a real threat that has to be considered by the processor designers. Moreover, these leakages affect all CPU vendors likewise. The race for CPU performance led to the development of many hardware and software optimization mechanisms. These mechanisms rely on underlying shared hardware resources such as: caches, speculative buffers, load ports, line fill buffers, store buffers, etc. The security implications of the addition of these memories have not been studied thoroughly beforehand and led to the discovery of many leakages that can be exploited using micro architectural covert-channels. These leakages have an important impact on the security and privacy of the recent processor architectures. Moreover, cache and other small inner micro architectural buffers have been pinpointed as a major source of micro architectural leakages over the past years, and the interest for security surrounding these elements has risen equally. In addition to that, the fast ascension of open-source initiatives such as the RISC-V ISA led to considering several new challenges. Indeed, RISC-V cores are as vulnerable as closed cores (e.g., Intel, ARM, AMD) yet their source code is available to anyone, attackers and mitigation developers alike. This introduces several advantages and drawbacks that need to be taken into account when considering micro architectural threats.

The literature contains a multitude of mitigations with respect to micro architectural attacks (see section 2). These protections either take place at the application level, the operating-system and hypervisor level, or the hardware level. All the above have their own advantages and drawbacks. When considering the overall state-of-the-art of micro architectural mitigations, it appears that only a few works have tried to propose CPU-wide protections. Most of the existing works focus on a very specific attack they are trying to hinder, or on protecting a specific part of the design itself such as the data cache. Despite having a wide array of available protections, there are no options providing extensive defensive solutions. RISC-V is also starting to be studied in terms of attacks and mitigations alike. We are only at an early stage for the studies of open-source initiatives, but there is a real trend to propose resilient open-sourced designs for the community.

The work proposed in this document attempts to exhibit the drawbacks and advantages of open approaches when conducting micro architectural cache covert-channels, while pinpointing the root cause of such vulnerabilities. It studies the methodology for applying a well-known attack on a new platform while detailing how an open-source context facilitates or hinders the implementation work. This thesis also tries to contribute to the general effort towards proposing building blocks for resilient CPU designs with respect to micro architectural threats.

In this work, we first proposed an in-depth reverse engineering study of the CVA6 data cache. This study was necessary for carrying out the implementation work of the attacks that were later developed. We provide detailed behavior analysis as well as experimental results

of typical use cases on the data cache. The proposed study is aimed at being the starting point for developers willing to either replicate micro architectural attacks or propose mitigations revolving around the data cache.

We then implemented a first simulated baremetal version of the Prime+Probe cache covert-channel on CVA6. This implementation work shed light on the simulation possibilities offered by open-source initiatives, enabling infinite replicability for a potential attacker without even needing to get the physical product. Our attack is practical and show a high success rate of 85% for values between 0 and 131. A preliminary study showed the impact of the addition of an OS. The additional noise caused by concurrent processes and the scheduling are two challenges to be considered when tailoring a micro architectural covert-channel.

The results of the previous study were leveraged in order to propose another implementation of the Prime+Probe covert-channel on an FPGA instantiated CVA6 running Linux. This implementation aims at showing the effectiveness of such an attack in a more realistic context, targeting an AES cryptographic algorithm. This attack is also practical and has a success rate of 97,9% for 2000 samples in a noiseless environment (e.g., without additional processes causing unwanted cache evictions). Readers can access, reproduce and distribute the source code for this implementation at the Github repository [214].The consequences of the addition of an OS have also been studied. The addition of concurrent processes generating perturbations as well as the impact of the OS' scheduling were experimentally tested out. Moreover, we carried out experimentations to observe whether the cache architecture itself could influence the level of leakage of a given cache implementation. Cache parameters such as the cache size or the associativity proved to have an impact on the vulnerability of the cache with respect to micro architectural covert-channels. This study laid the foundations for the mitigation works that are presented in this thesis.

The work presented in this document studied some existing solutions to protect RISC-V platforms against micro architectural attacks. Moreover, it proposed some ideas to reinforce mitigation possibilities at the user-space, system and hardware levels. Finally, this thesis introduced a new micro architectural design idea to reduce the risk of micro architectural leakages being exploited. This proposal comes in the form of an additional secure bus that bypasses the leaky micro architectural buffers and memories. This proposition tries to participate in the general effort of laying down the foundations for designing micro architecturally resilient CPUs based on open initiatives.

This thesis brings value-addition and novelty in multiple ways. We proposed a methodological analysis on the implementation of a micro architectural Prime+Probe covert-channel. We also studied the preliminary reverse engineering step in an open-source context and pointed out the differences with closed designs. The results of this reverse engineering step will enrich the available documentation of the CVA6 core. To the best of our knowledge, this is the first effort proposing a detailed documentation of the CVA6 data cache structure and behavior [197]. Despite proposing implementations of well-known attacks, this work leverages the new possibilities introduced by the open-source context such as simulated attacks. It draws conclusions and details the advantages and limitations of such approaches. Moreover, this thesis proposes a more realistic implementation on a

realistic use-case. This study details all the implementation process and the codes can be accessed online for reproducibility purposes. These experimentations also pointed out the challenges to overcome when applying a well-known attack to a new hardware/software scenario. This thesis also enriches the state-of-art on the mitigation aspects by proposing additional results based on a RISC-V platform to complete already existing results and polarize them in the specific case of open-source cores. In addition, it also introduces new mitigation ideas at all the potential levels where protections can be introduced. Finally, this thesis proposes a new micro architectural design modification that aims at reducing micro architectural leakages on sensitive computations. This proposition lead to a patent application at the European Patent Office.

8.2. Future work

8.2.1. Research perspectives

One of the main aspects of micro architectural threats is there multiplicity. As demonstrated by the current state-of-the-art, attacks have been targeting a very wide array of micro architectural elements, making almost every CPU but the simplest ones vulnerable. The recent rise of open approaches show an evolution in the design and management of CPU architectures. Micro architectural attacks being related to the implementation of a given ISA, cores based on open-sourced ISAs such as RISC-V are also subject to micro architectural threats. Thus, open source implementations are now being studied as alternatives and their security implications are becoming a major focus. Previously under documented micro architectures are now openly accessible to a wide audience, raising several challenges. Even if the trend is beginning to change over the last few years, most works related to micro architectural threats still focus Intel or ARM architectures. An interesting dimension of the future research work can be to see the impact of open-source designs and architectures on the development of micro architectural attacks and mitigations. Moreover, there are now open-sourced projects that aim at replacing the existing proprietary TEE such as Intel SGX. Keystone [215], Open Titan [188, 189], Mi6 [216] or Morpheus [217] are some examples of interesting candidate to provide strong software isolation on RISC-V platforms. Their implications in terms of attacks and defenses would be very interesting for future research works.

When considering mitigations techniques, a growing trend is to focus on proposing alternative cache architectures and designs to reduce the risk of micro architectural covert-channels being carried out. These initiatives are the result of a more secure-by-design approach that has been developed over the past years. Cache-based mitigation techniques have been studied thoroughly in the literature. Cache-based mitigation research is now expanding towards open-source initiatives. As mitigation propositions are increasing in number rapidly, the focus for future mitigation research is shifting. The first new main objective is now to both implement and characterize the proposed mitigations in realistic use cases to demonstrate to what extent they are effective. The mitigation research is not also beginning to study and develop performance-friendly mitigation that propose optimized defenses. An interesting future research trend would be to see to what extent proposed mitigations that exist today are effective in realistic contexts, and how they can be optimized

to minimize their impact on the systems. Another interesting aspect for future research is the evolution of works trying to propose building blocks for a micro architecturally resilient CPU design. Such initiative will tend to grow in the upcoming years and will yield valuable research results towards the development of an original and potentially resilient hardware CPU designs. As shown by an extensive study of the state-of-the-art, developers now have plenty of mitigation solutions to choose from. However, most of them are specific and aim at mitigating a precise variant of an attack, or reduce the leakages from a specific hardware element. Future research works studying the mitigation possibilities at the micro architectural level must propose more generalizable approaches and mitigations that will eventually lead to the development of more building blocks for resilient CPU designs.

8.2.2. Industrial perspectives

This thesis has been conducted in partnership with Thales DIS. As such, the industrial perspectives revolving around the micro architectural research have been studied throughout the work. These perspectives will be detailed in this section.

From an industrial standpoint, micro architectural attacks represent a new type of threat that has to be added to the spectrum. Indeed, it threatens most of the existing CPUs, and depending on the applicative scenario, they can present a major threat. For this reason, there is a real need for industrial-grade protections. Such protections are expected to provide a high level of security while reasonably reducing performances. This industrial trend will affect and orient future research initiatives as already described in section 8.2.1. Micro architectural vulnerabilities are specifically dangerous for application such as cloud computing, confidential computing and embedded devices.

Closed and proprietary architectures might fall short in a near future as open-source codes and hardware are gaining in popularity. More and more industrial actors are now joining the open-source effort. Intel, Amazon, Alibaba are only a few examples. The usage of open-source codes or hardware will be inevitable in a near future in order to stay competitive on the global market. However, this introduces some new challenges, especially during the integration phase. Indeed, it is very complicated and costly to proceed to a complete verification of all the open-sourced codes and updates used inside a product. These additions can potentially be compromised by a malicious developer, or contain bugs deteriorating the overall security of the product. For these reasons, there is a need for developing protections against micro architectural attacks (and more) to alleviate the risks related to integrating open-sourced contents.

Publications, communications and other contributions

International journal publication

- V. Martinoli, E. Tourneur, Y. Teglia, R. Leveugle, CCALK: (When) CVA6 Cache Associativity Leaks the Key, Journal Low Power Electronics and Applications, 2023, https://doi.org/10.3390/jlpea13010001

International conference

- V. Martinoli, Y. Teglia, A. Bouagoun and R. Leveugle, Recovering Information on the CVA6 RISC-V CPU with a Baremetal Micro-Architectural Covert Channel, 2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2022, pp. 1-6, doi: 10.1109/IOLTS56730.2022.9897297.

Patent application

- SECURED SEMICONDUCTOR DEVICE AND METHOD, Patent application n° 22305710.0, European Patent Office

Open archive publication

- V. Martinoli, Y. Teglia, A. Bouagoun, R. Leveugle, CVA6's Data cache: Structure and Behavior. ArXiv, abs/2202.03749, 2022

National Workshop

- V. Martinoli, Y. Teglia, R. Leveugle, SGX face aux menaces micro architecturales : efficacité et limitations, 15ème Colloque National du GDR SOC2, INSA Rennes, June 08 – 10, 2021

- V. Martinoli, Y. Teglia, A. Bouagoun and R. Leveugle, Recovering Information on a RISC-V CPU with a Baremetal Micro-Architectural Covert Channel, PHISIC'2022: Workshop on Practical Hardware Innovations in Security Implementation and Characterization, May 17 – 18, 2022, Ecole Nationale Supérieure des Mines de Saint-Etienne Campus Georges Charpak, France, Gardanne

Industrial Workshop

- Demonstration of the attack during the Thales Research Days, Oct 11-14, 2022, Palaiseau, France.

Other communications

- V. Martinoli, Y. Teglia, R. Leveugle, How SGX security claims meet real life scenarios, 26th IEEE European Test Symposium, PhD Forum, Virtual Interactive Event, May 24 – 28, 2021, Belgium
- Internal Thales company events and presentations

Other contribution

- Open-source codes for the Prime+Probe implementation targeting the FPGA instantiated CVA6 running Linux, available: https://github.com/CCALK-work/CCALK

References

[1] Eric Osterweil, Angelos Stavrou, Lixia Zhang: 20 Years of DDoS: a Call to Action. CoRR abs/1904.02739 (2019)

[2] Eugene H., The Internet Worm Program: An AnalysisPurdue Technical Report CSD-TR-823, Spafford Department of Computer Sciences, Purdue University West Lafayette, IN 47907-2004

[3] Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in ajava based smart card, Computers & Security, 50 :33-46, 2015.

[4] Paul C. Kocher, Joshua Jaffe, Benjamin Jun: Differential Power Analysis. CRYPTO 1999: 388-397

[5] Markus Kuhn, Oliver Kömmerling: Physical security of smartcards, Inf. Sec. Techn. Report 4(2): 28-41 (1999)

[6] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, Claire Whelan: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2): 370-382 (2006)

[7] Daniel Genkin, Adi Shamir, Eran Tromer: RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. IACR Cryptology ePrint Archive 2013: 857 (2013)

[8] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, Aurélien Francillon: Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers. ACM Conference on Computer and Communications Security 2018: 163-177

[9] Paul C. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. CRYPTO 1996: 104-113

[10] Costan, V., & Devadas, S. (2016). Intel SGX Explained. IACR Cryptol. ePrint Arch., 2016.

[11] Jo Van Bulck, Marina Minkin, Offir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. USENIX Security Symposium, 2018.

[12] S. van Schaik et al., "RIDL: Rogue In-Flight Data Load," 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 88-105, doi: 10.1109/SP.2019.00087.

[13] RISC-V Foundation, “RISC-V”, [Online]. Available at: https://riscv.org/, accessed on: 18 Feb. 2022.

[14] Zaruba, F., & Benini, L. (2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 27, 2629-2640.

[15] OpenHW Group, “CVA6 RISC-V CPU”, [Online]. Available at: https://github.com/openhwgroup/cva6, accessed on: 20 Feb. 2021.

[16] Allaf, Zirak & Adda, Mo & Gegov, Alexander. (2017). A Comparison Study on Flush+Reload and Prime+Probe Attacks on AES Using Machine Learning Approaches. Advances in Intelligent Systems and Computing. 650. 10.1007/978-3-319-66939-7_17.

[17] Red Hat Developer, "Reducing Memory Access Times with Caches", [Online]. Available at: https://developers.redhat.com/blog/2016/03/01/reducing-memory-access-times-with-caches#:~:text=Modern%20processors%20typically%20have%20a,from%20memory%20must%20be%20reduced, accessed on: 05 Jan 2022.

[18] 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).

[19] Onur Aciicmez. « Yet Another Microarchitectural Attack : Exploiting I-Cache ». In : Proceedings of the 2007 ACM Workshop on Computer Security Architecture. 2007, p. 11-18.

[20] Ge, Q., Yarom, Y., Cock, D. et al. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J Cryptogr Eng 8, 1–27 (2018). https://doi.org/10.1007/s13389-016-0141-6

[21] Maria Mushtaq. Software-based Detection and Mitigation of Microarchitectural Attacks on Intel's x86 Architecture. Cryptography and Security [cs.CR]. Université de Bretagne Sud, 2019. English.

[22] Percival, C. (2005). CACHE MISSING FOR FUN AND PROFIT.

[23] 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

[24] 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.

[25] 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.

[26] 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.

[27] Gruss, D., Maurice, C., Wagner, K., and Mangard, S., "Flush+Flush: A Fast and Stealthy Cache Attack", https://arxiv.org/pdf/1511.04594.pdf, 2015.

[28] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen, University of California, San Diego, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX", USENIX Security '17, pp. 51-67, 2017

[29] Intel Corp., "Intel® Transactional Synchronization Extensions (Intel® TSX) Memory and Performance Monitoring Update for Intel® Processors", [Online]. Available at:

https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html, accessed on: 21 Jun. 2022.

[30] Gruss, Daniel & Spreitzer, Raphael & Mangard, Stefan. (2015). Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. 897-912.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom, "Spectre Attacks: Exploiting Speculative Execution", 40th IEEE Symposium on Security and Privacy, 2019.

[32] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19). USENIX Association, USA, 249–266.

[33] Kiriansky, Vladimir and Carl A. Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses." ArXiv abs/1807.03757 (2018).

[34] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In Proceedings of the 12th USENIX Conference on Offensive Technologies (WOOT'18). USENIX Association, USA, 3.

[35] Maisuradze, Giorgi and Christian Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (2018)

[36] Horn J., "Speculative execution, variant 4: speculative store bypass", Project Zero, [Online]. Available at: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, accessed on: 6 Dec. 2023.

[37] Schwarz, M., Schwarzl, M., Lipp, M., & Gruss, D. (2018). NetSpectre: Read Arbitrary Memory over Network. ArXiv, abs/1807.10535.

[38] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., & Lai, T. (2018). SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. 2019 IEEE European Symposium on Security and Privacy (EuroS&P), 142-157.

[39] Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; et al. Meltdown: Reading kernel memory from user space. In Proceedings of the USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018.

[40] INTEL Q2 2018 Speculative Execution Side Channel Update, Intel, [Online]. Available at: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html, accessed on: 6 dec. 2023.

[41] Stecklina, J., & Prescher, T. (2018). LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. ArXiv, abs/1806.07480.

[42] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., & Yarom, Y. (2018). Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution.

[43] Schwarzl, M., Schuster, T., Schwarz, M., & Gruss, D. (2020). Speculative Dereferencing of Registers: Reviving Foreshadow. ArXiv, abs/2008.02307.

[44] Schwarz, M.; Lipp, M.; Moghimi, D.; Van Bulck, J.; Stecklina, J.; Prescher, T.; Gruss, D. ZombieLoad: Cross-privilege-boundary data sampling. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019. https://doi.org/10.1145/3319535.3354252

[45] Canella, C.; Genkin, D.; Giner, L.; Gruss, D.; Lipp, M.; Minkin, M.; Moghimi, D.; Piessens, F.; Schwarz, M.; Sunar, B.; et al. Fallout: Reading kernel writes from user space. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019. https://doi.org/10.1145/3319535.3363219.

[46] J. Van Bulck et al., "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020, pp. 54-72, doi: 10.1109/SP40000.2020.00089.

[47] Zhang, Z., Cheng, Y., Zhang, Y., & Nepal, S. (2020). GhostKnight: Breaching Data Integrity via Speculative Execution. ArXiv, abs/2002.00524.

[48] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them. ACM SIGARCH Computer Architecture News, 2014.

[49] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: speculative load hazards boost rowhammer and cache attacks. In Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19). USENIX Association, USA, 621–637.

[50] Schaik, S.V., Minkin, M., Kwong, A., Genkin, D., & Yarom, Y. (2020). CacheOut: Leaking Data on Intel CPUs via Cache Evictions. 2021 IEEE Symposium on Security and Privacy (SP), 339-354.

[51] Schaik, S.V., Kwong, A., & Genkin, D., SGAxe: How SGX Fails in Practice, 2020.

[52] H. Ragab, A. Milburn, K. Razavi, H. Bos and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 1852-1867, doi: 10.1109/SP40001.2021.00020.

[53] Dmitry Evtyushkin, Ryan Riley, Nael CSE and ECE Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). Association for Computing Machinery, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204

[54] Huo, Tianlin & Meng, Xiaoni & Wang, Wenhao & Hao, Chunliang & Zhao, Pei & Zhai, Jian & Li, Mingshu. (2019). Bluethunder: A 2-level Directional Predictor Based Side-Channel

Attack against SGX. IACR Transactions on Cryptographic Hardware and Embedded Systems. 321-347. 10.46586/tches.v2020.i1.321-347.

[55] Dhem, JF., Koeune, F., Leroux, PA., Mestré, P., Quisquater, JJ., Willems, JL. (2000). A Practical Implementation of the Timing Attack. In: Quisquater, JJ., Schneier, B. (eds) Smart Card Research and Applications. CARDIS 1998. Lecture Notes in Computer Science, vol 1820. Springer, Berlin, Heidelberg. https://doi.org/10.1007/10721064_15

[56] Schindler, W. (2000). A Timing Attack against RSA with the Chinese Remainder Theorem. In: Koç, Ç.K., Paar, C. (eds) Cryptographic Hardware and Embedded Systems — CHES 2000. CHES 2000. Lecture Notes in Computer Science, vol 1965. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44499-8_8

[57] David Brumley, Dan Boneh, Remote timing attacks are practical, Computer Networks, Volume 48, Issue 5, 2005, Pages 701-716, ISSN 1389-1286, https://doi.org/10.1016/j.comnet.2005.01.010.

[58] Hevia, A., Kiwi, M. (1998). Strength of two Data Encryption Standard implementations under timing attacks. In: Lucchesi, C.L., Moura, A.V. (eds) LATIN'98: Theoretical Informatics. LATIN 1998. Lecture Notes in Computer Science, vol 1380. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0054321

[59] Walter Tuchman (1997). "A brief history of the data encryption standard". Internet besieged: countering cyberspace scofflaws. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA. pp. 275–280.

[60] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. 2001. Timing analysis of keystrokes and timing attacks on SSH. In Proceedings of the 10th conference on USENIX Security Symposium - Volume 10 (SSYM'01). USENIX Association, USA, Article 25.

[61] Joseph Gravellier. Remote Hardware Attacks on Connected Devices. Cryptography and Security [cs.CR]. Ecole des Mines de Saint-Etienne, 2021. English.

[62] Lipp, M., Aga, M.T., Schwarz, M., Gruss, D., Maurice, C., Raab, L., & Lamster, L. (2018). Nethammer: Inducing Rowhammer Faults through Network Requests. 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 710-719.

[63] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: hammering a needle in the software stack. In Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16). USENIX Association, USA, 1–18.

[64] Gruss, D., Maurice, C., & Mangard, S. (2015). Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. ArXiv, abs/1507.06955.

[65] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 1675–1689. https://doi.org/10.1145/2976749.2978406

[66] A. Kwong, D. Genkin, D. Gruss and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 695-711, doi: 10.1109/SP40000.2020.00020.

[67] D. R. E. Gnad, F. Oboril and M. B. Tahoori, "Voltage drop-based fault attacks on FPGAs using valid bitstreams," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), 2017, pp. 1-7, doi: 10.23919/FPL.2017.8056840.

[68] Krautter, J., Gnad, D.R., & Tahoori, M.B. (2018). FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2018, 44-68.

[69] M. M. Alam, S. Tajik, F. Ganji, M. Tehranipoor and D. Forte, "RAM-Jam: Remote Temperature and Voltage Fault Attack on FPGAs using Memory Collisions," 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2019, pp. 48-55, doi: 10.1109/FDTC.2019.00015.

[70] Tang, A., Sethumadhavan, S., & Stolfo, S. (2017). CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. USENIX Security Symposium.

[71] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0LTpwn: attacking x86 processor integrity from software. In Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20). USENIX Association, USA, Article 82, 1445–1461.

[72] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). Association for Computing Machinery, New York, NY, USA, 195–209. https://doi.org/10.1145/3319535.3354201

[73] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1466-1482, doi: 10.1109/SP40000.2020.00057.

[74] J. Gravellier, J. -M. Dutertre, Y. Teglia and P. L. Moundi, "FaultLine: Software-Based Fault Injection on Memory Transfers," 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2021, pp. 46-55, doi: 10.1109/HOST49136.2021.9702295.

[75] F. Schellenberg, D. R. E. Gnad, A. Moradi and M. B. Tahoori, "An inside job: Remote power analysis attacks on FPGAs," 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018, pp. 1111-1116, doi: 10.23919/DATE.2018.8342177.

[76] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, Philippe Loubet-Moundi, Olivier Francis. Remote Side-Channel Attacks on Heterogeneous SoC. Smart Card Research and Advanced Applications, 18th International Conference, CARDIS 2019, Nov 2019, Pragues, Czech Republic. ⟨hal-02380092⟩

[77] Zhao, Mark & Suh, G.. (2018). FPGA-Based Remote Power Side-Channel Attacks. 229-244. doi: 10.1109/SP.2018.00049.

[78] I. Giechaskiel, K. B. Rasmussen and J. Szefer, "C3APSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage," 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1728-1741, doi: 10.1109/SP40000.2020.00070.

[79] Giechaskiel, Ilias & Rasmussen, Kasper & Eguro, Ken. (2018). Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires. 15-27. doi: 10.1145/3196494.3196518.

[80] Gnad, D.R., Krautter, J., & Tahoori, M.B. (2019). Leaky Noise: New Side-Channel Attack Vectors in Mixed-Signal IoT Devices. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2019, 305-339.

[81] O'Flynn, C., & Dewar, A.D. (2019). On-Device Power Analysis Across Hardware Security Domains. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2019, 126-153.

[82] M. Lipp et al., "PLATYPUS: Software-based Power Side-Channel Attacks on x86," 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 355-371, doi: 10.1109/SP40001.2021.00063.

[83] Gravellier, J., Dutertre, J., Teglia, Y., & Loubet-Moundi, P. (2020). SideLine: How Delay-Lines (May) Leak Secrets from your SoC. ArXiv, abs/2009.07773.

[84] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, and V. Bertocci. Security best practices for developing windows azure applications. In: Microsoft Corp (2010).

[85] D. J. Bernstein. Cache-timing attacks on AES. Tech. rep. Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005.

[86] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: S&P’09 45-60 (2009).

[87] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In: International Conference on Cryptology and Information Security in Latin America. 2012.

[88] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. CrossVM side channels and their use to extract private keys. In: CCS’12. 2012.

[89] G. Agosta, L. Breveglieri, G. Pelosi, and I. Koren. Countermeasures against branch target buffer attacks. In: IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC’07). 2007.

[90] O. Aciiçmez. Advances in Side-Channel Cryptanalysis: MicroArchitectural Attacks. PhD Thesis. Oregon State University, 2007.

[91] M. Joye and M. Tunstall. Securing OpenSSL against MicroArchitectural Attacks. In: SECRYPT. 2007.

[92] A. Hilton, B. Lee, and T. Lehman. PoisonIvy: Safe speculation for secure memory. In: Proceedings of the 49th International Symposium on Microarchitecture (MICRO’16). 2016.

[93] Gruss, D., (2018). Software-based microarchitectural attacks. it - Information Technology: Vol. 60, No. 5-6. Berlin: De Gruyter. (S. 335-341). DOI: 10.1515/itit-2018-0034

[94] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21). Association for Computing Machinery, New York, NY, USA, 715–733. https://doi.org/10.1145/3460120.3484583.

[95] Cauligi, S., Disselkoen, C., Gleissenthall, K.V., Stefan, D., Rezk, T., & Barthe, G. (2019). Constant-time foundations for the new spectre era. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.

[96] Intel Corporation, Retpoline: A Branch Target Injection Mitigation, White Paper Revision 003, June, 2018, [Online]. Available at: https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf, accessed on: 06 Dec 2023.

[97] Bualucea, R., & Irofti, P. (2022). Software Mitigation of RISC-V Spectre Attacks. ArXiv, abs/2206.04507.

[98] "BOOM core," Berkley univesity, [Online]. Available: https://boom-core.org/, accessed : 12 Nov 2021.

[99] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS'16. 2016.

[100] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA'16. 2016.

[101] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini and G. Heiser, "Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021, pp. 627-632, doi: 10.23919/DATE51398.2021.9474214.

[102] N. Wistoff, M. Schneider, F. K. Gürkaynak, G. Heiser and L. Benini, "Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning," in IEEE Transactions on Computers, 2022, doi: 10.1109/TC.2022.3212636.

[103] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH Computer Architecture News 35.2 (June 2007), p. 494.

[104] J. Kong, O. Acıi¸cmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cachebased side channel attacks. In: Proceedings of the 2nd ACM Computer Security Architectures Workshop (2008).

[105] F. Liu and R. B. Lee. Random Fill Cache Architecture. In: IEEE/ACM International Symposium on Microarchitecture (MICRO'14). 2014, pp. 203–215.

[106] Moinuddin K. Qureshi. 2018. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51). IEEE Press, 775–787. https://doi.org/10.1109/MICRO.2018.00068

[107] Mukhtar, Muhammad & Bhatti, Muhammad & Gogniat, Guy. (2020). IE-Cache: Counteracting Eviction-Based Cache Side-Channel Attacks Through Indirect Eviction. 10.1007/978-3-030-58201-2_3.

[108] Javed, Saqib & Mukhtar, Muhammad & Bhatti, Muhammad & Gogniat, Guy. (2022). Novel Design for IE-Cache to Mitigate Conflict-based Cache-side Channel Attacks with Reduced Energy Consumption. 675-680. 10.5220/0011326700003283.

[109] A. Purnal, L. Giner, D. Gruss and I. Verbauwhede, "Systematic Analysis of Randomization-based Protected Cache Architectures," 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 987-1002, doi: 10.1109/SP40001.2021.00011.

[110] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. SCATTERCACHE: thwarting cache attacks via cache set randomization. In Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19). USENIX Association, USA, 675–692.

[111] Thoma, J.P., Niesler, C., Funke, D.A., Leander, G., Mayr, P., Pohl, N., Davi, L., & Güneysu, T. (2021). ClepsydraCache - Preventing Cache Attacks with Time-Based Evictions. ArXiv, abs/2104.11469.

[112] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. Cryptology ePrint Archive, Report 2005/280. 2005. url: http://eprint.iacr.org/2005/280.

[113] Z. Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In: IEEE/ACM International Symposium on Microarchitecture (MICRO'08). 2008, pp. 83–93.

[114] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In: 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10). 2010 (p. 51).

[115] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. In: ACM Transactions on Architecture and Code Optimization (TACO) 8.4 (2011).

[116] C. Miao, K. Bu, M. Li, S. Mao and J. Jia, "SwiftDir: Secure Cache Coherence without Overprotection," 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2022, pp. 662-677, doi: 10.1109/MICRO56248.2022.00052.

[117] Zhou, Z., Reiter, M.K., & Zhang, Y. (2016). A Software Approach to Defeating Side Channels in Last-Level Caches. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.

[118] Y. Tan, J. Wei, and W. Guo. The Micro-architectural Support Countermeasures against the Branch Prediction Analysis Attack. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. 2014.

[119] Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier, Jean-Louis Lanet. Under the dome: preventing hardware timing information leakage. CARDIS 2021 - 20th Smart Card Research and Advanced Application Conference, Nov 2021, Lübeck, Germany. pp.1-20.

[120] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In: USENIX Security Symposium. 2016.

[121] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In: S&P'15. 2015 (p. 43).

[122] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In: Proceedings of the 2nd ACM Cloud Computing Security Workshop (CCSW'10). 2010, pp. 103–108.

[123] W. Wu, E. Zhai, D. Jackowitz, D. I. Wolinsky, L. Gu, and B. Ford. Warding off timing attacks in Deterland. In: arXiv:1504.07070 (2015).

[124] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In: DNS'13. 2013.

[125] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14. 2014.

[126] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'16). 2016.

[127] S. van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida. Reverse Engineering Hardware Page Table Caches Using Side-Channel Attacks on the MMU. 2017. url: http://www. cs.vu.nl/~herbertb/download/papers/revanc_ir-cs77.pdf.

[128] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA'17. (to appear). 2017.

[129] J. C. Wray. An analysis of covert timing channels. In: Journal of Computer Security 1.3-4 (1992), pp. 219–232.

[130] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016.

[131] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In: 22nd IEEE Computer Security Foundations Symposium. 2009.

[132] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In: CCS'10. 2010.

[133] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In: CCS'11. 2011.

[134] D. Cock, Q. Ge, T. Murray, and G. Heiser. The last mile: an empirical study of timing channels on seL4. In: CCS'14. 2014.

[135] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cachebased side-channel in multi-tenant cloud using dynamic page coloring. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W). 2011.

[136] T. Kim, M. Peinado, and G. Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security Symposium. 2012 (p. 46).

[137] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: minimal hardware extensions for strong software isolation. In Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16). USENIX Association, USA, 857–874.

[138] Y. Zhang and M. Reiter. D¨uppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: CCS'13. 2013.

[139] M. M. Godfrey and M. Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. In: IEEE Transactions on Cloud Computing (2014).

[140] "Resources and Response to Side Channel L1 Terminal Fault," Intel, [Online]. Available at: https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html, accessed on: 25 Jul. 2019.

[141] B. A. Braun, S. Jana, and D. Boneh. Robust and Efficient Elimination of Cache and Timing Side Channels. In: arXiv:1506.00189 (2015).

[142] Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2020. SmokeBomb: effective mitigation against cache side-channel attacks on the ARM architecture. In Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys '20). Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3386901.3388888.

[143] Ayaz Akram, Maria Mushtaq, Muhammad Khurram Bhatti, Vianney Lapotre, Guy Gogniat. Meet the Sherlock Holmes' of Side Channel Leakage: A Survey of Cache SCA Detection Techniques. IEEE Access, 2020, 8, pp.70836-70860. ⟨10.1109/ACCESS.2020.2980522⟩. ⟨hal-02508889⟩.

[144] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: a tool for the static analysis of cache side channels. In: ACM Transactions on Information and System Security (2015).

[145] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? Cryptology ePrint Archive, Report 2016/1123. 2016. url: http://eprint.iacr.org/2016/ 1123.

[146] A. Zankl, J. Heyszl, and G. Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In: International Conference on Smart Card Research and Advanced Applications. Springer. 2016.

[147] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In: ACM SIGARCH Computer Architecture News 40.3 (2012), pp. 106–117.

[148] G. Doychev and B. Köpf. Rigorous Analysis of Software Countermeasures against Cache Attacks. In: arXiv:1603.021 87 (2016).

[149] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the Information Leak in Cache Attacks through Symbolic Execution. In: arXiv:1611.04426 (2016).

[150] Yuan, Y., Liu, Z., & Wang, S. (2022). CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software. ArXiv, abs/2209.14952.

[151] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In: ACM SIGARCH Computer Architecture News 41.3 (2013), pp. 559–570.

[152] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In: RAID'14. 2014.

[153] C. Cardenas and R. V. Boppana. Detection and mitigation of performance attacks in multi-tenant cloud computing. In: 1st International IBM Cloud Academy Conference, Research Triangle Park, NC, US. 2012.

[154] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Cryptology ePrint Archive, Report 2015/1034. 2015.

[155] T. Zhang, Y. Zhang, and R. B. Lee. CloudRadar: A RealTime Side-Channel Attack Detection System in Clouds. In: International Symposium on Research in Attacks, Intrusions, and Defenses. 2016.

[156] M. Mushtaq et al., "Machine Learning For Security: The Case of Side-Channel Attack Detection at Run-time," 2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Bordeaux, France, 2018, pp. 485-488, doi: 10.1109/ICECS.2018.8617994.

[157] Z. Tong, Z. Zhu, Z. Wang, L. Wang, Y. Zhang and Y. Liu, "Cache side-channel attacks detection based on machine learning," 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Guangzhou, China, 2020, pp. 919-926, doi: 10.1109/TrustCom50675.2020.00123.

[158] Marco Chiappetta, Erkay Savas, Cemal Yilmaz, Real time detection of cache-based side-channel attacks using hardware performance counters, Applied Soft Computing, Volume 49, 2016, Pages 1162-1174, ISSN 1568-4946, https://doi.org/10.1016/j.asoc.2016.09.014.

[159] A. Fogh. Detecting stealth mode cache attacks: Flush+Flush. 2015. url: http://dreamsofastone.blogspot.co.at/2015/11/detecting-stealth-mode-cache-attacks.html.

[160] M. Payer. HexPADS: a platform to detect "stealth" attacks. In: ESSoS'16. 2016.

[161] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with D´ej`a Vu. In: Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17). 2017.

[162] Gueron, S., A Memory Encryption Engine Suitable for General Purpose Processors. IACR Cryptol. ePrint Arch., 2016, 204.

[163] J. V. Bulck, "jovanbulck/sgx-step," GitHub, [Online]. Available: https://github.com/jovanbulck/sgx-step/tree/master/app/foreshadow, accessed on: 22/01/2022.

[164] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21, 2 (Feb. 1978), 120–126. https://doi.org/10.1145/359340.359342

[165] Coron, JS., Naccache, D., Stern, J.P. (1999). On the Security of RSA Padding. In: Wiener, M. (eds) Advances in Cryptology — CRYPTO' 99. CRYPTO 1999. Lecture Notes in Computer Science, vol 1666. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48405-1_1.

[166] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, Vincent Verneuil. Square Always Exponentiation. 12th International Conference on Cryptology in India - INDOCRYPT 2011, Dec 2011, Chennai, India. pp.40-57, ff10.1007/978-3-642-25578-6_5ff. ffinria-00633545f.

[167] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb.–Mar. 2017.

[168] Dworkin, M. , Barker, E. , Nechvatal, J. , Foti, J. , Bassham, L. , Roback, E. and Dray, J., "Advanced Encryption Standard (AES)", Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, 2001, Gaithersburg, MD, [Online]. Available at: https://doi.org/10.6028/NIST.FIPS.197, accessed on: 4 May 2022.

[169] W. Stallings, "Reduced instruction set computer architecture," in Proceedings of the IEEE, vol. 76, no. 1, pp. 38-55, Jan. 1988, doi: 10.1109/5.3287.

[170] RISC-V Foundation, "RISC-V Sees Significant Growth and Technical Progress in 2022 with Billions of RISC-V Cores in Market", [Online]. Available at: https://riscv.org/announcements/2022/12/risc-v-sees-significant-growth-and-technical-progress-in-2022-with-billions-of-risc-v-cores-in-market/, accessed on: 22 Dec. 2022.

[171] PULP Platform, "PULP Platform Open hardware, the way it should be!", [Online]. Available at: https://pulp-platform.org/, accessed on: 12 Jan. 2022.

[172] ETH Zürich, "ETH Zürich", [Online]. Available at: https://ethz.ch/en.html, accessed on: 21 Apr. 2022.

[173] University of Bologna, "Alma Mater Studiorum Università Di Bologna", [Online]. Available at: https://www.unibo.it/en, accessed on: 09 Feb. 2022.

[174] OpenHW Group, "OpenHW Group CORE-V CV32E40P RISC-V IP", [Online]. Available at: https://github.com/openhwgroup/cv32e40p, accessed on: 21 Mar. 2022.

[175] lowRISC, "Ibex RISC-V Core", [Online]. Available at: https://github.com/lowRISC/ibex, accessed on: 24 Apr. 2022.

[176] PULP Platform, "PULPissimo", [Online]. Available at: https://github.com/pulp-platform/pulpissimo, accessed on: 03 Jan. 2023.

[177] PULP Platform, "PULPino", [Online]. Available at: https://github.com/pulp-platform/pulpino, accessed on: 03 Jan. 2023.

[178] A. Pullini, D. Rossi, I. Loi, G. Tagliavini and L. Benini, "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing," in IEEE Journal of Solid-State Circuits, vol. 54, no. 7, pp. 1970-1981, July 2019, doi: 10.1109/JSSC.2019.2912307.

[179] PULP Platform, “HERO: Open Heterogeneous Research Platform”, [Online]. Available at: https://pulp-platform.org/hero.html, accessed on: 03 Jan. 2023.

[180] OpenHW Group, “CORE-V Family of Open-Source RISC-V Cores”, [Online]. Available at: https://github.com/openhwgroup/core-v-cores, accessed on: 03 Jan. 2023.

[181] OpenHW Group, “OpenHW Group Proven processor IP”, [Online]. Available at: https://www.openhwgroup.org/, accessed on: 11 May 2022.

[182] CHIPS Alliance, “CHIPS (Common Hardware for Interfaces, Processors and Systems) Alliance harnesses the energy of open source collaboration to accelerate hardware development.”, [Online]. Available at: https://chipsalliance.org/, accessed on: 04 Jan. 2023.

[183] Western Digital, “RISC-V SweRV CoreTM Available to Open Source Community”, [Online]. Available at: https://blog.westerndigital.com/risc-v-swerv-core-open-source/, accessed on: 23 Mar. 2022.

[184] Western Digital, “Western Digital”, [Online]. Available at: https://www.westerndigital.com/, accessed on: 22 Jan. 2023.

[185] Western Digital, “SweRV RISC-V CoreTM 1.1 from Western Digital”, [Online]. Available at: https://github.com/westerndigitalcorporation/swerv_eh1, accessed on: 05 Apr. 2022.

[186] MEEP, “MEEP| MareNostrum Experimental Exascale Platform”, [Online]. Available at: https://meep-project.eu/about, accessed on: 12 Dec. 2022.

[187] lowRISC, “lowRISC”, [Online]. Available at: https://lowrisc.org/, accessed on: 15 Dec. 2022.

[188] Open Titan, “Open Titan”, [Online]. Available at: https://opentitan.org/, accessed on: 15 Dec. 2022.

[189] Open Titan, “OpenTitan”, [Online]. Available at: https://github.com/lowRISC/opentitan, accessed on: 15 Dec. 2022.

[190] Alibaba, “Alibaba.com”, [Online]. Available at: https://www.alibaba.com/, accessed on: 16 Dec. 2022.

[191] RISC-V Foundation, “Alibaba open sources four RISC-V cores: XuanTie E902, E906, C906 and C910 | Jean-Luc Aufranc, CNX Software”, [Online]. Available at: https://riscv.org/news/2021/10/alibaba-open-sources-four-risc-v-cores-xuantie-e902-e906-c906-and-c910-jean-luc-aufranc-cnx-software/, accessed on: 16 Dec. 2022.

[192] Intel, “Nios® V Processors”, [Online]. Available at: https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor/v.html, accessed: 05 Dec. 2022.

[193] SiFive, “SiFive”, [Online]. Available at: https://www.sifive.com/, accessed on: 07 Apr. 2022.

[194] Esperanto Technologies, “Esperanto.ai”, [Online]. Available at: https://www.esperanto.ai/, accessed on: 07 Apr. 2022.

[195] SiFive, “HiFive Boards.”, [Online]. Available at: https://www.sifive.com/boards, accessed on: 08 Apr. 2022.

[196] Esperanto Technologies, “Esperanto is leading the RISC-V revolution for AI

and enabling a new level of AI performance.”, [Online]. Available at: https://www.esperanto.ai/technology/, accessed on: 08 Apr. 2022.

[197] OpenHW Group, “CVA6”, [Online]. Available at: https://docs.openhwgroup.org/_/downloads/cva6-user-manual/en/latest/pdf/, accessed on: 03 Jan. 2023.

[198] JSOF, “19 Zero-Day Vulnerabilities Amplified by the Supply Chain”, [Online]. Available at: https://www.jsof-tech.com/disclosures/ripple20/, accessed: 19 Apr. 2022.

[199] Martinoli, V., Teglia, Y., Bouagoun, A., and Leveugle, R. (2022). CVA6's Data cache: Structure and Behavior. *ArXiv, abs/2202.03749*.

[200] Mastik Toolkit, “Mastik: A Micro-Architectural Side-Channel Toolkit”, [Online]. Available at: https://github.com/0xADE1A1DE/Mastik, accessed on: 20 Apr. 2022.

[201] Veripool, “Verilator”, [Online]. Available at: https://www.veripool.org/verilator/, accessed on: 27 Mar. 2021.

[202] RISC-V Foundation, “Spike RISC-V ISA Simulator”, [Online]. Available at: https://github.com/riscv-software-src/riscv-isa-sim, accessed on: 27 Mar. 2021.

[203] D. F. Aranha et al., “LadderLeak: Breaking ECDSA With Less Then One Bit Of Nonce Leakage,” CCS 20: Conf. on Computer and Communication Security, pp. 225-242, 2020.

[204] Digilent, “Genesys 2 Kintex-7 FPGA Development Board”, [Online]. Available at: https://digilent.com/shop/genesys-2-kintex-7-fpga-development-board/#:~:text=The%20Genesys%202%20is%20a,a%20wide%20array%20of%20projects, accessed on: 28 Mar. 2022.

[205] AMD Xilinx, “VIVADO ML Editions”, [Online]. Available at: https://www.xilinx.com/products/design-tools/vivado.html, accessed on: 05 Feb. 2022.

[206] OpenHW Group, “CVA6 SDK”, [Online]. Available at: https://github.com/openhwgroup/cva6-sdk, accessed on: 18 Feb. 2022.

[207] Kokke, “Tiny AES in C”, [Online]. Available at: https://github.com/kokke/tiny-AES-c, accessed on: 07 Mar. 2022.

[208] Cantora, “avr-crypto-lib/rsa/”, [Online]. Available at : https://github.com/cantora/avr-crypto-lib/tree/master/rsa, accessed on : 08 Mar. 2022.

[209] Tromer, E., Osvik, D.A. & Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. J Cryptol 23, 37–71 (2010). https://doi.org/10.1007/s00145-009-9049-y

[210] AMD, “AMD Athlon™ 64 X2 Dual-Core Processor Product Data Sheet”, [Online]. Available at: https://www.amd.com/system/files/TechDocs/33425.pdf, accessed on: 19 Apr. 2022.

[211] Intel Corp., “Intel® Pentium® 4 Processor 2.80 GHz, 512K Cache, 533 MHz FSB”, [Online]. Available at: https://www.intel.com/content/www/us/en/products/sku/27447/intel-pentium-4-processor-2-80-ghz-512k-cache-533-mhz-fsb/specifications.html, accessed on: 22 Apr. 2022.

[212] Heninger, N., Shacham, H. (2009). Reconstructing RSA Private Keys from Random Key Bits. In: Halevi, S. (eds) Advances in Cryptology - CRYPTO 2009. CRYPTO 2009. Lecture Notes in Computer Science, vol 5677. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03356-8_1.

[213] European Patent Office, “epo.org”, [Online]. Available at: https://www.epo.org/, accessed on: 28 May 2022.

[214] V. Martinoli et E. Tourneur, “CCALK-PP”, [Online]. Available at: https://github.com/CCALK-work/CCALK, accessed on: 04 Jul. 2022.

[215] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 38, 1–16. https://doi.org/10.1145/3342195.3387532.

[216] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor, 2018.

[217] NVIDIA DEVELOPER, “NVIDIA Morpheus – Cybersecurity and AI”, [Online]. Available at: https://developer.nvidia.com/morpheus-cybersecurity, accessed on: 02 Jan. 2023.

Appendices

Appendix 1. Main characteristics of the existing micro architectural attacks and comparison with related attacks

Characteristics / Attack type	Requires Physical access to the target	Exploited element/phenomenon	Required privilege level	Location of the secret recovered
Spectre-type	**No**	Speculative execution	User/Stronger if root	Value residing in a normally inaccessible memory page
Meltdown-type	**No**	Out-of-order execution	User/Stronger if root	Value residing in a normally inaccessible memory page
MDS-type side-channel analysis	**No**	Shared inner micro architectural buffers (Line Fill Buffer, Load ports, Store buffers…)	Root privileges for most variants	Incorrectly forwarded values from a totally different context/security domain
SW-based HW side-channel analysis	**No**	Information-dependent physical signals	User/Root depending on the variant	Value residing in an inaccessible memory area
MDS-type fault attacks (LVI)	**No**	Transient forwarding of incorrect values inside micro architectural buffers	User/Stronger if root	Arbitrary data or code when using a redirection of the execution flow
SW-based HW fault attacks	**No**	Perturbation of the physical signals (voltage glitch for example)	User/Root depending on the variant	Value residing in an inaccessible memory area
Traditional Side-Channel Attacks	**Yes**	Information-dependent physical signals	X	Value residing in an inaccessible memory area

Appendix 2. Overview of the existing micro architectural mitigations and their main characteristics

Characteristics / Defense type	Layer where the protection is located	Effect of the mitigation	Type of attacks countered	Cost
Constant-time cryptographic implementations	User-space	Protects cryptographic implementations	Traditional timing side-channels	Moderate
Eliminating secret-dependent computations	User-space	Protects cryptographic implementation	Some traditional cache side-channels	High
Preloading all data into the cache	User-space	Protects cryptographic implementation	Some single-core cache side-channels	Low
Constantine	User-space	Compiler-based - Linearizes secret-dependent data and control flow	Some cache side-channels	Low
Pitchfork	User-space	Detects Spectre-type attacks	Spectre-type variants	Low
Retpoline	User-space	Modifies the behavior of indirect jumps and indirect calls	Some Spectre-type variants	Low
Removing timing differences inside instruction sets	Instruction set	Develop constant-time instructions for optimization mechanisms	Several variants of Spectre, Meltdown and cache covert-channels	Low
Fence.t instruction	Instruction set	Possibility for the programmer to flush all the leaky micro architectural resources	Several single-core cache attacks	Low

Characteristics / Defense type	Layer where the protection is located	Effect of the mitigation	Type of attacks countered	Cost
PLcache	Cache design	Partition-locked cache design making cache lines impossible to evict	Eviction-based covert-channels (Flush + Reload, Prime + Probe…)	Moderate
RPcache (improved version)	Cache design	Randomized mapping between cache sets and addresses at runtime	Many cache covert-channel variants	Moderate
CEASER	Cache design	Mapping translating physical line-addresses into encrypted line-addresses Makes cache mapping harder to determine	Eviction-based attacks	Moderate
IE-Cache	Cache design	Introducing a random memory-to-cache mapping	Any variants requiring the determination of an eviction set	Moderate
ClepsydraCache	Cache design	Linking every cache entry to a Time-To-Live (TTL) + index randomization	Cache side-channels that defeat randomized cache designs (Prime + Prune + Probe)	Moderate
Disable shared resources and optimization mechanisms	Cache design	Removing caches, disabling system-multithreading	Most variants of all types of micro architectural attacks	Very high
Dynamic cache mappings	Cache design	Prohibiting different processes from sharing a cache set	Several cache-covert-channel variants	Moderate
Sanchez et al. cache design	Cache design	Decoupling cache ways and cache associativity	Eviction-based covert-channels	Moderate
Non-monopolizable cache	Cache design	Preventing any process from allocating enough cache lines to observe collision with other processes	Covert-channels based on contention	Moderate
SwiftDir	Cache design	Introducing a constant-time latency when serving requests from the Last Level Cache (LLC) Write-protected data are forced to be served from the LLC	Timing cache covert-channel on the LLC	Moderate

Characteristics / Defense type	Layer where the protection is located	Effect of the mitigation	Type of attacks countered	Cost
CacheBar	Kernel	Memory-management subsystem that disables line-sharing in the LLC	Flush + Reload, cross-tenant Prime + Probe on the LLC	Moderate
Constant-time libraries	System-level	Introducing libraries performing constant-time computations	Timing cache side-channel attacks	Low
Reduce the effectiveness of hardware timers	System-level	Simulating hardware timers to render them less accurate	Some timing cache side-channel variants (effectiveness is limited)	Low
Worst-timing algorithms	System-level	Making algorithms always have the worst timing behavior	Evict + Time covert-channels	Moderate/High
Disable cache-line sharing	System-level	Disabling the sharing of cache lines among different processes	Many cache covert-channel and micro architectural attacks variants	Very high
Cache coloring	System-level	Making the OS use supplementary bits in the physical address to attribute a color to each memory page Reducing in-application conflicts on the cache	Cache covert-channels relying on set sharing (Prime + Probe)	Moderate

Characteristics / Defense type	Layer where the protection is located	Effect of the mitigation	Type of attacks countered	Cost
SmokeBomb	System-level Compilation	Adding a collection of applications to find and patch vulnerable code + Creating a private L1 cache for each process + Cache flushes	Most variants of eviction and timing based cache covert-channels	Moderate
CacheQL	System-level Detection mechanism	Quantifying and localizing information leaks on binary code using cache side-channel traces	Potentially most cache-based covert-channels, depends on the developer's ability	Low
Monitoring performance counters	System-level Detection mechanism	Detecting ongoing attacks at runtime using performance counters	Some micro architectural and covert-channel variants (proved insufficient)	Moderate/High
Machine learning for automated detection	System-level Detection mechanism	Detecting ongoing attacks at runtime using machine learning	Most already known micro architectural attacks and covert-channels	Moderate/High
HexPADS	System-level Detection mechanism	Detecting ongoing cache attacks and Rowhammer exploits at runtime by monitoring cache events, misses and page faults	Most already known cache covert-channels and Rowhammer exploits	Moderate/High

Résumé

Afin d’augmenter la puissance de calcul des processeurs au-delà des progrès des technologies, des mécanismes architecturaux d’optimisation ont été ajoutés depuis de nombreuses années. Cependant, ces mécanismes reposent le plus souvent sur des ressources matérielles partagées qui peuvent donner lieu à des fuites d'information.

L’exploitation de ces fuites permet à un attaquant d'obtenir des informations sensibles en provenance d’autres applications. Ces attaques dites micro architecturales emploient des moyens logiciels pour provoquer l’utilisation de mécanismes matériels de très bas niveau. Les informations recherchées sont alors collectées par le biais d’attaques par canaux cachés.

Cette thèse vise à répliquer et à se défendre contre certaines de ces attaques sur le cœur RISC-V CVA6. Une analyse approfondie de la micro architecture de ce processeur a mené à la découverte de chemins d’attaque potentiels. Les deux attaques réalisées exploitent ces chemins afin de retrouver des informations sensibles en provenance d’un algorithme cryptographique. La première a été analysée grâce à des outils de simulation, permettant de comprendre dans le détail le fonctionnement de l’attaque et ses implications sur la micro architecture. Une première version vise l'exploitation directe d'une implantation "bare-metal" et une seconde version ajoute un "pseudo-ordonnancement" permettant d'analyser l'effet de bruits induits par d'autres processus pendant l'attaque. La seconde réalisation vise une cible plus réaliste où le cœur est instancié sur une carte FPGA avec un système d’exploitation Linux afin de prouver le réalisme de la menace micro architecturale et d’évaluer les possibilités de défense au niveau logiciel. Ces travaux montrent aussi que répliquer sur un nouveau processeur une attaque dont le principe est connu reste complexe et demande un investissement notable en durée d'analyse, même si ce processeur est libre de droits et disponible au niveau du code source.

L’implémentation de ces attaques a mené à l’élaboration de protections telles la remise à zéro des ressources ciblées par les attaques. Finalement, une modification micro architecturale est proposée pour contourner les ressources présentant des fuites lors de la manipulation de données sensibles. Pour ce faire, des instructions supplémentaires sont ajoutées et employées par le programmeur lors de l’utilisation de données critiques dans son code.

Mots clés : Processeurs sécurisés, Micro architecture, RISC-V, Attaques par canal caché, Attaques sur les mémoires caches, CVA6, Open-source hardware.

Abstract

To achieve ever-higher performances, architectural optimization mechanisms have been embedded in high-end CPUs for years. However, these mechanisms rely on shared hardware resources that might yield information leakage.

An attacker can leverage these leakages to gather sensitive information belonging to other unrelated applications. These so-called micro architectural attacks rely on software components to trigger very low-level hardware mechanisms. The leaked information can then be recovered through covert-channel attacks.

This thesis aims at replicating and mitigating some of these attacks on the CVA6 RISC-V core. An in-depth analysis of the CVA6 micro architecture led to the discovery of potential attack paths. The two implemented attacks leverage these attack paths to extract sensitive information from a cryptographic algorithm. The first one has been analyzed with simulation tools, enabling a deeper understanding of the attack's behavior and its implications on the micro architecture. A first version consisted in a direct baremetal implementation and a second one added a pseudo scheduling enabling an analysis of the impact of perturbations induced by other applications on the attack. The second attack that was implemented targeted a more realistic victim. The core has been instantiated on an FPGA board running a Linux OS. The goal was to show the realism of the micro architectural threat and evaluate the possibilities of mitigation at the software level. This work also demonstrates the difficulty and time investment that are required to replicate a known attack on a new processor, even when it is open-source and its description is fully available, since such a replication still requires a lengthy analysis period.

The replication of these attacks led to the elaboration of protections such as flushing the resources targeted by the attack. Ultimately, a micro architectural modification has been proposed. It enables the processor to bypass leaky micro architectural resources when handling out security-critical information. To this end, two instructions are added for the programmer to use when handling sensitive data.

Key words: Secure processors, Micro architecture, RISC-V, Covert-channels, Cache attacks, CVA6, Open-source hardware.