Modeling of micro-architecture for security with gem5

Quentin Forcioli

To cite this version:

Quentin Forcioli. Modeling of micro-architecture for security with gem5. Embedded Systems. Institut Polytechnique de Paris, 2024. English. NNT : 2024IPPAT033 . tel-04913269

HAL Id: tel-04913269

https://theses.hal.science/tel-04913269v1

Submitted on 27 Jan 2025

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

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

NNT : 2024IPPAT033

Thèse de doctorat

Modeling of micro-architecture for security with gem5

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Telecom Paris

École doctorale n°626 École Doctorale de l'Institut Polytechnique de Paris (IP Paris)
Spécialité de doctorat: Réseaux, Informations et Communications

Thèse présentée et soutenue à Palaiseau, le 21/11/2024, par

Quentin Forcioli

Composition du Jury :

Lilian Bossuet Professeur des université, Laboratoire Hubert Curien (UMR 5516)	Président/Examinateur
Gilles Sassatelli Directeur de recherche, Laboratoire dInformatique, de Robotique et de Microélectronique de Montpellier (UMR 5506)	Rapporteur
Guillaume Hiet Professeur, CentraleSupélec, IRISA (UMR 6074)	Rapporteur
Damien Couroussé Ingénieur de recherche, CEA-LIST	Examinateur
Jean-Luc Danger Professeur des université, Telecom Paris (UMR 5141)	Directeur de thèse
Sumanta Chaudhuri Maitre de conference, Telecom Paris (UMR 5141)	Directeur de thèse

Abstract

Embedded systems are the target of a wide variety of attacks, software, and hardware levels. Among these, micro-architectural attacks stand out as particularly challenging to investigate. Taking advantage of specific System-on-Chips behaviors, often not visible from an Instruction Set Architecture perspective, these attacks allow an attacker to gain control of protected resources, bypassing the mechanisms that are set up by the OS to isolate different processes. These attacks can target all the parts of a SoCs: CPU (flow control units, operators,...), caches, memory, accelerators (FPGA, GPU,), interfaces, etc. Understanding, replicating, and instrumenting these attacks and their associated scenario, greatly benefit from simulation. The **Archisec** project adopts this *simulation-for-security* methodology[For+21] leveraging the *gem5* simulator as a foundation to develop a virtual platform capable of reproducing typical micro-architectural attacks, thereby advancing research in this domain. A critical aspect of SoC security lies in its Trusted Execution Environment (TEE). In the TEE, specific tasks run isolated within secure enclaves, safeguarding them from attacks even if the OS is compromised. The TEE plays a vital role in safeguarding applications such as device updates, banking operations, etc. However, attackers are actively seeking ways to circumvent these protections, as documented instances of micro-architectural attacks against TEEs reveal.

For this reason, the **Archisec** platform needs to support TEEs while simulating micro-architectures. As the project focuses on ARM, we also decided to use ARMv8-A and its associated security framework TrustZone. With TrustZone, a TEE can be deployed on ARMv8-A and ARMv7-A platforms. We chose OP-TEE an open-source, TrustZone-compatible TEE.

As a contribution to the platform, I created a cache-timing attack library for ARMv8-A to compare cache-timing results between *gem5* and a raspberry PI. I also improved the ARM ISA implementation in *gem5* and created a compatible TEE-enable bootrom with OP-TEE. I created attack scenarios against OP-TEE that leverage the *gem5* simulator environments, to study them. After improving the *GDB* remote debugger in *gem5*, I developed an interface that uses *GDB* scripts and a programmable stub to study and analyze attack scenarios, extracting cache states and re-configuring the simulator on-the-fly. Utilizing this interface alongside existing *gem5* tools, we proposed a first contribution that uses the platform to study Third-party IP and cryptographic library [FDC23].

My second contribution is *TEE-Time* [FDC24b] a tool that analyzes cache-timing side-channel using my *gem5-GDB* interface. This tool uses *Key Execution Points (KEP)* that encompass all algorithmic knowledge, without any assumptions about the specific CPU architecture. Presumably, if an attacker could detect these *KEPs*, it would be able to reconstruct the totality or part of the secret. To evaluate how practical it would be for an attacker to spot these *KEPs*, and thus, how weak is a Victim application, *TEE-Time* produces reports that describe and assess an ideal attack scenario to retrieve *KEPs*. Then, an attack monitoring script, using the same *GDB-interface*, verifies the attack scenario described in the report. To achieve this, it produces labels for the attack traces to help identify *KEP* signals in cache timings. This two-step process was validated against toy attacks and then against standard cryptographic applications using RSA crypto-services in OP-TEE.

To validate my methodology against an actual system, I developed a virtual platform to simulate the *RK3399* SoC from Rockchip, present on the RockPi4 board. The *RK3399* features TrustZone memory protection, secure fuses, secure boot, and a lockable JTAG debugger. Given these functionalities, our simulation platform became indispensable for investigating cache-timing attacks on the *RK3399*. To build the *RK3399* virtual platform, I developed a new fast-prototyping tool for *gem5* called *PyDevices* that uses *gem5*'s Python interface to implement hardware devices. With *PyDevices* and *Ghidra*, I retro-engineered the *RK3399* `BootROM` and configured the simulation platform to accurately imitate the *RK3399* to the extent of booting the same SD card image across both physical and simulated platforms. Using my aforementioned attack library, I found out that the *RK3399* used an ARM-specific cache protocol called **AutoLock**. Incorporating this mechanism into *gem5* and refining the *TEE-Time* scripts, I reproduced an ideal attack scenario against OP-TEE RSA crypto services. This is my third contribution [FDC24a], which demonstrates a real-world attack built upon my simulation methodology. In this attack scenario, OP-TEE uses *mbedTLS* bignum exponentiation that implements the sliding-window exponentiation. Based on Bernstein et al.[Ber+17], I configured the *VicitimScan* tool to tailor a cache-timing attack. The attack was then verified in the simulated environment using the monitoring script. Running the same attack without any modification on a RockPi4, successfully leaked on average $\sim 1/3$ of the RSA key bits.

This final contribution serves as a pivotal step in bridging the gap between simulation and real hardware, thereby fulfilling a key goal for the Archisec project.

Résumé en français

Les systèmes embarqués sont la cible d'une grande variété d'attaques, tant au niveau logiciel que matériel. Parmi celles-ci, les attaques micro-architecturales se révèlent particulièrement difficiles à étudier. Tirant parti des comportements spécifiques des System-on-Chips (SoC: systèmes sur puce), souvent invisibles du point de vue du jeu d'instruction (ISA), ces attaques permettent à un attaquant de prendre le contrôle des ressources protégées, en contournant les mécanismes mis en place par le système d'exploitation (OS) pour isoler les différents processus. Ces attaques peuvent tirer parti de toutes les parties d'un SoC : CPU (flow control units, opérateurs,...), caches, mémoire, accélérateurs (FPGA, GPU,), interfaces, etc. La compréhension, la reproduction et l'instrumentation de ces attaques et de leurs scénarios associés, bénéficient beaucoup de la possibilité de les simuler sur plate-forme virtuelle. Le projet **Archisec** adopte cette méthodologie *simulation pour la sécurité*[For+21] en s'appuyant sur le simulateur *gem5* pour développer une plate-forme virtuelle capable de reproduire des attaques micro-architecturales. L'un des aspects essentiels de la sécurité d'un SoC réside dans son *Trusted Execution Environment* (TEE: environment d'exécution de confiance). Dans le TEE, des tâches spécifiques sont exécutées de manière isolée dans des enclaves sécurisées, ce qui les protège des attaques, même lorsque le système d'exploitation est compromis. Le TEE joue ainsi un rôle essentiel dans la protection des applications telles que les mises à jour du logiciel intégré, les opérations bancaires, etc. Cependant, les attaquants cherchent activement des moyens de contourner ces protections, comme le montrent les cas documentés d'attaques micro-architecturales contre les TEE.

C'est pourquoi la plate-forme **Archisec** doit prendre en charge les TEEs tout en simulant la micro-architecture. Le projet étant axé sur ARM, nous avons choisi d'utiliser OP-TEE, un TEE open-source standard pour ARM. Pour déployer OP-TEE sur une plate-forme ARMv8-A, elle doit supporter TrustZone, le framework de sécurité propre à ARMv8-A et ARMv7-A.

Pour évaluer cette plate-forme, j'ai créé une bibliothèque d'attaques de cache-timing pour ARMv8-A afin de comparer les résultats de cache-timing entre *gem5* et raspberry PI. Dans ce même objectif, j'ai amélioré le support de l'ISA ARM et de TrustZone dans *gem5* pour faire tourner une bootrom contenant OP-TEE, compilée pour l'occasion. Afin d'étudier les vulnérabilité des TEEs, j'ai créé des scénarios d'attaque contre OP-TEE en utilisant les simulations de *gem5*. Après avoir amélioré le module du débogueur *GDB* intégré dans *gem5*, j'ai développé une interface qui utilise le module *GDB* dans *gem5* au travers de scripts *GDB*. Ces scripts analysent des scénarios d'attaque, via l'extraction les états des caches et la reconfiguration du simulateur à la volée. En utilisant cette interface, et avec les outils *gem5* existants, j'ai construit une première contribution qui utilise *gem5* pour étudier des IPs third-party et des librairies cryptographiques [FDC23].

Ma deuxième contribution est *TEE-Time* [FDC24b], un outil qui analyse automatiquement les side-channels cache-timing en utilisant mon interface *gem5-GDB*. Cet outil opère à partir de *Key Execution Points (KEP)* qui englobent toutes les connaissances sur algorithme cryptographique, sans faire d'hypothèse sur l'architecture spécifique du processeur. On peut supposer que si un attaquant parvient à détecter ces *KEPs*, il sera en mesure de reconstituer tout ou partie du secret. Pour évaluer dans quelle mesure il serait possible pour un attaquant de repérer ces *KEPs*, et donc, compromettre l'application victime, *TEE-Time* produit des rapports qui décrivent et évaluent un scénario d'attaque idéal pour récupérer les *KEPs*. Ensuite, un script d'*attack monitoring*, utilisant la même interface *GDB-gem5*, valide le scénario d'attaque décrit dans le rapport. Pour ce faire, ce script rajoute aux traces de cache-timing des marques correspondant aux *KEPs* pour identifier des motifs dans les traces de cache. Ce processus en deux étapes a été validé contre des applications démos, puis contre des applications cryptographiques standards utilisant les services cryptographiques RSA présents dans OP-TEE.

Pour valider les prédictions de cette méthodologie sur un système réel, j'ai développé une plate-forme virtuelle capable de reproduire le SoC *RK3399* de Rockchip, au cur de la carte RockPi4. Le *RK3399* est doté d'une protection mémoire TrustZone, de fusibles sécurisés, d'un démarrage sécurisé et d'un débogueur JTAG verrouillable. Pour construire la plate-forme virtuelle du *RK3399*, j'ai développé un nouvel outil de prototypage rapide pour *gem5* appelé *PyDevices* qui utilise l'interface Python de *gem5* pour implémenter des blocs matériels. Avec mes *PyDevices* et *Ghidra*, j'ai rétro-ingénerer la `BootROM` du *RK3399* et j'ai ainsi pu construire une plate-forme de simulation qui imite fidèlement le *RK3399*. La plates-forme virtuelle peut ainsi booter la même image de carte SD que celle chargée sur la plate-forme réelle. En utilisant mes outils d'attaque cache, j'ai découvert que le *RK3399* réel utilisait un protocole de cache spécifique à ARM appelé **AutoLock**. Avec l'incorporation de ce mécanisme dans *gem5* et l'amélioration les scripts de *TEE-Time*, j'ai validé un scénario d'attaque idéal contre les services cryptographiques RSA de l'OP-TEE sur la plate-forme simulée et sur la carte RockPi4 réelle. Il s'agit ici de ma troisième contribution [FDC24a]. Elle présente une attaque réelle développée en utilisant ma méthodologie basée sur la simulation. Dans ce scénario d'attaque, OP-TEE utilise l'exponentiation de grand nombre entier (*bignum*) de la libraire *mbedTLS* qui met en uvre l'exponentiation modulaire dite *"sliding-window"*. Sur la base de [Ber+17], j'ai configuré l'outil *TEE-Time* pour confectionner une attaque de cache-timing. L'attaque a ensuite été vérifiée dans l'environnement simulé à l'aide du script *attack monitoring*. En exécutant la même attaque sans aucune modification sur un RockPi4, j'ai réussi à faire fuir en moyenne $\sim 1/3$ des bits de la clé RSA, contournant ainsi la protection d'OP-TEE.

Cette dernière contribution constitue une étape cruciale pour combler le fossé entre simulation d'attaque et leur application sur du matériel réel, remplissant ainsi un objectif clé du projet Archisec.

Acknowledgements

Writing this thesis was only possible thanks to the support of many people in my research environment and in my relatives. I first want to thank my advisors Jean-Luc Danger and Sumanta Chaudhuri who guided me in my research. Their expertise and ability to channel my efforts into meaningful and impactful research have been instrumental in shaping my work into proper academic publications.

I am also profoundly thankful to Chadi Jabbour, Florent Brugier, Guillaume Duc, Taric Graba for trusting me to teach and supervise lab classes. Their confidence in me allowed me to develop and refine my teaching and supervisory skills, testing them against the reality of the field. Additionally, I wish to thank Arnaud Varillon and Julien Béguinot, fellow PhD students, for the insightful discussions and collaborative exchanges we shared. Their input and perspectives have greatly contributed to my research. I am also grateful for their initiative in organizing the SSH seminar, where I had the opportunity to present and refine some of my work.

I would also like to extend my heartfelt thanks to Lilian Bossuet, David Novo, Maria Mushtaq, Clémentine Maurice, Philippe Nguyen, and Florent Bruguier, as well as their students Walid J. Ghandour, Carlos Andres Lara Niño, Loïc France, Pierre Ayoub, and Sammy Plat, for their trust in our collaboration for the Archi-Sec ANR project. The feedback and insights they provided during our regular meetings were invaluable, helping me refine my ideas and guiding my contributions toward our shared goal of developing a platform for embedded systems security.

I also wish to thank the Electrical Engineering Department at ENS Paris-Saclay, where I completed my graduate program. The department not only provided a supportive academic environment but also played a pivotal role in developing my engineering skills and broadening my knowledge of the field.

I am deeply grateful to Guillaume Hiet and Gilles Sassatelli for reviewing my thesis and providing essential feedback that allowed me to significantly improve my manuscript. Their detailed critiques and valuable suggestions were crucial in enhancing the quality and clarity of my work. I also wish to thank Damien Couroussé for participating as an examiner in my thesis jury and contributing his expertise to the evaluation process.

A special acknowledgement goes again to Lilian Bossuet for serving as the president of my thesis jury and for his insightful feedback through my thesis which greatly contributed to refining my work. Finally, I express my regrets that Clémentine Maurice, although invited, could not participate as an examiner in my jury due to unforeseen circumstances.

A final word of gratitude goes to my family, who provided me with a home and support throughout my thesis, especially during the challenges posed by the COVID-19 pandemic. I would also like to express my deep appreciation to my girlfriend, Sophie, for her unwavering support, despite the demanding nature of thesis work and the impromptu tasks it often entails.

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-19-CE39-0008 (project ARCHI-SEC)

Contents

abstract **1**

Résumé en français **2**

Acknowledgements **3**

Glossary **7**

1 Introduction **8**
1.1 Context . . . 9
1.1.1 Trusted Execution Environment . . . 9
1.1.2 Micro-architectural attacks . . . 10
1.1.3 Archisec Project . . . 10
1.2 Scope: The Archisec Virtual Platform . . . 11
1.3 Motivations . . . 11
1.4 Key contributions . . . 12
1.5 Thesis organization . . . 12

2 State of the art: security and simulation for embedded systems **13**
2.1 Introduction . . . 14
2.2 System-on-Chip simulation . . . 14
2.2.1 Simulators . . . 14
2.2.2 *gem5*: the SoC simulator . . . 16
2.2.2.1 Principle . . . 16
2.2.2.2 *SimObjects*: *gem5* primitives . . . 17
2.2.2.3 Models advantages and limits . . . 18
2.2.2.4 Memory model . . . 20
2.2.3 System element modeling . . . 21
2.2.4 *gem5* for security in literature . . . 23
2.3 Security in embedded systems . . . 23
2.3.1 Operating system security . . . 23
2.3.2 Attack scenarios . . . 24
2.3.3 Execution Privileges . . . 25
2.3.4 Trusted Execution Environment . . . 26
2.4 Micro-architectural attacks . . . 27
2.4.1 Side-channel attacks . . . 27
2.4.1.1 Cache timing attacks . . . 27
2.4.1.2 Higher-level cache attacks . . . 28
2.4.1.3 Static and dynamic cache analyzer . . . 29
2.4.1.4 Other side-channel attacks . . . 29
2.4.2 Transient execution attacks . . . 29
2.4.3 Fault injection attack . . . 30
2.4.3.1 Typical fault injections . . . 30
2.4.3.2 Hadware memory corruption: *RowHammer* . . . 31
2.4.4 Accelerator attacks . . . 31
2.4.5 Trusted Execution Environment attack in literature . . . 32
2.5 Conclusion . . . 32

3 Virtual platform on gem5 for ARMv8-A, TrustZone, and OP-TEE **33**
3.1 Introduction . . . 34
3.2 Platform and instrumentation with *gem5* . . . 34
3.2.1 Building a platform in *gem5* . . . 34
3.2.1.1 Writing a new config file . . . 35

3.2.1.2 Adding new *SimObject* . . . 35
3.2.2 Classical instrumentation on *gem5* . . . 37
3.2.2.1 *DebugFlag* . . . 37
3.2.2.2 *m5* instructions . . . 38
3.2.2.3 *CxxMethod* . . . 38
3.2.3 Our improvement to *GDB* in *gem5* . . . 39
3.2.3.1 *GDB* monitor call . . . 39
3.2.3.2 Interactive debug with *GDB* . . . 40
3.3 ARMv8-A security on *gem5* . . . 41
3.3.1 *aarch64* and its *gem5* model . . . 41
3.3.1.1 *aarch64* generalties . . . 42
3.3.1.2 *gem5* ARM platform model . . . 42
3.3.1.3 ARM cache model and *AutoLock* . . . 42
3.3.2 Cache timing attack on *aarch64* . . . 43
3.3.2.1 *Flush+Reload* . . . 43
3.3.2.2 *Prime+Probe* . . . 44
3.3.2.3 *Prime+Probe* direction and self-eviction . . . 45
3.3.3 Our baremetal prospects . . . 46
3.3.3.1 Principle . . . 46
3.3.3.2 Results . . . 47
3.4 ARM TrustZone and OP-TEE on *gem5* . . . 48
3.4.1 TrustZone . . . 48
3.4.2 Platform and boot model . . . 49
3.4.3 OP-TEE software model . . . 51
3.4.4 Refining TrustZone implementation in *gem5* to support OP-TEE . . . 53
3.4.5 Our typical OP-TEE scenarios . . . 54
3.4.6 Third Party IP simulation . . . 54
3.5 Conclusion . . . 58
3.A Appendix . . . 59
3.A.1 *GDB* API in *gem5*-Python . . . 59
3.A.2 ARM system devices in *gem5* . . . 59
3.A.3 Timing gadget on ARM . . . 60

4 TEE-Time: Simulating to get security insights 61
4.1 Introduction . . . 62
4.2 Key issues . . . 62
4.2.1 Cache timing attacks on Trusted Execution Environments . . . 63
4.2.2 Exploring attack complexity . . . 63
4.3 TEE-Time methodology . . . 64
4.3.1 Overview of TEE-Time process . . . 64
4.3.2 *Key Detectable States* . . . 65
4.3.2.1 VictimScan policy: 1hit . . . 66
4.3.2.2 VictimScan policy: nhit . . . 66
4.3.2.3 VictimScan policy: nhit_inclusive . . . 66
4.3.3 Ranking methodology . . . 67
4.3.4 Attack configuration and *Key Detectable States* . . . 67
4.4 TEE-Time implementation . . . 69
4.4.1 Instrumenting the attack scenario . . . 69
4.4.2 Dedicated *GDB* scripts . . . 69
4.4.2.1 *VictimScan* . . . 70
4.4.2.2 *Attack Monitoring* . . . 73
4.5 Example: demo cryptographic function . . . 74
4.5.1 Demo: *VictimScan* . . . 74
4.5.2 Demo: *Attack Monitoring* . . . 75
4.5.3 TEE-Time: Code coverage . . . 76
4.6 Attack against RSA signing in OP-TEE . . . 76
4.6.1 *mbedTLS* bignum exponentiation . . . 76

4.6.2 RSA: *VictimScan* . . . 78
4.6.3 RSA: *Attack Monitoring* . . . 78
4.7 Conclusion . . . 80

5 Rockchip-platform: An accurate simulation model for a real TEE hardware 81
5.1 Introduction . . . 82
5.2 About the RockPi4 and its *RK3399* . . . 82
5.2.1 CPUs, caches, and bus topology . . . 82
5.2.2 *RK3399* boot process . . . 83
5.2.3 Security features . . . 84
5.3 PyDevices: fast prototyping with *gem5* . . . 85
5.3.1 PyDevices: programming model . . . 85
5.3.2 Building a RockPi4 in *gem5* . . . 86
5.3.3 Retro engineering with PyDevices and *Ghidra* . . . 87
5.3.3.1 Bootstraping until the OS . . . 88
5.3.3.2 PyPowerState and Power Management Unit . . . 89
5.3.4 *Rockchip-platform* environment . . . 89
5.4 Using TEE-Time and Prime+Probe on the *Rockchip-platform* . . . 90
5.4.1 Detecting cache configuration . . . 90
5.4.2 *AutoLock* and *Prime+Probe* . . . 91
5.4.3 Pseudo-LRU: LRU implementation on real hardware . . . 92
5.4.4 Running an attack on the *RK3399* . . . 92
5.5 A bridge between theory and real-world: attacking OP-TEE on a *RK3399* . . . 93
5.5.1 Instrumented scenario . . . 93
5.5.2 Using TEE-Time to search for weaknesses . . . 94
5.5.2.1 Finding good *KEPs* against *AutoLock* . . . 94
5.5.2.2 *Attack Monitoring* and real hardware results . . . 96
5.5.3 Extracting a key from real traces . . . 96
5.6 Conclusion . . . 98

6 Conclusion and Perspectives 100
6.1 Introduction . . . 101
6.2 Overview of contribution . . . 101
6.2.1 Contributing to *gem5*: Trusted Execution Environment and *GDB* . . . 101
6.2.2 Virtual Security Platform . . . 101
6.2.3 TEE-Time tools . . . 101
6.2.4 PyDevices: building the *Rockchip-platform* . . . 102
6.2.5 Attacks against hash signing RSA scenarios with TEE . . . 102
6.3 Future works . . . 103
6.4 Concluding remarks . . . 103

Bibliography 105
My publications . . . 105
Other publications . . . 105

A Appendix 113
A.1 About *gem5* . . . 114
A.2 About PyDevices . . . 116
A.3 About ARM . . . 116
A.4 About *RK3399* . . . 118
A.4.1 Retro-engineering . . . 118
A.4.2 Covert-channel results . . . 120

List of Figures 122

List of Tables 126

Glossary

SoC : System-on-Chip

ISA : Instruction Set Architecture

TEE : Trusted Execution Environment

REE : Rich Execution Environment

TA : Trusted Application

Trustlet : trusted application

bignum : Large integer (arithmetic)

Vexpress : ARM Versatile Express Platform

KEP : Key Execution Point

KCL : Key Cache Line

KDS : Key Detectable State

WB : WriteBack

ROB : ReOrder Buffer

MMU : Memory Management Unit

DVFS : Dynamic Voltage and Frequency Scaler

RSA-CRT : RSA-Chinese Reminder Theorem

CLI : Command Line Interface

DTB : Device Tree Blob

GIC : General Interrupt controller

EL : Exception Level

LRU : Least Recently Used.

IRQ : Interrupt request

FIQ : Fast Interrupt request

BL : Boot Loader

TFA : TrustedFirmware-A

PMCCNTR : Performance Monitors Cycle Count Register

CNTPCT : Counter-timer Physical Count

ROI : Region-of-Interest

API : Application programming interface

SimObject : *gem5 system primitives using Python and C++.*

CcObject : *C++ element of SimObject.*

Chapter 1

Introduction

Contents

1.1 Context **9**
1.1.1 Trusted Execution Environment 9
1.1.2 Micro-architectural attacks 10
1.1.3 Archisec Project 10
1.2 Scope: The Archisec Virtual Platform **11**
1.3 Motivations **11**
1.4 Key contributions **12**
1.5 Thesis organization **12**

1.1 Context

Systems-On-Chip are single package computing elements that contain all the elements needed to build an embedded system: CPUs and core complexes (caches, TLB, MMU, interfaces), IO(DRAM controller, UART, etc.), Accelerators(GPUs)...

Systems-on-Chip (SoC) are used ubiquitously in consumer electronics: fridges, cars, smartphones, laptops, etc. In some of these devices, typically smartphones, but in increasingly more devices, SoCs present complex user interfaces (CarPlay, etc.) with non-monolithic firmware, often running a full operating system. These SoCs are hybrids between high-performance computing and embedded devices. these SoCs typically uses ARMv7-A, ARMv8-A and now RISC-V as their **Instruction Set Architecture** (ISA). To simplify development, the same SoCs can also be used in industrial settings. These SoCs usually rely on a full Operating System (OS), generally Linux. This development model is fully modular: their OS has modules and runs multiple applications and daemon. All of these elements can have their own update path, maintainer, and sources. These systems are generally locked down and strongly control what the user can do and run on them. This strongly isolates applications, allowing them to trust the system.

A weakness in one of the system elements could be leveraged to gain full control, thwarting any protection and isolation. This makes embedded devices a target for nefarious actors and their security a stake when designing their hardware and software.

1.1.1 Trusted Execution Environment

To strengthen security in embedded systems, new hardware, and ISA frameworks have been developed in modern systems. These frameworks, known commercially as Intel SGX [CD16] or ARM TrustZone[Nga+16], allow the deployment of a **Trusted Execution Environment (TEE)**[SAB15]. With a TEE, it is possible to protect an application against a privileged escalation[Dav+11]. Applications running in a TEE are called **Trusted Applications (TAs)**(figure 1.1), also known as **trustlets**.

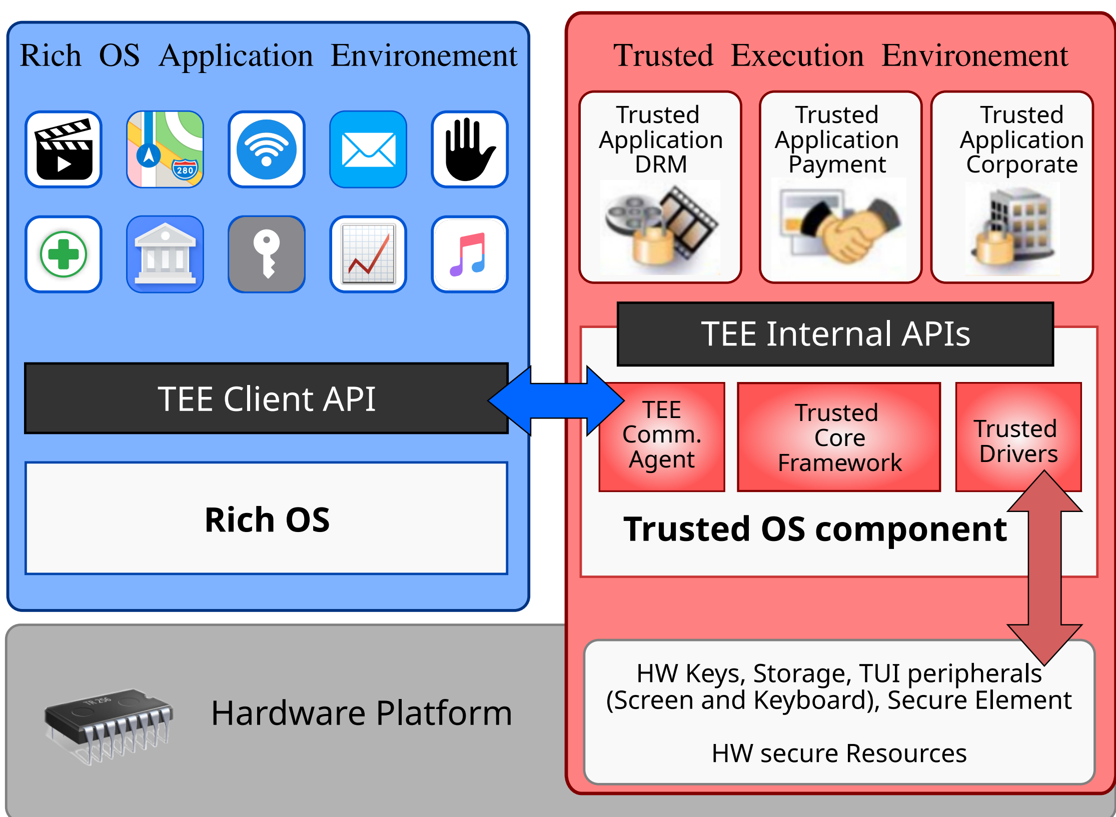


Figure 1.1: Overview of Trusted Execution Environment typical use cases.

TEEs are typically used by a wide range of applications: Digital-Right-Management (DRM), banking applications, secure web browsing, cryptographic libraries, etc. A Trusted Execution Environment generally guarantees for trusted application:

- **Integrity**: A trusted application cannot be tampered with after it has been packaged by its developers.
- **Confidentiality**: The data manipulated by a trusted application should not be accessed and/or altered by other applications (trusted or not).
- **Authentication**: Provenance of the trusted application can be validated with a system of signature.

Moreover, the different APIs used by a Trusted Execution Environment (on the *rich OS* side and on the TEE side) are thoroughly checked for any potential vulnerabilities. For example, SGX uses a system of *encrypted enclaves* to enforce these properties: the code and the data in the enclave are encrypted and decrypted on-the-fly by the CPU (using specific hardware mechanisms).

Other examples of TEEs are: Trusty[And16], Samsung Knox[Sam15], OP-TEE[YL20], Qualcomm QTEE[Qua15]. Besides Trusty, they all use the *GlobalPlatform* API definition.[lea21]

1.1.2 Micro-architectural attacks

Indeed, bugs and incorrect implementations of features are sources of weaknesses: such as buffer overflow[One96], use-after-free[Lee+15], ...; They have been commonly used and patched in OS and critical applications. Countermeasures[Cow+98] and design choices have even been developed to mitigate them. Indeed, Trusted Execution Environments are, by design, less prone to this type of vulnerability as they reduce their interface to the Rich Execution Environment. On the other hand, **Micro-architectural attacks** fixes and countermeasures are more complex to deploy. Indeed, micro-architectural attacks are tightly linked with the platform on which they are used. They take advantage of specific System-on-Chips behaviors. This behavior might be present as an error in the ISA or something not specified by it(like instruction latency). These attacks can target all the parts of a SoCs: CPU (flow control units, operators,...), caches[Per05], memory[GMM16], FPGA[Cha17], GPU[Lad+13], networks interface[DPM11]. etc.

Attacks on CPU cores include the famous *Meltdown* [Lip+18] which exploits an out-of-order execution vulnerability in modern Intel processors, and *Spectre* [Koc+19] which exploit branch prediction for speculative execution. Other well-known attacks in this class are *Zombieload* [Sch+19b], *Fallout* [Min+19], *RIDL* [Sch+19a]. Caches are also a weak element that has been exploited: *Prime+Probe* [Liu+15], *Flush+Reload* [YF14; Gru+16b] and *Evict+Time*[OST06]. Both *Spectre* and *Meltdown* use *Flush+Reload* [YF14] attack to retrieve the data. Indeed, these attacks have been demonstrated against TEEs: "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone"[Rya19].

1.1.3 Archisec Project

To study these micro-architectural attacks and the security of SoC platforms, classical tools like binary instrumentations are not enough. We need a full simulation platform to reproduce and study this attack in a controlled environment. This platform also allows the development and testing of countermeasures against these attacks. This is the main objective of the Archisec Project,*studying SoC security with the aid of a virtual platform.* This project covers (figure 1.2):

- Cache timing attacks.
- Trusted Execution Environment (TEE).[FDC23] [For+21] [FDC24b]
- Power side-channel attack[BGL23][BL23a]
- Speculative execution.[AM21]
- FPGA related attacks.[BL23b][FBL23]
- DRAM attacks [Fra+22][Fra+21a][Fra+21b] and Emerging technology.

This project has multiple aims:

- Producing reports and surveys about micro-architectural attacks on embedded system and their reproducibility.
- Developing a virtual platform to study said attacks.

Regarding ISA and manufacturers, considering the project's focus on embedded security, most of the work has been done on ARM platforms (ARMv7-A and ARMv8-A). Although, the project also mentions RISC-V. With the virtual platform, Archisec also aims to discover and demonstrate new attacks, using the platform to gain insight into vulnerabilities to finally demonstrate them on real hardware.

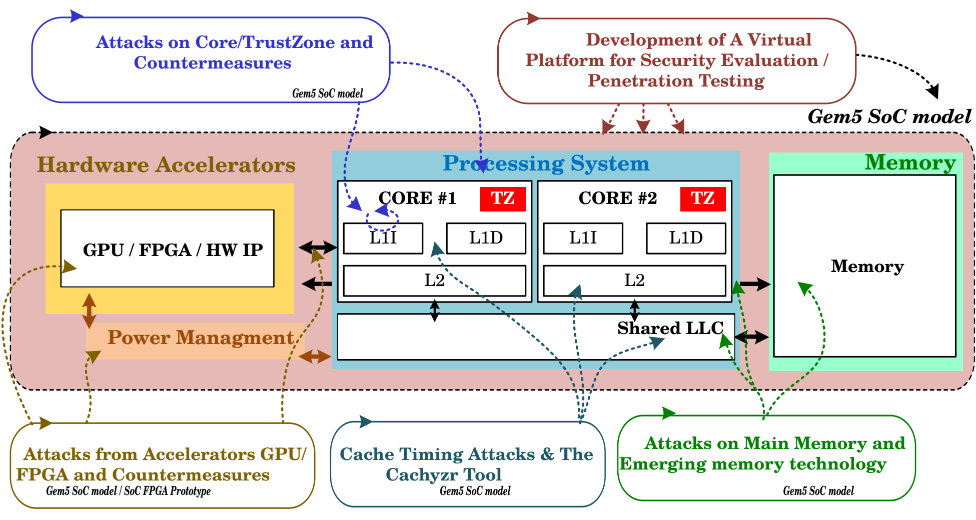


Figure 1.2: Scope of the Archisec project

1.2 Scope: The Archisec Virtual Platform

The simulator *gem5* [Low+20] has been chosen, to serve as a base for the platform (figure 1.3). Its industry-standard status and modularity were deciding factors for that. (please see section 2.2 for more detail). Most of the work of this thesis has been done using *gem5* and comparing its results against real platforms when necessary. Although gem5 support a wide variety of ISA (X86, ARMv8-A, RISC-V, ...), this thesis only focuses on ARMv8-A. More specifically, my examples and attacks use *aarch64* instructions. As I was in charge of developing the TEE component of the platform, this thesis mainly covers its implementation in *gem5* and the research related to it. Attack-wise, most of the thesis leverages cache timing attacks. With this platform, described in figure 1.3, we can integrate a real secure workload in a Python programmable SoC model using a wide variety of primitives which integrate with the *gem5* simulator. We can then use *GDB*, in tandem with the platform, to follow the execution of our unmodified secure workload and analyze its behavior in a simulated environment.

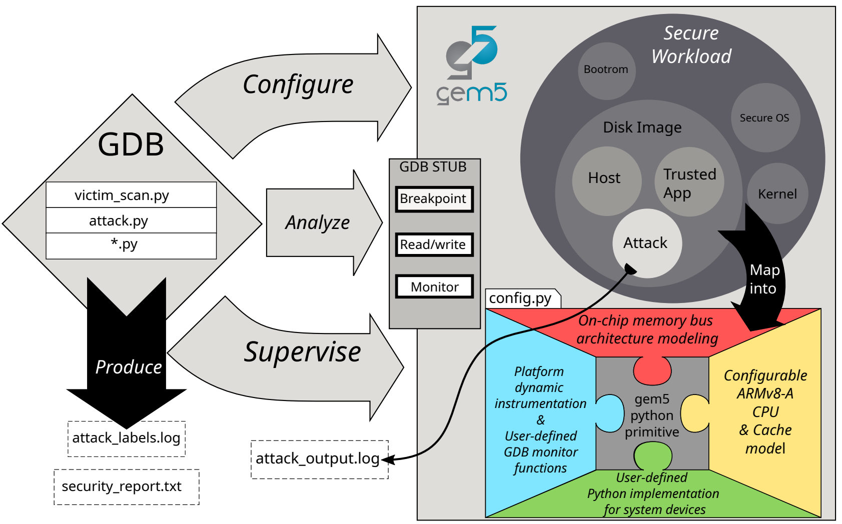


Figure 1.3: Overview of the Archisec platform, instrumentation tools, and simulation capabilities

1.3 Motivations

On customer devices, Trusted Execution Environments deployment and development is the privilege of the original equipment manufacturer. This is necessary to prevent any tempering once the device is in use. If the end user can deploy applications and use the TEE, they do not have full access to debug information and can not study an application that they did not create. In that regard, it becomes harder to study the security properties of TEE, although static analysis is still possible. However, micro-architectural attacks are intricate attacks and are demonstrated in proof of concept which often requires precise knowledge of how an application is deployed. This type of attack is thus complicated to study statistically without gathering information about their execution environments. Trying to deploy such attacks without any certainty of an expected result is illusory. With these assumptions, it seems that such attacks can be efficiently

demonstrated only by the original equipment manufacturer. To bypass this security by obscurity scheme, we proposed to leverage a configurable simulation model to imitate the micro-architectural behavior of the real hardware while keeping the same binary workloads (trusted applications, disk-image, bootrom, etc.). And, since simulation is widely slower than the real hardware, we have to design new methodologies that simplify the attack exploration without requiring too many simulation runs. In that regard, we can leverage the possibilities of a simulation environment to access internal micro-architectural states and exposed them to external tools for study.

1.4 Key contributions

In this thesis, I present the following contributions, which have been presented to peer-reviewed conferences and journals:

- For the first time (to my knowledge), I developed an open-source virtual platform capable of booting *GlobalPlatform*-compliant TEEs and *rich OSes* (Linux). (presented in [FDC23])
- Thanks to the *gem5 GDB*-integration, I opened up the possibility of gathering and analyzing cache traces on the fly during simulation. (presented in [FDC23])
- I built TEE-Time, a cache analyzer relying on this principle and demonstrated it against a real cryptographic implementation. (presented in [FDC24b])
- I developed using *gem5* a fast-prototyping method in order to port the *RK3399*[Roc21] platform to *gem5*. (presented in [FDC24a])
- I used this *RK3399* virtual platform and TEE-Time to build an attack against the RSA implementation in OP-TEE running on a real RockPi4. (presented in [FDC24a])

1.5 Thesis organization

In the first chapter, we present the state-of-art related to SoC and their simulation, specifically *gem5* and how it is used to study security issues. We then detail the typical security features of embedded systems. Finally, we cover the typical micro-architectural attacks that circumvent these security features. In the second chapter, we detail how we built our *gem5* virtual platform for security. In this chapter, we focus on three aspects. The *platform*: how to leverage the simulation environments in *gem5* to study security issues, the *attacks*: how to implement and run real *aarch64* attacks in *gem5*, the *TEEs*: how to run in *gem5* a Trusted Execution Environment that can represent how victim applications are protected in typical systems. Then, in the third chapter, we present a new methodology that leverages the platform and tools we built to study more efficiently a victim using all the information that can be extracted from *gem5* (cache states, internal CPU states, ...). This methodology generally allows to dynamically reconfigure *gem5* using *GDB*. This technology was proposed in multiple papers under different names: **VictimScan** or **TEE-Time** published in [FDC23] and [FDC24b]. It is a demonstration of the *gem5* virtual platform for security possibilities. In the final chapter, we cover how we improved our platform to simulate a typical attack scenario against a RSA application using recommended security practices without any modification. We explain then how we leverage the tools we presented in the third chapter to configure an attack . This attack successfully extracts a partial key in both simulation and on a real RockPi4 board. Finally, we conclude on all the possibilities rendered open by our new and now-proven methodologies and virtual platforms.

Chapter 2

State of the art: security and simulation for embedded systems

Contents

2.1 Introduction . . . **14**
2.2 System-on-Chip simulation . . . **14**
2.2.1 Simulators . . . 14
2.2.2 *gem5*: the SoC simulator . . . 16
2.2.2.1 Principle . . . 16
2.2.2.2 *SimObjects*: *gem5* primitives . . . 17
2.2.2.3 Models advantages and limits . . . 18
2.2.2.4 Memory model . . . 20
2.2.3 System element modeling . . . 21
2.2.4 *gem5* for security in literature . . . 23
2.3 Security in embedded systems . . . **23**
2.3.1 Operating system security . . . 23
2.3.2 Attack scenarios . . . 24
2.3.3 Execution Privileges . . . 25
2.3.4 Trusted Execution Environment . . . 26
2.4 Micro-architectural attacks . . . **27**
2.4.1 Side-channel attacks . . . 27
2.4.1.1 Cache timing attacks . . . 27
2.4.1.2 Higher-level cache attacks . . . 28
2.4.1.3 Static and dynamic cache analyzer . . . 29
2.4.1.4 Other side-channel attacks . . . 29
2.4.2 Transient execution attacks . . . 29
2.4.3 Fault injection attack . . . 30
2.4.3.1 Typical fault injections . . . 30
2.4.3.2 Hadware memory corruption: *RowHammer* . . . 31
2.4.4 Accelerator attacks . . . 31
2.4.5 Trusted Execution Environment attack in literature . . . 32
2.5 Conclusion . . . **32**

2.1 Introduction

Security and simulation technologies are widely explored themes on their own. To understand how micro-architectural attack works and how to simulate them: We will also have to cover the basic elements of system security.

2.2 System-on-Chip simulation

To thoroughly investigate the properties of a program's execution, particularly its interactions with hardware, leveraging a fully virtualized environment offers unprecedented opportunities for instrumentation. Unlike real hardware, constrained by the visibility afforded by integrated debuggers (if available), a virtual platform can simulate program executions while meticulously observing specific interactions with the hardware. This capability renders simulators indispensable tools for in-depth security analyses.

2.2.1 Simulators

Simulators try to replicate how a SoC executes a workload. They can model components at different level ([Vah10]):

- **Register Transfer Level** (figure 2.1): It represents a logical element as sets of net, registers, gates, and memory. It can be generated by a synthesis phase from a HDL (Hardware Description Language). RTL simulator models how signal and register states evolved in response to outside signals. An RTL model can represent a simple component (e.g., a PWM controller, a UART PHY, etc.) or a complex one (e.g., an out-of-order CPU) up to a full System-on-Chip. For that, RTL simulators use an event-driven approach that uses time at *ps* scale to organize events way below cycle time length. Figure 2.1 illustrates how the RTL simulators compute signal updates using sensitivity lists for each combinatorial operator. RTL model can encompass information about a placed and routed standard cell implementation. In this case, simulators can account for considerations related to an ASIC or FPGA design (clock analysis, power analysis, etc.) RTL simulators are typically integrated into synthesis tools like Platform Architect/VCS from Synopsys[Syn21]. They are generally closed-source.

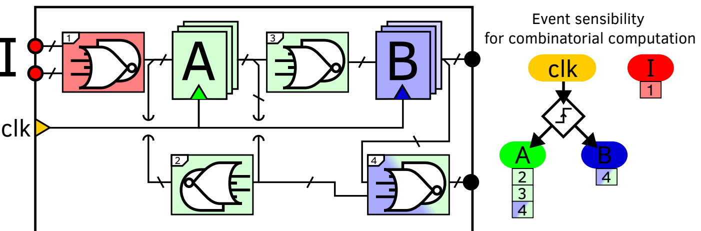


Figure 2.1: RTL simulator: when input is updated, all the logical gates that depend on it are computed, w.r.t clock-edge sensibility. Intermediate signal are stored for future computation, as any signal update is generally considered atomic.

- **Behavioral level** (figure 2.2): At this level, SoC elements are represented as behavioral descriptions. They detail, in Hardware Description Language(HDL) or other system languages (like systemC), how an element of a SoC behaves in response to outside signals. The same description can be used to generate a RTL model. On the contrary, behavioral level models can be *abstract*, unsynthesizable, only giving a higher-level view of actual hardware. Behavioral level models are simulated at the cycle level and below using an event-driven simulation. However, the simpler the model the faster the simulation is. This is why, outside of HDL verification, abstract models are preferred. As they still represent signal evolution at clocked interfaces, abstract models can be used to verify performances (typically operation per cycle). They can also model more efficiently a component at the interface of a synthesizable model

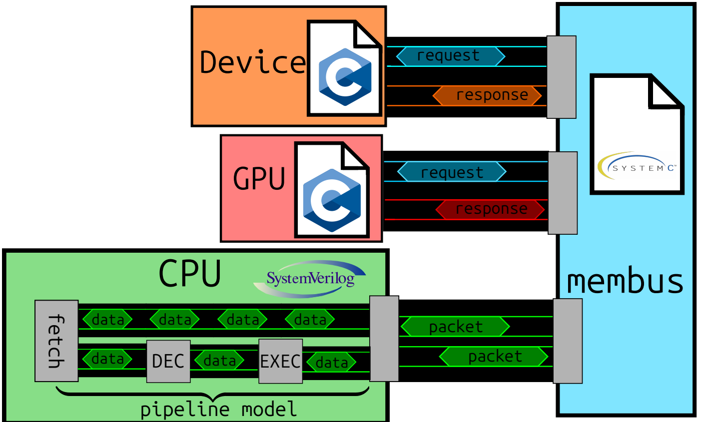


Figure 2.2: Behavioral simulator: component with different model types, exchange messages.

or a RTL simulation. This diversity of model is illustrated on figure 2.2 in which abstract C models for devices and GPU interact with a *SystemVerilog* model for the CPU and a *SystemC* model for the bus. An example of a behavioral simulator is Verilator[Sny13], which can simulate Verilog. SystemC[Swa01] was designed with system simulation in mind to simulate a full system: It contains both abstract and synthetizable models. SystemC is both a language and a library, with implementation in both open-source and closed-source tools.

- **ISA level** (figure 2.3): At the Instruction Set Architecture level, models only account for the functional effect of instructions including visible effects on peripheral devices (video output, terminal message, etc.). For example, emulators simulate CPUs at the ISA level, ignoring pipeline effects. As they only model the effect of instruction on the ISA state, they are used to test ISA and develop software. In that context, they are called *Virtual Model*. Because of their simpler nature, their simulations are marginally faster than behavioral models. In some cases, these models are integrated into behavioral and RTL simulators to accelerate simulation. They are often described as *Functional Model*. ARM FastModels[ARM21b] are examples of virtual models for ARMv8-A architecture. *QEMU*[Bel05] is an example of an open-source emulator. It can simulate different ISA and platform features by implementing functional models of peripheral devices. It uses ASM recompilation to accelerate emulation through its *Tiny Code Generator*. On figure 2.3, we compared the architecture of simulator (like *gem5*) which contains models for internal SoC elements and an emulator (like *QEMU*), which translates the binary to natively run on the host machine CPU with a *recompiler*. To reproduce the functional effects of the binary executions from the original platform, the emulator contains an ISA model that is updated by the recompiled binary. However, the original platform's unintended behaviors are lost because of the translation.

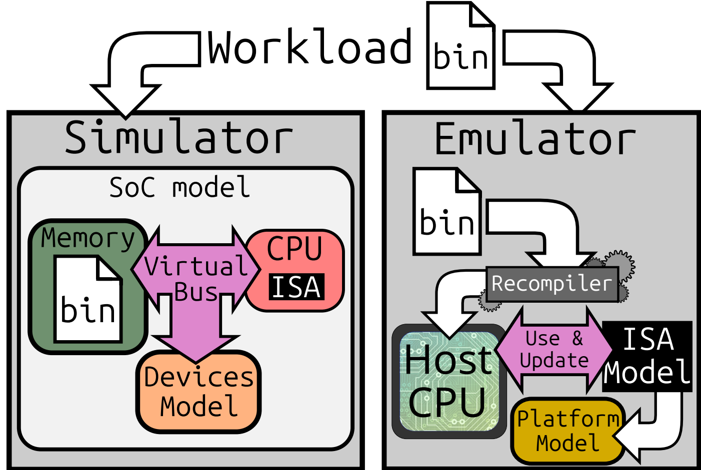


Figure 2.3: Emulator: an emulator only reproduces the functional effect of the ISA and system components. Aspects of systems that are platform dependant (performances, randomness, etc.) are not reproduced

This different type of simulation can be integrated into mixed simulators that simulate different parts of Systems-on-Chip with different precision to only study specific properties. In [Smi97], Smith describes lower-level simulations for ASICs: Gate-level simulation, switch-level simulation, and transistor-level simulation.

Our suggested simulator, *gem5*[Low+20], is widely used in computer architecture research. It is a system simulator, which means it simulates a complete system and not specific logic components. It mostly uses abstract models to model CPU, caches, and RAM. On the other hand, it uses functional models for peripheral devices. Considering that our requirements for our virtual platform are:

- To run full SoC workloads which interact with secure enclaves.
- To provide accurate models for the micro-architectural attacks that reproduce observable effects of real hardware.
- To instrument the model used for the simulation to gain more insight into security properties than what is possible on a real platform.
- To perform efficiently for the scenarios we want to reproduce.

In table 2.1, we compared different simulators to make educated choices on which simulator is suitable for our project. When factoring our ISA constrain (ARMv8-A) and open-source requirement, considering that we need to be able to modify the simulator, we concluded that *gem5* was the most adequate choices for our project.

	Real Platform	*gem5*[Low+20]	QEMU[Bel05]	SystemC Cycle Model[ARM17b]
Easy to use	✅: standard use case.	🟡: have to be configured to operate like our real platform.	✅: is compatible with most software	❌: we need to find or write a model for our real platform.
Speed	✅:fastest	🟡:slow	✅:fast	❌:slowest
Modifiable/Adaptable	❌: cannot be modified	✅: highly customizable	✅: customizable but no separate config files	🟡: modifiable but requires complex SystemC model.
Accuracy	✅:model for accuracy	🟡: model micro-arch to an extent	❌: does not model micro-arch	✅: only omit thermal effects and interface noises
Instrumentation	❌: JTAG are locked on secure platforms	✅: designed to extract architecture statistics using instrumented workloads	✅: implement hooks to dynamically instrument binaries	🟡: JTAG interface is available.

Table 2.1: Comparison between simulators and the real platform as a reference. We can see that *gem5* is a suitable tool for our use case.

2.2.2 *gem5*: the SoC simulator

gem5[Low+20] is an open-source simulator that comes from the fusion of m5[Bin+06] and GEMS[Mar+05] simulator. It is widely used in the computer architecture community to test and demonstrate functionalities and design, checking the effect on performance and software behavior.

2.2.2.1 Principle

As *gem5* separates hardware models and ISA, it features implementation for classical ISA (x86_64, ARMv8/v7, RISC-V, MIPS,...) [Low18] which can be applied to any CPU model (in-order, out-of-order,....).

It works as a Python interpreter that can be used to build a full architecture in Python and then simulate its behavior by invoking a Python function. It can be provided with Python files called **Config Files** or **Config Scripts**. They use the `m5` implemented by *gem5* using `pybind11` [JRM17]. When run by *gem5*, **Config Files** use the specific primitives provided in `m5` to: configure a SoC architecture, connect all the elements, run the simulation loops and react to its return value (e.g. taking checkpoints, compiling statistics). For all other considerations, these config files are normal Python files that can use other Python libraries. *gem5* has two main simulation modes:

- *SystemCall Emulation (SE)*: *gem5* runs a Linux/BSD application while only emulating the system calls (High-Level Emulation).
- *FullSystem (FS)*: *gem5* simulates the complete system. The loaded binaries are firmware and bootroms.

gem5 can also be implemented as a library. *gem5* simulation uses an event-based simulation with event queues behaving as a priority queue. Events can be scheduled at ticks, which organizes them in the queue. In each simulation loop, the event scheduled at the earliest ticks executes its workload, which corresponds to the model of a certain SoC element. This event is then retired, and the process can continue to the next simulation loop. These event queues are represented on figure 2.4. Each event is symbolized by a square and is ordered by tick. We see *Object4* scheduling an event on queue 1. Its event is inserted in the queue using its scheduled tick. Events are then popped out of the queue from the lowest tick, which corresponds to the queue *CurTick*. As shown on figure 2.9 and figure 2.8, when they are popped out of the queue, the event's `process()` method is called. This method triggers functions in their related object which in turn can interact with other objects and schedule other events (with `schedule(event, latency)`).

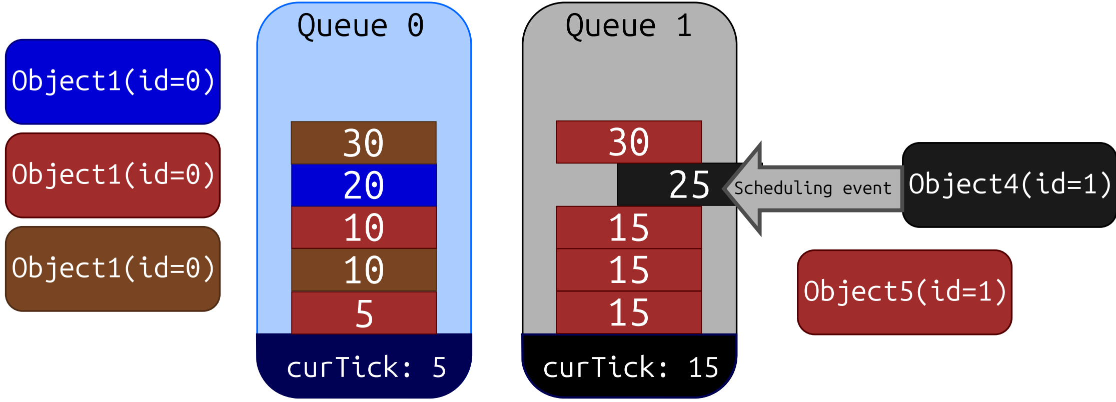


Figure 2.4: Simplified representation of *gem5* event queue (*gem5* only use the queue 0)

gem5 can be compiled in different binary depending on the optimization wanted:

- `gem5.debug`: Debug version with few optimizations
- `gem5.opt`: Classically optimized but with debug messages
- `gem5.fast`: Most optimized version, no debug message.

2.2.2.2 *SimObjects*: *gem5* primitives

SimObjects are the basic component of the *gem5* simulation. They are the main *gem5* primitives. Everything in a *gem5* simulation, CPUs, memory, cache, devices, etc. are *SimObjects*. *SimObjects* are assembled and linked in the Python config file. In the same `.py` file, *SimObjects* are then instantiated using a dedicated Python function. *SimObjects* consist of :

- **Python Class**: It is used in the Python config file to build the architecture
- **Param Class**: An automatic generated C++ class that goes with this Python class.
- **CcObject Class**: A manually written C++ class that is instantiated using the *Param Class* Object. (also called *CxxClass*)

Most of *gem5*'s execution is made of compiled C++ code, either in SimObjects or helper objects that are not visible from Python. In the config file, through explicit and hidden attributes, *SimObjects* constitute a tree of objects, with a special *Root SimObject* as the root of the tree. On figure 2.5, we represent this operation performed by *gem5* which build a *SimObject* tree when it runs the config file using *libpython*.

The node of this tree, *SimObjects*, can be configured using an attribute called *Params* in *gem5*. On figure 2.5, we see their affectation in Python as attributes of their *SimObject*. *Params* can be:

- Classical variable type: *String, Int, ...*
- Simulation-related variable type: *Tick, AddrRange, Enum, ...*
- Other *SimObject* Python classes.
- Port of different types that have to be paired with same-typed ports.

It is this *SimObject* as a *Param* relation that creates the *SimObject* tree. In addition, it is also possible to provide proxy values to this attribute, this is represented on figure 2.5 using the *cpu_clk SimObject* and its proxy alias *clk*. These proxies can also automatically search in the *SimObject* tree for a correct value for the attribute and then forward it. The *proxy link* is only resolved at the start of the exponentiation. When the architecture is completely defined, meaning that all the *SimObject* attributes have been filled in the config file. The user can called `m5.instantiate` to instantiate all the *SimObject*:

1. The Python class object transfers all the *Params* to the *Param Class* object.

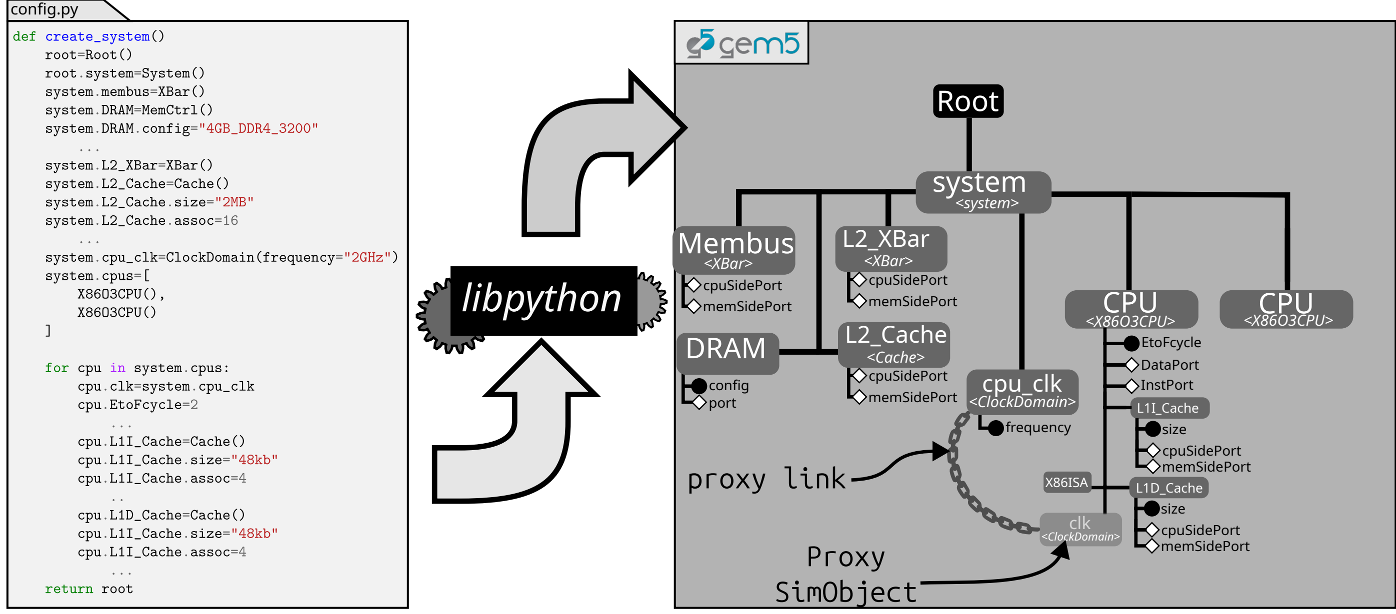


Figure 2.5: Image of the tree of *SimObject* and *Params* that is created by config files in *gem5*

2. The *Param Class* machinery produces the *CcObject class*, by using itself as arguments for its *CcObject* constructor.

3. Ports at the C++ level are paired with the one they are paired with in the config file.

When all the objects are ready, the user can call `m5.simulate()` to launch the simulation. The `startup()` function will be called for each *SimObject* at the first simulation loop. As we explained, *SimObjects* can then interact with the event queue by scheduling events (see figure A.1). Each *SimObject* schedules events on a queue that is automatically determined by parent *SimObjects* on the queue or user-specified manually. However, multiple event queues are generally a source of crashes and should not be used for closely interacting *SimObjects*. On figure 2.6 is a simplified typical *gem5* config file. These files are organized as we mentioned, with a part building the *SimObject* tree starting from the root, a called to `m5.instantiate()` to instantiate all the C++ models. and finally a called to `m5.simulate()` to launch the simulation. This last function terminates when specific events (simulation exit events) are reached or when there are no longer any event in the event queues.

config.py

```python
#instantiating root (the first SimObject)
root=Root(full_system=True)
#this function adds children SimObject to the
↪ root.
add_children(root)

#this instantiates all the children and resolves
↪ the proxies
m5.instantiate()

#launch the simulation
m5.simulate()
```

Figure 2.6: Typical config file for *gem5* written in Python.

2.2.2.3 Models advantages and limits

gem5 splits hardware models and the ISA implementation, which allows the CPU model to be independent of the ISA used. *gem5* provides different CPU models. We provided an illustration for each of them on figure 2.7 and a demonstration of their execution behavior:

- **SimpleCPUs**: called *AtomicSimple* and *TimingSimple*, these CPUs only have one pipeline stage, which is artificially separated inside between *fetch*, *decode*, and *execute*. It interacts with the two ports (instruction port and data port). This stage is triggered at each clock tick (program with a scheduled event), which is postponed if a memory transaction latency needs more time. Instructions can only be executed in order.

- **Minor**: an in-order CPU model. It uses a basic pipeline. Each stage is independently ticked and is only stalled by the further stages being busy. Each stage has an independent latency and throughput. Moreover, the execution

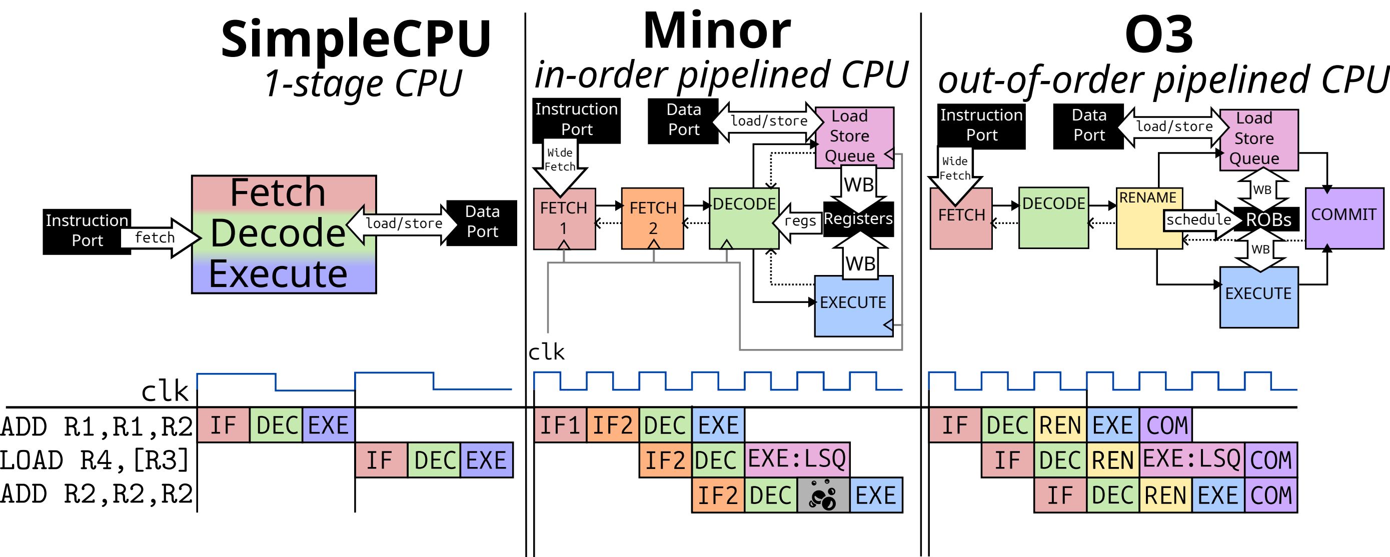


Figure 2.7: Overview of CPU models in *gem5*. They represent typical CPU architecture. Their pipelines tick following a clock which is modeled using regular events

stage can have a different latency and throughput for each type of instruction. This latency difference can create bubbles in the pipeline, as shown on figure 2.7. This CPU model can perform *wide instruction fetch*, which is then separated into multiple instructions by the *Fetch2* phase. ARM proposes their special configuration for Minor: HPI and ex5_little. Their latency and throughput have been fine-tuned to resemble ARM CPUs.

- **O3**: an out-of-order CPU model. It features a full out-of-order pipeline that can independently tick each stage and schedule instructions to be executed on the different execution units at the same time, out-of-order. Similarly to *Minor*, it can be configured independently of the ISA. For example, the count of ReOrder Buffer (ROB) and internal registers for renaming can be tuned using *Params*. It also supports branch predictor defined as a *Param SimObject*, which allows the user to propose their own branch predictor (see section 3.2.1.2). ARM also proposes its special implementations: ex5_big. It imitates a Samsung Exynos 5 *"big"* CPU.

These CPU models are not monolithic *SimObjects* and contain sub-*SimObjects* that handle some functionalities. These sub-*SimObjects* can be selected independently to specify these behaviors. In addition to the branch predictor we mentioned about *O3*, all the CPU models have a dedicated *SimObject* for the MMU which is, by default, the one provided for the associated ISA. *gem5* also proposes power models, taking the thermal feedback loop into account. These models have been built using theoretical knowledge of CPU design. They are not a close representation of any real CPU. Furthermore, although these model can be customized, their results, especially their performance, are not representative of a specific real platform. These models have to be used knowing the effects researched.

We mainly rely on *gem5*'s ability to model cache behavior and its interaction with speculative execution. To model cache, *gem5* contains parametrizable simple cache models. By connecting them with CPUs inside a `SubSystem` *SimObject*, they can model multi-level cache systems. Inside this cache model, there are other sub-*SimObject* that handle functionalities and that can be selected independently:

- Replacement policies: which select which cache line should be evicted among all the possibilities.
- Cache tags: which describes where a cache line is stored and what are all the possible aliases for this cache line.
- Hardware prefetcher: which describes how the caches prefetch line (it can also be absent).

Basic cache properties (associativity, size, latency, etc) are defined using simple *Param* value. Another complex cache system is also provided by *gem5*: it is called *RUBY*. Using a specific language called *SLICC*, the cache-coherency protocol is described and compiled. *gem5* then produces *SimObject* representing not only cache, controller and interconnect. They can be assembled to create a full cache system interfaced with their own type of port. The *RUBY* cache system can then be connected with the normal port system between memory and CPUs.

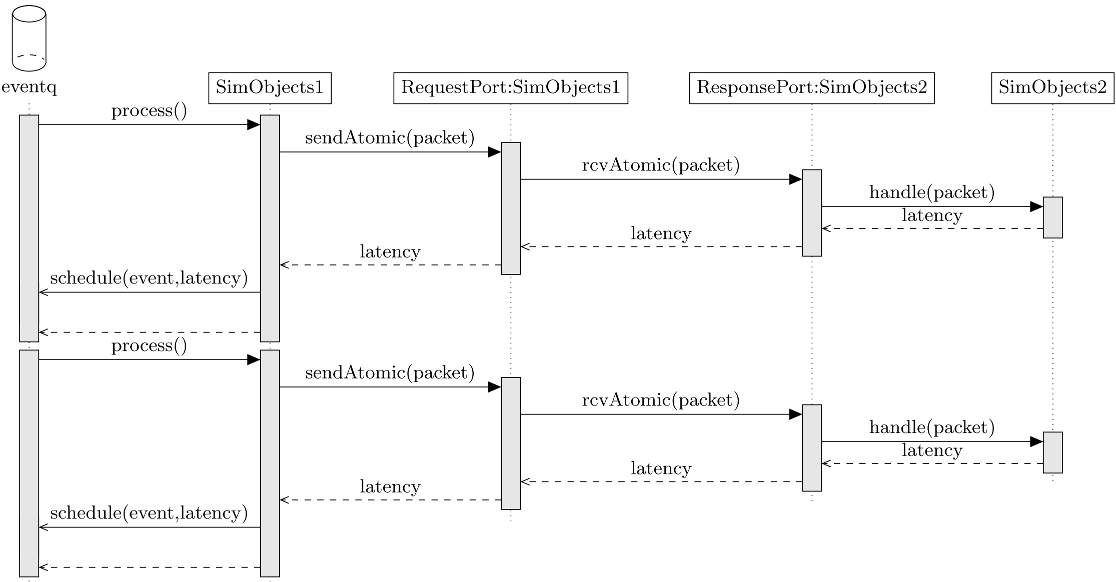


Figure 2.8: Atomic model: two memory transactions through ports between *SimObjects1* and *SimObjects2*. A *SimObjects1* event is processed, which requires sending a packet to *SimObjects2*.

2.2.2.4 Memory model

In *gem5*, the memory hierarchy is represented using port *Param*, we detailed in section 2.2.2.2. Memory *SimObjects*, like caches, memory-mapped devices, or membuses, come with ports which can be connected to represent a memory connection between two devices. During simulation, these ports pass packets representing memory transactions[Low24a]. However, the responses are not handled in the same way and are just provided to the requesters (CPU, cache, ...) by editing the packet and transforming it into a response packet. This response packet can, however, be transmitted to other *SimObject* in its "way" back to its original requester to account for cache coherency (e.g. snooping response packet). This memory protocol, and by extension the port *Params* associated with it, is called the **packet** protocol. There are two modes for handling timing:

- **Atomic** (figure 2.8): The communication between CPU and memory traveled once through caches with only one event scheduled. Latency is added along the way, with each device in the memory transaction adding its own latency. Then, the CPU decides how this latency is implemented (AtomicCPU, for example, postpones the next cycle if latency is bigger than a cycle length). On figure 2.8, we represented a simple atomic transaction between *SimObjects1* and *SimObjects2*.

- **Timing** (figure 2.9): At each step between two memory devices, some events are scheduled that take into account each device's added latency. So each device event handling function (`process()`) finishes when they have to communicate with another device. In that context, the receiving device has to schedule a new event to handle the rest of the transmission. This process continues until the transmission reaches its final destination, and then the response follows the same process but in reverse. On figure 2.9, we represented the different communication scheme between *SimObjects1* and *SimObjects2*. First, a successful packet transmission from *SimObjects1* to *SimObjects2*. *SimObjects2* schedules an event to respond later. The second transaction is a back-pressured transmission from *SimObjects1* to *SimObjects2*; *SimObjects1* wait until the *SimObjects2* authorized a retry. The third transaction is a response from *SimObjects2* to *SimObjects1* for the first transaction, which is then followed by a retry event, which will be in charge of warning *SimObjects1* that it can send the pending packet. Finally, the last transaction corresponds to the *SimObjects2* sending a retry request which is immediately followed by the *SimObjects1* sending its packet again and *SimObjects2* scheduling a respond event which will be similar to the third transaction.

There is a third mode called **Functional** that can be used in addition to other modes to only account for the functional effects of memory accesses. This mode generally behaves like an atomic access with no latency. The caches that use the packet protocol rely on snooping packets inside the crossbar that connects them. This snooping system represents a basic cache-coherency protocol. For example, it allows caches to share lines with other same-level caches.

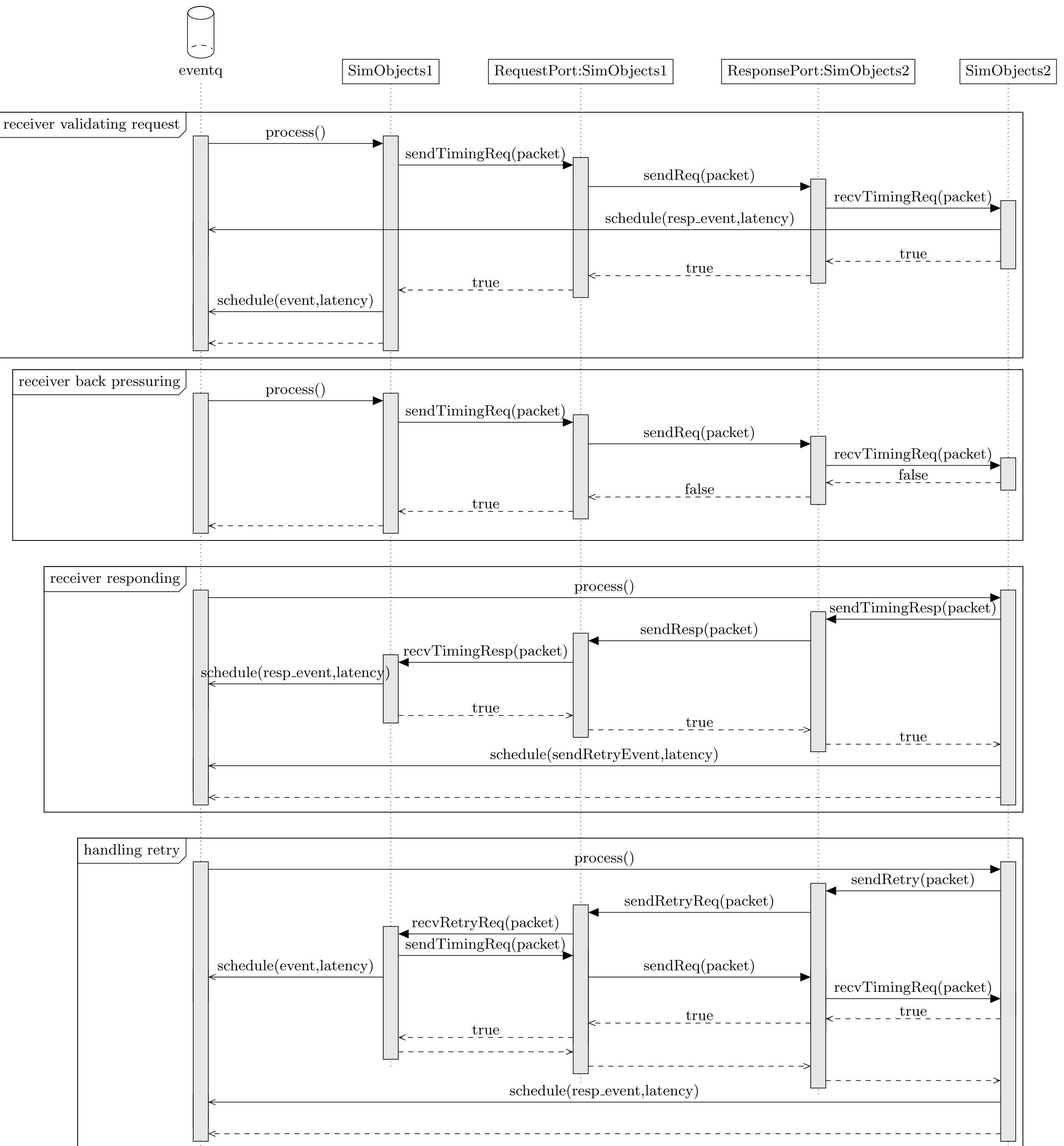


Figure 2.9: Timing model: two memory transactions through ports between *SimObjects1* and *SimObjects2*, the second one get back-pressured.

2.2.3 System element modeling

Full system workloads also contain interaction with other embedded devices. These devices, either memory-mapped or accessed through buses handled by other devices, mostly represent interfaces to the outside world for SoC (UART, GPU, disk, etc.). They also handle behaviors that are necessary for a system to function (timers, power controllers, etc.).

Indeed, *gem5* proposes a functional model for these devices in order to run their associated full system workloads. If we take, for example, the BCM2837[Pi22] from a Raspberry Pi 3B, we can identify how *gem5* would model each of its devices. It features :

- **UART0, UART1**: They allow connection to the Raspberry Pi using a serial port. For that *gem5*, contains multiple implementations of UART, they all connect to a `Terminal` Object which makes them accessible from a telnet

connection from the host running the simulation. Otherwise, they store everything printed to the terminal in a file in the *m5out* folder[1] .

- **Integrated memory**: They are integrated SRAM or ROM. *gem5* models them using `SimpleMemory` which can be preloaded with specific binary.
- **GPU** or other graphical devices: *gem5* proposes a MALI GPU from ARM implementation using the **nomali** library. It also features an ARM HDLcd controller. This is generally not used in *gem5*.
- **Interrupt controller** handles interrupts and can be configured to distribute them specifically if it is mostly a memory-mapped device on the BCM2837 (GiC), its main model in *gem5* is integrated into CPUs. CPUs models in *gem5* use an abstract interface for interrupt controllers. This way any ISA implemented in *gem5* can provide, as an ISA device, an implementation for its own interrupt controllers.
- **Timers**: They interface with the interrupt controller. In *gem5*, there are timer implementations that generally target a specific ISA, but they are implemented as normal devices. They interface with the interrupt controller in *gem5* using a `Pin` object which links them with a specific interrupt number.
- **USB**: Through a USB controller, devices can be accessed and communicate with a system. *gem5* can simulate a keyboard input, but it does not use USB and prefers to simulate a **PS/2** interface. Besides that, *gem5* does not model USB devices.
- **eMMC, SD card**: These interfaces are used to connect storage mediums. They are handled by an integrated controller. On the Raspberry Pi board, an SD card can be connected using an SDMMC controller. On *gem5*, disks can be connected using VirtIO devices(Virtio is a simplified interface for devices exposed by a hypervisor). *gem5* also supports IDE controllers as PCI devices to connect drives in a simulated IDE environment. *gem5* ARM implementation also provides a Universal Flash Storage (UFS) implementation through a memory-mapped controller.

  Similarly, on other systems, storage mediums can be connected using *SATA* or *NVMe*.
- **DMA controller**: This controls how devices access memory and allows programming transfer between different memory spaces. Some devices on *gem5* can have access to memory through DMA requests. These devices inherit from the `DmaDevice` class. They are, however, no specific controller that can monitor device access to memory.
- **I2C, SPI0, SPI1, SPI2, Pci-express**: using integrated controller the BCM2837 can handle these different type of buses. They can connect to memory, sensors, probes, storage, actuators, etc. *gem5* only models I2C and PCI controllers and, besides the aforementioned IDE controller, *gem5* only models: A copy engine, an AMD GPU, and an ethernet controller (no I2C devices are modeled).
- **PCM/I2S**: IšS is used to connect a system to audio devices by sending Pulse-Code Modulation data (PCM). This information can be used to play sound in a remote digital-to-analog converter. Audio-wie, *gem5* only proposes a `PcSpeaker` model, which does not account for audio output.
- **GPIO** & **PWM**: To connect more directly to sensors and actuators, General-Purpose IO and Pulse Wave Modulation can be used on configurable SoC physical pins. GPIO and PWM are not modeled by *gem5*.
- **PowerManagementUnit** & **Dynamic Voltage and Frequency Scaler**: they are used to monitor the power envelope of the SoC and adapt frequency and voltage to avoid overheating. In *gem5* a power model can be connected to CPU and GPU models to account for the relation between used processor cycles and thermal power dissipated. By connecting power models to a thermal model, *gem5* can simulate SoC temperature. Conjointly with a DVFS controller model, *gem5* can model DVFS retro-action on frequency and voltage.

In *gem5*, all of these devices are children of `Platform` *SimObject* which represents the type of hardware platform that *gem5* currently models. For X86, it is called `PC` . To run a FullSystem workload, we have to verify that it is compatible with the platform *SimObject* implemented in *gem5*. If not, we have to create our own platform, which may require our own devices.

[1] we explain more about this gem5 output folder in section 3.2.2.

2.2.4 *gem5* for security in literature

Several usages of *gem5* for security evaluation can be found in the literature. They can be divided into hardware and software categories. Firstly, *gem5* has been used to identify software vulnerabilities related to micro-architecture, such as cache timing. [Wu+18] presents such an approach. CacheD [Wan+17] is another static analysis tool for software cache information leakage, which uses *gem5* simulation to validate the results.

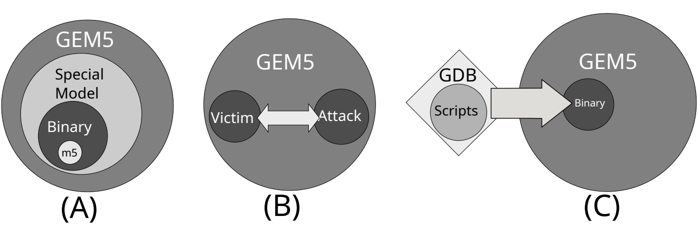


Figure 2.10: methodology comparison between article; Ours is *(C)*

For hardware vulnerabilities, [BY19] demonstrates a denial-of-service attack targeting writeback cache buffers, on *gem5*. [KDG20] demonstrates a covert channel using *gem5*. [Yan+19] reproduces *Spectre* attack in *gem5* simulator and proposes a countermeasure by adding a speculative buffer, changing the coherence protocol, and the D-TLB operation. Similarly, [FFY19] proposes a hardware/-software combined countermeasure, by modifying the out-of-order processor model and the Linux kernel to map non-speculative pages. [AJ20] is another countermeasure for *Spectre* demonstrated on *gem5*. [Wer+19] evaluates cache set randomization as a countermeasure for cache attacks on *gem5*. [Yu+19] is a work based on *gem5*, which explores the effect of various cache parameters and policies on information leakage.

There are two typical way to use *gem5* in the literature (figure 2.10):

(A) uses *gem5* as a special simulator by adding function or using m5 instructions to monitor directly the binary

(B) uses *gem5* as a way to simulate an attack, sometimes with a countermeasure implemented in *gem5*.

This thesis proposes a third method:*(C)*. In this method, monitoring is offloaded outside *gem5*, using it as a simulator (like *B*) while monitoring execution (like *A*).

2.3 Security in embedded systems

Compared to simpler micro-controllers, modern SoCs can run Operating Systems (OSs) and thus feature internal mechanisms to enforce process isolation. Without being necessary multi-tenant, a modern embedded system can run code provided by diverse actors, some being potentially nefarious or compromised.

In this section, we cover the common security features in a SoC. These are the tools that create isolation layers between programs/operating systems and secure monitors.

2.3.1 Operating system security

Tools to build an Operating System (OS) are integrated in most modern ISAs. These tools allow isolating user applications from the kernel through the implementation of virtual memory. This possibility is the main difference between ARM-M (ARMv8-M), which only allows baremetal programming, and ARM-A, which features a **Memory Management Unit**. For each application, the MMU creates a virtual memory space with its own *page table base*. An application can only access what is in its virtual memory. *page table base* points to an array of pages indexed by virtual addresses and pointing to physical pages. Each page can be either a last-level entry (with different page sizes depending on the level 4kB, 2MB, 1GB) or point to sub-page tables. The page table is explored either manually by the OS, which registers the translation in the MMU, or by a hardware page walker like on ARMv7-A and v8-A, which only requires OS intervention if page walking fails. The pagetable base is held in a dedicated MMU register, which is swapped when the OS switches between applications (applications page tables can have pages in common).

As access to MMU registers would allow an application to have access to all the memory space (including memory-mapped devices), the operating system keeps control over the MMU. It keeps its own pagetable in a dedicated MMU register. This is done through the execution privilege system. This page is only active when the OS takes over execution through interruptions (IRQ) and is disabled when switching back to unprivileged applications. As shared libraries are often made accessible to applications through shared page entries, MMU needs to enforce strict read-only access to their pages. This is done through **read-only** flags present on page table entries. In addition, to protect applications from buffer-overflow attacks[One96], the MMU also implements a strict NX flag (No eXecute) in pagetable entries. This flag prevents an application from executing data placed in an NX-flagged page. The OS prevents all pages from having none of the read-only or NX flags, implying that a page should always be either read-only or not executable.

However, buffer overflows are still possible with NX bit using execution gadget[One96] already present in executable pages. With these gadgets, an attacker-controlled executable page can be created in a victim application memory. OS made these types of attacks impractical by randomizing virtual memory layouts with Address Space Layout Randomization (ASLR)[NZ19]. Generally, it is not possible for a process to access any information about its own page table mappings.

As last-level CPU caches are generally physically indexed, this can represent an issue when trying to control cache content from a process with minimum privileges. In fact, on ARMv8-A, all the caches are *Physically Indexed, Physically Tagged* (PIPT) as the MMU operates before caches. Above the OS privilege level, there exist higher execution privileges, represented on figure 2.11. They also use the MMU for their own isolation:

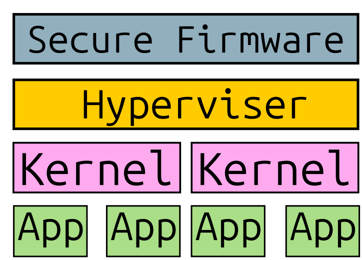


Figure 2.11: Execution privilege: using MMU isolation, multiple levels of application can run on a single system.

- **An hypervisor** supervises hardware isolation for multiple OS. With a new stage of MMU address translation, it gives each OS a virtual memory space that they can redistribute using their usual MMU address translation. This is reserved for server environments and is usually not present on embedded systems.
- **A secure firmware**: above hypervisor, it supervises system security features. It uses these security features to control workload validity at boot time. It tightly locks these functionalities to prevent any tampering from a potentially compromised OS.

2.3.2 Attack scenarios

The committee on National Security Systems[DUK15] defines an *attack* as *any kind of malicious activity that attempts to collect, disrupt, deny, degrade, or destroy information system resources or the information itself.* An attack can be an attempt to access data, functions, or other restricted areas of the system without authorization. An attacker can also monopolize or destroy the resources they were supposed to share with other users, typically in the case of a denial-of-service attack [BY22]. Indeed, an attacker can use a variety of ways to reach their goal: physical attacks, remote attacks, etc... [PMB15]

However, as we mainly focus on embedded systems, we discuss attack scenarios that are related to those. As shown in figure 2.12, typical attack scenarios against embedded systems, imply:

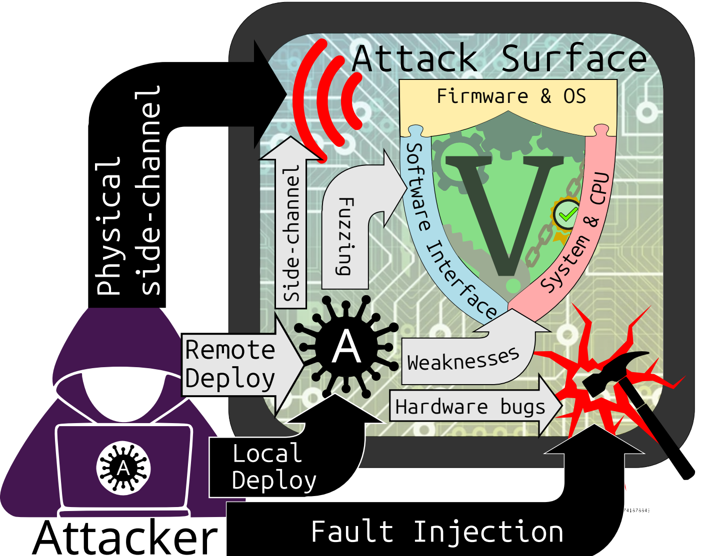


Figure 2.12: Embedded attack scenarios: An attacker tries to attack a program running in an embedded system (**V**). It can run an attack program (**A**) on the target or use a physical medium (black arrows) to attack the victim (**V**) in order to bypass its interfaces (puzzle pieces around the victim). These constitute the victim attack surface.

- A secret or functionality held by an application, the OS, or any system element that will be targeted by the attack. The target of the attack is called the **victim**. Generally, there are specific use cases circumvented by the attack: Some applications normally have access to this secret or functionality, but the attacker should not. A typical secret is a cryptographic key. This victim is protected from intruders using different system functionalities working in tandem to create an interface isolating the victim's secret: the victim *software interface* is programmed using the *firmware and OS* primitive implemented using the *CPU and system* functionalities. This green shielded V is used throughout the thesis to represent this idea of victim program.
- The attacker can have physical (black arrow) or remote access (white arrow) to the victim system. In our scenario, an attacker has execution rights but not at the highest privilege. The attacker can run an **attack** on the target system, but some system functionalities are not accessible to it. We represent this attack program as a black virus, like on figure 2.12. For physical attacks, the attacker may not even be able to run programs on the target system using physical side-channels.

- The attacker uses the direct or indirect interfaces it has with the victim. It circumvents their intended behavior to gain access to the secret or functionality it seeks. These interfaces from which the victim can be attacked are called **attack surfaces** or attack vectors. To explore interface limits, security researchers use *Fuzzing*[Dua+23] to detect security-compromising behaviors of incorrect requests. The attacker can also leverage hardware bugs or fault injection to bypass software-implemented behavior, as shown on figure 2.12. Finally, the attacker can use side-effect produced by the victim to recover its secrets. Represented by red semicircles on figure 2.12, these side-effects can be measured by an attack running in the target system or by performing physical measures on the target system.

Enumerating what the attacker can do and to what interfaces it can have access to is important to delimit and prioritize threats. Indeed, this guides the countermeasures to implement. Of course, an omnipotent attacker can generally bypass any protection. So, studying the security of an application starts by defining a *threat model*. A threat model proposes a model of an attacker and lists what functionalities and interfaces it can access.

For the attack scenario in this thesis, the attacker is assumed to be able to execute on the victim system:

- Natively or as an application.
- In a browser or any interpreter.

2.3.3 Execution Privileges

The OS uses privilege systems at the hardware level that restrict access to hardware resources (like the MMU), to isolate privilege levels. These privilege levels are represented as different colored circled on figure 2.13. Therefore, compromising the OS is a simple way to access information held by any application running below it. Indeed, OS are complex software elements with multiple components, e.g., kernel module on Linux. These modules can be loaded at run times and can be developed by different sources. As these components run at the same privileged level as the kernel, compromising any of these components gives access is enough to gain control over the OS. It widens all the possible vectors that can be used to compromise an OS's security. These attacks are called **privilege escalation**[Son+06]. The two black arrows on figure 2.13 represent two possible vectors for an attacker program: circumventing the hardware mechanism or finding vulnerabilities in the access API. For embedded devices, the root user is generally locked by the Original Equipment Manufacturer (OEM). As a matter of fact, gaining access to this user gives full control over the kernel: Devices for which this mode has been unlocked for the end-user are called *root'ed*. For Android smartphones and tablets, this is a frequent threat[Dav+11][Ran+14]. In these devices, an application that uses this vulnerability could take over the OS and spy on any Android apps. On figure 2.13, we also represented functionalities (IRQ, MMU, scheduler,...) that are controlled by each privilege level and to which a privilege escalation can give access. Operating systems on embedded devices are also attacked through the physical media storing them: the bootrom. In fact, bootrom contains multiple elements:

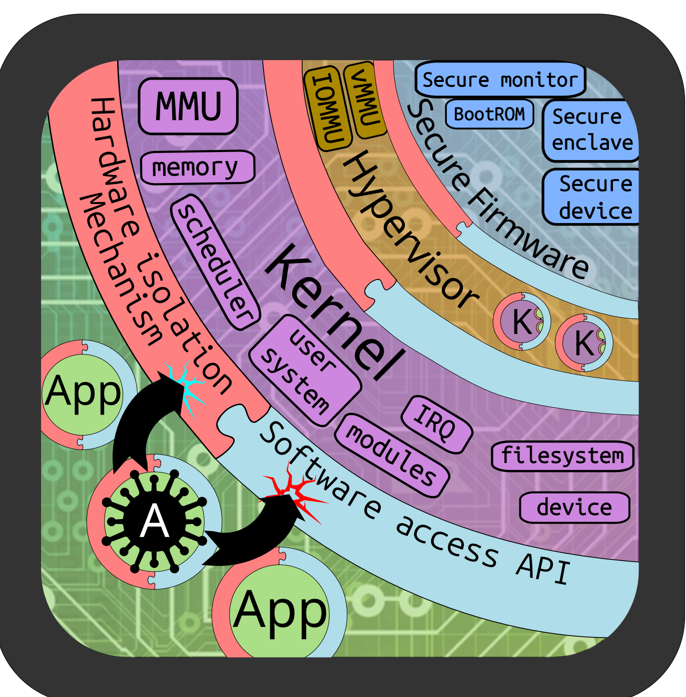


Figure 2.13: Execution privileges: Different execution privileges control access to system elements. Each creates isolation layers using hardware mechanisms. The tenant software in higher privilege levels can grant access to their elements using an API that restricts what interactions are allowed. An attacker, wanting to have unrestricted access to a system element, will try to bypass hardware mechanisms or find vulnerabilities in the access API.

- A firmware that handles basic hardware functionalities. It is where execution starts when booting.
- A bootloader that can contain multiple steps and that can load OS from filesystems.
- An operating system which can be updated independently of the rest of the bootrom.

To prevent any tampering with the bootrom, the earliest stages of the firmware, or the CPU itself, verify the bootloader integrity and validity[And19]. To protect this stage against attacks, it executes at a higher privileged level than the OS (and hypervisor). This privilege level is called **secure firmware** or **secure monitor**. Although the secure monitor is responsible for the first bootloader stage, it can remain active after the OS boot and safeguard functionalities relative to system integrity: Like CPU wake-up handler and bootrom update mechanism. To ensure that system updates cannot be reverted (to restore a patched out security vulnerability) the secure monitor often implements **rollback protection**[AIM22]. Integrated fuse-boxes in the SoCs are typically used to implement rollback protection. The attacks thwarting rollback protection, or lack thereof, are called **downgrade attacks**[Che+17]. Some of these hardware and software features, meant to be used by the secure monitor, are represented on figure 2.13, in the upper rightmost circle.

2.3.4 Trusted Execution Environment

To share the protection provided by the secure monitor in order to mitigate the effect of a privileged escalation attack at the OS-level, Trusted Execution Environments allow user-level or OS-level applications to perform operations in a more secure environment. Inside TEEs, these operations are handled by **Trusted Applications (TAs)** that can be user-developed or provided by the manufacturer. In some cases, only the former can develop applications which are then validated by the TEE with a signature system.

As we mentioned in the introduction, TEEs generally enforce three properties for these applications :

- **Integrity**
- **Confidentiality**
- **Authentication**

Integrity is generally enforced using an application packaging mechanism that encrypts the application to prevent tampering using the TEE internal key. The application is only decrypted inside the TEE. The privilege of signing applications for the TEE can be reserved for the OEM to improve security.

Confidentiality is enforced using a separate execution environment. Indeed, the TEE is isolated from the **Rich Execution Environments (REE)**, typically Linux. This separation relies on either a specific ISA-level execution mode or on physical separation using a different CPU, creating a secure enclave. With this enclave mechanism, they split the execution world in 2:

- The **normal world** or *unsecure* world, where the classical operating system and user application runs. The OS running in the *normal world* is called *rich OS* and generally is responsible for communication with the *secure world*.
- The **secure world** is where trusted applications runs. This world often has its own operating system called a **trusted OS** or **secure OS**.

To set up the *secure world* (initializing CPU and booting the trusted OS), the system generally relies on a secure boot mechanism that, when the device is powered on, verifies the original bootrom (BIOS, UEFI, etc.) and configure the TEE. This way, the OS is not responsible for the TEE initialization. As it is the bootrom that handles *secure world* initialization and also **secure-boot** it is considered as the **Root-of-Trust** for the system. This makes it the target of simulation attacks, cold-boot attacks, and downgrade attacks [Che+17]. To counter these attacks, a verification mechanism is sometimes incorporated in the SoC to verify the first stage of the boot: i.e., the bootrom.

TEE security is enforced by having the boundaries between the *secure world* and the *normal world* tightly closed through the use of:

- Different memory hierarchies (cache and RAM).
- Specific secure labeling for memory transaction: thus the caches and the RAM can physically separate memory space.
- Minimalist interfaces.
- Flushing caches when transitioning between REE and TEE if they share the same CPU.

With all these mechanisms, the attack surface is kept at a minimum for TEE and the TAs running in it.

2.4 Micro-architectural attacks

As we have seen in the section 2.3, system designers and OS architects use ISA elements to provide isolation between processes. As a result, attack surfaces for a victim module or application are supposed to be restrained to the interfaces it can have with an attacker. In this context, micro-architectural attacks allow to circumvent the intended interfaces, by using properties of the architecture. Shared medium, transient behavior, assumptions on valid programs, physical effects, etc., can be leveraged to detect data and/or hinder execution.

2.4.1 Side-channel attacks

Side-channel attacks rely on side effects due to computations using a secret, illustrated on figure 2.14. Detecting these side effects allows an attacker to reconstruct the secret. Side effects can be detected either using a shared resource between victims and attackers(e.g., caches) or using hardware probes (power metrics, frequency, etc.). As represented in figure 2.14, we symbolize throughout this thesis operations and functions as gears and secret as keys. Performing these operations and accessing the keys monopolizes unique resources marked as small squares. Sometimes, the victim resources footprint is unique to certain operations and can be used to detect which are performed or some properties of the secrets. Indeed, with the unique resources being monopolized, an attacker program will suffer performance loss if it tries to use the very same resources. As we illustrated on figure 2.14, the attacker can detect these changes, detect victims' operations, and extract critical information from them. On the other hand, physical resource usage can also have physical side effects (ex: electromagnetic emissions), which can be detected to determine a usage footprint. As we illustrated on figure 2.12, this medium can be used by an attacker program or by a physical attacker.

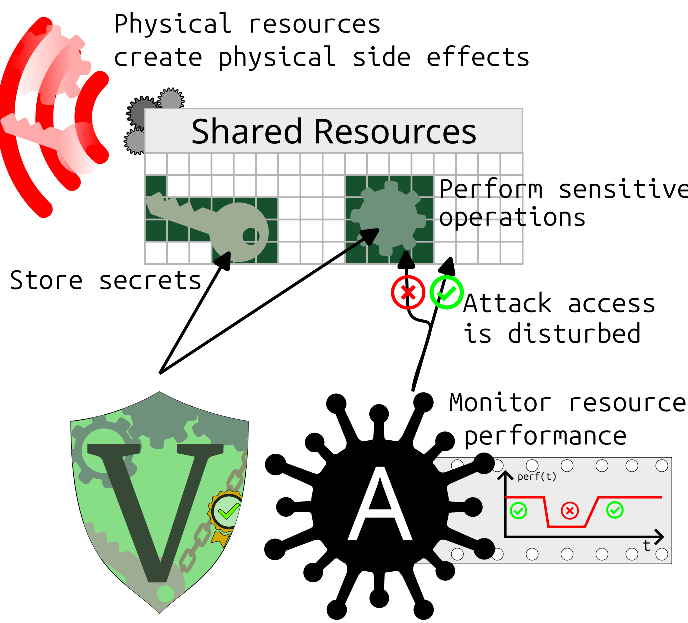


Figure 2.14: Representation of a side-channel: an attack program and a victim program use the same shared resource.

2.4.1.1 Cache timing attacks

Caches' side channel relies on victims and attackers sharing the same CPU caches. It can be any level of cache:

- L1I/L1D: applications sharing the same core through multi-tasking.
- Last level caches: applications that execute in parallel (cross-core cache attacks)

Cache-timing attacks use cache side effects caused by the victim's computation to gain information about it. They take advantage of the difference between having a piece of data in cache, a cache *hit* and not having it in cache a *miss*, to detect the cache side-effect. At the hardware level, it means that an instruction that interacts with data from memory, can take a different amount of time, depending on cache states. For example, a *load* instruction takes less time on a cache *hit* than on a cache *miss*. Other instructions, like cache-maintenance instructions, can also be used [Gru+16a]. By measuring the time that a specific instruction takes, it is possible to know some information about the cache state. As the cache is shared between processes and privilege levels, its state can leak information about other processes. At any time, cache lines from different processes and from the kernel are present in the caches. To set up an ideal situation for attack, the cache is placed in a control space by the attacker.

The two typical cache attacks: *Flush+Reload*[YF14], *Prime+Probe*[Liu+15], have two opposite cache set up. We compare then in figure 2.15:

- **Flush+reload**: the attacker removes a line that it knows the victim will access. It can then use cache-timing to determine if the victim accessed it.
- **Prime+Probe**: The attacker fills all possible slots in a cache for a specific line using a set of lines called *prime set*. It can then check if one of the lines in the prime set is missing, using cache timing. It indicates that the victim accessed the targeted line.

Of course, these attacks rely on the victim being the only other thread to conflict with the attacker. So, in practical situations, it may require multiple measures. These measures of cache-timing/activity are called *traces*.

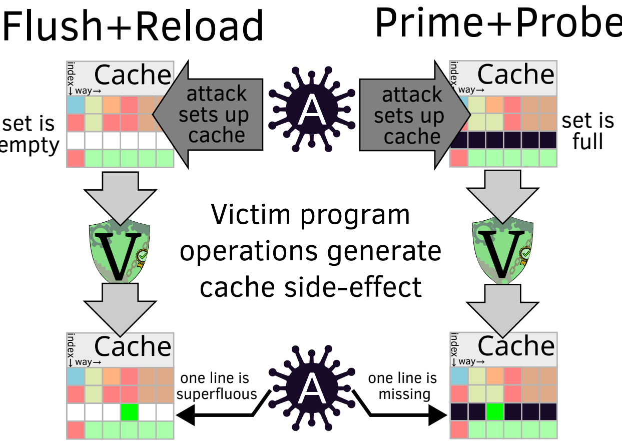


Figure 2.15: Comparison between *Flush+Reload* and *Prime+Probe*: We can see they exploit different cache set-up

When implementations are weak to cache timing attacks, they show differences in memory accesses that are influenced by a secret in the algorithm. Figure 2.16 lists different sources of memory accesses that can result in a cache line being allocated in the l2 unified cache. Each type of leak can be linked to specific operations (represented as gears) or secret values (represented as keys) to determine a cache footprint. The CPU fetching multiple instruction lines, also known as a *wide fetch*, is typically something that must be taken into account to determine a cache footprint for a specific operation. Depending on the length of the time window during which these differences can be detected, it can be possible to carry on a cache timing attack from an interpreted environment like a web browser[Ore+15]. *ARMageddon*[Lip+16] is a survey documenting cache attacks on mobile devices. It mentioned various variations on the typical *Prime+Probe* and *Flush+Reload*: *Flush+Flush*[Gru+16a], *Evict+Reload* [GSM15].

2.4.1.2 Higher-level cache attacks

Cache-timing attacks can also be used at a higher level: these attacks target the victim as a whole without specifically trying to detect directly an elementary computation in the victim. Examples of higher level cache attacks are:

- Cache template attack[GSM15]: by studying how *Prime+Probe* traces evolve depending on the secret, a cache template attack proposes *points of interest* in traces that can be used to determine the value of each bit of the key.
- *Evict+Time* [OST06]: By evicting different cache sets or cache lines, and measuring the execution time of a cryptographic process, an attacker can retrieve some bits from the key.

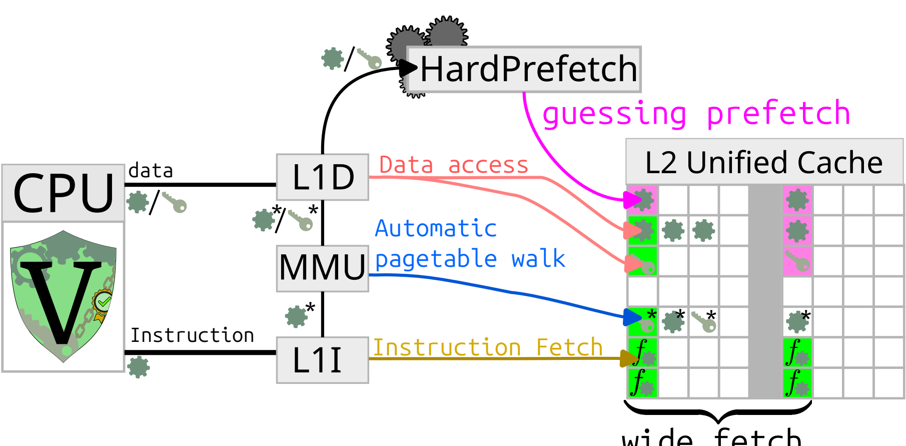


Figure 2.16: A weak implementation can leak information in the cache through a variety of means: data access, instruction fetch, MMU table walk, or hardware prefetcher.

- Fingerprinting cache. [Shu+21]: Using a *Prime+Probe* traces for certain cache sets, it is possible to correctly identify specific activities, in that case, which websites are currently loading in a browser tab.

These attacks may also use machine learning to reconstruct a secret from the full traces. In this situation, supervised learning is possible using traces labeled with the secret in order to train a model to reconstruct the secret for an unlabeled trace[Per+21].

2.4.1.3 Static and dynamic cache analyzer

Cache analyzers are tools that are used to scan for potentially vulnerable code for patterns that leak secrets in the way they access memory, explicitly (*load* instruction, etc.) and implicitly (instruction fetch, etc.). We compared how cache-analyzers from [Gei+23] model CPUs and caches with *gem5* 's model. These tools can scan the code statically or analyze how it behaves when the application is running:

- **Static cache analyzer**: *CacheAudit* [Doy+15] is an example of static analyzer. It uses its own model for the CPU that can fail to predict leaks caused by the architecture (such as prefetching or automatic table walking). Because we use *gem5* to model our CPU, we have an accurate enough model that represents these types of phenomena. On the contrary, we need to run an application to study how it uses cache, and we also need our execution trajectory to be representative of what can happen. Static analyzers, like *CacheAudit* because they analyze dependency directly in the code, are able to see theoretical leaks that dynamic analyzers will not see because they rarely happen. Well-known static cache analyzers, *ct-verif* [Alm+16],*Binsec/Rel*[DBR20], or *CaSym*[Bro+19] share the same downside when compared to a *gem5* simulation.

- **Dynamic cache analyzer**: Dynamic cache analyzers, like [Bos+16], use incomplete CPU/cache models that do not take into account some modern CPU features like branch prediction. Also, the fact that we use *gem5* allows us to directly test the effect of the countermeasures that are implemented with it. On the other hand, *gem5* is a vastly more complicated model that may be harder to debug and analyze. Some dynamic cache analyzers, like *CacheD*[Wan+17] or *"Dude, is my code constant time?"*[RBV17], do not include an execution model and directly work on execution traces to detect leakage due to control flow or secret-dependent memory traffic. Other dynamic analyzers, like *ABSynthe*[Gra+20] propose their own CPU model with specific focus,

2.4.1.4 Other side-channel attacks

Side-channel attacks can be generalized to any internal buffer or side-effect [Lav+21]. If a side-effect of any nature is correlated with a specific operation or value and detectable from an attacker standpoint, it can be used to build a side-channel attack. A weaker version of these attacks is **covert channels** attacks in which two isolated processes collaborate to pass data using hardware side effects. For cache, [Mau+17] uses *Prime+Probe* to exchange information between a sender and a receiver process. Examples of hardware side-effects that can be used to extract data are:

- Power side-channel and physical side-channel: Attackers use power consumption to detect which operation is currently being processed in a CPU or in dedicated hardware. [ZS18] uses power consumption, to perform a power analysis attack against a RSA crypto module in an FPGA.

- Dynamic Voltage and Frequency Scaler (DVFS) can also be used to leak information using frequency or voltage switching in response to a workload modification. It is used as a covert channel in[Ala+17].

- Instruction micro-op buffer, which is integrated into the CPU decoder. An attacker can use the instruction decode delay to detect if a similar instruction was decoded recently[Ren+21].

For Archisec, Bossuet, Grosso, and Lara-Nino have shown that power can be used as covert channels in modern SoCs like the Zynq UltraScale[BL23a] and that *gem5* can be used to simulate power side-channel attacks[BGL23].

2.4.2 Transient execution attacks

Most modern processors feature a speculative execution system to improve performance. The CPU can execute instructions speculatively, and if its assumption is shown wrong, speculatively executed instructions are not committed to memory, and all the ISA-level effects (registers, PC state,...) are reverted. However, the side effects of these instructions on the microarchitecture (buffer, caches, etc..) are not reverted. Transient execution attacks use this speculative execution process to execute code and create a pattern that causes deterministic side effects on the architecture. The most famous transient execution attack are *Meltdown*: [Lip+18] and *Spectre*:[Koc+19]

spectre.c

```c
struct_t kernel_level_function(int user_ind){
    if(user_ind<SIZE){
        struct_index kern_index=user_ind_to_kernel_ind[user_ind];
        /* if user_ind is illegal, we can access the full kernel memory*/
        return struct_tab[kern_index];//loading real data using something that can be a secret;
    }else{
        return NULL;
    }
}
```

Figure 2.18: Typical *Spectre* Gadget

Meltdown allows an instruction that would cause a *Fault*, typically a *Data Abort*(or *Page Fault*) to proceed speculatively. Although the instructions are reverted, as we mentioned above, the side-channel effects on the cache are not. Thus, an attacker can perform an illegal memory access (outside its own memory) to load a secret and then directly load from memory at an address offset'ed by this secret. This creates a cache-side effect directly function of the secret that will not be reverted when the CPU realizes that a data abort has happened, and it triggers an interrupt. This cache side-effect can then be detected using a cache-timing attack. This way, the attacker can recover the secret by determining which address offset was used to cause the side-effect. *Spectre* types attack relies on a specific fragment of code in the victim called **a gadget** (see figure 2.18). These gadgets are necessary to create secret-dependent memory access, which will then be detected by the attack. With *Spectre*, a branch in the gadget is incorrectly taken and causes erroneous access (see figure 2.17). This can be forced reliably by poisoning the branch predictor to force it to make the desired incorrect prediction. The gadget on figure 2.18 can leak elements of the kernel memory by poisoning the branch predictor to bypass the `if(user_ind<SIZE)` check and proceeds in spite of an incorrect `user_ind`. By carefully choosing this incorrect `user_ind` data from the secret zone shown on figure 2.17, is used as an offset and allocates an offset-dependent cache line. Although the execution will be rollbacked by the CPU, the allocated cache line remains which can be detected by the attacker using *Prime+Probe* as shown in figure 2.17. This access causes a cache-side channel which can be detected. If this gadget load depends on secret data, which is possible even if the code checked for out-of-bound accesses, as this happens in a speculative execution, the side-effect caused will depend on said secret. Like *Meltdown*, this effect can be used to leak a secret from a victim. Since *Spectre* and *Meltdown*, a wide variety of transient execution attacks have been discovered: [Can+19] keeps a list of *Spectre* and *meltdown*-derived attacks.

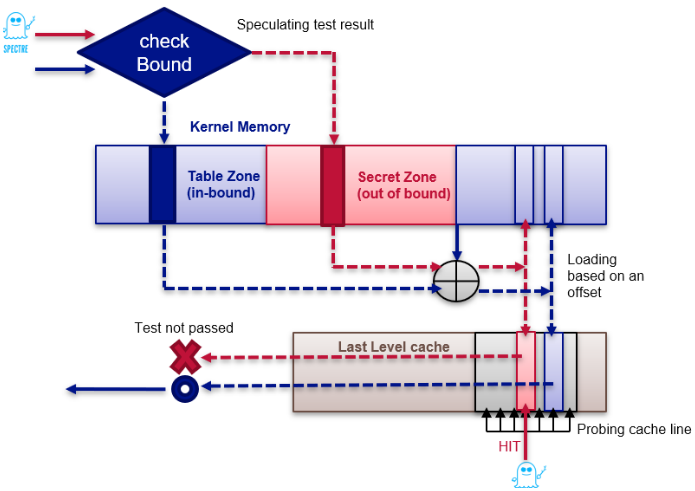


Figure 2.17: *Spectre* attack representation

For Archisec,Ayoub and Maurice demonstrated the possibilities and benefits of using *gem5* to model *Spectre*-like attack in [AM21].

2.4.3 Fault injection attack

As CPUs are physical systems, it is possible to disturb their inner workings by causing physical fault inside them. These faults manifest themselves generally as transient bitflips. Correctly used in an attack, these bitflips can cascade into a security bypass[Bak+22]: corrupting memory resources, altering results, skipping instructions, etc.

2.4.3.1 Typical fault injections

To inject fault in a CPU, different medium can be used [Bak+22]:

- Laser, ion, or electromagnetic radiation-based. With a focused beam, it is possible to localize fault up to a single transistor[Anc+17]. This could be used to corrupt the content of a RAM or EEPROM[Anc+17]. With less focus

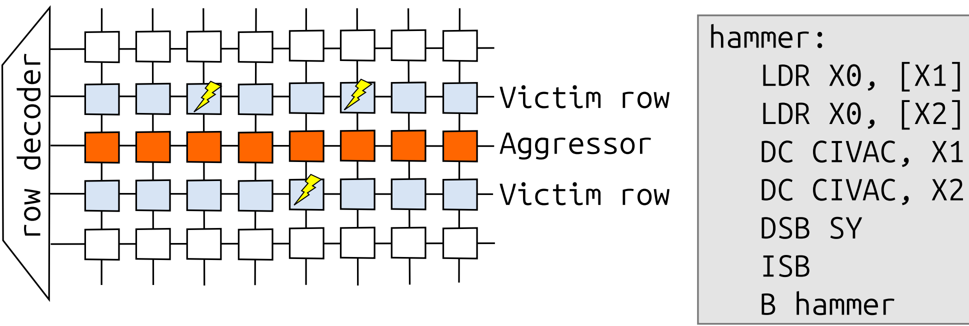


Figure 2.19: *RowHammer* effect and DRAM structure

energy beam, using radio waves, it is possible to cause glitches [Mor+13] in the execution of an ARM Cortex-M3 processor.

- Power or clock-based: using *glitch* or underfeeding can cause transient faults. For example, [Dob+18] uses clock-glitch to attack an AES co-processor, while [OFl16] uses power-glitch to attack an AVR 8-bit controller and a FPGA.

These fault injection attacks require physical access to the hardware to be performed.

2.4.3.2 Hadware memory corruption: *RowHammer*

RowHammer[Kim+14] is a memory corruption bug present in DRAM that manifests itself as bitflip. It is caused by adjacent rows influencing the capacitor charge, making it deep below the valid voltage level (see figure 2.19). As it is caused by memory accesses, it can be used remotely and does not require physical access. It is possible for an attacker to reliably trigger this bug by forcing the memory controller to repeatedly access the same row, this can be done in multiple ways (figure 2.19):

- Using cache flush [Kim+14] as represented on code extract in figure 2.19.
- Using uncached memory [Van+16].
- Using cache eviction [GMM16].

By choosing specific data to target specific rows, the attacker can control which row the bitflip happens in. This bitflip creates a memory corruption, which can then be used in a variety of ways:

- [SD15] uses the *RowHammer* to corrupt the page table in order to alter the attacker's pages so it can access kernel memory space, creating a privilege escalation for the attacker.
- [Car17] uses the *RowHammer* to create a bitflip the key to then uses the incorrect behavior of the RSA algorithm to reconstruct the key using the encrypted data: This is the *Bellcore* attack.[BDL97]

For Archisec, France et al. implemented the *RowHammer* effect in *gem5*[Fra+22; Fra+21a]. France et al. uses said model in *gem5* to assess vulnerabilities using machine learning. This model is always integrated into our *gem5*, but it is rarely used in this research.

2.4.4 Accelerator attacks

Accelerators, as their name suggests, are specific processors that can be programmed to accelerate tasks. GPU and FPGA are typically used in embedded applications. However, an attacker can use them to target a victim running in the main CPU or to attack another component of the main SoC. Accelerator attacks are harder to generalize as accelerators can be widely different. Accelerator attack can also use the same type of attack we mentioned before but executed from the FPGA: Cache-timing (through the ACP port [ARM17a]), *RowHammer* [Wei+19], etc. Some examples of accelerator attacks are:

- [Lad+13] and [VPI15] use GPU to attack applications running in the main CPU. They have a malware program running in the GPU to spy on user activity.

- [Jac+17] uses a malicious IP integrated into an FPGA, which overrides memory sections. This trojan IP can then tamper with the system update verification by replacing a public key used to verify the system.

For Archisec, Fellah-Touta, Bossuet, and Lara-Nino[FBL23] uses FPGA to attack an AES application using the power domain they share with the main CPU core.

2.4.5 Trusted Execution Environment attack in literature

Trusted Execution Environments have smaller attack surfaces than operating systems, Although they still are weak to classical attacks[Ben17][Che+17] mentioned in section 2.3, they have become increasingly more resistant to them. Indeed, they are generally attacked using micro-architectural attacks or other types of hardware attacks, bypassing the constraints of their interfaces.

Some examples of micro-architectural attacks against TEE are:

- [Rya19] uses cache-timing attacks and *Spectre* attacks to analyze cache traces from L1D and BTB to attack ECDSA[JMV01] in Qualcomm TEE. This relies on the TEE and the attack sharing the same CPU at different time slices. It uses interrupts to force the secure monitor and the TEE to transfer execution to the attack in the *normal world* (without any cache flush).

- [LW18] uses a *Prime+Probe* and *Flush+Reload* to attack Samsung TrustZone Keymaster (in Trustonic's Kinibi Secure OS). They reverse-engineered the Galaxy S6 bootrom to study the AES implementation in the Keymaster truslet. It relies on the attack running in the same time-sliced CPU and the shared memory between *normal* and *secure world*.

- [BBA19] demonstrates how FPGA can be used to attack TrustZone. Using specifically designed IPs, it is possible to access trusted memory spaces or devices by tampering with AXI interconnects. [Gro+22] uses a similar principle to break secure boot and TA authentication.

- [Car17] uses *RowHammer* to attack Trusty RSA implementation using corruption in the *key* to trigger a Bellcore attack [BDL97].

- [Kou+21] and [KOU+23] attacks the RSA implementation in the *mbedTLS* library incorporated into a trusted application. [Kou+21] developed a novel attack, flush-evict, using ARM CCI (Cache Interconnect Interface) performance metrics. Because the CCI reports the eviction count, it can be used as a probe to detect line conflict caused by the loading of prime sets. [KOU+23] elaborate on the weakness in RSA implementation in cryptographic libraries that are leveraged by TEE attacks.

These attacks rely on weak implementations of the TEE and are not possible on OP-TEE[YL20], our open-source TEE, except [Car17] and [Kou+21; KOU+23] which ignore or disable OP-TEE protections.

2.5 Conclusion

We have seen what are micro-architectural attacks and how they have been used in literature. Specifically, we showed that such attacks have been demonstrated against Trusted Execution Environments. However, we see that there is a gap between how they have been demonstrated and how they can be practically used. As there exists *uncharted lands at the junction between TEE, attack, and simulation*, we suggest that we could leverage simulation to study TEE more efficiently. In the following part, we will present our prospects for building such a simulation platform.

Chapter 3

Virtual platform on *gem5* for ARMv8-A, TrustZone, and OP-TEE

Contents

3.1 Introduction **34**
3.2 Platform and instrumentation with gem5 . . . . **34**
3.2.1 Building a platform in *gem5* 34
3.2.1.1 Writing a new config file 35
3.2.1.2 Adding new *SimObject* 35
3.2.2 Classical instrumentation on *gem5* 37
3.2.2.1 *DebugFlag* 37
3.2.2.2 *m5* instructions 38
3.2.2.3 *CxxMethod* 38
3.2.3 Our improvement to *GDB* in *gem5* 39
3.2.3.1 *GDB* monitor call 39
3.2.3.2 Interactive debug with *GDB* 40
3.3 ARMv8-A security on gem5 . . . . **41**
3.3.1 *aarch64* and its *gem5* model 41
3.3.1.1 *aarch64* generalties 42
3.3.1.2 *gem5* ARM platform model 42
3.3.1.3 ARM cache model and *AutoLock* 42
3.3.2 Cache timing attack on *aarch64* 43
3.3.2.1 *Flush+Reload* 43
3.3.2.2 *Prime+Probe* 44
3.3.2.3 *Prime+Probe* direction and self-eviction 45
3.3.3 Our baremetal prospects 46
3.3.3.1 Principle 46
3.3.3.2 Results 47
3.4 ARM TrustZone and OP-TEE on gem5 . . . . **48**
3.4.1 TrustZone 48
3.4.2 Platform and boot model 49
3.4.3 OP-TEE software model 51
3.4.4 Refining TrustZone implementation in *gem5* to support OP-TEE 53
3.4.5 Our typical OP-TEE scenarios 54
3.4.6 Third Party IP simulation 54
3.5 Conclusion **58**
3.A Appendix **59**
3.A.1 *GDB* API in *gem5*-Python 59
3.A.2 ARM system devices in *gem5* 59
3.A.3 Timing gadget on ARM 60

3.1 Introduction

Our goal was to develop a platform that could reproduce a state-of-the-art embedded platform, being able to re-create the countermeasures and the attacks that such a platform would have in reality and anticipate the potential threats by finding vulnerabilities at microarchitecture level. However, this platform, besides being able to run realistic scenarios with as little modifications as possible while supporting typical hardware countermeasure, should also be used to gain more information and control on these scenarios: to study, reproduce, and defeat "security by obscurity" countermeasures. In this part, we highlight an exploratory process that reassembles the different axes present in the state-of-the-art. Each axis element in the context of the virtual platform (*gem5*, OP-TEE, and ARM) is detailed and linked with their state-of-art before developing my contribution for each of them. Finally, they can be re-aggregated to assemble our *virtual security platform*. This structure is represented on figure 3.1.

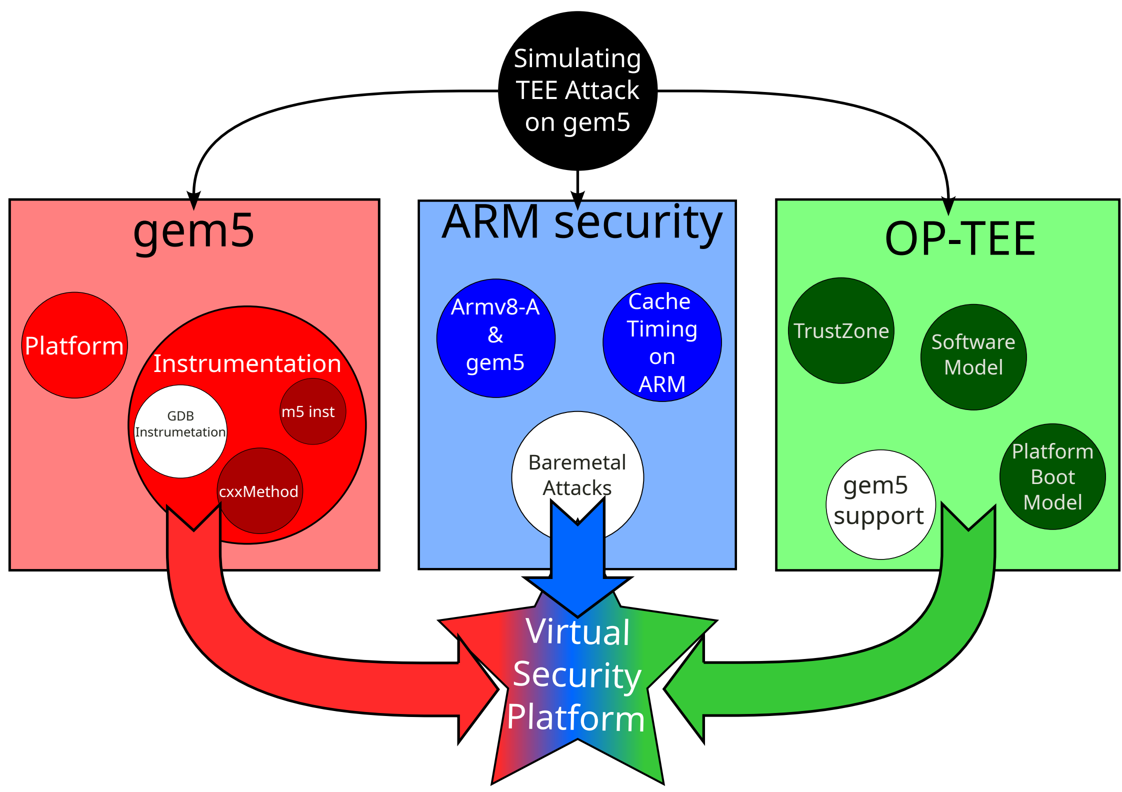


Figure 3.1: Structure of Chapter 3: white circles represent the key contributions for each of the subject axis.

3.2 Platform and instrumentation with *gem5*

gem5 provides simple config files stored in the `config` folder. These config scripts provide a basic platform to test directly binaries and produce basic statistics based on what models (*SimObjects*) are used. Typically used config scripts are provided by *gem5* in the `configs/example` folder (see section 2.2.2.1):

- `se.py`: This script configures a *SystemCall Emulation* config.
- `fs.py`: This script configures a *FullSystem* config using command-line arguments to specify

However, *gem5* encourages building your own config script, and in the following part, we detail how to build one. And then, with our own config script, we explain how to implement, customize, and use *gem5* simulation and its instrumentation tools. In this part, we also detail our contribution to instrumentation with *gem5*.

3.2.1 Building a platform in *gem5*

gem5, after a short initialization, runs the provided Python config file in its command line argument. Indeed *gem5* acts like a Python interpreter running the config file, which can import other modules from other `.py` files. Command-line

arguments are split between those intended for *gem5* and those intended for the config file (which can be processed with the `argparse` Python module.)

```
gem5.opt <--gem5 arguments--> config_file.py <--config file argument-->
```

The config file can use its command line arguments to configure the platform it builds, while *gem5*, command argument mainly configures how the simulator primitives (*SimObjects*, etc.) behave. To build `gem5` executable, `scons`[Fou24] have to be used. For example, the following builds `gem5.opt` build with only the `ARM` ISA included:

```
scons build/ARM/gem5.opt
```

3.2.1.1 Writing a new config file

`m5` module provides the necessary primitives which are implemented through the `_m5` module. This module contains the `pybind` implementation for **Param Class** and **CcObject class** Python interface. These implementations are generally hidden in config files, and only **Python Classes** are used. As config files are executed by *gem5* they have several purposes:

- *SimObject* can be defined directly in config files, they have to inherit from a `m5` imported *SimObject*. These new config-defined *SimObjects* can implement methods to automatize their deployment, configuration, or exploitation. (e.g. `Cache` can be used to implement `L2Cache` directly in config)
- Starting from the *Root SimObject*, a tree of *SimObjects* have to be built using *Python Classes*. Among *Params*, each *SimObject* expects other *SimObjects* as attributes. They are called *children* (e.g. a cache expects a tag *SimObject* as a child). In this context, a device is a child of a platform, which in turn is a child of a system. This hierarchical relation gives a *SimObject* its unique name (e.g. `system.cpu1.dcache.tags`).
- Independently of the hierarchical relationship between *SimObjects*, some *SimObjects* have ports that must be connected to other *SimObjects*'(see figure A.3). There are two types of ports: `cpu_side` (*master*) `mem_side` (*slave*). Each *SimObject* `cpu_side` has to be connected to a `mem_side` port. Ports can be vector ports to represent buses.
- When the *SimObject* tree is complete and connected, we can request *gem5* instantiation of *CcObject Classes* corresponding to each of our *Python Classes*. Using `m5.instantiate`, we can optionally provide a checkpoint. This checkpoint is handled per *SimObject*. They are expected to restore the data that they serialized in the checkpoint. This data is conveyed to the right *SimObject* in the tree using its name (e.g., `system.cluster0.cpu1`).
- When *SimObjects* are instantiated, *gem5* can no longer modify the tree structure outside well-defined functions (e.g. `m5.cpuSwitch()`). Config files can only launch the simulation using `m5.simulate` for an explicit amount of ticks or implicitly until the max possible number of ticks (2^{64}). *gem5* will process `Events` scheduled by *SimObject* until a `SimLoopExitEvent` is encountered. This `SimLoopExitEvent` can be scheduled by *SimObjects* to halt the simulation at a specific tick, providing a reason for the termination. As it returns because of the simulation exit event, the result of the `m5.simulate` function carries the reason given by the *SimObject*. Thus, this reason can be exploited directly in the Python config file. While the simulation is stopped, the config file can interact with *SimObjects* or *gem5* instrumentation tools depending on the reason provided for the simulation exit. After utilizing the `ExitEvent`, the config file can recall `m5.simulate` to continue the simulation or exit *gem5* like any Python interpreter (`exit`).

Therefore, config files represent more than just a platform memory bus architecture: They represent its detailed configuration, its initialization, its integrated tools, and how it reacts to events in simulations. To implement TEE security functionalities in *gem5*, we can not ignore the possibilities provided by writing our own config files.

3.2.1.2 Adding new *SimObject*

The *SimObjects* that we manipulated in our config files mostly rely on a C++ implementation. During the simulation loop, execution does not leave the C++-compiled code to take advantage of C++ compiled code speed. To allow this system of flexible configuration system, while keeping the C++ efficiency, *SimObjects* use a system of base classes that define interfaces between C++ classes, allowing them to be assembled into full systems. This system of interfaces allows a user to modify directly the simulator to add new *SimObjects*. For a slot in a parent *SimObject* class, the user can propose its own model.

MySimObject.py

```python
class MySimObject(SimObject):
    type = 'MySimObject'
    cxx_header = "my_sim_object.hh"
    cxx_class = 'gem5::MySimObject'
    addr = Param.Addr("Device Address")
```

my_sim_object.hh

```cpp
class MySimObject: public SimObject{
    Addr addr1;
    Addr addr2;
    public:
    PARAMS( MySimObject);//using
    ↪ Params=MySimObjectParams;
    MySimObject(const Params &p);//launch at
    ↪ initialization
    void startup();//launch at the fist
    ↪ simulation
 }
```

Figure 3.2: Minimalist *SimObject* definition and declaration in *gem5*

An example of that is cache replacement policies: the cache *SimObjects* expect a `replacement_policy` *SimObject* that inherits from `gem5::replacement_policy::Base`. A user can thus implement its own replacement policy, which can then be used in place of already implemented policies. To create a new *SimObject*, we only need to create two elements:

- **A CcObject Class**: A C++ class which has to inherit from the correct interfaces class or at least from the `SimObject` class which is all *SimObject* base class. This is the `MySimObject` from `my_sim_object.hh` in figure 3.2.
- **A Python class** which referenced the *CcObject* class header and which has to also inherit from the same equivalent Python class that corresponds to the *CcObject* class parent. This Python class is what config files manipulate. Therefore, it is where all *Params* are defined. This is the `MySimObject` from `MySimObject.py` in figure 3.2.

gem5 uses *scons*[1] as its *build automation system*. It creates macros to simplify incorporating new *SimObject* into *gem5*. In the `src` folder, each subfolder contains `Sconscript` files which control what adjacent files will be included in *gem5* and to what *SimObject* they correspond. To compile a *SimObject* in *gem5*, we have to add it to its coincident `Sconscript` (see figure A.2).

Scons will automatically generate the **Param Class** C++ implementation mentioned in section 2.2.2 with its Python binding and the *CcObject* using `pybind11`. For `MySimObject`, the *Param Class* is `MySimObjectParams`. Scons only uses the Python class to create the *Param Class*. It exposes through its C++-binding equivalent to all the *Param* defined in the **Python Class**. This means that the C++ variables that correspond to the *Param* can be set directly from Python. The only *CcObject* constructor argument is an object from the *Param Class* containing all the Params. (e.g. `MySimObject(const Params &p);`) *SimObject* ports behave differently from the other *Params*. They are directly referenced in the *CcObject* definition in C++. Ports also have to be linked when initialized by their constructor using their *Param* name. After being added to *gem5*, our *SimObject* on figure 3.2 can then be used in config files. Each instance provides *Params* as keyword argument (`obj=MySimObject(addr=0x1000)`). When calling `m5.instantiate()`, all the *CcObjects* associated with *SimObject* descendants from the Root *SimObject* in Python are created using the following process:

- *SimObjects* try to resolve the *Param* proxy.
- If a *SimObject* has all its *Params* resolved, its related *Param Object* is instantiated in Python and attributes are filled with params and corresponding *CcObject* for *SimObject* descendants.
- the *Param Object* `create()` Python method is used to instantiate the *CcObject*. This method calls the *CcObject* constructor with the *Param Object* as argument and returns the constructed object.
- The *SimObject* is not fully instantiated and can be used to resolve pending proxies. *Param Object* and *CcObject* are stored in the Python *SimObject* respectively in `_ccParams` and `_ccObject` attributes.

At the end of this phase, all the *SimObject* have been initialized. Simulation can now start using `m5.simulate()`. During simulation, *SimObject* methods can be called in two context:

- They Schedule an event in the event queue that calls their method at a specific tick[2].

[1] *scons* is a Python-based build automation system similar to `cmake` or `graddle`. It handles building object files (`*.o`) and linking them to build a program.

[2] a *SimObject* event queue is specified in the `eventq_index` *SimObject* attribute (*Param*). Although this usage creates instabilities, *gem5* supports multiple event queues

```
31000:system.cluster0.cpus0:T0: 0x...fdf0: movz x2, #192, #0     :IntAlu:D=0x00000000000000c0
32000:system.cluster0.cpus0:T0: 0x...fdf8: add  x0, sp, #32      :IntAlu:D=0xffff000974737970
33000:system.cluster0.cpus0:T0: 0x...fdfc: bl 0xffff80001047b640 :IntAlu:D=0xffff80001006fe00
```

Figure 3.4: Sequence of executed instructions when the *DebugFlag:Exec* is set

- They can be called by another *SimObject* method which was scheduled in their event queue.

All the *SimObjects* are also visited at tick 0 of simulation calling `void startup()` method. This is when *SimObjects* can schedule their first event. *SimObjects* are responsible for rescheduling events, if they do not have any event in the queue, only other *SimObjects* scheduled events are able to interact with their methods. Therefore, when the event queue is empty *gem5* instantly stops the simulation and returns from `m5.simulate`

3.2.2 Classical instrumentation on *gem5*

gem5 provides multiple tools to instrument a simulation. They serve 3 purposes:

- Debug *gem5* to verify how a feature (often represented as a new SimObject) behaves.
- Produce data to study how a feature behaves when running a specific scenario.
- Adapt or configure a feature to specific parts of a scenario. (e.g. having a performance-costing feature only enabled for a Region-of-Interest (ROI))

When running *gem5*, you can define a *m5out* folder using the command line (`--outdir=|$m5out$|`). this folder is the preferred folder for *gem5* to store all the reports and results it produces for the simulation. For example, *gem5* will store reports that contain the configuration used in human-readable format or as figures (like figure A.3). *gem5* also allows *SimObject* to monitor statistics. These are implemented using variables in C++ that *SimObject* can update to compute totals, averages, etc. At the end of the simulation, these statistics are reported in a result file stored in the *m5out* directory.

In this section, we present the different instrumentation tools present in *gem5* and how we use them in our scenario. This section highlights functionalities that are not necessarily documented properly and highlighted in *gem5* tutorial. As we did not make use of statistics in our instrumentation, they are only mentioned here for exhaustivity.

3.2.2.1 *DebugFlag*

DebugFlags are key elements in using and debugging *gem5*: they allow the activation of specific debug messages related to a specific feature. These messages are printed in the *gem5* console (`stdout`). For example, the `Fetch` *DebugFlag* activates messages related to CPU fetching instructions. In *gem5* C++ source code, these latent messages are made using `DPRINTF`:

```
DPRINTF(Fetch, "Fetch: Inst PC:%08p, Fetch PC:%08p\n",
↪  instAddr, fetchPC);
```

debug_flag.py

```
import m5
m5.debug.flags["Fetch"].enable()
m5.debug.flags["Fetch"].disable()
```

Figure 3.3: Enabling and disabling *DebugFlag* from config files

They also track the object that triggered the `DPRINTF` and add it to the message:

```
31000:system.cluster0.cpus0:Fetch: Inst PC:0x00002334, Fetch PC:0x00002338
```

DebugFlags are not available in the `gem5.fast` binary only in `gem5.opt` and `gem5.debug`. To activate the them, the `--debug-flags=` *gem5* command-line argument have to be used, with a comma-separated list of flags.

```
gem5.opt --debug-flags=Fetch,Exec config.py --cpu 4 test.bin
```

Although not documented, *DebugFlags* can also be activated and deactivated directly from the Python config file as shown in figure 3.3. This is done using a Python dictionary containing all the *DebugFlags* located in the `m5` module in *gem5*. In this context, they can be enabled and disabled dynamically between `m5.simulate()` call. For example, in response to a CPU switch or a m5 instruction.

3.2.2.2 *m5* instructions

m5 instructions are *gem5*-specific instructions that are integrated into the workloads' binaries through a static-link library. They are also accessible in a dedicated executable (`m5`) for command-line usage. These instructions allow a program to communicate with the simulator either directly or in the Python config script by exiting the simulation loop. With them, instructions can pass messages between the host and simulated environments dynamically upon request by the simulated program. All *m5* instructions are listed in table A.1, including `m5_env` that we added. *m5* instructions can also be used directly in a bash script inside the simulation, using the *m5* tool (also mentioned in table A.1). Some *m5* instructions exit the simulation and require their intended behavior to be implemented in the config file. This is done as follows in config files: When an exit event caused by `m5` instruction has been received, `m5.simulate()` exits returning the event. The config file can then use `event.getCause()` and `event.getCode()` to dispatch and handle the event appropriately before calling `m5.simulate()` and resuming the simulation where it exited. For example, in figure 3.5, a `m5_exit` causes the simulation to exit with `"m5_exit instruction encountered"` which is handler by the config file by a `exit(0)` which closes *gem5*.

As shown in figure 3.5, taking checkpoint uses `m5.checkpoint(cpkt_save_folder)` from the `m5` module. We exposed its internal logic on figure 3.6. First, this function causes all the *SimObject* to be *drained* before taking the checkpoint: **Draining** forces all the *SimObject* to advance their internal pipeline until it is emptied and push all pending data to non-volatile memory. Then, this function saves the simulation state using functions implemented in the *CcObjects* (`serialize( CheckpointOut &cp)` method). A checkpoint can be restored using `m5.instantiate(ckpt_folder)` providing the checkpoint folder to the `m5.instantiate` method. After the normal instantiation, any data that was serialized during checkpointing by *CcObject* will be provided back to the *SimObject* using the method `unserialize( CheckpointIn &cp)`. This manner of restoring checkpoint is only possible at startup (before any simulation loop with `m5.simulate`). We exploring restore a checkpoint manually at any point by directly using the `m5` internal logic, presented on figure 3.6. This enabled us to restore checkpoints in the middle of the simulation. Considering that it sometimes caused crashes and lacked a use case for this functionality, we only use checkpoint restoration at the start of a simulation run.

checkpoint.py

```python
m5.drain()#draining simulation
ckpt = _m5.core.getCheckpoint(ckpt_dir)
for obj in root.descendants():
   obj.loadState(ckpt)
print("Checkpoint restored")
for obj in root.descendants():
   obj.startup()
```

Figure 3.6: Internal logic for gem5 to restore a checkpoint from `ckpt_dir`.

3.2.2.3 *CxxMethod*

Not mentioned in *gem5* presentation of *SimObject*, `cxxmethod` is a Python decorator[3] that can be used inside the *SimObject* Python definition, allowing a C++ function to be called from the Python config files. With these functions, we can implement methods to dump information or change *SimObject* behavior on demand (figure 3.7).

main.py

```python
while True:
   event=m5.simulate()
   exit_msg = event.getCause()
   code=event.getCode()
   if exit_msg == "checkpoint":
      #here we have to handle the m5_checkpoint instruction
      m5.checkpoint(cpkt_save_folder)
   elif exit_msg == "m5_exit instruction encountered"
      #receive a M5 exit so we stop the simulation loop
      exit(0)
```

Figure 3.5: Extract from a config file which handles the simulation loop. It follows the building and configuration of the system model and its instantiation.

[3]In Python, a **decorator** is a design pattern that allows you to modify the functionality of a function by wrapping it in another function. The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it. The `@` symbol can be used to automatically decorate a function, replacing it with its decorated variant.

MySimObject.py

```python
class MySimObject(SimObject):
  type = 'MySimObject'
  cxx_header = "my_sim_object.hh"
  cxx_class = 'gem5::MySimObject'
  addr = Param.Addr("Device Address")
  @cxxmethod
  def dump_state(self,format):
    pass
```

my_sim_object.cc

```cpp
class MySimObject: public SimObject{
  Addr addr1;
  Addr addr2;
  PARAMS( MySimObject);//using Params=MySimObjectParams;
  MySimObject(const Params &p);//launched at
  ↪  initialization
  void startup();//launch at the fist simulation
  std::string dump_state(std::string format);
}
```

Figure 3.7: How to use `cxxmethod` decorator to implement Python callable C++ methods

main.py

```python
#launch the simulation
exit_event=m5.simulate()
msg= exit_event.getCause()
if msg == "m5_exit":
   print(my_sim_object.dump_state())
```

Figure 3.8: Example of *GDB* monitor call handling in *gem5* config files

This decorator exposes the C++ method that shares the same name as the *decorated* method using the Python binding for the **CcObject**. The *SimObject* Python implementation then automatically calls the bound C++ method when the corresponding Python method is used, and it transfers its arguments and return values between C++ and Python. Because the *CcObjects* need to be initialized for this method to be called, it can be called only after stopping the simulation. If the simulation stops, it means that the `m5.simulate` command returns, and we can react to its exit message. On figure 3.7, we show how to add a `dump_state(self,format)` method to a *SimObject*, which can then be called in Python config files, like the following example on figure 3.8. In this example, the *CxxMethod* is called when the execution reaches a `m5_exit` instruction. With *CxxMethod*, we can configure and extract from *SimObject* dynamically in response to simulation events. To enable more interaction with *gem5* mid-simulation, we added *CxxMethod* to *gem5* standard *SimObjects*, giving us direct access to their internal states when the simulation is paused.

3.2.3 Our improvement to *GDB* in *gem5*

gem5 features a **stub** (represented on figure 3.9) that allows connecting *GDB* to the simulation, similar to a development board. On a development board, SoCs can implement specific functions inside their stub and integrate them into *GDB* as *monitor* queries. By default, the *GDB* stub in *gem5* allows a remote *GDB* to debug the program running in *gem5*.

This *gem5* stub supports:

- Connecting to a remote *GDB* running on the host machine using localhost TCP port.
- Debugging the program running in *gem5*: in *FullSystem mode* and in *SystemCall Emulation mode*.

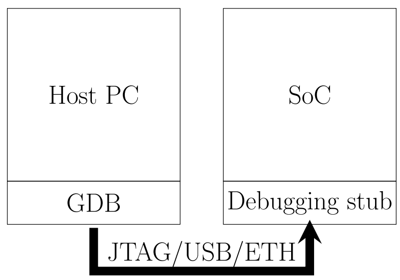


Figure 3.9: A *GDB-stub* in a SoC or in *gem5* connected to *GDB*

GDB also supports scripting in Python using the `gdb` module, importable in Python files run from *GDB*. We fixed several issues in the *GDB-stub* implementation in *gem5*:

- Multi-thread implementation was incomplete.
- CPU-switch was not compatible with using *GDB*.
- We added memory watchpoint support (which are breakpoints triggered by memory accesses to specific addresses)

3.2.3.1 *GDB* monitor call

Development boards, or more specifically, development SoCs can implement specific functions inside their stub and integrate them into *GDB* using the *monitor* queries. In our *gem5* build for our virtual platform, we implemented the *monitor* query in *GDB* such that it results in a new cause for exiting the main *gem5* simulation `m5.simulate()` thread. As illustrated on figure 3.11, monitor query sends a message from *GDB* to *gem5* to which gem5 can then respond. As shown on figure 3.10 and figure 3.11, we used the Python interpreter inside *gem5* to parse the message and respond

to it using Python methods. This Python methods have access to *CxxMethod*, such as `obj.dump_state(format)` described in figure 3.7, to configure and extract from the simulation state.

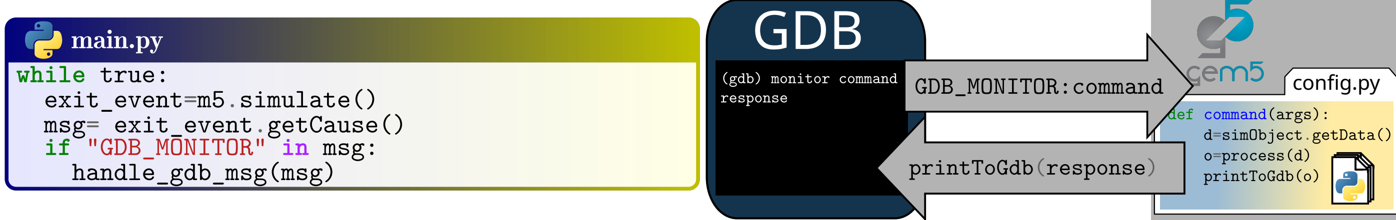


Figure 3.10: GDB monitor message reception and handling in gem5 config files.

Figure 3.11: *GDB* monitor command: sending and receiving message during the simulation.

As a contribution, we also added this feature to the stable branch of *gem5*. It was presented in [Forne].

3.2.3.2 Interactive debug with *GDB*

With the Python API, more than simple console commands, a full *interactive debugging session* can be set up between *gem5* and Python.

It contains two Python interpreters running a program:

- **gem5-Python** running a *config file* which implement monitor commands to modify and extract information from the simulation.
- GDB with a real user or with **GDB-Python** running a specific script file: In both cases, monitor commands can be issued to communicate with *gem5*.

With a script file in *GDB-Python*, it is possible to fully automatize a monitoring process. In an interactive debugging session, the *monitor call* can be used to control the simulation from *GDB*, more specifically to change the precision/speed of simulation, change to simpler CPU models, access the performance counters, for text I/O through a terminal, to flush caches, to dump cache/execution traces, etc. These monitor calls are implemented directly in *gem5-Python* with the *GDB* API described in section 3.A.1. With this GDB API, *gem5-Python* can control the GDB stub to implement functionalities like the ones illustrated on figure 3.13.

Figure 3.12 shows the typical interactions between *gem5-Python* and *GDB-Python*, the sequence is as follows:

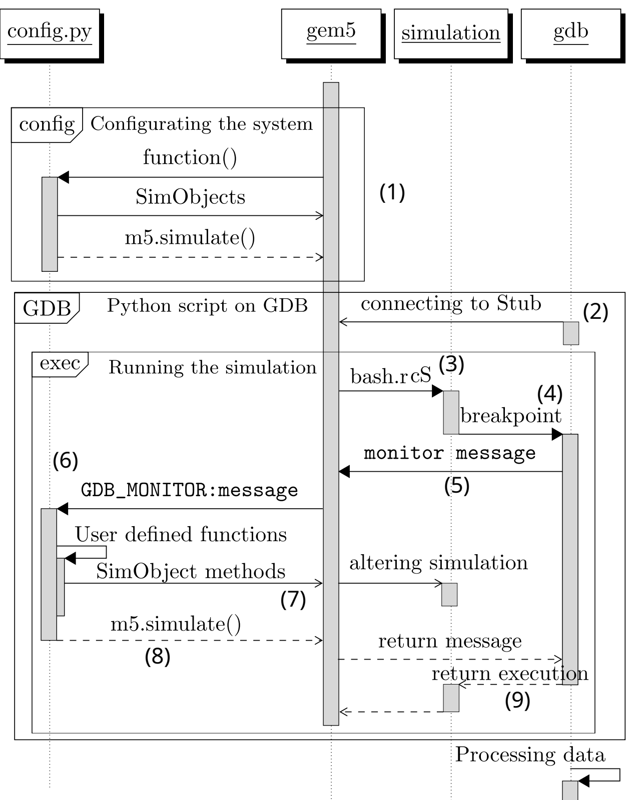


Figure 3.12: Simulation scripting: we use *GDB* to modified simulation parameters on-the-flight, the binary is never modified.

(1) *gem5-Python* sets up the system and call `m5.simulate()` to start the simulation
(2) The system waits for a *GDB* to connect before starting. Multiple breakpoints can then be set through *GDB*.
(3) The system loads a bash script, `bash.rcS`, from outside the simulation.
(4) After reaching a breakpoint, either the user or an automated Python script can debug the simulation.
(5) Using the `monitor` command, they can send specific commands to the *gem5-Python*. (e.g. `monitor dumpCache`)
(6) On the the *gem5-Python* side, the `m5.simulate()` command finishes with an event that contains the monitor message.
(7) This message is then interpreted using *gem5-Python* and the method implemented in the *SimObjects* in C++.

(8) The *gem5-Python* can then process this data and respond to the *GDB*. Before recalling `m5.simulate()`.

(9) *GDB* can then allow the simulation to progress.

As one can see, the binary workloads are never modified and can be configured dynamically with *GDB* and the loaded `bash.rcS` script. This methodology resembles what Mihajlovi, ili, and Gross [MG14] called *GDB instrumentation*. While theirs was integrated directly inside QEMU source code, our implementation mostly uses *gem5-Python* interpreter to dispatch and execute the commands sent by *GDB*.

With an interactive debugging session and the right monitor commands, the *gem5* user can then:

- Take checkpoint manually before a critical event.
- Automatically dump the content of the L2 cache when a specific function is executed.
- Switch CPU types only inside a specific function in a binary loaded from the OS.
- Enable and disable *DebugFlag* manually on the fly through *GDB*.
- Use any *CxxMethod* of your *SimObjects* on the fly. By implementing a monitor `exec` command, Python code can be sent directly to be executed by *gem5-Python* from *GDB*.

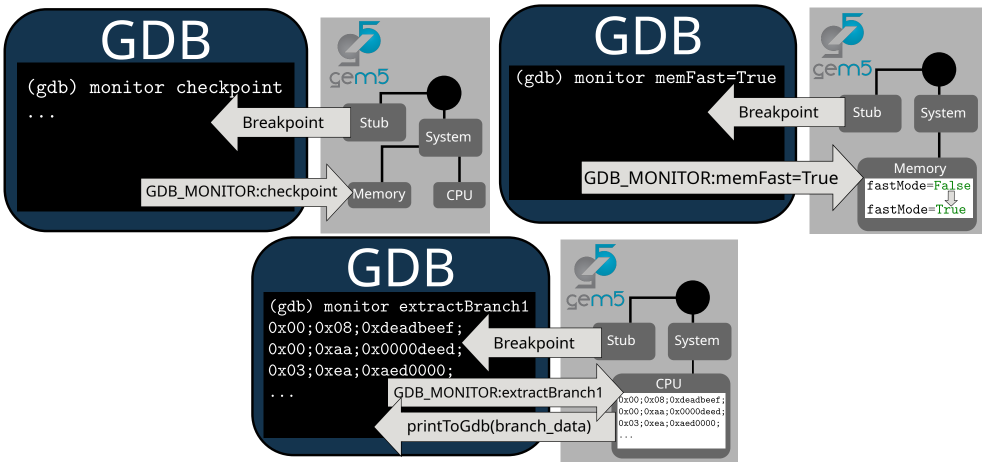


Figure 3.13: Typical use case for the interactive debugging session between *GDB* and *gem5*

3.3 ARMv8-A security on *gem5*

ARMv8-A is the most predominant ISA on smartphones. Contrary to the M variant, it can support classical OS like Linux as it features a MMU. It support a 64bit and a 32-bit mode:

- *aarch32* is the 32-bit mode and is similar to ARMv7. its associated GCCs have generally the `arm` prefix.
- *aarch64* is the 64-bit mode. It is the default ISA for ARMv8 and its associated GCCs generally have the `aarch64` prefix.

We decided to mainly cover *aarch64* attack scenarios, as they are the most representative of modern threats and still similar to ARMv7 and *aarch32* scenarios.

3.3.1 *aarch64* and its *gem5* model

To simulate these scenario, we use the ARMv8-A implementation in *gem5*. This implementation is also ARMv7-A compatible. This ARMv8-A implementation can thus run *aarch64* and *aarch32* binaries.

3.3.1.1 *aarch64* generalties

ARMv8-A has multiple privilege levels called **Exception Level (EL)** from EL0 to EL3 the higher the number, the more privileged the exception level is. Each exception level has a specific role for the system:

- EL0: User applications
- EL1: Kernel Level
- EL2: Hypervisor
- EL3: Firmware

Each EL can have its own exception handling (exception vector) and *Service Call* instructions : SVC for EL1, HVC for EL2, SMC for EL3. MMU settings and pagetables are also unique for each EL. *aarch64* has a set of 31 *work-registers*, from x0-x31 which can be accessed by in both 32bit (w0-w31) and 64bit (x0-x31) mode. X31 is denoted as SP and is used as the stack pointer. X30 is denoted as LR and is used as *link-register*. It is automatically set to the return address when doing a Branch-with-Link (BL). The *aarch64* ABI uses x0 as the return register and x0 to x7 for parameters. These registers have to be saved manually when transitioning ELs. *Exception Levels (ELs)* are configured using system register. In *aarch64*, the system registers' names contain the EL they apply to: `VBAR_EL3`, `VBAR_EL1`, etc. System registers are accessed and modified using MSR/MRS (usage for these instructions is detailed in figure A.6) In *gem5*, the support for *aarch64* is implemented in `arch/arm/isa.cc`. In this file, `readMiscRegs` and `setMiscRegs` implement system registers behavior. These functionalities are integrated into the ArmISA *SimObject*. This Object is present as a sub-*SimObject* in each CPU object.

gem5 ARM implementation also includes other system device models for ARM MMU (including its automatic table walker) and ARM configurable interrupt controller called General Interrupt Controller (GIC). We give more detail on these devices in section 3.A.2.

3.3.1.2 *gem5* ARM platform model

On ARMv7-A and ARMv8-A, *device discovery* is implemented using a Device Tree Blob (DTB). The DTB is a binary file compiled from a readable device tree. This is a tree structure that organizes devices as nodes in the tree, which can have properties that describe them and other sub-nodes/sub-devices. The DTB is provided to Linux by the bootloader. The different drivers (or modules) loaded by Linux parse the DTB, searching for compatible devices (using the `compatible` properties in DTB nodes). Each compatible devices are then initialized using the driver implementation and will then be visible in Linux, typically in `/dev`. The DTB allows ARM to have modularity on how ARM processors BOOT and what devices they have at their disposal. These platforms and the way they boot can be organized in categories that only differ by the devices they implement. *gem5* ARM implementation is thus part of the *ARM Versatile express* (*Vexpress*) family, which contains mostly development boards (e.g., CoreTile Express boards) and *Virtual Models* (e.g. Versatile Express Fixed Virtual Platforms). Some key characteristics of the *Vexpress* platform are:

- *Vexpress* platform boot from a memory-mapped bootrom that contains all the bootloader elements needed before Linux (figure 3.25).
- The *Vexpress* platform in *gem5* uses the *Fixed Virtual Platforms Base_PowerController* (`FVPBasePwrCtrl`). This controller directly interacts with the *GIC* to handle the Power State Coordination Interface (PSCI).
- *gem5 Versatile express* memory map is based on the *Versatile Express RS1* (V2M-P1), with both off-chip devices and on-chip devices (based on those featured on the ARM CoreTile Express A15x2 daughterboard (V2P-CA15)).

3.3.1.3 ARM cache model and *AutoLock*

On ARMv8-A, caches are split into two domains: inner and outer. For each domain, each page in the pagetable has dedicated cachability and ordering behavior. In *gem5*, *aarch64* pages are either assumed to be *uncachable-strictly-order* or cachable with read or write allocability determined by the cache model used. Write-back or write-through behavior, which is also normally defined by the page table entry, is not implemented in the *aarch64 gem5* model and is instead determined by cache models. This is generally not an issue, as this is how Linux and OP-TEE configure caches. In *gem5*, inclusivity and exclusivity have a simple implementation. They differ only on how they treat requests:

- Inclusivity makes both read and write allocating for the caches.

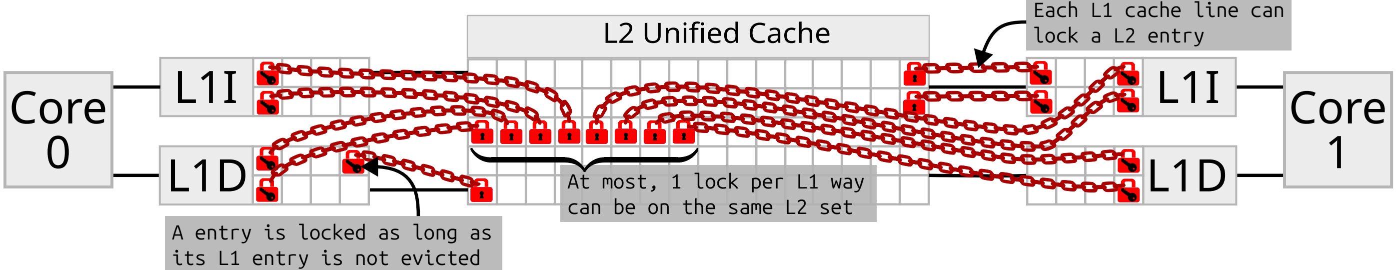


Figure 3.14: *AutoLock*: This cache replacement policy prevents eviction of L2 lines that are still present in a cache L1. This lock is set up when a cache L1 receive a *miss* response from the L2. This lock on the L2 line is opened when all the L1 lines that lock a L2 line are evicted.

- Exclusivity makes all the read request "read and clean" requests, preventing caches from responding with dirty-lines.

Cache on ARM can be inclusive or exclusive. However, *gem5* can not accurately model certain caches for ARMv8 because they enforce inclusivity in a stronger sense than *gem5*'s cache implementations. This is called *AutoLock* [Gre+17] (figure 3.14). For example, it is present in the RK3399 on its A72 core complex.

AutoLock, as described by [Gre+17], is an ARM-specific replacement policy designed to enforce inclusivity by preventing the eviction of L2 cache entries if they are present in a connected L1 cache (see figure 3.14). This policy ensures that L2 entries remain locked until they are evicted from the L1 caches. Only after their eviction from the L1 caches can they be considered for eviction from the L2 cache. Consequently, up to one entry per associativity of L1 caches can be locked in the L2 cache (illustrated in figure 3.14). For a complete eviction of a L2 cache set (i.e., all ways not locked), the L1 entries corresponding to the current L2 entries must have been evicted by lines that share the same L1 index but not the same L2 index. This scenario is much less likely to occur with *AutoLock*, resulting in L2 sets rarely being fully evictable. To reproduce this behavior, we implemented *AutoLock* as a replacement policy in *gem5*. *AutoLock* in *gem5* couple L1 and L2 replacement policies (which have to be *AutoLock* enabled) and when evicting cache line will automatically avoid lines that are in lower level caches.

3.3.2 Cache timing attack on *aarch64*

Cache timing attacks have been studied on ARM and, more specifically, on *aarch64* [Lip+16]. The two classical examples are:

- *Flush+Reload*[ZXZ16] (or its variant *Evict+Reload*[GSM15])
- *Prime+Probe*[LJ18][Rei+16]

These two examples require a way to measure execution time for a specific *LOAD* instruction. We detailed the needed gadget in section 3.A.3

3.3.2.1 *Flush+Reload*

Flush+Reload requires the attacker to be able to flush the victim lines (figure 3.15). The attack processes as follows:

1 **Loading Shared memory spaces**: The attacker have to add to its memory space a section that it can share with the victim: generally a dynamic library shared between the two by Linux as represented on figure 3.15.

2 **Flushing** a target address in the shared memory space is flushed while the victim is computing. This address is linked to the key as the loading of this target address only happens if certain conditions on the secret value are met, typically a branch. After the memory is flushed, the attacker knows that the target address is no longer in caches. On figure 3.15, this target address is associated with a function represented as a magenta-colored gear, highlighted in red.

3 **Reloading** a target address in the shared memory space while measuring access time. If it takes less time (a cache *hit*), it means that the victim *"accessed"* the target address: implying certain constraints on the secret value. On figure 3.15, we represented the attacker measuring the magenta gear function access time as a red plot on the bottom plot. If the branch is taken, the access time is measured with a lower value corresponding to the solid

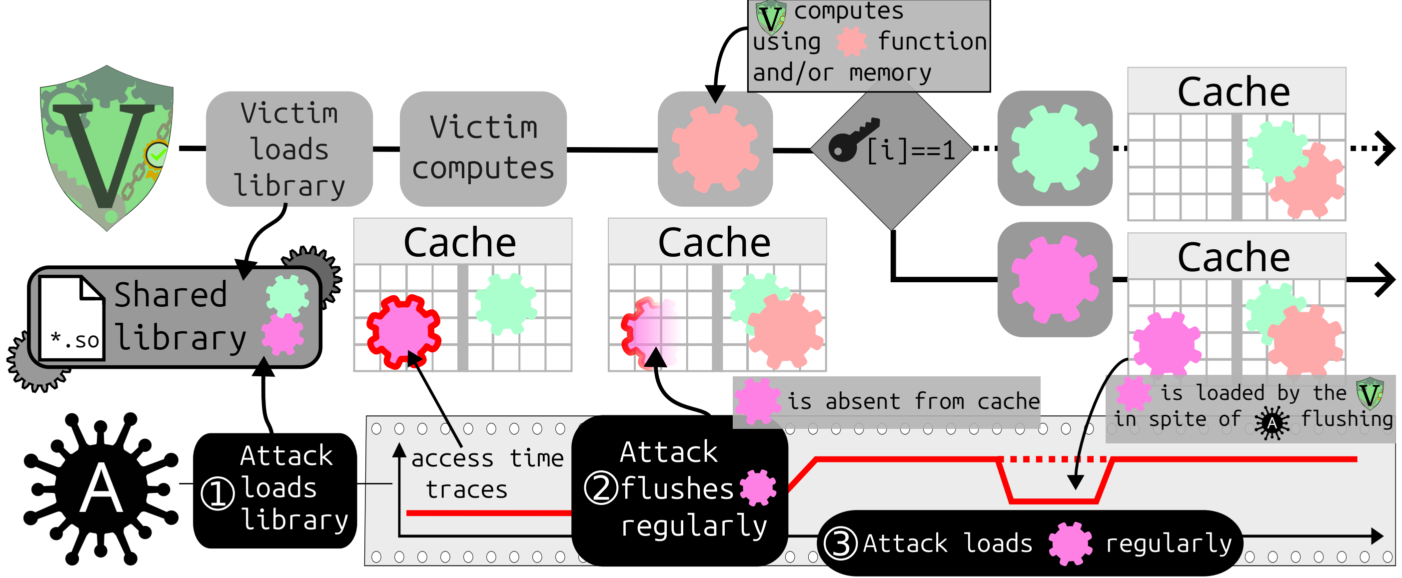


Figure 3.15: Representation of how a *Flush+Reload* works. The attacker knows that the victim took the branch because it detected the victim loaded the magenta gear.

line because the magenta gear function is used. If the branch is not taken, the timing measured correspond to the dotted line and is not changed because the gear operation used is not the *Flush+Reload* function. This access time measure is done using the primitive described on figure 3.36.

3.3.2.2 *Prime+Probe*

For *Prime+Probe*: the attacker only needs to know in which cache index the victim address will be put. The attack proceeds as follows:

1 **Allocating**: We allocate our *prime set*. A *prime set* consists of data with addresses specifically chosen to share the same cache index as the victim address[4]. There is an entry in the prime set for each associative way in the cache. (i.e. if the cache is 16-way associative, we need 16-entries prime set). If there are multiple victim addresses, each needs its own prime set.

2 **Priming** We access our prime set for the victim address we want to attack. This way, our prime set fills each possible alias for this victim address in the cache. Thus, the victim address is forced to evict one of the entries to be cached.

3 **Probing** Measuring access time to all the lines in our prime set to check if one was evicted by our victim. To do that, we use the code as illustrated in figure 3.36.

since **Probing** also fills the cache with the prime set like **Priming** would do. We do not need to **Prime** after **Probing** as long as we are only doing that.

We implemented our own *Prime+Probe* to ensure minimum noise and maximum performance. Following [TOS10] recommendation, we use a double-linked list data structure:

With the structure on figure 3.16, we can *probe* the set as we are going through it and directly store the timing result in `startT` (timestamp before the load happened) and `endT` (timestamp after the load happened) without interacting with any other cache lines. Each entry of the *prime set* fully uses its line of cache to store all the necessary properties for traversing it. Figure 3.17 shows how we probe each entry of the prime set. We start from the last entry we probed, knowing it is in our cache set, since all entries of the prime set share the same set index (here $0x38$). To continue to the next entry, we first have to *probe* it as we are not sure if it is still in the cache since last *probe* or *prime*. Using the code on figure 3.36, we measure the `LDR` execution time for the next entry in the prime set using the `next` pointer. When this *probe* measurement is complete, we know that the associated entry is in the cache set. Thus, we

[4] As ARM SoCs have physically indexed caches, we can use `/proc/self/pagemap` to reconstruct physical addresses (require root to access physical mapping information since Linux 4.0). Otherwise, we have to *probe* timings to verify that our prime set effectively filled the cache index.

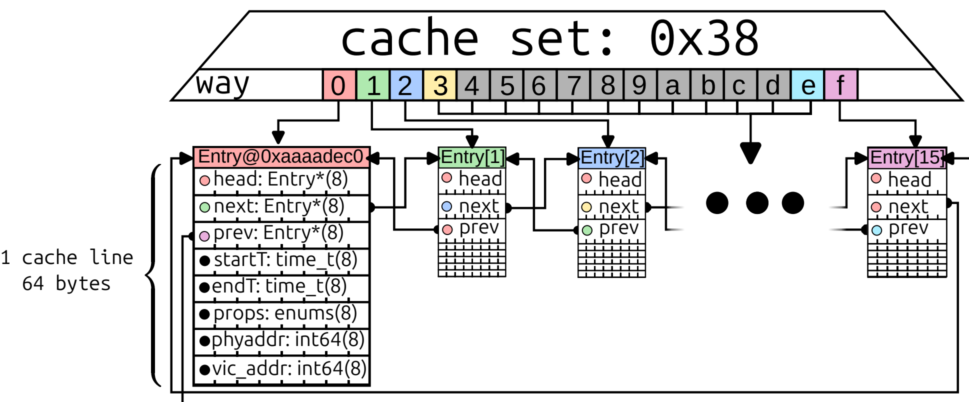


Figure 3.16: Our prime set uses a double-linked list. Its elements are allocated in such a way that they have all the same cache index (here is 0x38). If enough are allocated, they fill out all the possible ways for their index.

can store the result of this measurement directly in this entry. We then use the `head` and `next` pointer to determine if we have finished. Otherwise, we proceed to the next entry as we described before.

When the prime set has been traversed and probed, its results are transferred to the result table. Each result point is stored in a single cache line to ensure minimum noise. They contain: start-time, length, id (to identify which cache set it belongs to), entry count, and entries (time length of each *LOAD* operation result of the *probe*).

3.3.2.3 *Prime+Probe* direction and self-eviction

Already mention in [Liu+15], under the name *thrashing*, self-eviction happens when probing a prime set entry evicts another prime set entry. It can cause an entry to be wrongfully considered as been evicted by the victim. Figure 3.18 presents a situation in which *Prime+Probe* can cause self-eviction depending on the direction of *probing*. After priming, a victim evicts some element from the prime set (0 and 1) with its lines labeled "V". The cache handle this process using *Least-Recently-Used* replacement policy, whose timestamps are represented as small clocks on figure 3.18. The figure then presents the two directions for probing, **Forward** and **Reverse**, and how they interact with victim lines:

- **Forward**: Probing is always done in the same direction. If a *miss* is encountered, the following element will be bumped out of the cache due to self-eviction, and therefore all the following entries will be *misses*. On figure 3.18, the victim lines from the victim are not evicted first by the prime set because they were accessed the most recently. Instead, the prime set evicts all its lines until it reaches the end to finally evict the victim lines. This produces all the red timing values indicating all the entries in the prime set *missed*.

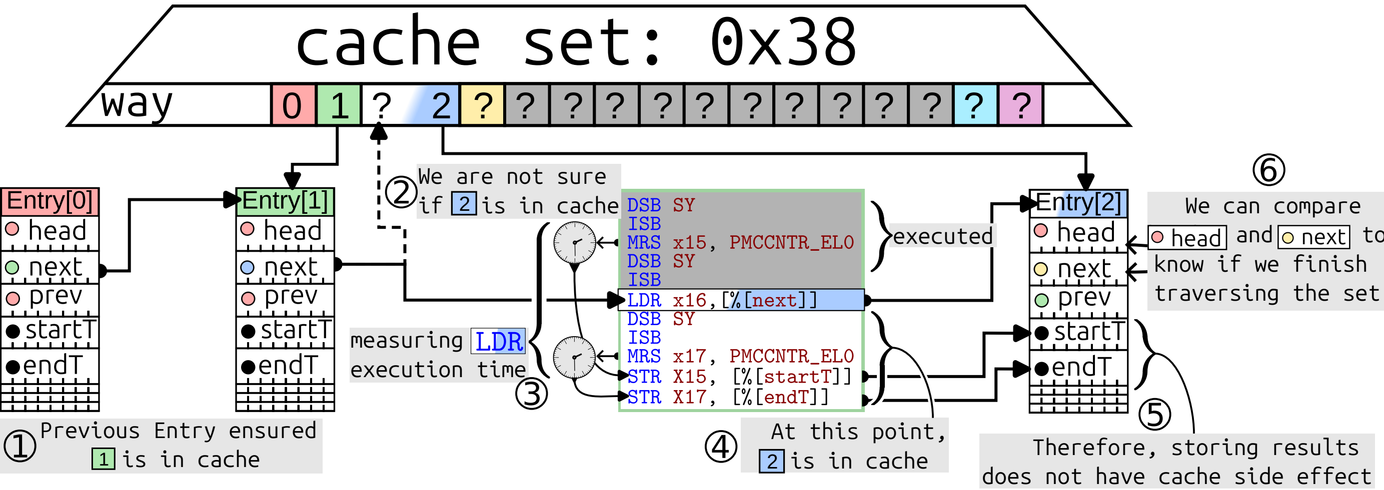


Figure 3.17: With the structure on figure 3.16, we can *probe* the set while traversing it. Timing measures for a prime entry are directly stored in it

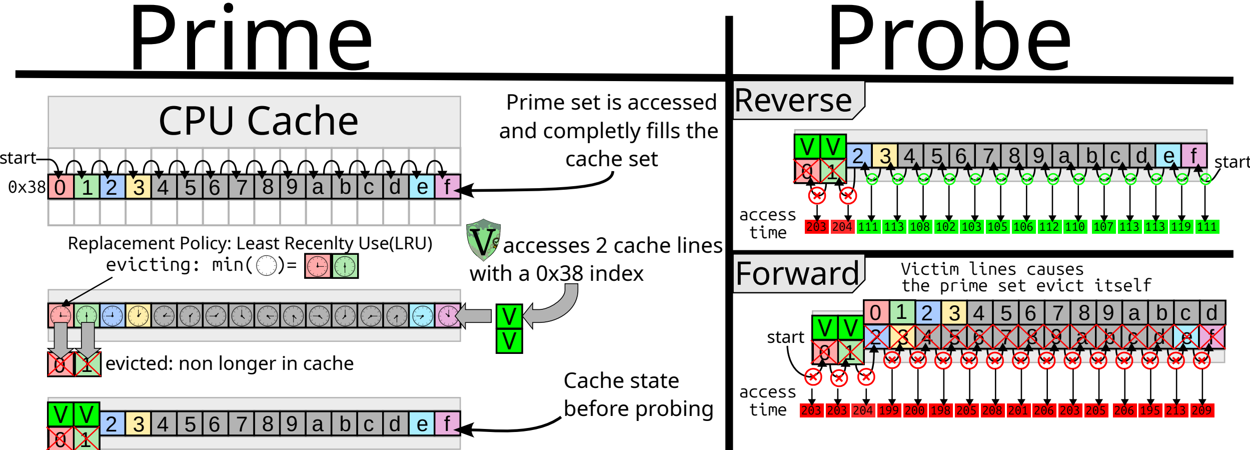


Figure 3.18: How direction of probing control self-eviction: after a victim accessed two lines, different *probe* directions produce different results.

- **Reverse** We change direction each time we finish probing. Thus, elements are never bumped, and we can observe the sensibility of each element. On figure 3.18, no additional entries are evicted because the entry accessed is always the next one that would be evicted. Thus, the victim lines are accessed last. This produces the green timing values (indicating the entries in the prime set *hit*) followed by two red timing values associated with the two prime set entries that *missed* because of the victim lines.

Depending on the victim we want to observe, we can choose between the two directions of *probe*. Using the forward probing creates bumping, which improves the Signal-to-Noise ratio[TOS10] [5]. On the other hand, reverse probing allows measuring if each element of the prime set is present or has been evicted by the victim. Therefore, with reverse probing, we can count how many victim lines share the same index have been used[6].

3.3.3 Our baremetal prospects

We propose first to run an attack on baremetal comparing a Raspberry PI and *gem5*: to verify if the *gem5* platform can simulate cache timing attacks.

For that, we use a small micro-kernel. To allow simulation of the attack, this kernel needs to:

- Transfer execution to EL1 as the SoC starts in EL3 which does not allow cachability.
- Set up the Memory Management Unit (MMU) to allow the cachability of certain parts of the memory using the pagetable properties.
- Configure automatically the UART and memory using the DTB provided by both systems.

The program can be run both in *gem5* and on Raspberry Pi without any modification.

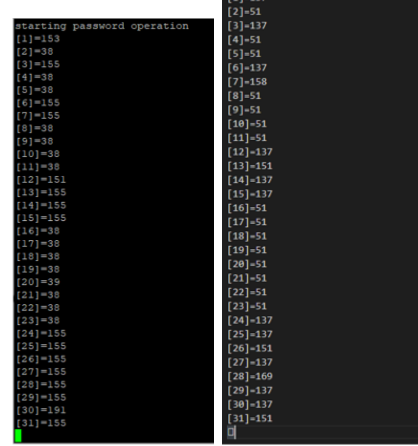


Figure 3.19: Comparison of the timing result for the cache timing attack between a Raspberry Pi 3B+(left) and *gem5*(right)

3.3.3.1 Principle

Using our baremetal kernel, we can run a basic *Flush+Reload* attack. This is the function we are trying to attack:

[5]This bigger signal can be seen in the top plot of figure 4.5

[6]The different signal associated with each number of lines evicted is visible on the bottom plot of figure 4.5

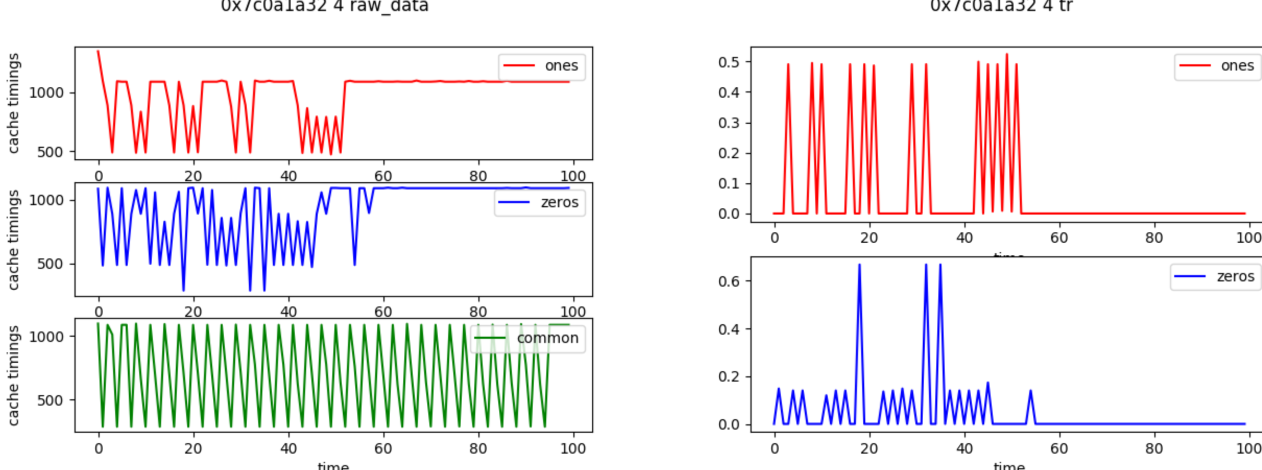


Figure 3.20: Result obtained with a baremetal attack (on the left) and a fast treatment to make result more visible on the right

victim_fun.c

```c
for(int i=0;i<SECRET_SIZE;i++){
    if(!secret[i]){
        /* Branch 0 */
    }else{//Branch 0 and 1 do not share a line
        /*Branch 1*/
    }
}
```

We propose a first synchronous variant of our *Flush+Reload* attack to demonstrate the similarities between a *gem5* simulation and a Raspberry Pi 3B+ run. On figure 3.19, we present the results of *Flush+Reload* attack running in baremetal compared between *gem5*(right) and a Raspberry Pi 3B+(left). This demonstrates *gem5* ability to simulate cache timing attacks.

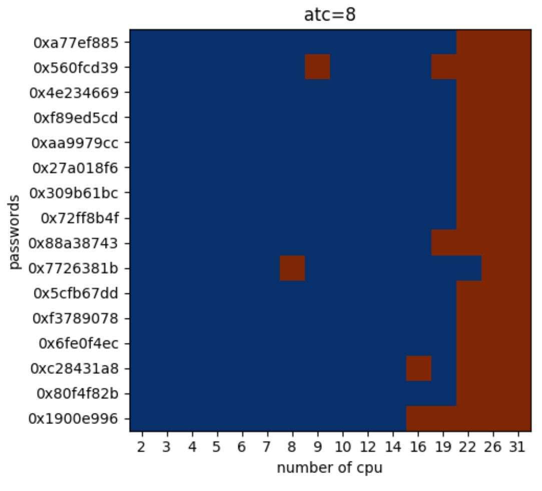
We also designed an asynchronous variant of the attack to test further on *gem5*. We experimented with the trace data generated by this asynchronous variant.

3.3.3.2 Results

In this demo, we use the *BasicTimingCPU* model for our CPUs. The traces we have for the asynchronous method allow us to rebuild the secrets easily through basic thresholding methods. On figure 3.20, we plotted timing results for our *Flush+Reload* attack running in *gem5* in baremetal. We extracted the right figure using basic thresholding on the raw result present in the left figure. We then proposed to consider the noise that could be added by having others CPU works in parallel and use the last-level cache. We built a success matrix and plotted the success rate of our methods when we changed the number of added CPU cores generating noise (figure 3.21). On this matrix, we see for different keys on the y-axis, if it was possible (blue squares) or not (red squares) to correctly recover the key while having other CPUs working in parallel and using the last-level cache (x-axis). This effect of noisy CPUs on our attack is represented on the right plot in (figure 3.21), which represents the success rate of our attack depending on the number of added CPU cores generating noise.

We also tested machine learning model attacks to have a more automatic and less ad hoc approach for the attack. We reached a 75% success rate on test data with (8000 traces for the training) The ML model we used mainly relied on convolution neural networks and ReLU/logistic activation functions (5 convolutions with a 3x3 kernel followed by a linear sum to produce the full key). We took the traces as a whole as an entry and output a possible password.

These experiments demonstrate that *gem5* can be used to model cache timing attacks but they also show the limits of using *gem5* to generate data for ML, since *gem5* is vastly slower even when running multiple simulations in parallel.



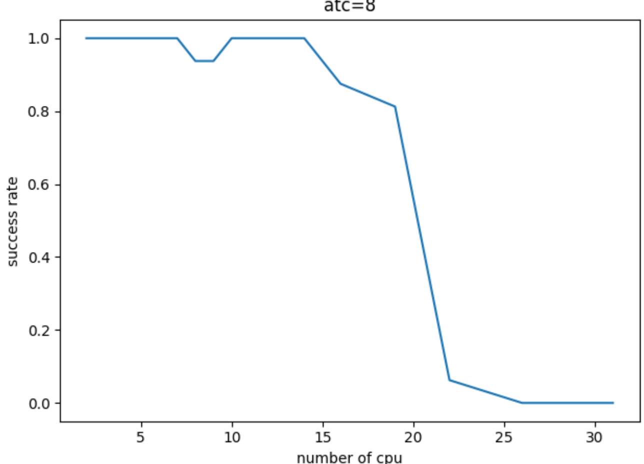


Figure 3.21: Success of the attack when multiple CPUs are generating noise

3.4 ARM TrustZone and OP-TEE on *gem5*

As we mentioned in section 2.3.4, there are multiple TEEs developed for ARMv7-A and ARMv8-A. For our project, we chose OP-TEE [YL20], which is an open-source TEE that follows the *GlobalPlatform* specifications [lea21]. It is now maintained directly by ARM as a part of the *TrustedFirmware-A Project* [Lin23c]. Like other TEEs on ARMv8-A, it uses TrustZone to operate.

3.4.1 TrustZone

TrustZone is the commercial name for all the software and hardware elements that are needed to implement a secure enclave in ARM. It mainly relies on a specific execution mode. On ARMv8-A, they are called **Exception Levels** (EL) (see figure 3.22). To support TrustZone, ARM implementations add secure EL variants to EL0 and EL1 (and sometimes to EL2). These variants are called EL0S and EL1S (and EL2S). ARM also adds an EL3 level. The EL3 level is always considered secure with TrustZone and becomes responsible for switching between *secure EL* and *unsecure EL*. On figure 3.22, we represented the different ELs, showing what system program they are designed to run and which are secure. The program running in EL3 level is called *secure monitor* or *trusted firmware*. These secure ELs are all included in the *secure world*. By contrast, the *unsecure ELs* are considered in the *normal world*. In the *secure world*, MMU translation tables (section 2.3.1) and their associated TLB/page entries have an extra bit to indicate security. This bit is the NS bit which indicates, only in secure EL, that the page is assumed not secure. In that context, pages that are not flagged with NS are thus assumed secure. This secure bit is then used to label any memory request that uses this entry.

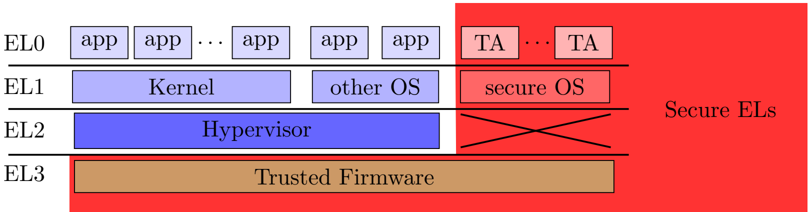


Figure 3.22: ARMv8-A Exception Levels

This secure bit is kept throughout the full memory transaction until it reaches its destination. We represented an example of a full memory transaction on figure 3.23. In this TrustZone memory model, the secure and unsecure transactions can share the same interconnect and use the secure bit to be distinguished by devices. For example, this secure bit is present in the AXI protocol in `AxPROT` vector (see the AXI manual extract on figure 3.24). The final device can then react to a possibly unauthorized memory request by checking if the secure bit is set and reacts appropriately (see the discarded transaction on figure 3.23). The key idea behind TrustZone is that the CPU starts in the bootrom running in EL3 and uses secure devices to verify the booting process and configure system elements. The bootrom then transfers execution to the OS, which runs in the *normal world*. The *secure monitor* in EL3 can still provide services to the OS using its dedicated *SystemCall* called *Secure Monitor Call* with the instruction ASM `SMC`. For example, the

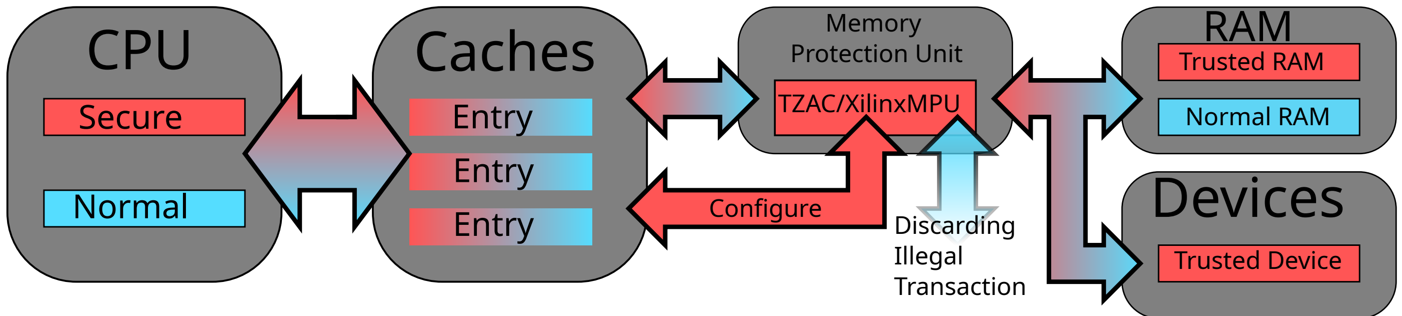


Figure 3.23: TrustZone memory model: secure labeling is propagated along the memory hierarchy.

Secure Monitor typically handles waking up and putting CPUs to sleep through `SMC`, using the Power State Control Interface (PSCI) interface.

TrustZone also specifies a set of devices that can interact with secure EL and secure transactions:

- TrustZone Memory Access (TZMA) or TrustZone Access controller (TZAC) which acts as an access control between CPUs and the system bus. It allows any device to be put behind a TrustZone memory protection dynamically.
- Generic Interrupt Controller (GIC) directly takes CPU EL into account through the GIC interface in the CPU. When accessed through its memory-mapped interfaces, security is also checked. It also allows the system to route certain interrupts to the *secure world*.
- Caches account for secure labeling by distinguishing lines that have different secure labels: This means that two lines that share the same addresses but have different secure labels are considered different by ARM caches.

Table A4-6 Protection encoding

AxPROT	Value	Function
[0]	0	Unprivileged access
	1	Privileged access
[1]	0	Secure access
	1	Non-secure access
[2]	0	Data access
	1	Instruction access

Figure 3.24: explanation of the AxPROT signal from the AXI4 norm [ARM21a]

Of course, manufacturers can implement their own secure devices, typically fuse devices.

The set of features implemented by TrustZone-enabled devices vary widely. For example, the Raspberry PI, although it supports TrustZone at an ISA level, does not provide any memory protection: *memory transactions are not blocked: All the memory and all the devices are accessible from both worlds.*

3.4.2 Platform and boot model

As we mentioned in section 3.3, gem5 boot and platform imitate ARM Versatile Express Platform (*Vexpress*). On this platform, the CPU starts in EL3. It is a secure Exception Level (EL); thus, the CPU cannot directly start in *U-Boot*[Eng23] or Linux. It needs a special bootrom called the *trusted firmware*. For ARMv8-A, there is a standard open source *trusted firmware* called *TrustedFirmware-A*[Lin23c][7]. *TrustedFirmware-A* supports *Vexpress* platform, producing a full bootrom (from system start to OS bootloader).

[7] Other platforms, like the Samsung SoC, can use proprietary *trusted firmware* to deploy their TEE (e.g.: S-boot[sof19] for Samsung Teegris[Sam20]).

This firmware contains different bootloaders called *BL1, BL2, BL31, etc.*. These elements and the boot flow between them is represented on figure 3.25. The *TrustedFirmware-A* bootrom can optionally integrate:

- **BL33**: A typical bootloader like *U-Boot*[Eng23] which run in EL2 (not in the *secure world*).
- **BL32**: A secure payload like OP-TEE *secure OS*: which will be deployed in EL1S. OP-TEE *secure OS* is the operating system responsible for the Trusted Execution Environment inside which TA runs. Besides OP-TEE, *TrustedFirmware-A* can deploy [Lin23b]: Trusty[And16], Trusted Little Kernel[Nah12] and ProvenCore[Les15]

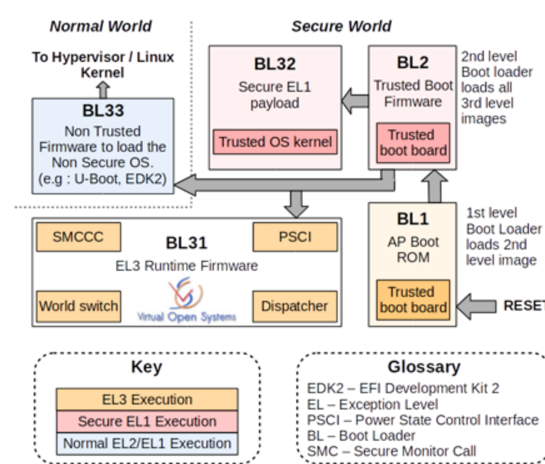


Figure 3.25: Representation of the trusted boot process for *Vexpress* platforms

The *TrustedFirmware-A* as a whole runs from a trusted RAM. Each bootloader step has different functions:

- **BL1**: *AP Boot ROM* which configure the platform to load the BL2 or perform a firmware upgrade (FWU).
- **BL2**: *Boot Trusted Firmware* which loads all the BL3X payloads, configuring memory if needed. It then passes the information about all the loaded payloads to the BL31.
- **BL31**:*EL3 Runtime Firmware*: This will remain in the trusted RAM after the boot process is finished. During boot, it initializes its own services before booting the BL32 payload that BL2 prepared. When BL32 returns, after finishing its own boot process, BL31 continues with BL33.

The BL33 for our *gem5* platform is *U-Boot*. It has to load the kernel image and Device Tree Blob (DTB), which lists all the platform properties to allow Linux to discover devices (loading their associated driver). It loads them from a classical file system (e.g., EXT2). *U-Boot* will then boot Linux with the right boot arguments, including the DTB.

The BL31, *EL3 Runtime Firmware*, is the element in the *TrustedFirmware-A*, which functions as the *secure monitor*. As mentioned before, It stays in a trusted DRAM while Linux is running. It provides services to *rich OSes* and *secure OSes*:

- SMC Calling Convention (SMCCC): Secure Monitor Calls (SMC) are instructions that trigger a synchronous abort like SVC but are routed to EL3. They are used to implement communication between *secure OSes*, *rich OSes*, and secure monitors. The SMC Calling Convention specifies how to use the SMC instruction in link with other work registers in order to send messages.
- Power State Control Interface (PSCI) is one of the possible destinations for SMCCC messages. It handles putting CPUs to sleep (using platform-specific hardware) and waking up CPUs. CPU that wakes up will start in a warm boot state in BL31 and will be redirected to Linux.

While Linux is booting, it probes for BL31 functionalities if they are mentioned in the DTB. OP-TEE is mentioned here as *firmware* node. This *firmware* node is compiled in the DTB from the source on figure 3.26. This node is necessary to enable OP-TEE functionalities in Linux. A root partition is also mentioned in the Linux command line provided by *U-Boot*. Then, Linux mounts this partition at root (/). For OP-TEE, Buildroot can be used to build a root disk image, which can then be loaded into an SD card, for example. Buildroot creates a BusyBox distribution that contains basic tools to run a C program and which can also be configured to include a wide variety of packages (compiler, editor, sound, graphics, etc.). OP-TEE integrates itself as a package for Buildroot. This ensures that the necessary libraries are available and that the necessary *daemons* are started using `init.d`. The OP-TEE packages can also integrate development tools into the generated root image.

vexpress.dts

```
firmware {
optee {
    compatible = "linaro,optee-tz";
    method = "smc";
};
};
```

Figure 3.26: Node to add to DTB which declare *SMC* as a way to access OP-TEE

3.4.3 OP-TEE software model

OP-TEE mainly relies on TrustZone to implement its Trusted Execution Environment (TEE). figure 3.27 represents how OP-TEE uses TrustZone exception level to isolate trusted applications. It utilizes EL1S and EL0S to create an execution environment that is isolated from the normal OS. The normal OS, which is Linux for OP-TEE, is called *rich OS*. Linux applications execute in the Rich Execution Environment, which uses EL0. OP-TEE's trusted applications run in the Trusted Execution Environment, which uses EL0S. The EL1S is used to run a *secure OS*, which is responsible for configuring the EL0S exception level. Like a normal OS, trusted applications running in EL0S have access to service calls through libraries such as *libc* and *libutee*. OP-TEE's *secure OS* is called *OP-TEE OS*. It is initialized by the *secure monitor* before booting Linux. After it has booted, it can only resume execution in two cases:

- Linux emits a request for OP-TEE through SMCCC.
- A secure IRQ is triggered and is routed to the *secure OS*.

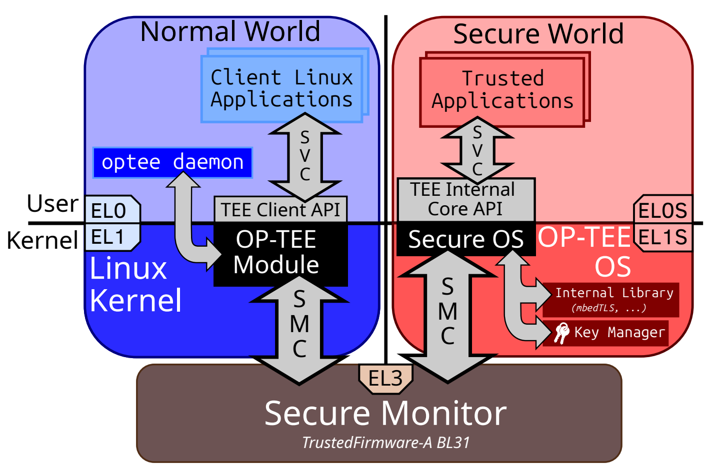


Figure 3.27: OP-TEE programming model: how TA communicates with Linux client applications

These two situations are handled by the *secure monitor* as shown on figure 3.27. To use these communication methods, our Linux kernel contains an OP-TEE module. This module provides hooks to send and receive messages to and from the *secure OS*. To do that, It uses the SMCCC module in the *secure monitor* (as declared in figure 3.26). SMCCC uses *Secure Monitor Calls*, which means that the CPU handling the request changes from EL1 to EL3. The SMCCC handler will then deliver the message using an exception return (`ERET`), but not before saving the *rich OS* execution context. CPU has now resumed in the *secure OS* EL1S, which can service the request from the *rich OS* OP-TEE module. The same exception path can be done in reverse to return the *secure OS* response to the *rich OS* message. At startup, this message exchange is used by the OP-TEE kernel module to probe the *secure OS* and determine its features. At runtime, this mechanism is mainly used by client applications running on Linux to start trusted applications and then exchange with them using commands. This message exchange is also used in reverse, from the *secure OS* to the *rich OS*, allowing the *secure OS* to use OP-TEE's Linux daemon services. Illustrated on figure 3.27, this daemon is started as a Linux service by `init.d`. Called `tee-supplicant`, it provides:

- Rich filesystem access for the *secure OS*, to load TA or simple files from the TEE. It also includes access to replay-protected memory blocks on a MMC[8]. This is a key feature to implement rollback protection as mentioned in section 2.3.3.

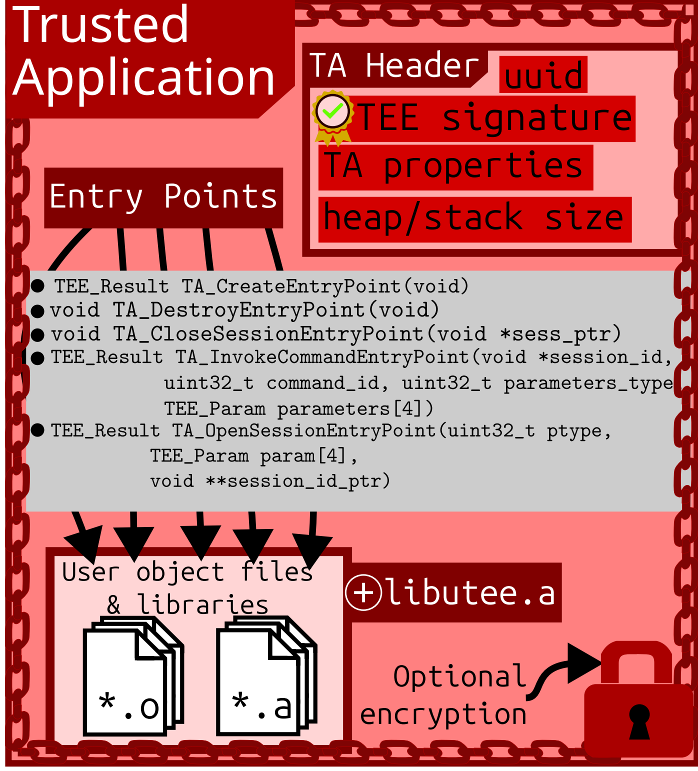


Figure 3.28: Structure of a Trusted Application (TA): User object files and libraries are bundled in a signed container. It provides the element to verify its content and load it in OP-TEE.

[8] Replay Protected Memory Blocks (RPMB) is provided as a means for a system to store data in a specific memory area in an authenticated and replay-protected manner and can only be read and written via successfully authenticated read and write accesses. The data may be overwritten by the host but can never be erased. This is implemented by the UFS specification [ASS12].

- Network access for the *secure OS*, which can then open IP sockets.

- Shared memory allocations which can only be done by the *rich OS*.

- User-defined plugins that can be integrated into the supplicant. These can then be accessed from the TA using the *GlobalPlatform* API.

This software architecture is detailed in figure 3.27. To help TA and TEE development, OP-TEE also provides a command-line tool, `xtest`, found in the *optee_test* GitHub. Following *GlobalPlatform* specifications[lea21], as shown on figure 3.27, OP-TEE provides one API for each execution environnement:

- The **TEE Client API** allows a Linux application to invoke and communicate with a trusted application identified by its UUID. This is provided to Linux applications through *libtee*.

- The **TEE Internal Core API** specifies how to implement a trusted application. It also specified what services should be provided by the TEE and their interfaces with the TA. This is provided to TA through *libutee*.

While the **TEE Client API** mainly specified communication with TA, the **TEE Internal Core API** also specified services that the TA can use:

- The **Trusted Core Framework API** defines basic application tools (e.g. memory management). It also defines TA interfaces with the client and with data directly incorporated into the TA package.

- The **Trusted Storage API for Data and Keys** provides primitive storage to manipulate complex data in the TA. Using these primitives, it is possible to handle data persistence using the *Trusted Storage* mechanism (which provides rollback protection for these objects).

- The **Cryptographic Operations API** provides Cryptographic tools for standard *Cryptography Operations*: *Hashing*, *Symmetric* and *Asymmetric cryptography*, *Message Authentication Codes(MACs)*, *Key Generation and Derivation*,... These tools are directly implemented in the TEE and are invoked using *SystemCalls*. Therefore, they can benefit from hardware primitives to improve their performance and security.

- The **TEE Time API** provides tools to access *System Time*. *System Time* can rely on secure devices, preventing any tampering from the REE. TA can also verify if the TEE supports this feature.

- The **TEE Arithmetical API** provides tools to interact with *Big Integer* (integer larger than the register size). These can leverage integrated cryptographic hardware.

- The **Peripheral and Events API**: It provides tools to interact directly with hardware devices (camera, NFC, fingerprint sensors, etc.). This feature prevents any tampering from the REE when interacting with hardware devices and performing critical operations (e.g., NFC payment).

These APIs are implemented using *secure OS* service calls in the TA. These service calls are integrated into the libraries that are incorporated in TAs. *GlobalPlatform* also specifies TA entry points, which are where a TA begins after a session was started by a Linux application or when a command is invoked during a session (these entry points are listed in figure 3.28). TAs are built using the development kit produced when compiling the OP-TEE OS. This development kit is in charge of:

- Integrating the libraries needed for the *GlobalPlatform* (presented as `libutee` in figure 3.28.)

- Linking and assembling the TA, ensuring it follows the correct binary layout (as described in figure 3.28, the different `*.o` and `*.a` files have to integrate the entry points).

- Integrate the TA in its secure package, which includes descriptors for the OP-TEE OS. It also includes a final signing, which protects the TA from tampering (these descriptors go in the TA header visualized in figure 3.28).

Each TA is identified by its UUID (Universally Unique IDentifier). To interact with TA, a Linux application has to include the OP-TEE client library which implements the *GlobalPlatform Client API*. We give an example of a client Linux source code in figure 3.29. Using the *GlobalPlatform Client API*, the client Linux application can start a session using the UUID of the TA it wants to interact with. When the session has started, the client Linux application can invoke commands in the TA. When the TA has finished processing the required command, it returns a success value, which the Linux application will receive in the form of the return value of the `TEEC_InvokeCommand` function. Trusted applications can be loaded from the *rich file system* through OP-TEE daemon (`tee-supplicant`). Otherwise, they are directly integrated in OP-TEE either inside the *secure OS* blob next to the *secure OS* binary, these are **early TAs**, or as specific UUID directly in the *secure OS*,These are **pseudo-TAs**. Pseudo-TAs's functionalities (sessions and message dispatch) are directly integrated as *secure OS* functions: No real TAs are deployed when a client opens a session.

host.c

```c
//Opening a TAA session from the HOST
res = TEEC_OpenSession(&ctx, &sess, &uuid,
    TEEC_LOGIN_PUBLIC, NULL, NULL, &eo);
//Preparing TA commands parameters
op.paramTypes = TEEC_PARAM_TYPES(
    TEEC_MEMREF_TEMP_INPUT,
    TEEC_MEMREF_TEMP_OUTPUT,
    TEEC_NONE, TEEC_NONE);
op.params[0].tmpref.buffer = inbuf;
op.params[0].tmpref.size = inbuf_len;
op.params[1].tmpref.size = 0;
//Invoking the TA command (the execution go to
↪  the secure OS)
res = TEEC_InvokeCommand(sess,
    TA_ACIPHER_CMD_ENCRYPT,
    &op, &eo);
```

Figure 3.29: Opening a TA session and launching command from Linux

The Linux application can send parameters with the command invocation, which can contain references to non-secure memory, which can be accessed from the TA in the *secure world* without issues. Of course, it is recommended for TAs to copy any data provided by the Linux applications to their own *secure* memory.

3.4.4 Refining TrustZone implementation in *gem5* to support OP-TEE

As we showed in section 3.3, the *gem5* model supports the EL3 firmware level. Supporting this level implies it supports the secure extension of ARM ISA, which is the ISA part of TrustZone. Therefore, *gem5* also supports EL1S and EL0S as expected. Moreover, *gem5* ARM MMU supports secure labeling, and it can use the NS bit to access unsecure regions from secure Exception Levels (EL). Although it seems to support TrustZone at the ISA level, TrustZone workloads designed for the *Vexpress* platform expect devices that are not present in *gem5* implementation. Thus, to support the booting of unmodified TEE binaries in *gem5*, we have made the following changes in the standard *gem5* boot flow:

- We use the *Semihosting* feature of *U-Boot* which enables the user of an embedded system to load files from the host computer. With *Semihosting*, *gem5* loads Linux kernel and DTB. DTBs for *gem5 Vexpress* are generated directly using the Python config files.
- We added a secure DRAM to the hardware configuration, and a simple secure memory system to *gem5* that makes our trusted memory refuse unsecure transactions.
- We modified the *packet protocol* (see section 2.2.2.4) inside *gem5* and allowed *GDB* memory transactions to be considered as both secure and unsecure.
- We also corrected the following bugs in *gem5* :
  - The deactivation of EL2S was not correctly handled. This stayed undetected in *gem5* as it could not boot a proper TEE.
  - The generic ARM interrupt controller was not correctly synchronized when switching between secure and unsecure ELs.

These modifications have been committed to *gem5* stable git branch 21.1. OP-TEE needs to be built into the bootrom. We use *TrustedFirmware-A* [Lin23c] for that matter. We also need a bootloader program to load the kernel, and we chose *U-Boot* [Eng23]. *U-Boot* enables us to use *Semihosting*, a feature that allows an embedded system to load files from the host computer (which is supported by *gem5*). In our case, we load the Linux image and the *Device Tree Blob* (The DTB is used by Linux to probe the platform and detect all its devices on ARMv8) using *Semihosting*. We also built a dedicated **Buildroot** disk image, which is configured to have all the OP-TEE tools. The Linux version we load features a specific driver and is provided by Linaro [Lin23a], which is the main OP-TEE maintainer. All of this process is automated in a `makefile` that imports everything from their dedicated repository and then builds the disk image,

the bootrom, and the kernel. When bootstrapping a TA, OP-TEE randomly places the TA into the address space. So we need *GDB* to grab from OP-TEE the offset to be able to debug the TA. Using our **packet protocol** fixes, we implemented TA ELF loading in *GDB*.

3.4.5 Our typical OP-TEE scenarios

To analyze security, we developed simple OP-TEE scenarios that demonstrate how OP-TEE can be used to secure critical applications. These scenarios that leverage OP-TEE functionalities were made building upon optee_example.

We have two scenarios that use a TA to implement a secure critical functionality:

Figure 3.30: `xtest` allows to install TA at run time. `/mnt/ta` is a folder containing the TA to install

- *sec-store*: A secure storage implementation (figure 3.31) that reads and writes files from the REE-filesystem using the OP-TEE Daemon and decrypts/encrypts them using an integrated key in OP-TEE. This mechanism allows OP-TEE to store files in the REE-filesystem without any risk to their integrity. These functions are integrated into a TA that a host calls to access the OP-TEE-secured files.
- *sec-sign*: A secure signing application (figure 3.32) that uses an OP-TEE crypto service to hash a message and sign it. This is implemented in a TA to which a host sends commands to configure the key and send the message to be signed. A RSA key is used to sign the message. Signing and hashing are implemented in the *GlobalPlatform* API and are used by the TA through service calls.

In *gem5*, all these scenarios start from a past-boot checkpoint: we called it *BootPoint* or *boot-checkpoint*. This checkpoint is produced using `m5 checkpoint` in a `init.d` script. The `init.d` script loads the scenario-specific bash script after the checkpoint is restored using: `m5 readfile | sh`. The scenario-specific bash script is provided by *gem5* when relaunching *gem5* and restoring from the *boot-checkpoint*. The boot phase, which takes the*boot-checkpoint*, is done using a simpler CPU model (atomic). Runtimes for the boot phase that generates the *BootPoint* are listed in table 3.1. We also listed runtimes for the previous scenario and the demo function showed in section 4.5. In the scenarios' bash script, we mount a second disk that contains all the elements necessary for the scenario. Indeed, the root disk is left unchanged between scenarios and runs because if it was modified, it would require regenerating the *boot-checkpoint*.

This disk contains the TA and a host that will invoke commands in the TA that runs in OP-TEE in the *secure world*. The TA can be installed at run times using the `xtest` (presented on figure 3.30) program from *optee_test*. The host program can then interact with the TA by starting a session using its dedicated UUID. Commands can then be invoked, the TA being responsible for decoding and dispatching them as they all arrive at the same entry point. The scenarios' disk also contains an attack implemented using the *aarch64* attack tools mentioned before in section 3.3. This attack can be configured to attack the OP-TEE scenario. As we can see, in figure 3.32, OP-TEE reverts cache side-effects when returning to the REE. This countermeasure requires our attack to run in parallel. This makes interrupt-driven attacks described against other TEEs [Rya19] [LW18]. Kou et al. [Kou+21][KOU+23] uses interrupt-driven attacks against OP-TEE but does not mention bypassing this security feature.

	Configurations
gem5	version 21.2
Software Stack	optee-3.21.0 (based on Linux v6.2-rc3)
	U-Boot : v2020.07-rc3
	ARM TrustedFirmware-A v2.7
OP-TEE noticeable flags	`CFG_CORE_WORKAROUND_NSITR_CACHE_PRIME=y` *Adds protection against a tool like Cachegrab (https://github.com/nccgroup/cachegrab), which uses non-secure interrupts to prime and later analyze the L1D, L1I and BTB caches to gain information from secure world execution.*
	`CFG_CRYPTOLIB_NAME=tomcrypt` *By default use tomcrypt as the main crypto lib providing an implementation for the API in <crypto/crypto.h> CFG_CRYPTOLIB_NAME is used as libname when compiling the library It's also possible to configure to use mbedtls instead of tomcrypt. Then the variables should be assigned as CFG_CRYPTOLIB_NAME=mbedtls*
	Runtimes
boot	2244.26s (1.042778s in simulation)
sec-sign scenario	4805.94s (4.33s in simulation)
sec-store scenario	1332.58s (4.332691s in simulation)
demo-function scenario	982.60s (4.34s in simulation)

Table 3.1: Simulation Configuration and Runtime. Times measured by *gem5*. No *GDB* acceleration is used in these runs. We run our examples on a *Intel(R) Xeon(R) Gold 6128* with 256GB of DDR4

3.4.6 Third Party IP simulation

Almost all hardware IP vendors provide a SystemC TLM model of their IPs, which can be assembled in a tool like Platform Architect [Syn21] or in an ad-hoc manner to generate virtual platforms. This is the main virtual prototyping technology in use among the industrial players. For this

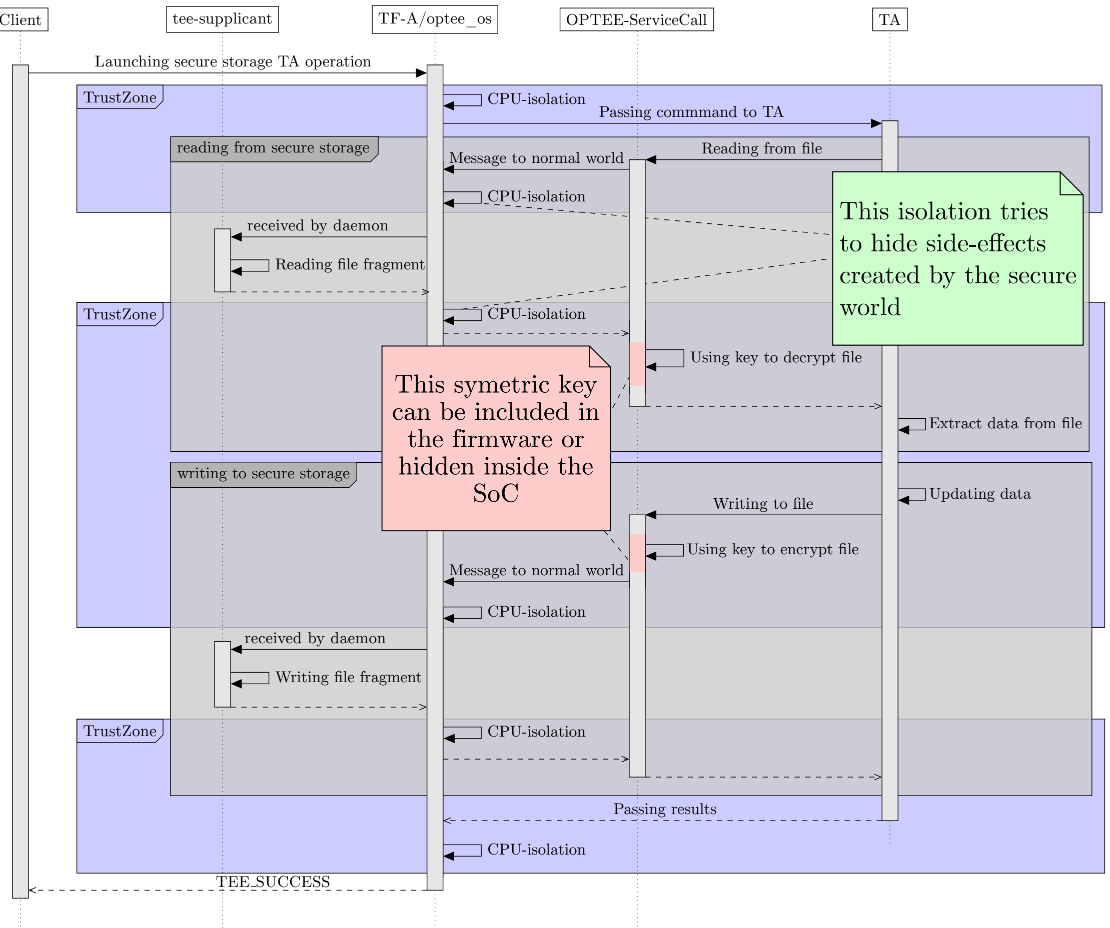


Figure 3.31: Implementation of the *sec-store* TA in OP-TEE: It tests reading and writing from secure storage. It thus uses the `tee-supplicant` to access the REE filesystem without exposing the encryption key.

reason, and due to an increasing number of security vulnerabilities emanating from 3rd party IPs [VPI15; Lad+13; DPM11; SB13] it is highly desirable to provide an interface in *gem5* to integrate them.

Fortunately, the *gem5* simulator and SystemC TLM have strikingly similar simulation mechanisms, although using different terminologies. SystemC TLM 2.0 uses *initiator* and *target* sockets while *gem5* uses *master/slaves* with *requestor* and *responder* port. SystemC TLM has three different timing modes, namely *blocking*, *non-blocking*, *debug*, which corresponds to *gem5 atomic*, *timing*, and *functional* modes. TLM also has a *DMI, Direct Memory Interface* mode, which has no counterpart in *gem5*.

We show the correspondence between the timing modes in *gem5* and TLM in figure 3.34. The *gem5* atomic mode is clearly equivalent to the *blocking transport* mode in TLM, also known as *Loosely Timed*. Since each memory transaction is modeled with a fixed latency. The *timing* mode in *gem5* is almost equivalent to TLM *non-blocking* mode, which models a memory transaction with backpressure. The *gem5 timing* uses retries to handle backpressure, whereas TLM uses four distinct phases for the same. This is also known as *approximately timed*,see ref [Men+17] for more detail.

The recent version of *gem5* [Low+20] provides two different ways of interfacing with SystemC TLM 2.0 blocks.

- *gem5 to SystemC Bridge:* The co-simulation is achieved by hosting the *gem5* executable within a standard SystemC simulation. The *gem5* is compiled as a shared library, which is called by the SystemC simulation and becomes another SystemC process. A set of translators translate the *gem5* packets to TLM 2.0 generic payload and vice-versa as shown in 3.33.

- *SystemC in gem5:* Alternatively, the TLM IPs have to be recompiled with *gem5*'s SystemC header files which represent *gem5*'s own TLM2.0 implementation. In this way, the co-simulation is achieved natively in *gem5* and can benefit from *gem5* dynamic reconfiguration mechanism using Python script.

Although the second approach is better from the *gem5* point of view, we have chosen to use the first approach since it allows integration of IPs/models in standard SystemC TLM 2.0 provided by vendors. In the future, it might be

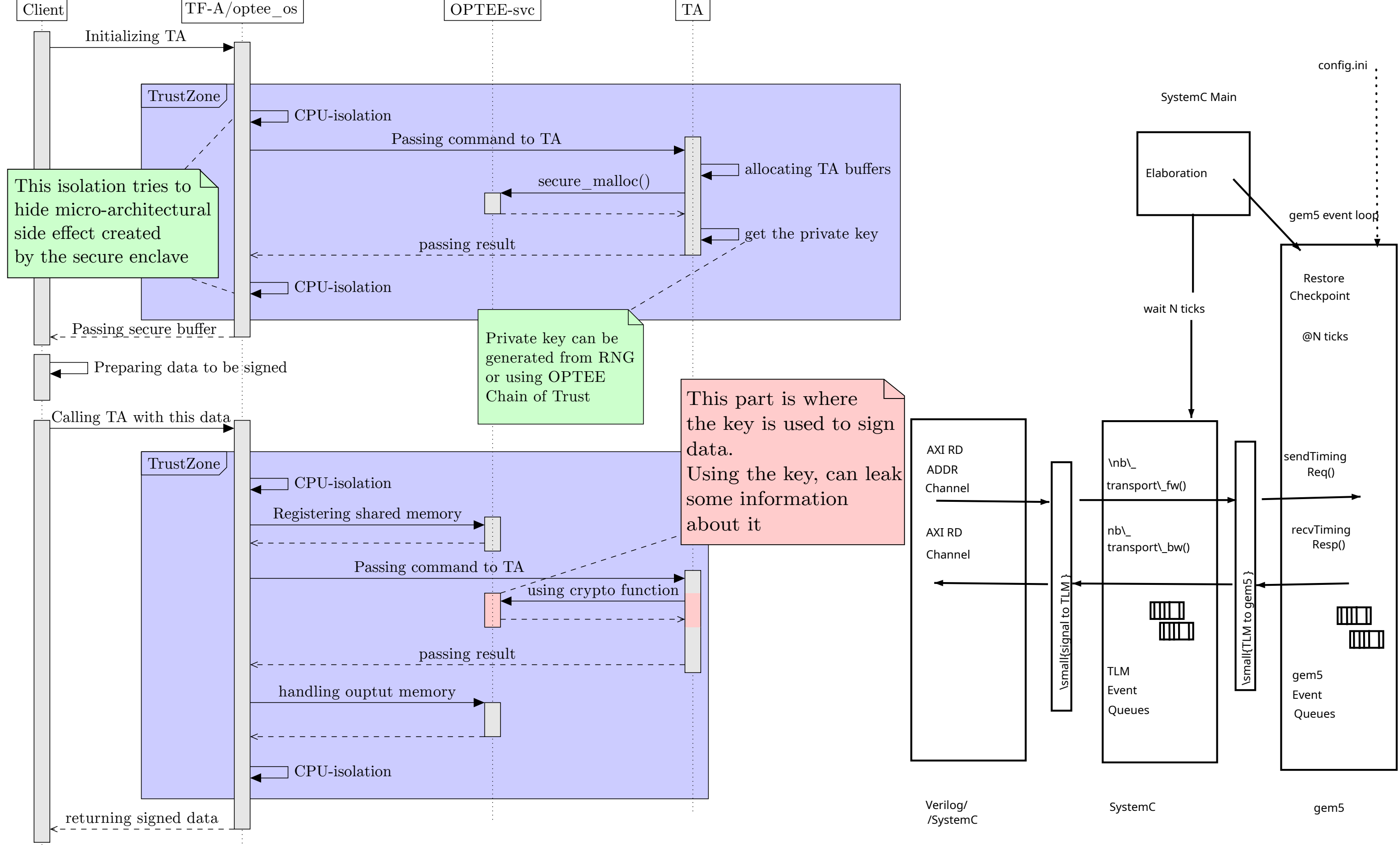


Figure 3.32: Implementation of the *sec-sign* TA in OP-TEE: the TA prepares a buffer for the client. In this buffer, the client loads a message, which is then sent back to the TA to be signed using a RSA key never exposed to the client.

Figure 3.33: *gem5* platform configuration to emulate 3rd Party IPs.

beneficial to translate these off-the-shelf IPs/Models to *gem5* native description. Following the first approach, we achieve co-simulation of 3rd-Party IPs and *gem5* booting unmodified OP-TEE. We used the process described on figure 3.33:

- We build *gem5* both in executable and shared library format.
- First, we use standard *gem5* executable with Python configuration script to boot OP-TEE and save a checkpoint. We also note the *gem5 ticks* necessary to arrive at this checkpoint (let's say N).
- Next, we launch the SystemC simulation calling the *gem5* shared library.
- We wait for N ticks to synchronize the SystemC and the checkpoint time. This phase is quite fast since it does not simulate any transactions.
- We restore the checkpoint, and at this point, we have a fully functional processor subsystem running OP-TEE and third-party IPs in SystemC TLM.

The models for third-party IPs can be obtained in the following fashion

- From the IP vendor in standard TLM 2.0 format.
- IPs in RTL format can be converted to SystemC using Verilator [Sny13]. Then, a *Signal-to-TLM* bridge can be used to integrate these IPs in the simulation (figure 3.33).

With this platform, we made an experimental attack. Using our secure storage TA, *sec-store* (see figure 3.31), we reproduce the attack on OP-TEE secure storage functionality from a SocFPGA as described in [Gro+22]. *Trusted*

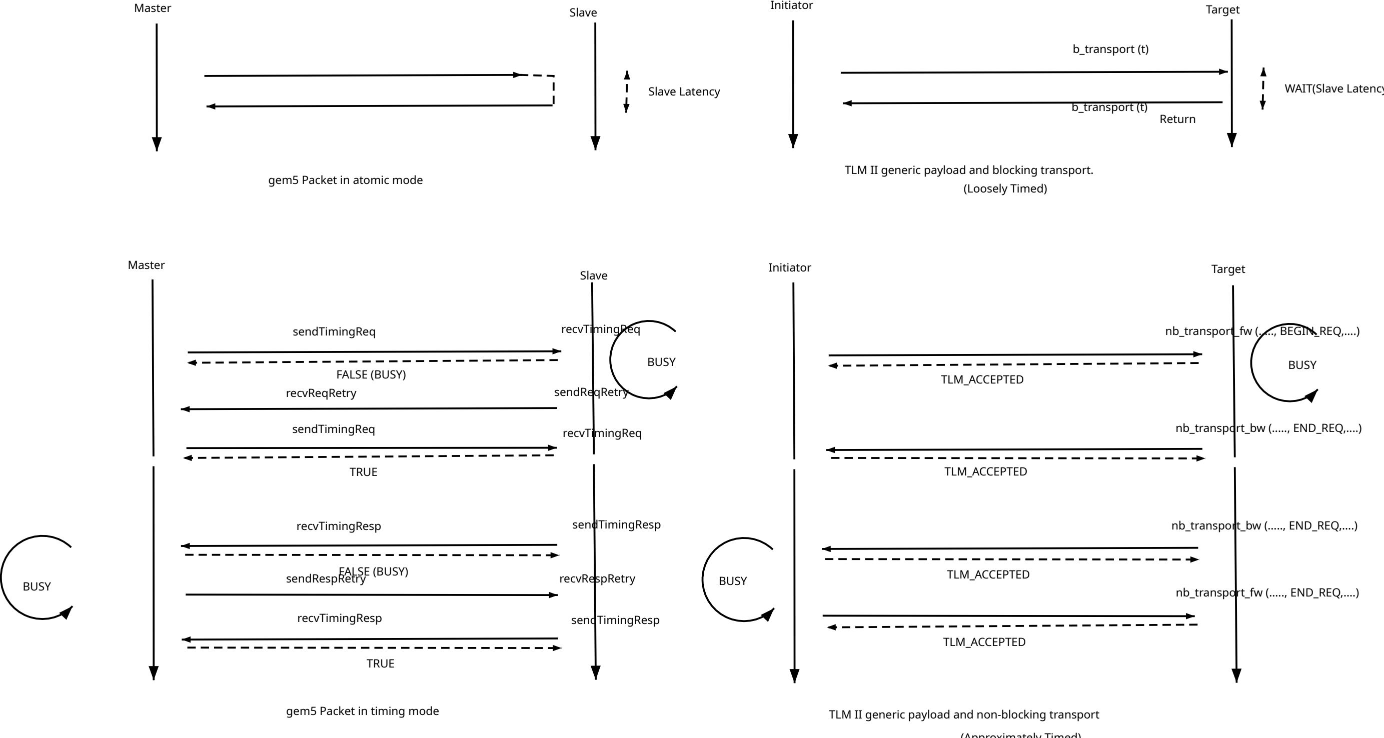


Figure 3.34: Different timing modes for *gem5* taken from [Low24b]. The atomic mode closely resembles the SystemC Loosely Timed(LT), and the timing mode resembles the approximately timed(AT).

Storage is a required functionality defined in the *GlobalPlatform TEE Internal Core API* (see section 3.4.3). The UML sequence diagram for the secure storage TA is shown in figure 3.31.

- The REE client provides an encrypted file, initialization vector, and a key ID.
- The trustlet (TA) first reads the encrypted file. It also gets the key required to decrypt the file based on key ID. The File Encryption Key (FEK) is derived from the device-specific secure storage key, which again is derived from the Hardware Unique Key (HUK) and chip identification. [Gro+22]
- Internally OP-TEE uses *libtomcrypt* [lib23] for AES, and since *libtomcrypt* uses a precomputed key schedule in memory, this structure is observable in the memory dump of the secure memory.

This is a direct memory access attack performed from the SoCFPGA ACP (Accelerator Coherency Port). Due to a bug in Xilinx FPGAs [Gro+22] ACP is able to access the secure memory. In our experiment, we model the third-party IP from FPGA using the ACP port with TLM. The attack is then performed in the following fashion:

- We assume that the attacker is able to run an IP in the FPGA accessing the ACP port, and she has access to an encrypted file.
- The goal of the attacker is to find the encryption key.
- After calling the trustlet for decryption, the attacker performs a memory dump of the *secure world* memory(32MB), which contains the precomputed key schedule.
- By following the scanning method presented in [Gro+22; Hal+08], she finds the original key.

In table 3.1, we present the *gem5* configuration parameters and the runtime of this experiment (listed under *sec-store* scenario). Since it is an in-vivo experiment, it has been performed in *FullSystem* mode.

3.5 Conclusion

When we combine work from our study of *aarch64*, our study of *gem5* tools, and finally enabling OP-TEE on a*gem5* platform, we create the first TEE-enable simulation platform. This platform required multiple fixes to *gem5* implementation and a dedicated config script, which create an OP-TEE-compatible *Vexpress* model. As we confirm using our experimentation with *aarch64* attacks on *gem5* and a Raspberry Pi, cache-timing attacks can be simulated using our *gem5* platform. Therefore, since we are assured that cache-timing attacks can be simulated we are now ready to propose an example of such attacks against OP-TEE using our *aarch64* attack tools. To find these attacks, we can create a rigorous methodology that leverage our simulated environment. To implement this methodology, we can take advantage of the instrumentation we developed for our simulation platform. It gives us access to micro-architectural information that would be otherwise inaccessible. Using our *GDB* interface, a tool developed as a *GDB-Python* script, could leverage this information to study an unmodified TA to find weaknesses that an *aarch64* cache-timing attack could then use. Another tool could also study how an *aarch64* cache-timing attack interacts with a victim TA, combining execution and micro-architectural information from both *secure world* and *normal world*. Indeed, this is the main use case for a *virtual security platform*, creating rigorous automation that automatically discovers vulnerabilities at the microarchitecture level. We also demonstrated the ability of our platform to reproduce vulnerabilities due to trojan IPs using *gem5* TLM models.

3.A Appendix

This appendix contains technical elements that can help understand how our virtual security platform works.

3.A.1 *GDB* API in *gem5*-Python

`RemoteGDB` is the *GDB-stub* class in *gem5*. `RemoteGDB` is the *GDB-stub* class in *gem5*. It is not a *SimObject* but a helper object initialized by the `Workload` SimObject. Using the *CxxMethod* in `Workload`, we implemented a *GDB* API to be used in *config files*. There are three main methods in this API:

- `sendToGdb(self,message)` : to print a message in the *GDB* terminal.
- `triggerGDB(self,signal,ctx_id,stopReason,skipped_inst)` which set up an instruction counting event which will then trigger a hardware breakpoint in *GDB* after `skipped_inst` instruction.
- `getGDBStopReason(self)` : recovering stop reason for a currently pending breakpoint. This information is not accessible by the user in *GDB*.

With these functions, we can write elaborate config scripts that handle the `GDB_MONITOR` exit events and respond to them. Figure 3.13 described message exchange between *GDB* and *gem5* to implement typical use cases for this interface. In tandem with *GDB* monitor call, the *GDB* API that can be used to:

- Interact with the simulator functions using *GDB* console (enabling *DebugFlags*, triggering checkpoints, CPU-switching...).
- Configure the *SimObjects* with `cxxmethod` or with dynamically referenced parameters.
- Extracts data from the *SimObjects*, using a `cxxmethod` and then sending the data to *GDB* using `sendToGdb`.
- Creating other type of hardware breakpoints using `triggerGDB` and `getGDBStopReason`.

All of these functionalities can be hidden in command using the Python API in *GDB* which can then use: `gdb.execute("monitor cmd",to_string=True)` It executes a monitor command and retrieves what is printed to the *GDB* console by *gem5* to place it into a variable for use by the Python script in *GDB*

3.A.2 ARM system devices in *gem5*

ARMv8-A model in *gem5* includes ARM's MMU and its automatic table walker. With the table walker, the MMU automatically goes through the pagetable structure to update the TLB entries. ARMv8-A has different granularity for pages which allows different number of page-level. Generally, a 4k-granule mode is used: which makes the smallest page size 4096 bytes. ARMv8-A also supports HUGEPAGES with 2MB and 2GB size when in 4k-granule mode. ARMv8-A page entries contain classical write, read, and execute permissions but also feature privilege control flags (PXN and UXN) which prevent execution in EL0 (unprivileged) or in EL1, EL2, and EL3 (privileged) mode. ARMv8-A has 4 types of exceptions that are handled by the exception vector (interrupt table):

- Synchronous: Exception by the MMU or the CPU (service calls, illegal instruction, translation error, ...)
- FIQ: Fast Interrupt request.
- IRQ: Normal interrupt request.

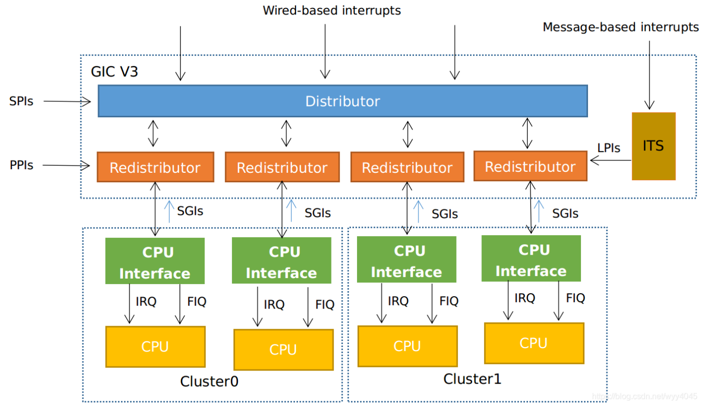


Figure 3.35: GIC architecture in *gem5* and ARMv8-A: each CPU as an interface which communicates the GICv3. There are only two types of interrupts FIQ and IRQ. The GIC distributes interrupts to the interface and controls to which CPU an interrupt will be routed and what type of CPU interrupts will be used.

- SEerror: System error/memory bus error.

These 4 types each have a dedicated handler in the exception vector. FIQ and IRQ are generated by the GIC and its local interface in each CPU. The General Interrupt Controller (GIC) is a memory-mapped device that controls: interrupt priority and routing. It also provides inter-processor interrupt. With the GIC, an interrupt can be routed to a specific CPU. It can also choose which of the FIQ or IRQ handlers is used depending on the current EL running in the CPU. It has a similar architecture in *gem5*. This different element present in both *gem5* as *SimObject* and in real hardware as IP are detailed on figure 3.35. Each CPU has its CPU interface which triggers the assigned CPU IRQ trigger signal (either FIQ or IRQ), which can depend on CPU current EL. These CPU interfaces are connected to redistributor which handle *Per Processor Interrupt (PPI)* and *Locality-specific Peripheral Interrupts (LPI)* from the *Interrupt Translation Service (ITS)*. The ITS is a memory-mapped device that allows other peripherals to trigger interrupts through the memory interconnect. Finally, the distributor routes all wired-bases interrupts, called *Shared Peripheral Interrupts(SPIs)*. It also distributes the *Software Generated Interrupts (SGI)* produced by CPU interfaces and that are used as *Inter-Processor Interrupts (IPI)*[9].

3.A.3 Timing gadget on ARM

Cache timing attack requires a way to measure execution time for a specific *LOAD* instruction. To measure time, ARM provides two timing sources:

- `CNTPCT_EL0` (or `CNTVCT_EL0`): This register is provided by the mandatory system timer. It can be accessed in EL0 as long as EL2 does not prohibit its access (This is generally the case). It generally uses a 24MHz clock.
- `PMCCNTR_EL0`: This is a register that is located in the Performance Monitor Unit (PMU). It counts the core cycle and thus the core clock. However, accessing this register requires authorization from EL1 (Linux): `PMUSERENR_EL0` has to be set to 1 in EL1. This generally requires OS-level access: which means on Android, a *rooted device*.

timings.S

```
    DSB SY
    ISB
    MRS x15, PMCCNTR_EL0 # Start time in x15
    DSB SY
    ISB
    LDR x16,[%[addr]]
    DSB SY
    ISB
    MRS x17, PMCCNTR_EL0 # End time in x17
```

Figure 3.36: Gadget to measure access time: using memory barriers (`DSB SY`) and instruction barriers `ISB`, the execution time of a single `LDR` instruction is measured.

As an alternative to native timing sources, we could use a different thread to increment a counter[LW18]. As `PMCCNTR` is more precise than the `CNTPCT`, we chose to use the former to measure time. To measure the time taken by a *LOAD* instruction (`LDR`), we use the code as illustrated in figure 3.36. This code uses memory barriers (`DSB SY`) and instruction barriers (`ISB`) to ensure that the *LOAD* instruction is executed after the first time stamp is taken (with `MRS x15, PMCCNTR_EL0`) and before the second time stamp is taken (`MRS x17, PMCCNTR_EL0`). This primitive is used in our implementation of *Flush+Reload* and *Prime+Probe*.

[9]For example, signals between processes running in different CPU use IPIs, in Linux.

Chapter 4

TEE-Time: Simulating to get security insights

Contents

4.1 Introduction **62**
4.2 Key issues **62**
4.2.1 Cache timing attacks on Trusted Execution Environments 63
4.2.2 Exploring attack complexity 63
4.3 TEE-Time methodology **64**
4.3.1 Overview of TEE-Time process 64
4.3.2 *Key Detectable States* 65
4.3.2.1 VictimScan policy: 1hit 66
4.3.2.2 VictimScan policy: nhit 66
4.3.2.3 VictimScan policy: nhit_inclusive 66
4.3.3 Ranking methodology 67
4.3.4 Attack configuration and *Key Detectable States* 67
4.4 TEE-Time implementation **69**
4.4.1 Instrumenting the attack scenario 69
4.4.2 Dedicated *GDB* scripts 69
4.4.2.1 *VictimScan* 70
4.4.2.2 *Attack Monitoring* 73
4.5 Example: demo cryptographic function **74**
4.5.1 Demo: *VictimScan* 74
4.5.2 Demo: *Attack Monitoring* 75
4.5.3 TEE-Time: Code coverage 76
4.6 Attack against RSA signing in OP-TEE **76**
4.6.1 *mbedTLS* bignum exponentiation 76
4.6.2 RSA: *VictimScan* 78
4.6.3 RSA: *Attack Monitoring* 78
4.7 Conclusion **80**

4.1 Introduction

In this part is detailed how we use *gem5* to study attackability of applications running in a Trusted Execution Environment. The tool that analyzes this attack scenario is called TEE-Time. It consists of two phases: **VictimScan** and **Attack Monitoring**. *VictimScan* extracts information from *gem5* while it is running a potentially weak scenario: A scenario where an attack running in parallel could spy on a victim program. *VictimScan* will then produce a report using the data gathered during the execution of the victim process. This report documents the attackability of a specific implementation of a cryptographic algorithm. As illustrated on figure 4.1, *VictimScan* uses a ranking methodology based on cache dumps to evaluate these properties. After that, *Attack Monitoring* can use the *VictimScan* report to verify if a simulated attack build upon *VictimScan* findings produces the expected results.

4.2 Key issues

To carry on cache timing attacks or, more generally, any side-channel attacks, We have to identify two things:

- A small code extract that can be linked to a secret in our victim algorithms, generally the key.
- How this small code can be detected using the shared medium between attacker and victim. In cache timing attacks, this shared medium is generally the last-level cache.

The first part has been widely explored in the literature: *cache static analyzers* can determine if a certain algorithm implementation is not **Time constant**. For example, CacheAudit[Doy+15] uses a formal model to study information propagation in the algorithm implementation. With a cache analyzer, it is possible to study in-vitro potential victims and find out if they are attackable.

The second part, identifying how to detect these leaks, is harder to generalize. In fact, this largely depends on the platforms and how the CPU and caches fetch lines. For example, *AutoLock*[Gre+17] complicates the victim's leak detection, although it remains possible. In addition, we have to account for other sources of noise that would consistently obscure the data. Therefore, fine-tuning an attack to work on a specific platform requires a specific model. This leads attack demonstrations to be hardly scalable in more practical use cases. These issues culminate in Trusted Execution Environments which prevent most of the tricks used to deploy attacks more easily:

- Sharing CPU between *victim* and *attacker*: The Trusted Execution Environment will preempt the core for its execution and only release it after it is finished and has flushed all caches.

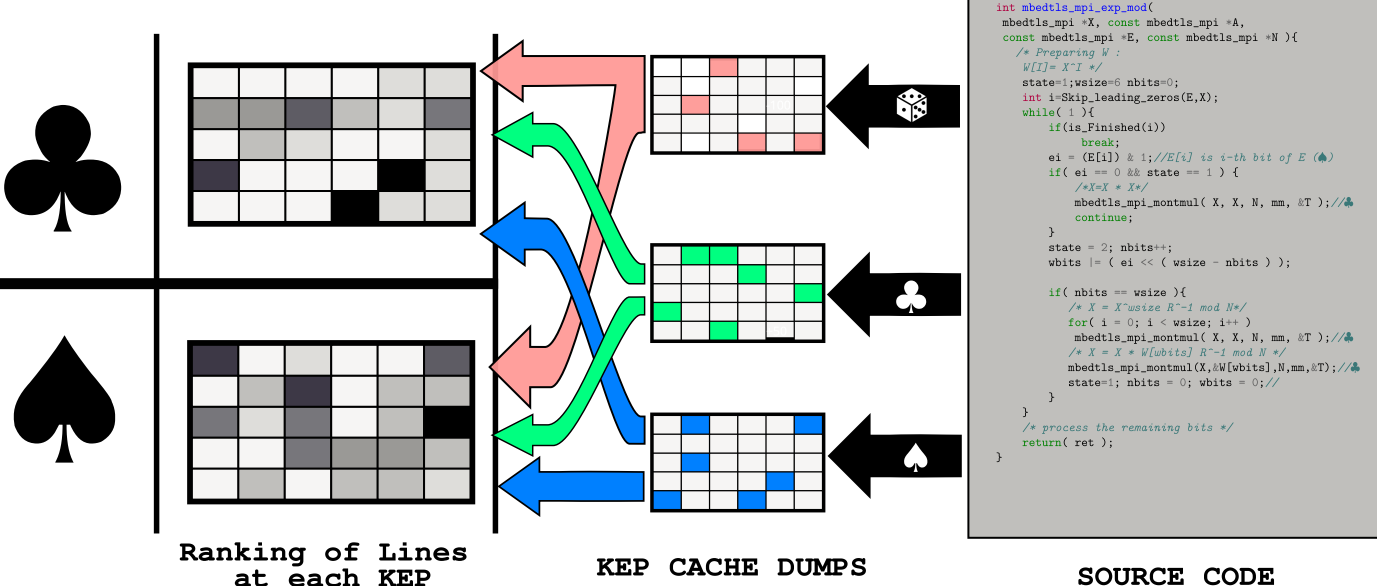


Figure 4.1: Illustration of the *VictimScan* component in TEE-Time. Using simulator access to cache dump, *VictimScan* evaluates vulnerabilities using the source code.

- Being able to gather multiple traces without knowing what we are searching for.
- Being able to study the victim directly on real hardware: TEE can block the debugger on ARMv8-A.

All these elements without making attack impossible, largely hinder attack deployment, and therefore, security studies on real hardware.

4.2.1 Cache timing attacks on Trusted Execution Environments

For our *threat model*, we assume that our attacker does not have access to the TEE, it can only run applications in Linux. This consideration already prevents the attacker from using *Flush+Reload* as caches consider lines with different security attributes as being different. Therefore, a *rich OS* application can not have cache 'hits' for a *trusted application* address. Moreover, when transitioning from *rich OS* to *secure OS*, all the caches are flushed. Thus, only a *Prime+Probe* attack is possible (see section 3.4.4). This distinguishes our threat model from other cache-attack in literature [Rya19], [LW18], [Kou+21] and [KOU+23] which uses the same CPU core shared between *attack* and *victim*, leveraging interrupts to preempt OP-TEE while it is running and execute the attack on the same CPU. To measure cache-side effects during the execution of OP-TEE, our *Prime+Probe* attack has to run in parallel to our victim while sharing the last-level caches. To be able to control which CPU the TEE runs in, the attacker has to control on which thread the host application invoking the TA runs in Linux. This assumes being able to call this application or just filling CPU occupancies with dummy thread to constrain the victim to the only remaining CPU.

For this reason, our attack scenario is a cross-core *Prime+Probe* attack (as shown in figure 4.2). In this type of attack, victim and attacker run on different CPUs and the last-level cache (here the L2) is used as the side-channel. Therefore, the *Prime+Probe* set has to fill the last level cache and thus contains at least as many entries as its associativity.

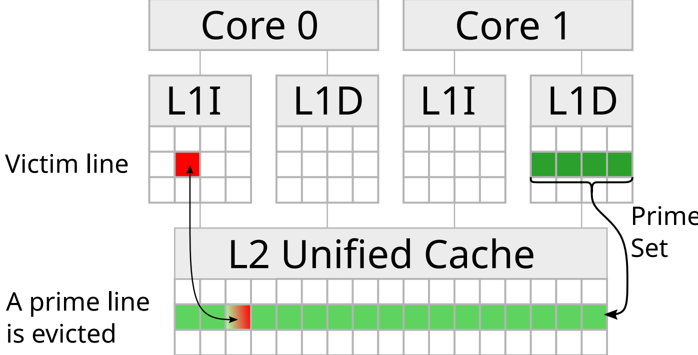


Figure 4.2: Cross-core *Prime+Probe* attacks: an attack running in CPU1 is trying to attack the TEE in CPU0.

In our scenarios, the attack program is run while a *host* first installs a TA (using `xtest --install-ta`) and then opens a session with the TA it just installed. In that session, the host client app in Linux will use the TA by sending commands for a critical operation that we want to attack using our cross-core *Prime+Probe* attack. The attack program, which was waiting in the background, starts monitoring caches using *Prime+Probe* just before the *victim* execution enters the trusted OS area. To do that, the Linux *host* is instrumented to communicate with the attack. It signals the attack when it is about to start the critical TA commands. This is possible as the Linux *host* application can be modified by the attacker, whereas the TA is unalterable. Indeed, this threat model introduces complexity(synchronization, memory placement,...) that can only be resolved through an automatic cache analyzer like TEE-Time.

4.2.2 Exploring attack complexity

To deploy our *Prime+Probe* attack, we have to choose which cache set to attack. A cryptographic algorithm is, generally, the target for our attack. So, our goal is to recover the key (or secret) using differences in timing. This problem is shown in figure 4.3, which illustrates how a *Prime+Probe* attack can be used to detect a branch. To do so, we have to choose the right cache set to attack in order to effectively detect the branch taken by the victim. This cache set has to be accessed by an operation (symbolized as gears) happening in the branch. By monitoring the cache set, the attack produces a signal that, if done correctly, only corresponds to the branch being taken. If this cache set is also part of an operation footprint outside the branch, it creates additional signals that have to be ignored. As we can see on the red plot in figure 4.3 if we do not choose carefully the monitored cache set, the cache timing traces can contain additional signals that do not correspond to the branch. This red signal is incorrect as it presents when the branch is not taken and when the yellow gear is used. On the other hand, the green signal is only present when the branch is taken.

A cryptographic algorithm that is leaking information through cache and thus attackable, is considered **non-constant time**. To be **constant time**, an algorithm has to:

- Take all the branches independently of the key.
- Do not use any key-offset'ed memory access.

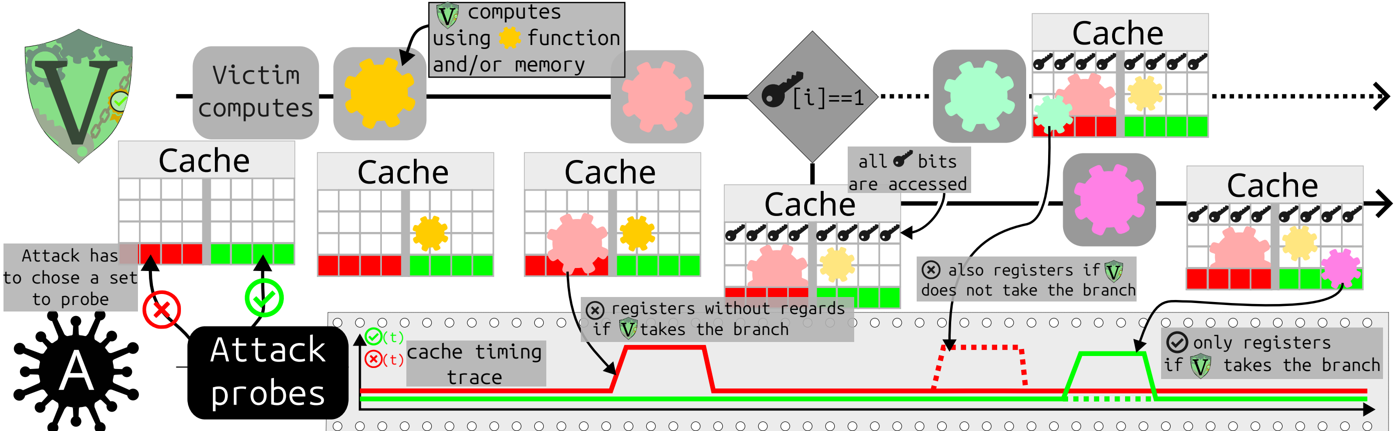


Figure 4.3: An attacker uses a cache timing attack to determine which branch a victim takes. To do that, it has to choose the right line to detect only an operation that happens in that branch.

To find these sources of weaknesses, a static cache analyzer can be used (e.g., *FlowTracker*[RQA16], *CacheAudit*[Doy+15]). These tools rely on static analysis to find the vulnerabilities, propagating information through the program to find non-time constant parts. Some static cache analyzers, with the help of cache model can also suggest the associated cache set. A dynamic cache analyzer, like [Ira+17], can also detect cache sets responsible for a cache leak. They rely generally on differential analysis to find differences between two runs with different keys.

Another possibility to find these cache vulnerabilities is to study arbitrary attack traces for different cryptographic keys. Called template attacks, they study a large amount of traces to propose correct cache sets associated with points-of-interest in said traces. These points-of-interest correspond to time instants in traces where differences in timing represent bits values for a key (red and green bumps on the figure 4.3).

To explore attack complexity and find the right configuration to attack a specific cryptographic algorithm, we can leverage using our platform described in Chapter 3 and its precise *CPU and cache model*. But we can take advantage of our knowledge of the algorithm and our access to a micro-architectural debugger in a secure environment. Bypassing the need to produce millions of traces, we can directly propose points-of-interest based, for example, on code segments we want to detect (like the branches on figure 4.3). Through automation, we can monitor and trace an algorithm's micro-architectural states without any human interventions. This last point solves the problem with the largely slower execution speed that comes with simulation: with basic assumption on the algorithm leakage model, we can gather as much information in a single unattended run as what could be done in millions of cache template profiling runs.

This is our tool, TEE-Time. It deduces from potentially non-constant time section, which cache set to attack and what signal to expect for the non-constant time section we want to detect.

4.3 TEE-Time methodology

TEE-Time is first designed as a methodology that leverages a simulation environment while trying to overcome the limitations caused by said simulation environment. In this section, we present the core concepts and model that shape the TEE-Time methodology.

4.3.1 Overview of TEE-Time process

TEE-time uses a three-step methodology (see figure 4.4) to propose a reasonable attack that can be run on a real platform:

- **I** : **Choosing KEPs**: Based on the knowledge of the underlying algorithm, we propose points of interest in the algorithm called **Key Execution Point** (*KEP*) which potentially leaks information about the key. These points are regrouped in *KEP classes* that denote similar operations.

- **II** : **VictimScan**: During an automatic/interactive debug session, the victim running in *gem5* is analyzed to extract key features associated with each *class of KEPs*, with respect to an attack information model. We call this information model a *VictimScan policy*. These key features are called **Key Detectable States (KDS)**, and can be used to configure a cache timing attack.

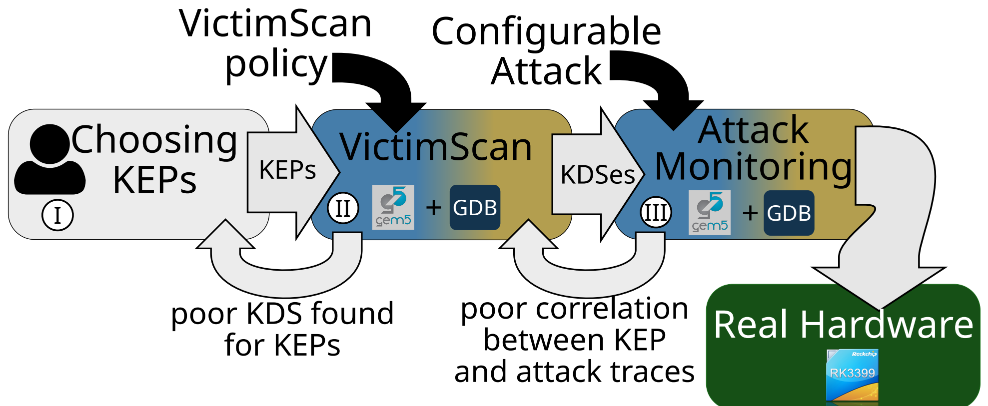


Figure 4.4: Overview of TEE-Time: With this process we use our simulation platform (*gem5*+*GDB*) to craft an attack that we can test on real hardware.

III : **Attack Monitoring**: During an automatic interactive debug session, the attack scenario (victim + attack) running in *gem5* is monitored to supplement the traces of an attack with victim *KEP* execution data. These information can then be processed to confirm the correlation between *KEPs* and attack results.

TEE-time introduces two concepts that serve at the interface between our three steps:

- **Key Execution Points (KEP)**: They mark the non-constant time section we want to detect. They symbolize potential cache timing weaknesses associated with an execution path and/or a specific variable value. Representing point-of-interest directly in code or in disassembled binaries, We denote them as ♠,♥,♣, [$\mathbb{S}$] or [$\mathbb{M}$]. *KEPs* sharing the same label are considered in the same class. *KEPs* in the same class represent similar information at the algorithm level: e.g. *which S-box is used for AES*, *which operation is performed by the square and multiply algorithm*, etc. Often symbolized as breakpoints, *KEPs* are chosen by the user to control the behavior of *VictimScan* during step I.

- **Key Detectable States (KDS)**: produced by *VictimScan* from a set of *KEPs*, they represent key features or properties of the cache states associated with each *class of KEPs*. A *KDS* is, therefore, a property of a cache state, which directly corresponds to a specific attack outputting a characteristic signal when this cache state occurs. Therefore, with a *KDS*, an attack can be tuned to recover its associated signals without interferences from different *KDS*. Indeed, different attacks and cache replacement policies have different associated *KDS* definitions and sets. This definition is provided in a *VictimScan policy* on figure 4.4, which has to correspond to the attack provided in step III.

KDS are extracted from cache states in the vicinities of *KEPs*. These cache states are acquired using cache dumps or by tracking the cache state for the length of the *KEP* segment. These different possible behaviors are controlled by a *KEP toolbox*, detailed in section 4.4.2.1.

4.3.2 *Key Detectable States*

To define what are the key features in cache state, we developed a model for the relation between cache states and attack traces. With this model, we exploit the dumps produced on each *KEPs* to produce a set of *Key Detectable States (KDS)*. This model is integrated into *VictimScan* as different *VictimScan policies*. Let $\mathbb{D}$ be the cache dump entry set. An element d of the cache dump set $\mathbb{D}$ represents the state of a specific way w from a cache set with index i occupied by a line which corresponds to the address a is a 3-tuple as follows:

$$d = (i, w, a) \in \mathbb{D} \tag{4.1}$$

Each dump produced, D_u, is a set of $d \in \mathbb{D}$ which we call $\mathbb{U}$, the set of cache dumps, such that $D_u \in \mathbb{U}$. Therefore, we have:

$$\forall D_u \in \mathbb{U}, D_u = \{d_1, d_2, d_3, ..., d_n\} \text{ with } (d_1, d_2, d_3, ..., d_n) \in \mathbb{D}^n \tag{4.2}$$

A *VictimScan policy* x is thus a function f_x of $\mathbb{U}$ to a set of elements from a simpler set called $\mathbb{K}_x$, the key detectable state set.

$$f_x : D_u \in \mathbb{U} \mapsto \{k_1, k_2, ..., k_n\} \text{ with } (k_1, k_2, ..., k_m) \in \mathbb{K}_x^m \tag{4.3}$$

Each element of $k \in \mathbb{K}_x$ can be distinguished using an attack $A_k \in \mathbb{A}_x$, with $\mathbb{A}_x$ being the set of attacks that can be configured to detect an element of $\mathbb{K}_x$. This attack A_k produces traces along the execution of the victim, influenced by the shared cache state. For a given point of execution p, we can define the result of the attack:

$$A_k : p \mapsto \mathbb{T}_A \tag{4.4}$$

$\mathbb{T}_A$ is the A attack output space, such as $A_k(p) \in \mathbb{T}_A$ is the output of the attack for a point p. An attack trace is therefore a list of execution points $\{p_1, p_2, ...\}$, and attack result $\{A_k(p_1), A_k(p_2), ...\}$. In that regard, key execution points ($p_{\spadesuit 1}$) are specific points in the execution that can be organized into classes that the attacker wants to distinguish using the output of the attack ($A_k(p_{\spadesuit 1})$).

KDS Property: Given two *KEPs*, $p_\spadesuit$ from *KEP* class $\spadesuit$ and $p_\heartsuit$ from *KEP* class $\heartsuit$ that each produced a dump, $D_{u\spadesuit}$ and $D_{u\heartsuit}$, $A_k \in \mathbb{A}_x$ with f_x its associated *VictimScan policy*, is equivalent to the following:

$$\forall k' \in \mathbb{K}_x, k' \in f_x(D_{u\spadesuit}) \text{ and } k' \notin f_x(D_{u\heartsuit}) \;\Rightarrow\; A_{k'}(p_\spadesuit) \neq A_{k'}(p_\heartsuit). \tag{4.5}$$

if the property 4.5 is true for an attack A_k, and therefore $A_k \in \mathbb{A}_x$, it means that it can be used to detect *KEPs* using *KDS* k from the image of their dump through the policy x.

With these settings, the *VictimScan* ranking proposes a set of attack configuration (A_k), one for each *KEP* class which can be used in tandem to detect and distinguish *KEPs*. To represent our cache timing attacks, we define the following *VictimScan policies*.

4.3.2.1 VictimScan policy: 1hit

1hit is the simplest *VictimScan policy*. In [FDC23] and [FDC24b], this type of *KDS* was called *Key Cache Lines (KCL)* and represented the minimum configuration for a *Prime+Probe* attack. With this policy, *KDS* are only made of non-empty cache indices (0x1, 0x23, ...) with no regard for the number of occupied ways or set occupancy. Their associated f_{1hit} function is as follows:

$$\begin{aligned} &f_{\text{1hit}} : D_u \in \mathbb{U} \mapsto \{k, ...\} \\ &f_{\text{1hit}}(D_u) = \{(i) | \exists (w, a), (i, w, a) \in D_u\} \end{aligned} \tag{4.6}$$

The attacks $A^{\text{1hit}}_{(i)}$ for this policy are attacks which can distinguish between *hit* and *miss* for a specific set with index i.

4.3.2.2 VictimScan policy: nhit

nhit is the second *VictimScan policy* which takes into account set occupancy. For each cache dump, the *KDS*, that it produces are composed of: a cache index and the number of occupied ways for this index ((0x1,1), (0x23,4), ...). Given $\mathbb{O}_i(D_u) = \{w | \exists a, (i, w, a) \in D_u\}$, their associated f_{nhit} function is as follows:

$$\begin{aligned} &f_{\text{nhit}} : D_u \in \mathbb{U} \mapsto \{k, ...\} \\ &f_{\text{nhit}}(D_u) = \{(i, card(\mathbb{O}_i(D_u))) | card(\mathbb{O}_i(D_u)) > 1\} \end{aligned} \tag{4.7}$$

The attacks $A^{\text{nhit}}_{(i,o)}$ for this policy are attacks which can distinguish between different occupancies o (the number of ways filled) for a set of index i.

4.3.2.3 VictimScan policy: nhit_inclusive

This is a derived *VictimScan* policy from the *nhit* policy. For each cache dump, its *KDS* have the same definition as *nhit*. However, for each *nhit KDS*, additional *KDS* are added for included occupancies: For (0x23,4), (0x23,3), (0x23,2), and(0x23,1) are also emitted. Their associated $f_{\text{nhit_inclusive}}$ function is as follows:

$$\begin{aligned} &f_{\text{nhit_inclusive}} : D_u \in \mathbb{U} \mapsto \{k, ...\} \\ &f_{\text{nhit_inclusive}}(D_u) = \{(i, o_{th}) | (i, w) \in f_{\text{nhit}}(D_u), o_{th} \in [1, w]\} \end{aligned} \tag{4.8}$$

The attacks $A^{\text{nhit_inclusive}}_{(i,o_{th})}$ can be seen as a variation on $A^{\text{nhit}}_{(i,o)}$. This means that an attack $A^{\text{nhit_inclusive}}_{(i,o_{th})}$ which has a number of filled ways for set index i of at least an occupancy threshold o_{th}, can be defined using a sum of $A^{\text{nhit}}_{(i,w)}$, with assoc the cache associativity:

$$A^{\text{nhit_inclusive}}_{(i,o_{th})} = \sum_{w=o_{th}}^{\text{assoc}-1} A^{\text{nhit}}_{(i,w)} \tag{4.9}$$

4.3.3 Ranking methodology

With *KDS* isolated for each *KEP*'s dump, we want to determine which *KDS* is ideal to detect a *class of KEPs*. Our goal is thus to identify which *KDS* are more likely to be triggered around a *KEP* and are less likely to be triggered by other *KEPs* or randomly. If no such *KDS* can be found, we can safely declare the associated *KEP* is not detectable by *Prime+Probe* attacks. For that, we use a simple scoring system that is computed along the execution, i.e., each time a new cache dump is collected on a *KEP*.

Given $\mathbb{S}$ is the set of all *KEPs*: $\mathbb{S} = \{\spadesuit, \heartsuit\}$

Given $h_x(k)$ which is the number of times a *KDS* k is present in cache dump corresponding to a *KEP* class x.

Given w_x which is the number of times a *KEP* of class x has been found and thus, a cache dump has been made.

This score function presented in equation 4.10 has three components:

- $\frac{h_x(k)}{w_x}$: The *KDS* which are present in the cache dump during the *KEP* gets a positive score, normalized by the number of times the associated *KEP* has been triggered.
- $\frac{h_{\text{⚂}}(k)}{w_{\text{⚂}}}$: The *KDS* present in random dump get a negative score, normalized by the number of times this random dump has been done.
- $\sum\limits_{s \in \mathbb{S}, s \neq x} \frac{h_s(k)}{w_x \times card(\mathbb{S})}$: is the *conflict contribution*: The *KDS* which are found in other *KEPs* get a negative score and are normalized by their related *KEP*'s trigger count and by the number of *KEPs*' classes.

Overall, the score function is given by:

$$\text{score}_x(k) = \frac{h_x(k)}{w_x} - \frac{h_{\text{⚂}}(k)}{w_{\text{⚂}}} - \sum_{s \in \mathbb{S}, s \neq x} \frac{h_s(k)}{w_x \times card(\mathbb{S})} \tag{4.10}$$

This score is also generally indicative of how much a **class of KEPs** is identifiable using a cache timing attack configured with *KDS* k. A negative score indicates that the **class of KEPs** cannot be identified.

4.3.4 Attack configuration and *Key Detectable States*

This score function can be used to rank *KDS* for each *class of KEPs*. The highest ranked *KDS* for each *class of KEPs* can then be used to configure an attack, in our case, a cross-core *Prime+Probe* attack. This attack is then supposed to detect *KEP* using the signal associated with its *KDS*. Depending on the *KDS*, different attacks can be configured to detect it.

Our different *KDS* definitions in section 4.3.2 account in fact for the different ways of using a *Prime+Probe* attack. A *Prime+Probe* attack is configured using a prime set index i, which corresponds to a *1hit KDS* (k_i) or the first element of *nhit KDS* ($k_{(i,o)}$). Indeed, the output of *Prime+Probe* for each execution point p is the access time for each element of the prime set. Given *assoc* the last-level cache associativity, we have:

$$A_{k_i}^{\text{Prime+Probe}}(p) = \underbrace{\{t_0, t_1, .., t_{\text{assoc}-1}\}}_{\text{assoc}} \tag{4.11}$$

Making abstraction of noise, we can propose a model for these expected timing results depending on the probing direction.

When probing forward, due to self-eviction, all the access timing for the prime set will have the same value, either t_{hit} or t_{miss}. Thus, *Prime+Probe* forward can output only two possible value T_{miss} or T_{hit}

$$T_{miss} = \underbrace{\{t_{miss}, t_{miss}, .., t_{miss}\}}_{\text{assoc}} \qquad T_{hit} = \underbrace{\{t_{hit}, t_{hit}, .., t_{hit}\}}_{\text{assoc}} \tag{4.12}$$

This means that when probing forward, the *Prime+Probe* attack can only reliably detect between a set being empty and being filled with one or more entries. This behavior links the *Prime+Probe* attack to the *1hit VictimScan policy*. In that context, *KDS* are made of only a single index. And if we take two points of execution p_1 and p_2 whose dumps only differ by a single cache line in an otherwise empty cache set with index i. We have $k_i \in f_{\text{1hit}}(\text{Dump}_{p_1})$ and $k_i \notin f_{\text{1hit}}(\text{Dump}_{p_2})$ and our attack $A_{k_i}^{\text{Prime+Probe-forward}}$ produce the following results:

$$\begin{aligned} A_{k_i}^{\text{Prime+Probe-forward}}(p_1) &= T_{miss} \\ A_{k_i}^{\text{Prime+Probe-forward}}(p_2) &= T_{hit} \end{aligned} \tag{4.13}$$

Therefore, we have $A_k^{\text{Prime+Probe-forward}} \in \mathbb{A}_{\text{1hit}}$. This property is still valid when using the sum of timing over the prime set. In that, case we have $\sum T_{miss} = \text{assoc} \times t_{miss}$ and $\sum T_{hit} = \text{assoc} \times t_{hit}$. Therefore, we can plot only the sum of the timing without losing information.

On the other hand, for *Prime+Probe* in reverse, timing values can differ between entries in the set. Each entry can be a *hit* or a *miss*. However, due to the LRU (Least Recently Used) cache replacement policy and the direction of probing, the victim program evicts elements of the prime set in order, from the least recently probed to the most recently probed. This results in all entries after the first *miss* being *misses* because the prime set is evicted from the extremity where the last probe started. Consequently, the number of prime set entries evicted is directly linked to the number of occupied cache ways o by the victim for their associated index. Thus, we can define:

$$T_{hit\text{-}miss}(o) = \{\underbrace{t_{hit}, .., t_{hit}}_{\text{assoc}-o}, \underbrace{t_{miss}, .., t_{miss}}_{o}\} \tag{4.14}$$

For $o \in [0, \text{assoc} - 1[$, $T_{hit\text{-}miss}(o)$ represents all possible outputs for the *Prime+Probe* reverse attack ($A^{\text{Prime+Probe-reverse}}_{k_i}$). Each of these outputs is linked with a number of occupied ways o for the cache index i which was used to allocate the prime set. The attack has, therefore, a different output for each occupancy of the cache set. This behavior links the *Prime+Probe* reverse attack to the *nhit VictimScan policy*. In that context, *KDS* made of index i and occupancy o correspond to the attack $A^{\text{Prime+Probe-reverse}}_{k_i}$ outputting $T_{hit\text{-}miss}(o)$.

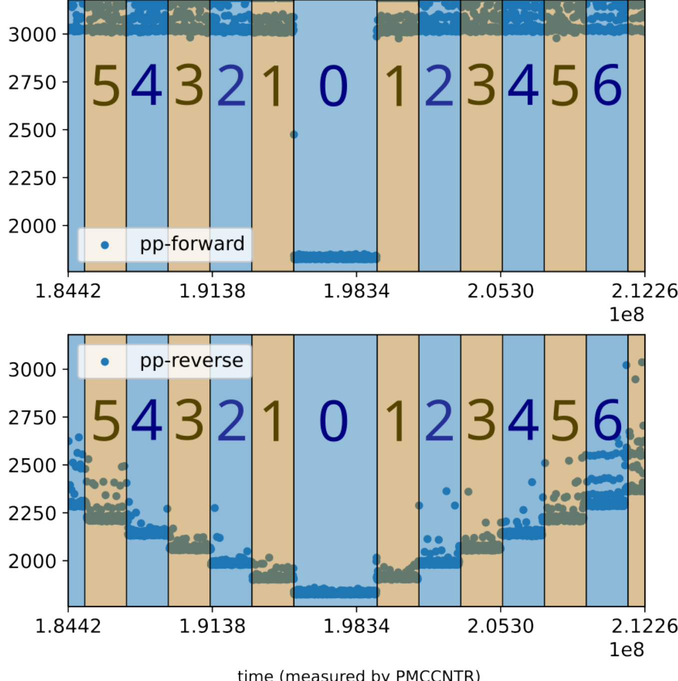


Figure 4.5: *Prime+Probe* directions: above are *Prime+Probe* forward and below are *Prime+Probe* reverse. The victim uses cache occupancy, indicated as colored rectangles, to send a stair signal clearly visible on *pp-reverse*.

Given two execution points, p_1 and p_2, whose dumps differ by only a single cache line in cache set i. In p_2, this cache line occupies an additional way o, assuming that all ways from 0 to $o-1$ are already filled. We have: $k_{(i,o)} \in f_{\text{nhit}}(\text{Dump}(p_1))$ and $k_{(i,o)} \notin f_{\text{nhit}}(\text{Dump}(p_2))$ and our attack $A^{\text{Prime+Probe-reverse}}_{k_{(i,o)}} = A^{\text{Prime+Probe-reverse}}_{k_i}$ produces the following results:

$$\begin{aligned} A^{\text{Prime+Probe-reverse}}_{k_{(i,o)}}(p_1) &= A^{\text{Prime+Probe-reverse}}_{k_i}(p_1) = T_{hit\text{-}miss}(o) \\ A^{\text{Prime+Probe-reverse}}_{k_{(i,o)}}(p_2) &= A^{\text{Prime+Probe-reverse}}_{k_i}(p_2) = T_{hit\text{-}miss}(o-1) \end{aligned} \tag{4.15}$$

Therefore, we have $A^{\text{Prime+Probe-reverse}}_{k} \in \mathbb{A}_{\text{nhit}}$. This property is still valid when using the sum of timing over the prime set. In that case, we have:

$$\begin{aligned} \sum T_{hit\text{-}miss}(o) &= o \times t_{miss} + (\text{assoc} - o) \times t_{hit} \\ &= (t_{miss} - t_{hit}) \times o + (\text{assoc} \times t_{hit}) \end{aligned} \tag{4.16}$$

Therefore, if we use as an attack trace *the sum of the prime set timing values*, there will be a distinct trace point value for each $T_{hit\text{-}miss}(o)$.

The *nhit_inclusive* policy, is similar and shares the same *KDS* definitions as the *nhit* policy. Indeed, we have $\mathbb{A}_{nhit} \subset \mathbb{A}_{nhit_inclusive}$. In that regard, *nhit_inclusive* mostly differs on what attack from $\mathbb{A}_{nhit_inclusive}$ is searching for. Whereas attacks from $\mathbb{A}_{nhit}$ search for exact $T_{hit\text{-}miss}(o)$ values associated with the *KDS* (i, o), attacks from $\mathbb{A}_{nhit_inclusive}$ search for $T_{hit\text{-}miss}(w)$ higher with w higher than a certain o_{th} associated with the *KDS* (i, o_{th}). For *Prime+Probe* reverse, this can be computed as a $\sum T_{hit\text{-}miss}(o_{th})$ threshold value for the acquired $\sum T_{hit\text{-}miss}(w)$. In that case, a $\sum T_{hit\text{-}miss}(w) \geq \sum T_{hit\text{-}miss}(o_{th})$ is our signal for the *KDS* (i, o_{th}).

We sum up the link between policies and *Prime+Probe* direction in the table 4.1. It also contains the trace points we use, and how it is linked with the *KDS* we want to detect. In this table, we also give the signal we are searching for to detect a *KDS*, although in real measures, we would have to account for the noise.

We propose the figure 4.5 to visualize, the different output values for *Prime+Probe* forward and reverse and their correspondence with KDS. On the bottom, which corresponds to *Prime+Probe* reverse traces, each stair level corresponds to a $\sum T_{hit\text{-}miss}(o)$ (with o from 0 to 7) associated with a cache occupancy o. Each $\sum T_{hit\text{-}miss}(o)$ for the set i is associated with the *nhit* KDS (i, o). The same victim behavior produces the trace on the top when using *Prime+Probe* forward, with only two values $\sum T_{hit}$, the lowest, and $\sum T_{miss}$, the highest. $\sum T_{miss}$ for the set i is associated with the *1hit* KDS (i).

Policy	*KDS*	Attack	Output	Trace (Σ)	Signal
1hit	(i)	*Prime+Probe* forward	T_{hit} or T_{miss}	$\sum T_{hit}$ or $\sum T_{miss}$	$\sum T = \sum T_{miss}$
nhit	(i, o)	*Prime+Probe* reverse	$T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w) = \sum T_{hit\text{-}miss}(o)$
nhit_inclusive	(i, o_{th})	*Prime+Probe* reverse	$T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w) \geq \sum T_{hit\text{-}miss}(o_{th})$

Table 4.1: Correspondence between *VictimScan policy* and *Prime+Probe*

4.4 TEE-Time implementation

TEE-Time is implemented in instrumentation scripts running in *GDB-Python* (section 3.2.3.2). These scripts connect to the *gem5* platform to analyze an attack scenario.

4.4.1 Instrumenting the attack scenario

We implemented certain elements in the simulated image to allow *GDB* to configure the attack scenario at run time. An attack scenario run starts at a checkpoint that happened in an `init.d` script. After the checkpoint, this script continues: It loads, then executes a bash script using `m5 readfile`. This bash script is controlled directly by *gem5* and allows us to modify the scenario without having to re-build the boot checkpoint (*BootPoint*). To avoid issues with the root disk image, we have to manually mount another disk that contains all the necessary tools needed for our attack scenario and that we are likely to modify (Linux client, TA, attack, etc.).

Our scenario is described as a bash script loaded in the simulation with `m5 readfile`. To configure this scenario through *GDB* at run time, we use the `m5 env` (presented on figure 4.6) in our script to load environment variables. These variables are contained in a *dict* in *gem5-Python*, which we set using *GDB-Python*. They specify for the rest of the bash script:

bash.rcS

```
attack_cli=$(m5 env attack_cli)
victim_cli=$(m5 env victim_cli)
```

Figure 4.6: `m5 env`: our new *m5* instruction to load environment variable at runtime

- If the attack should be run
- The command line of the victim.
- The command line of the attacks.
- In which thread to launch the attack and the victim.

GDB changes its current directory to correspond with the *gem5 m5out* directory. It gets this information using the *GDB-instrumentation* interface. This setup allows the use of the same *gem5* configuration, platform, and disk images for different behaviors configured by *GDB*. Each *gem5* run only differs by the *m5out* directory: It makes simulation runs independently of each other. After this initialization phase, in which *GDB-Python* also creates breakpoints to monitor the attack and the victim, the attack scenario proceeds normally, only interrupted by *GDB* breakpoints. During these breakpoints, *GDB-Python* can use the *GDB-instrumentation* to extract information or reconfigure the simulator, but it can no longer modify the attack scenario. To improve run times the *GDB-Python* script can command *gem5* to perform a CPU switch: *gem5* temporally changes the CPU model for a faster but less precise model (atomic model). When the victim program starts, *GDB-Python* commands *gem5* to switch back to the more precise CPU model. Finally, to make results more accessible, the attack can use `m5_writefile` to output traces directly in the host directory (*m5out*)

4.4.2 Dedicated *GDB* scripts

To use TEE-Time, a dedicated Python script is loaded in *GDB*, while execution starts in *gem5* (*gem5* can wait for a *GDB* connection before starting the simulation). This dedicated script connects automatically to *gem5*, loads ELF images in *GDB*, configures environment variables in *gem5*, initializes breakpoints, and then resumes simulation. TEE-Time uses two different *GDB* instrumentation scripts, one for each step of the process (see figure 4.4):

- `VictimScan` uses the *KEPs* provided as a CSV file, gathers dump as the victim program is running on *gem5*, and finally produces a report that summarizes the information contained in dumps. This report's main feature is a *Key Detectable State (KDS)* ranking using the score described in section 4.3.3.

- `AttackMonitor` monitors an attack scenario. It adds label points based on a set of *KEPs* to the cache traces produced by the attack in order to verify the correlation between *KEP* and attack timing traces.

These two scripts load the same label file containing *KEPs* to initialize breakpoints associated with each *KEP*; they only differ in the behaviors associated with each *KEP* type. Each script has to be launched from the start of a *gem5* run, which represents an attack scenario.

4.4.2.1 *VictimScan*

The *VictimScan* is handled by a dedicated *GDB-Python* script: `victim_scan.py`. Its inner workings and interactions with *gem5* are represented in figure 4.7. *VictimScan* uses the *KEP* chosen by the user to delimit potentially non-constant-time or critical code segments handling the secret/key. *VictimScan* creates breakpoints to extract cache states for each *KEP*. The user can choose how *VictimScan* gathers cache data around *KEP* using the *KEP toolbox* that describes multiple types of *KEP* with different behaviors. The *KEP toolbox* can also deploy random points to deploy noise *KEPs*. Noise *KEPs* generate dumps that are used as references to reject cache states correlated with random execution points and not specific *KEPs*. These dumps contain reconstructed address data, which allows tracking each cache line to its corresponding source. As we illustrated in figure 2.16, cache states depend on a wide variety of sources:

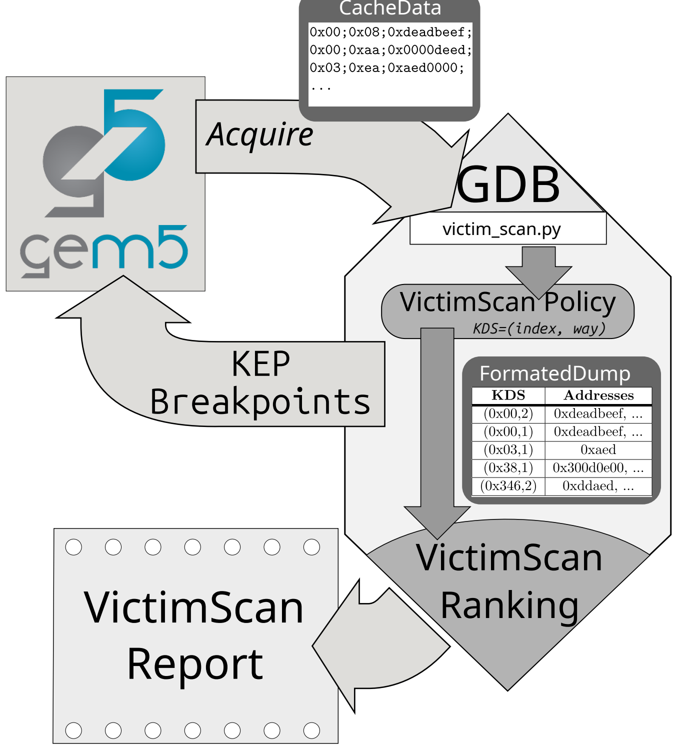


Figure 4.7: Structure of *VictimScan*: running in *GDB*; *VictimScan* programs breakpoint on *KEP* in *gem5*. From these breakpoints, *VictimScan* extracts raw cache data which are formatted using *VictimScan policy*. These formatted dumps, now made of a set of *KDS*, can be presented to the ranking algorithm. These *KDS* are processed to produce the *VictimScan* report

- Data access

- Instruction fetch (wide fetch can fetch instruction speculatively).

- MMU table walk: o ARMv8-A the MMU has a hardware table walker that automatically tries to fill the TLB by exploring the page table.

- Hardware prefetcher: caches have prefetchers that can preventively fetch lines.

Thanks to our virtual platform modeling all these sources, TEE-Time can account for cache states that other analyzers might have missed. These sources follow the cache state and then their related *KDS* along the process to propose causes for *KEP* detectability (e.g., specific memory access that frequently causes a line to be present in caches). Independently of the *KEP* breakpoints' internal behavior, a unique cache dump is produced containing one or more cache states. To extract *KDS* from the cache state needed to classify its contents with respect to a cache timing attack, a *VictimScan policy* arranges the information from the dumps producing one or more formatted dumps, each containing a set of *KDS*. The user can choose between different *VictimScan policy* that we described in section 4.3.2. Each formatted dump is associated with a *KEP* class. As mentioned before, these *KDS* come with one or more source addresses that designate a reason for the associated cache state. Using this set of *KDS*, the ranking can be updated using the score function described in section 4.3.3. This ranking orders *KDS* for each class of *KEP* such as:

- The best *KDS* can be used to distinguish between two classes. It is rarely present in other classes.

- The best *KDS* is not present in random/noise *KEP* classes (⚂,⚄, etc.). *KDS* present in random *KEP* classes are triggered randomly and therefore can not be used to distinguish and detect a *KEP*.

To consider different types of *KEP*-cache state correlation, The *KEP toolbox* proposes different *Key Execution Point* types:

- *Punctual Key Execution Point*: Only study current cache states (using a cache dump) when the point occurs. This *KEP* is followed by a flush to ensure the same cache state is not registered by multiple *KEPs* if it did not re-enter the cache.
- *Delay Key Execution Point*: Study what is in the cache but only after a certain number of instructions.
- *Scope Key Execution Point*: Study the difference between what was in caches when the point occurred and what is in caches when leaving a scope.
- *Tracker Key Execution Point*: study what entered and leaves the cache in a section (using scope breakpoint or normal breakpoints). When entering a tracker section, a cache flush is generally performed to force a line to re-enter the cache.

In *VictimScan*, *cache flushes* use a *CxxMethod* to flush all the evictable lines to be evicted, imitating what a *Prime+Probe* attack could do. This forces potentially leaking lines to be evicted, allowing them to be fetched again only if they are linked with the *KEP* we are trying to monitor. As mentioned before, *KEPs* are identified by their name which represents the event they want to spot. Thus, each *KEP* implemented by the *KEP toolbox* has a name (this name can be a parameter of local variable values). In the toolbox, *KEPs*' positions are represented as a breakpoint location in *GDB*: `demo.c:2`, `mbeltls_mul`, etc.

KEPs have to be placed precisely to correctly organize cache data. Their position around a function is important to account for different types of cache signals that could leak the victim function secret. Therefore, depending on the assumption made on the victim handling of the secret, we have to deploy *KEP* different type of *KEP* setup:

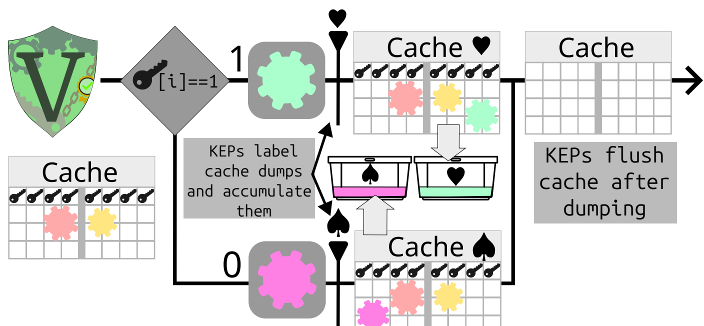


Figure 4.8: Classifying cache data on whether the victim function takes a branch using *Punctual Key Execution Point*

- **Punctual event**: We could assume that leaking computations happen at a single instant of the victim computation, which could be directly linked to the secret leak. In that case, we can deploy *punctual KEP* after the computations. These *KEPs* will classify the cache data with the assumption that it still contains the data from the previous computation. Therefore, *KEPs* are named depending on the secret the computation was leaking. A cache flush is done on each *KEP* after gathering the data to ensure that the acquired data is no longer in caches. Otherwise, the same access could be counted by multiple *KEP* triggers. This typically happens when a victim performs computations that have different cache access patterns in a branch, like the situation represented on figure 4.8. In this situation, we deploy a *Punctual KEPs* after the computation in each branch (represented as colored gears). Each *KEPs* classifies the cache contents (which have been altered by the computation), using its name (♥ or ♠). These cache states are archived in their two categories to be then processed by the *VictimScan policy* to extract their *KDS*. Cache state is then flushed to avoid archiving the same cache state multiple times.

Code associated with the situations described by figure 4.8

```
if(key[i]==1){
    do_1();//KEP:♥ here
}else{
    do_0();//KEP:♠ here
}
```

- **Scoped event**: We could assume that the leaking computation happens after the secret access, in a scope that can not be easily linked to the secret it leaks. In that case, dual *KEPs* have to be deployed. Either manually, by putting them where we could determine the secret after the leak happened, or automatically if this happens when entering and leaving a scope (branch, function, etc.). In this context, one *KEP* defines the name of the section, which is associated with the secret it leaks, and another *KEP* defines the end of the section where cache data are linked with the secret. A cache flush is done on entering the *KEP*-scoped section to ensure that the detected *KDS* can only be caused by the scoped section. *Cache tracker* KEPs can also be used in this context as they can register all the internal cache states while in the scope section. This typically happens when a function is called with an argument that depends on a secret and behaves differently cache-wise depending on this secret, like the situation represented on figure 4.9. In this situation, the victim function only prepares which function or data it will use in the branch (represented as key-labeled gear) and effectively uses them outside the branch (represented as the two-colored gear in the execution line). To correctly classify this situation, we use three *KEPs*. Two *KEPs* in the branch (♥ and ♠) labels that cache state in advances and perform a cache flush. A final *KEP*, after the leaking computation, archived the current cache state, using the label previously defined by one of the first *KEP*. This setup ensures that the cache state is put in the category corresponding to the branch it traversed. The *VictimScan policy* will then try to extract *KDS*, which can highlight a possible correlation between cache state after the leaking computation and the branch taken. These *KEPs* can also be cache trackers to follow all the cache state between the start *KEPs* in the branches and the end *KEP* after the leaking computation.

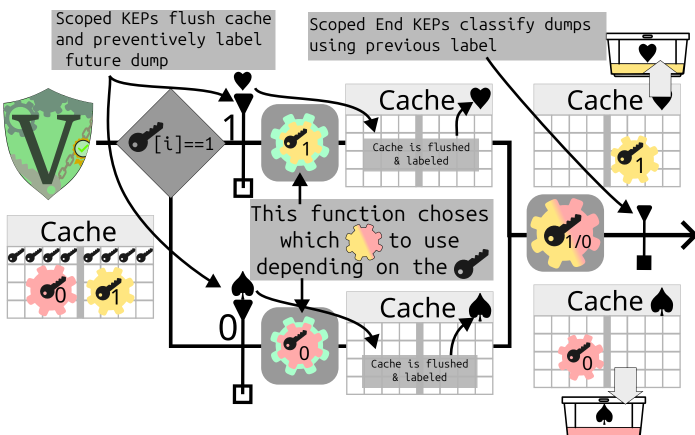


Figure 4.9: Classifying cache data on whether a victim function takes branch when the leaking computation happens after the branch.

Code associated with the situations described by figure 4.9

```c
if(key[i]==1){
  x=f1;//start scope KEP:♥
}else{
  x=f0;//start scope KEP:♠
}
do_(x);//scope end KEP
```

- **Backward scoped event**: Coincidentally, we could assume that the leaking computation happens before we can determine the value of the secret. In fact, a preparatory computation might be necessary to access the secret. This preparatory computation may have a different internal cache access pattern while still having its final cache state not correlated with the key. In such cases, when the secret is determinable, the cache state does not show any correlation with it. In this situation, we rely on *cache tracker KEPs*. One *KEP* starts the sections with a placeholder name, and another *KEP* names the cache data accumulated to classify it. A cache flush is done on entering the *KEP*-scoped section to ensure that the detected *KDS* can only be caused by the scoped section. This typically happens when a

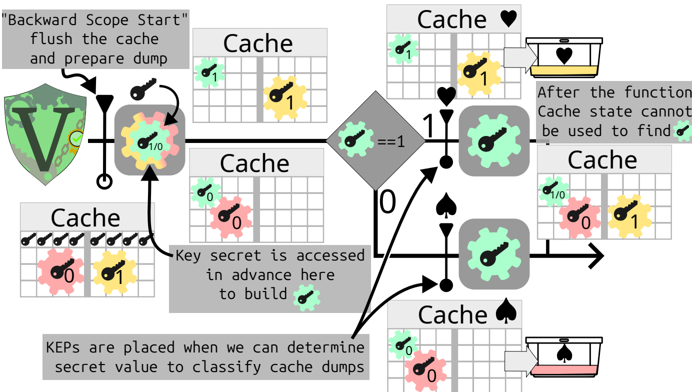


Figure 4.10: Classifying cache data on whether a victim takes branch when the leaking computation happens before the branch was taken

function is used to determine the value of the secret, and such a function has different cache pattern access depending on the value of the secret. This corresponds to the situation on figure 4.10. In this function, the victim prefetches or pre-computes a gear (yellow or pink), depending on the value of the key. It then performs a constant time operation (green gear) that ensures that the cache state contains both gears, however the first computing operation (green gear overlaid on a two-colored gear) leaks which operation will be performed in the constant time section. In this context, we can use three *KEPs* to classify the cache states. A first *KEP* is deployed before the leaking pre-compute operation. This *KEP* does not have a name and prepares the future dump by flushing the cache and starting the tracker if it is a *Tracker KEP*. Two following *KEPs* are then deployed before the constant time operation. In the branches, these two *KEPs* will label the pending cache data and archive it in their respective categories (♥ and ♠). If they are trackers, they end the cache tracking process and archive all the cache states that happened between the first *KEP* and them. This way, the cache states that are passed to the *VictimScan policy* contains the effect of the pre-compute operation (two-colored gear) but not the constant time operation (green gear). The deduced *KDS* can thus show a correlation between the pre-compute operation cache side effect and branch taken.

Code associated with the situations described by figure 4.10

```c
//Scope start KEP
k=access_k(key,i);
if(k==1){
    //Scope end KEP:♥
    do_ct1(k);
}else{
    //Scope end KEP:♠
    do_ct0(k);
}
```

Although, on figure 4.8, figure 4.9 and figure 4.10, cache states correlated with secrets are trivially determinable, *VictimScan* can account for all the causes on figure 2.16 which may be harder to apprehend only by studying victim binary images. We can also define *KEPs*' name (the class to which they register their associated dump) to be dependent on the value of a *GDB* accessible variable. These are *parametric KEPs*. In some cases, we can choose to also run an unconfigured attack while *Victim Scanning* as attack cache behavior might disturb some victim signals, making them less detectable.

VictimScan policies can also be configured to exclude unsecure lines when monitoring a *secure OS* operation. In that case, they produce a new *noise KEP* dump entry (that we noted ⚄). This is called *REJECT_ UNSECURE*. With that settings, for each dump produced on *KEP*, the produced *KDS*es are separated between the real *KEP* (♠, ♥, ...) contribution (secure cache lines), and the noise (⚄) contribution (unsecure cache lines). It improves *VictimScan KDS* detection performance by depreciating *KDS* that are linked with unsecure access. At the end, *VictimScan* produces a report that contains all the information it gathered and a *KDS* ranking, linking each *KDS* to its reported sources (addresses, prefetch, ...).

4.4.2.2 *Attack Monitoring*

Attack Monitoring is performed by a Python script called `attack_monitor.py`. This script is loaded in *GDB* and configured breakpoints using the same *KEPs* as *VictimScan*. This script automatically loads environment variables using `m5 env`. It adds additional environment variables containing the attack configuration. It chooses them using the last *VictimScan* report. Indeed, the *Attack Monitoring* phase is used in a new run after the *VictimScan* run to verify its results and the *KDS* it found.

This *GDB* script automatically monitors the attack program while also monitoring the victim. When a *KEP* is triggered by the victim, the script logs when it happens using the attack time counter. The script can also extract the attack timing results while they are being gathered by the attack, or at the end of the runs. Using cache timings results, and *KEP* logged by *GDB*, we can produce annotated attack traces which plot on the same time graph: the attack timing results and when *KEP* are encountered by the victim. With this graph, we can study if an attack configured with *KDS* found by *VictimScan* effectively results in a correlation between their associated timings and *KEPs* being triggered.

With the *KEP* instant recorded, we can plot timing results w.r.t. each *KEP* class trigger. Using that plot, we can observe if timings results behave consistently around *KEP*, which demonstrates *KDS*'s effectiveness in detecting and distinguishing said *KEPs*. Indeed, using *Attack Monitoring* before running the attack on an actual platform helps to verify key properties of the attack. It gives the necessary confidence in case this correlation is not visible on the actual platform that it can still be present statistically and therefore require multiple traces to be observed.

4.5 Example: demo cryptographic function

We propose the following function, which is reminiscent of the *Square and Multiply Algorithm* as an example to show how TEE-Time works. It is placed inside a trusted application and launched from Linux using a host application.

src/ta/crypto_f.c

```
17 void crypto_f(big_int_t* A, big_int_t* B,big_int_t* E){
18   for(size_t i=0;i<BIG_N_B;i++){
19     if(bit(E,i)){
20       add(A,B,A);//♠
21     }
22     add(B,B,B);//♥
23 }}
```

KEP	type	pos
♥	scoped	demo_fun.c:22
♠	scoped	demo_fun.c:20

The host application initializes and launches the TA while an attack is notified. The attack being notified that the victim started, it can begin its *Prime+Probe* process. In this application, we used *scoped KEPs* as we want to detect the `add` functions, without any assumption on the previous cache state. For this demo we are using *1hit VictimScan policy* and its associated attack *Prime+Probe* forward as described in table 4.1.

4.5.1 Demo: *VictimScan*

The *KEPs* in this function, based on typical square and multiply algorithm weaknesses, are represented as ♥ and ♠. Because we use the *GDB* format, we can generate them from typical IDEs (like VsCode). They are provided as a CSV file to the *VictimScan* script running in *GDB*.

In this example, we have two classes of *KEPs*: ♥ and ♠, with each only containing one *KEP*. With this configuration, *VictimScan* produces the report in figure 4.11. *VictimScan* suggests the best *KDS* to attack. Here, we only displayed the top two for the two classes of *KEPs*. Thanks to *gem5* integration, we are able to trace the main source for *KDS*: Attributing it to an address (virtual and physical) and, if possible, a code line. We added this information to *gem5* packets and stored it in the cache model in their associated cache line. In most situations, *VictimScan* also finds *KDS* that have hidden causes, like:

- Automatic translation table walking: Address sources are table addresses.
- Prefetching: Sources are instructions outside the function.
- Heap and stack addresses: Sources are typically in the function accesses around the *KEP*.

m5out/report.txt

```
♥->max_hit:('0x212', 64)
  (1):0x210
     score:0.6666666666666667
     hit_count:64
     top_addr:
      1@128=S#0x40093400[S#0x30218400]:add + 76 in section .text
  (2):0x211
     score:0.6666666666666667
     hit_count:64
     top_addr:
      1@128=S#0x40093440[S#0x30218440]:add + 140 in section .text
♠->max_hit:('0x20e', 35)
  (1):0x58
     score:1.0
     hit_count:35
     top_addr:
      1@70=S#0x400fc600[S#0x30281600]:__ta_no_share_heap + 130992 in section .bss
  (2):0x210
     score:0.6666666666666667
     hit_count:35
     top_addr:
      1@70=S#0x40093400[S#0x30218400]:add + 76 in section .text
```

Figure 4.11: Typical report from *VictimScan*, showing the *KEPs* classes, and the associated *KDS* with their scores; ranked in decreasing order. Each *KDS* also specifies the corresponding address in the binary, e.g. instructions from `.text` section, or variables in the heap.

These causes reflect what was described in figure 2.16. In the report in figure 4.11, *KDS*es correspond to cache lines present in cache dumps that we define in section 4.3.2 as *1hit KDS*. By checking `top_addr`, which reports the cache state sources, we can see that: `0x210` is due to an instruction in the `add` function and `0x58` is due to a heap variable (`__ta_no_share_heap`).

4.5.2 Demo: *Attack Monitoring*

Once the *KDS* are detected and ranked, TEE-Time proceeds to the *Attack Monitoring* phase (Step III). During this phase, TEE-Time configures the attack using the best candidate *KDS* and then simulates it, producing real-attack traces. In the context of *1hit*, the *Prime+Probe* attack is configured to target the cache sets associated with each *KDS*. The attack traces are shown in figure 4.12. The bottom figure is zoomed for better visibility. The attack is automatically configured using the *VictimScan* report shown in figure 4.11, with the highest ranked KDS. ♥:`0x210` and ♠: `0x58`. Cache set uses the prime-set signal described in table 4.1 for *1hit* which corresponds to the sum over the prime set ($\sum T$).

Attack Monitoring combines cache timing results from the *Prime+Probe* attack and the victim monitoring information from *GDB* to produce the traces in figure 4.12. On figure 4.12 and figure 4.13 the cache timings are referred by the *KDS* they are trying to detect. TEE-Time uses the *KEPs* events (shown in figure 4.12 as vertical bars) to create a window around each *KEP*. All the windows belonging to the same *KEP* are then superposed and we obtain one graph per class of *KEP*. This graph for the example attack is shown in figure 4.13. It shows the relation between cache timing and labels: we expect that, if a *KDS* is really linked with a *KEP*, its associated cache timings will rise around the *KEP* while resting at a low value everywhere else. As expected, we can easily see in figure 4.13 that there are *KDS* whose associated cache timing values are higher around *KEPs*. This demonstrates that *VictimScan* effectively found *KDS* that can be used to detect the *KEPs*. However, we see an asymmetric conflict since `0x210` is triggered by both *KEPs*. This is not an issue and is a consequence of the conflict contribution in the equation 4.10. This is one of the reasons for the sign of the contribution in the equation. So for our demo function:

- `0x210` will be used to characterize both ♥ and ♠
- `0x58` will be used to distinguish between ♥ and ♠

Sometimes, the *Attack Monitoring* step fails to show a clear correlation even when traces are generated with the highest ranking *KDS*. It could be because of noise or countermeasures. For this reason, both steps of TEE-Time are necessary to find and assess a cache timing vulnerability. Though it depends on the complexity of the function analyzed, *VictimScan* is a rather quick process. Meanwhile, *Attack Monitoring* is noticeably slower (as seen on table 4.2). This is why overseeing results in the report produced by *VictimScan* is important before continuing the process. Table 4.2 shows the experimental configurations and the corresponding run times.

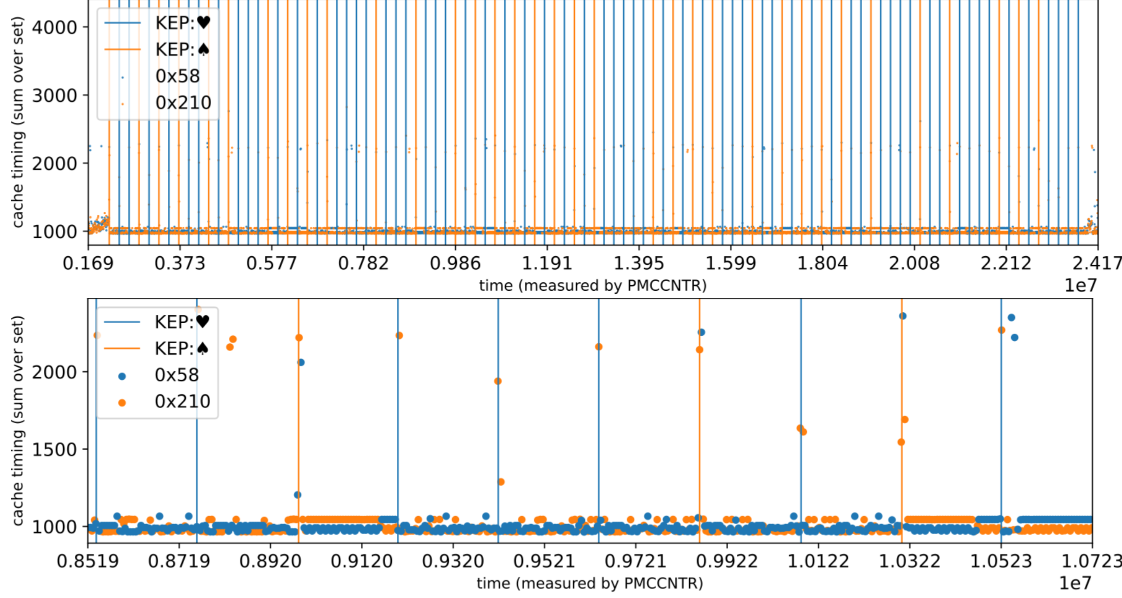


Figure 4.12: Cache timing traces for the simple example, the bottom figure being the top zoomed. The X-axis is the time. The moments when execution reaches a *KEP* are indicated with vertical lines. Prime set timings are shown with colored dots, with their Y-value corresponding to the sum access time for the prime set ($\sum T$).

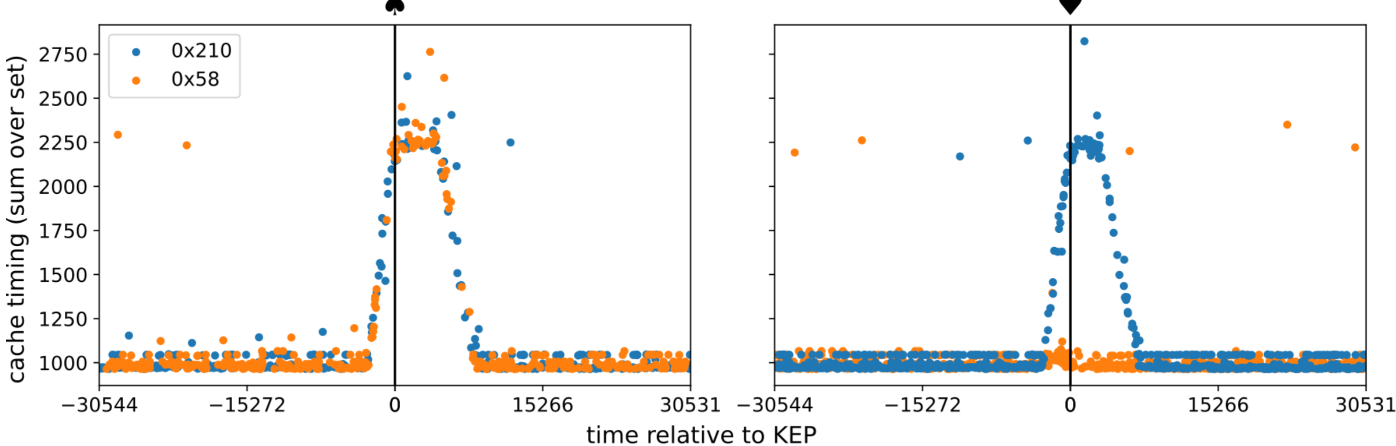


Figure 4.13: Zoomed in timing traces plotted relatively to *KEPs*. The attack traces for 0x210 sense a signal for both ♥ and ♠. The attack traces for 0x268 sense a signal for only ♠. The black vertical lines mark the moment the *KEP* was triggered and the timings are plotted relative to this moment.

4.5.3 TEE-Time: Code coverage

For a production-quality TA, full code coverage for cache timing vulnerabilities is required. Here is a brief outline of the method to achieve such complete code coverage:

- First, we can eliminate lines in the code that do not depend on any secret information(e.g. cryptographic key). This step can be done using static analysis.
- We can designate the remaining lines as *KEPs* belonging to a particular *class of KEP* (label).
- By running *VictimScan*, we can check if these *KEPs* are secure (a negative score) or vulnerable.

Runtimes	
Demo Victim TA	1212.52 s
Victim TA + *VictimScan* (*GDB*)	1809.89 s
Victim TA + *Attack Monitoring* (*GDB*)	2680.65 s
Victim TA + Attack	2359.49 s

Table 4.2: Simulation runtime. Times measured by *gem5*. When using *GDB*, the acceleration methodology is used. We run our examples on a *Intel(R) Xeon(R) Gold 6128* with 256GB of DDR4.

The negative scores can be the result of a software countermeasure or hardware optimization. If TEE-Time fails for these reasons, as described in figure 4.4, we have to go back to the previous step and change its configurations: changing *VictimScan policies*, changing KEPs or their implementation with the *KEP toolbox*.

4.6 Attack against RSA signing in OP-TEE

Our *sec-sign* TA uses RSA to sign a hash. This service can be used by a Linux application to hash and sign a message as shown in the UML diagram on figure 3.32. We propose to use TEE-Time to analyze this TA security against an attacker trying to recover the private key using *Prime+Probe*. As recommended, our *sec-sign* TA uses the *GlobalPlatform* API's cryptographic functions to implement hash signing. To sign a hash, OP-TEE uses the function `rsa_exptmod` in *libtomcrypt* directly incorporated in OP-TEE. To sign, `rsa_exptmod` will use the private exponent which should be kept secret. If the key provided does not contain RSA-CRT factors (dQ, dP, qP, Q and P), `rsa_exptmod` uses a simple *bignum* exponentiation provided by *libmbedTLS* after blinding the base. When RSA-CRT factors are provided, two *bignum* exponentiation are used with dP and dQ. We chose to only provide basic RSA factors D, E, and N, to simplify our study. In this situation, *libtomcrypt* perform a single operation with D using *libmbedTLS bignum*:

$$blind_sign = blind_hash^{D} mod\ N \tag{4.17}$$

Blinding and unblinding is done using E and N.

4.6.1 *mbedTLS* bignum exponentiation

The exponentiation function in *mbedTLS*, `mbedtls_mpi_exp_mod`, uses the sliding-window algorithm to compute the bignum exponentiation [MOV01], shown in figure 4.14(a). In our case, the exponent is the private exponent (D). This

Sliding-window exponentiation

Require: $E \geq 0$
Ensure: $X = A^E$
  $X \leftarrow 1$
  while $i \geq 0$ **do**
    if $n_i = 0$ **then**
      $X \leftarrow X \times X$
      $i \leftarrow i - 1$
    else
      $s \leftarrow \max\{i - k + 1, 0\}$
      for $h = [1; (i - s + 1)]$ **do**
        $X \leftarrow X \times X$
      end for
      $wbits \leftarrow (n_i \dots n_s)$ $\triangleright\ i - s + 1 \leq wsize$
      $X \leftarrow X \times A^{wbits}$
      $i \leftarrow s - 1$
    end if
  end while
  return X

(a) Sliding window algorithm

optee_os/lib/libmbedtls/mbedtls/library/bignum.c

```
int mbedtls_mpi_exp_mod( mbedtls_mpi *X, const mbedtls_mpi
↪  *A, const mbedtls_mpi *E, const mbedtls_mpi *N ){
  /* Preparing W :
  W[I]= X^I */
  state=1;wsize=6 nbits=0;
  int i=Skip_leading_zeros(E,X);
  while( 1 ){
    if(is_Finished(i))
        break;
    ei = (E[i]) & 1;//E[i] is i-th bit of E
    if( ei == 0 && state == 1 ) {
        /*X=X * X*/
        mbedtls_mpi_montmul( X, X, N, mm, &T );   Square
        continue;
    }
    state = 2; nbits++;
    wbits |= ( ei << ( wsize - nbits ) );//
    //
    if( nbits == wsize ){                         Window section
      /* X = X^wsize R^-1 mod N*/
      for( i = 0; i < wsize; i++ )//
        mbedtls_mpi_montmul( X, X, N, mm, &T );   Square
      /* X = X * W[wbits] R^-1 mod N */
      mpi_select( &WW, W, (size_t) 1 << wsize, wbits );
      mbedtls_mpi_montmul(X,&WW,N,mm,&T);         Multiply
      state=1; nbits = 0; wbits = 0;//
     }
  }
  /* process the remaining bits */
  return( ret );
}
```

(b) Sliding window implementation: we call the section in red the window section.

Figure 4.14: Algorithm and implementation from *mbedTLS* for the sliding window exponentiation algorithm from [MOV01].

algorithm exploits a window (wbits) to accumulate multiple bits of the key (n_i) together and then uses them to do the exponentiation using a precomputed value (A^{wbits}). When a leading 1 is found, the following $wsize$ bits are accumulated in $wbits$ the associated precomputed value is then multiplied with X: $X \leftarrow X \times A^{wbits}$. Zeros outside the accumulation phase are skipped by just squaring X.

This algorithm implementation (taken from OP-TEE 3.21) is presented on figure 4.14. The following function is from the implementation on figure 4.14(b):

```
mpi_select( &WW, W, (size_t) 1 << wsize, wbits );
```

It ensures that accessing the precomputed window using the accumulated window bits is time-constant. This is called *multiplier obfuscation* by KOU et al.[KOU+23]. This function uses a conditional move operation and accesses all the possible windows. It was changed since the first experiment, we did in OP-TEE 3.12. However, this algorithm is known to leak some information about the key. [UH23] and [Ber+17] suggest detecting *montgomery multiplication call* and categorizing them: *square*([$\mathbb{S}$]) or *multiply*([$\mathbb{M}$]). These are the *KEPs* that we will use to extract the partial keys, which can be then used to reconstruct the key as mentioned by [UH23]. The series of *square* and *multiply* can then be used to extract a partial key as represented on figure 4.15. This figure represents how our *KEP* segments divide the execution in order to allow us to retrieve

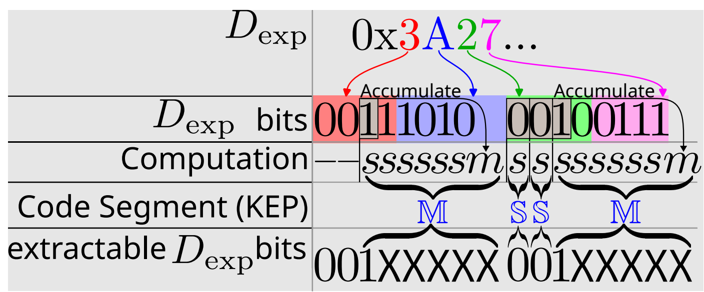


Figure 4.15: S are square operation and M are multiply operation

optee_os/lib/libmbedtls/mbedtls/library/bignum.c

```c
int mbedtls_mpi_exp_mod( mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *E, const
↪  mbedtls_mpi *N ){
  /* Preparing W :
  W[I]= X^I */
  state=1;wsize=6 nbits=0;
  int i=Skip_leading_zeros(E,X);
  while( 1 ){
    if(is_Finished(i))
        break;
    ei = (E[i]) & 1;//E[i] is i-th bit of E
    if( ei == 0 && state == 1 ) {
[S]     /*X=X * X*/
        mbedtls_mpi_montmul( X, X, N, mm, &T );
        continue;
    }
    state = 2; nbits++;
    wbits |= ( ei << ( wsize - nbits ) );

    if( nbits == wsize ){
      /* X = X^wsize R^-1 mod N*/
      for( i = 0; i < wsize; i++ )
        mbedtls_mpi_montmul( X, X, N, mm, &T );
      /* X = X * W[wbits] R^-1 mod N */
[M]   mpi_select( &WW, W, (size_t) 1 << wsize, wbits );
      mbedtls_mpi_montmul(X,&WW,N,mm,&T);
      state=1; nbits = 0; wbits = 0;//
    }
  }
  /* process the remaining bits */
  return( ret );
}
```

Figure 4.16: These are the *Key Execution Points*, we use to scan `mbedtls_mpi_exp_mod`. They are all scoped key execution points defined by the highlighted sections.

4.6.2 RSA: *VictimScan*

Based on [Ber+17], we propose the *KEPs* shown in figure 4.16 as red sections. These *KEPs* correspond to different types of multiplication:

[S] is the squaring multiplication outside the window.

[M] is the multiplication with the precomputed window.

For this demo, we are using *1hit VictimScan policy* and its associated attack *Prime+Probe* forward as described in table 4.1.

In fact, the series of [S] and [M] that the algorithm goes through contains information about the key. Using TEE-Time, we found the following *KDS* as shown in the report of figure 4.17:

[S] :0x289

[M] :0x2a9

Both of these *KDS* correspond to set indices used by instructions around each *KEP*.

4.6.3 RSA: *Attack Monitoring*

With these results, we can proceed to the *Attack Monitoring* phase (Step III). TEE-Time automatically configures the *Prime+Probe* attack with the two *KDS* on report figure 4.17. The results from the *Prime+Probe* attack are compiled with *Attack Monitoring* information to produce the trace on figure 4.18.

In figure 4.19, we plot each *KEP* window from the cache timing traces from figure 4.18 in a superposed manner. This way, we have the corresponding relative cache timings to *KEP*, for the `mbedtls_mpi_exp_mod` function. In this figure, we can see that the timing associated with the prime set `0x289` is only at a high value around [M]. On the other hand, although the timing associated with the prime set `0x2a9` produces spikes around the two *KEPs*. However, it only

>_ m5out/report.txt

```
[𝕄]->max_hit:(0x2ab, 149)
(1):(0x289)
     score:1.0
     hit_count:149
     top_addr:
        1@149=S#0x3008a240[S#0x3008a240]:mpi_select + 12 in section .text
[𝕊]->max_hit:(0x2a9, 129)
(1):(0x2a9)
     score:0.9818933404586392
     hit_count:129
     top_addr:
        1@129=S#0x3008aa40[S#0x3008aa40]:mbedtls_mpi_exp_mod + 1884 in section .text
        2@0=S#0x5aa40[S#0x5aa40]:UKN
        3@0=0xffff80000a51aa40[0x471aa40]:UKN
```

Figure 4.17: TEE-Time report generated for *mbedTLS* and the *KEP* specified in the code extract. `UKN` implies that the cache line belongs to code outside *GDB* knowledge (e.g Linux kernel)

produces two spikes around [𝕊]. We can see that these spikes always happen by noticing that the `0x2a9` is never at a low value for these two instants. By using the method proposed in [UH23] on these traces a partial key can be recovered which can be used to rebuild the full key. So, we conclude that, with these lines, we can perfectly follow the execution of the *mbedTLS* function only using a *Prime+Probe* attack. [UH23] guarantees us that with these points (our *KEPs*), we can get a partial key that can be used to rebuild the full key using [MH20]. This way we have the configuration to do an attack against this OP-TEE trusted application. This theoretical attack can also be used to study how *gem5* parameters affect attackability, reproducing different platforms to tailor the attack and scenario for them (sharing CPU clusters with other processes, interruption, etc.).

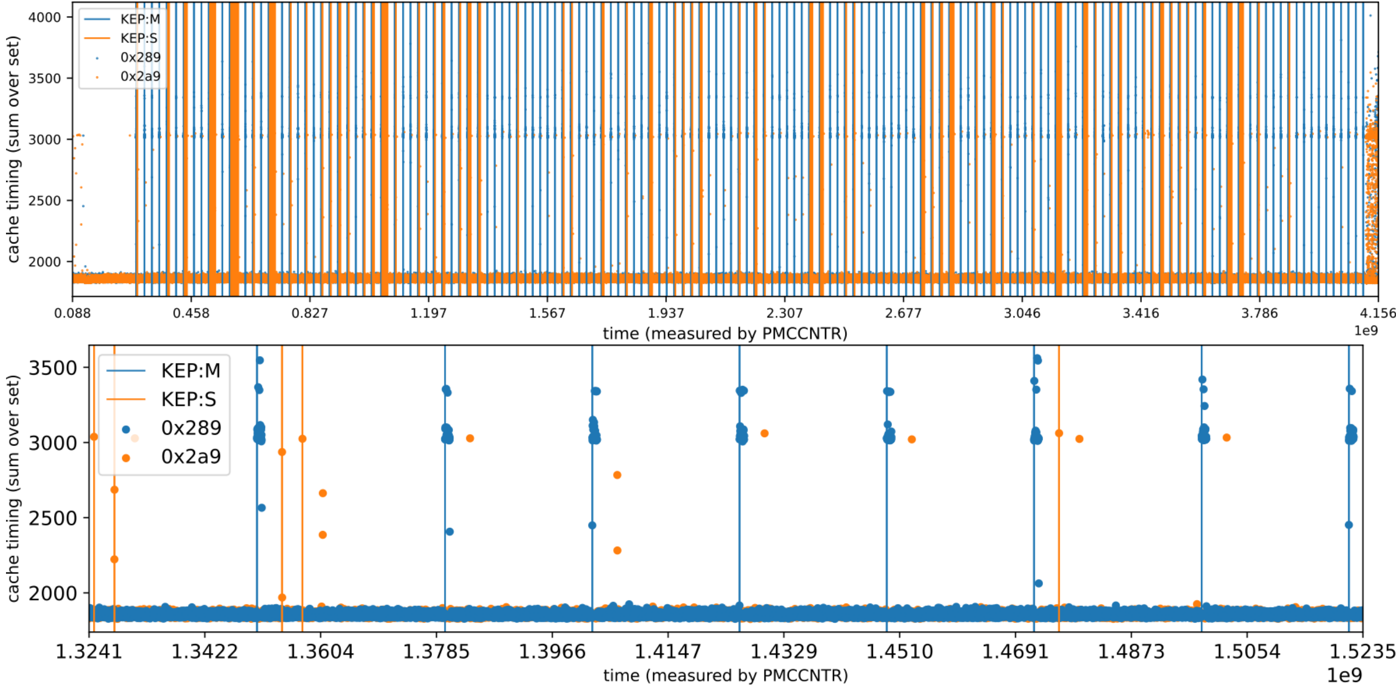


Figure 4.18: Cache timing traces for the mbedTLS attack, the bottom figure being the top zoomed. The X-axis is the time. The moments when execution reaches a *KEP*, are indicated with ticks. *Prime+Probe* timings are shown with colored dots, with their Y-value corresponding to the sum access time for the prime set ($\sum T$).

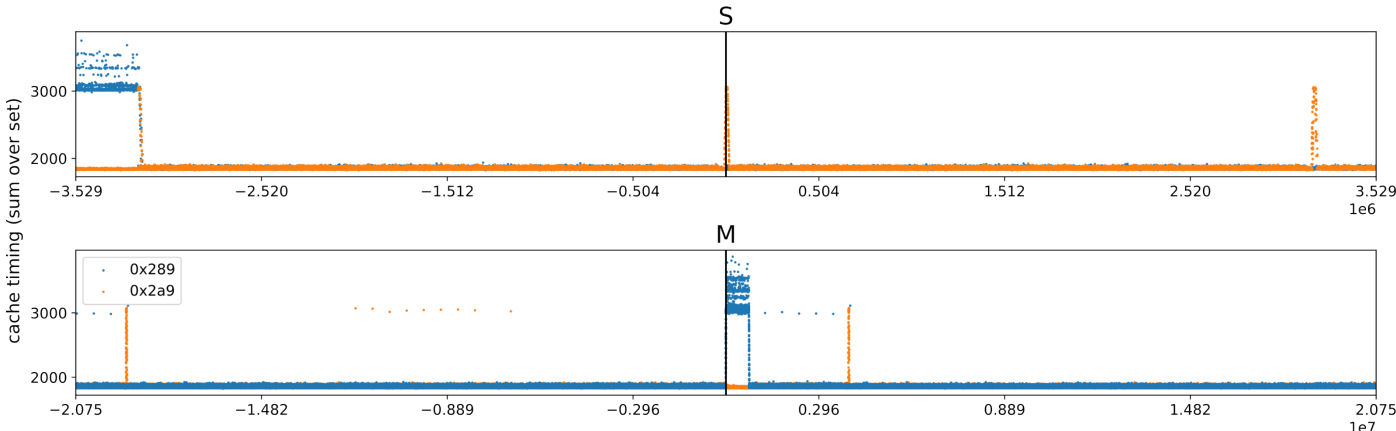


Figure 4.19: Zoomed in timing traces plotted relatively to *KEPs*. The black vertical lines mark the moment a *KEP* was triggered. Timings are plotted relative to this moment.

4.7 Conclusion

Built upon our virtual Platform, TEE-Time shows how we can use simulation and instrumentation to build tools that allow us to gain insight into the security properties of complex cryptographic scenario. It creates an environment where we can easily explore attack complexity and build demonstration attacks against cryptographic implementations that would have seen otherwise impossible to study. With this environment, TEE-Time can propagate algorithm security properties symbolized by *Key Execution Points (KEP)* to properties observable in the simulator caches without any human intervention. The first phase of TEE-Time, *VictimScan*, tries to find cache states that would allow detection of these *KEPs*, breaching the security properties of the scanned algorithm. This is possible thanks to a theoretical model of cache timing attacks, which exploit key features of cache dumps called *Key Detectable States (KDS)*. These *KDS* can then be linked to *Prime+Probe* attack configurations. This way, *VictimScan* proposes an attack configuration aiming at detecting our set of *KEPs*. The second phase of TEE-Time, *Attack Monitoring*, verifies that the previously found cache timing attack can be used by an attacker to detect *KEPs* in the victim algorithm. It runs an attack in the simulated environment and combines its results with direct victim monitoring. These two phases can be used on any program running in the TEE to automatically verify if it is leaking information and propose an attack that can be used to verify that. Indeed, we demonstrated them first on a small demo and then on a real cryptographic function (a large integer exponentiation). To further our research, we have to choose a real platform that applies *security through obscurity* to demonstrate that our findings with TEE-Time can scale to real hardware.

Chapter 5

Rockchip-platform: An accurate simulation model for a real TEE hardware

Contents

5.1 Introduction **82**
5.2 About the RockPi4 and its RK3399 . . . . . . **82**
5.2.1 CPUs, caches, and bus topology 82
5.2.2 *RK3399* boot process 83
5.2.3 Security features 84
5.3 PyDevices: fast prototyping with gem5 . . . . . . **85**
5.3.1 PyDevices: programming model 85
5.3.2 Building a RockPi4 in *gem5* 86
5.3.3 Retro engineering with PyDevices and *Ghidra* 87
5.3.3.1 Bootstraping until the OS 88
5.3.3.2 PyPowerState and Power Management Unit 89
5.3.4 *Rockchip-platform* environment 89
5.4 Using TEE-Time and Prime+Probe on the Rockchip-platform **90**
5.4.1 Detecting cache configuration 90
5.4.2 *AutoLock* and *Prime+Probe* 91
5.4.3 Pseudo-LRU: LRU implementation on real hardware 92
5.4.4 Running an attack on the *RK3399* 92
5.5 A bridge between theory and real-world: attacking OP-TEE on a RK3399 . . . . . . **93**
5.5.1 Instrumented scenario 93
5.5.2 Using TEE-Time to search for weaknesses 94
5.5.2.1 Finding good *KEPs* against *AutoLock* 94
5.5.2.2 *Attack Monitoring* and real hardware results 96
5.5.3 Extracting a key from real traces 96
5.6 Conclusion **98**

5.1 Introduction

To validate our methodology and our tool TEE-Time, we have to apply it to real hardware. This way, we could compare our simulation results with those of a real-world device. We had multiple expectations for this platform: it needs to be supported by OP-TEE to implement a secure memory protection scheme, to be used in actual consumer devices, and finally, to have available documentation.

This is why we chose the RockPi4 from RadXA with *RK3399* SoC from Rockchip. The *RK3399* is part of a line-up of chips from Rockchip with similar design and features: *RK3288*, *RK3399-T*, *RK3399PRO* and *OP1*, *RK3588*. They share similar system devices and mostly differ in their memory layout. Moreover, the *RK3399-T* and the *OP1* are functionally identical to the *RK3399*. The *RK3399* is used in multiple devices :

- Chromebooks: Samsung Chromebook Plus, ASUS Chromebook Flip C101PA,...
- Android TV boxes: H96 MAX RK3399.
- Tablet: Acer D651N-K9WT 9.7IN 4GB 32GB OP1.

RK3399 Technical Reference Manual(TRM) is widely available. We can also rely on the *U-Boot* source code for the *RK3399* which mentions hidden devices that are absent from the TRM.

5.2 About the RockPi4 and its *RK3399*

The RockPi4 variant we use is the RockPi4 C plus. It uses the *RK3399-T*, which is identical to *RK3399* in everything but CPU clock and voltages. The *RK3399* uses the ARM BIG.little architecture with a Cortex-A53 aimed at power efficiency and a Cortex-A72 aimed at performance. The *RK3399* also features: two Cortex M0, ARMv8-M CPU to be used for low-energy sleep, and a Mali T860MP4 GPU for rendering to one of the multimedia interface ports (HDMI, DP, ...). Figure 5.1 is a photo of our RockPi4 C plus, with visible *RK3399* heatspreader, DRAM chips and wifi antenna. Like the Raspberry Pi, it has a wide variety of digital high-speed interfaces (Ethernet, USB, ...) and industrial/GPIO interfaces (SPI, UART, I2C, etc.). In the *RK3399* TRM manual, we can find a memory map of all the devices in the *RK3399*(see figure A.7(a)).

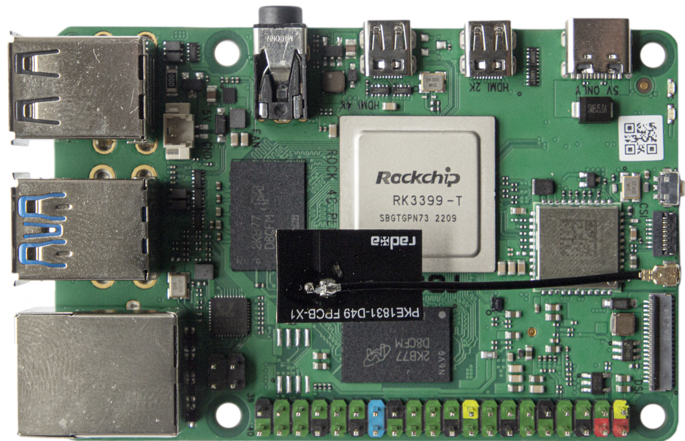


Figure 5.1: Image of the RockPi4 C plus. It has GPIO pins like the Raspberry Pi, which can be used to access a UART.

To reproduce the RockPi4 C and its *RK3399* in *gem5*, we noted several things. First, the RockPi4 is equipped with 4GB of RAM. They are located at the start of the address space. This differs from the *Vexpress* platform[1], used by default in *gem5* (mentioned in section 3.3.1.1). The GIC used in the *RK3399* is the *GIC500* which is compatible with *GICv3* standard used by OP-TEE and implemented in *gem5* (described in section 3.A.2). To reproduce the GIC in the *RK3399*, we can use the GIC *SimObject*, instantiating it at the expected *RK3399* address instead which differ from *Vexpress*. The *RK3399* also contains 2 integrated SRAMs[2]. We can reproduce them using the `SimpleMemory` model. *RK3399* contains a bootrom, noted `BootROM`, mapped at two different addresses. If we extract this bootrom, we can reproduce its implementation in *gem5* also using `SimpleMemory` mapped at the two addresses and set as read-only memories. Finally, we can exclude the two Cortex M0 and Mali T860MP4 GPU from our model as they are not used in our OP-TEE demos. Thus, communication with the RockPi4 is only done using the UART2 through GPIOs and a RS232 adapter.

5.2.1 CPUs, caches, and bus topology

By combining the TRM manual and ARM documentation for the Cortex-A53 and Cortex-A72 CPU clusters, we can have an overview of the CPU and cache topology of the *RK3399-T*, detailed in table 5.1 and figure 5.2.

[1] *Vexpress* DRAM starts at 0x80000000
[2] named *INTMEM0* and *INTMEM1*

RK3399-T	
Cortex-A53	4 CPUs at 1GHz
	In-order CPUs: -Armv8-A ISA including NEON and Crypto ext. -2 instructions fetch per cycle
	Split L1 cache: -instruction: 32kB L1(4-way) -data: 32kB L1(4-way) -Replacement policy: pseudo-random
	L2 cache: -512kB (16-way) -Cache coherency: exclusive (enforced) -Replacement policy: pseudo-least-recently-used
Cortex-A72	2 CPUs at 1.5 GHz
	Out-of-order: -Armv8-A ISA including NEON and Crypto ext. -Variable-length pipeline & Dynamic Branch Prediction
	Split L1 cache: -Instruction: 48kB (3-way) -Data: 32kB (2-way)
	L2 cache: -1MB (16-way) -Cache coherency: inclusive (AutoLock[Gre+17]) -Replacement policy: pseudo-least-recently-used
Cache line	64 bytes

Table 5.1: RK3399-T: CPU and cache information gathered from ARM and Rockchip TRM documentation.

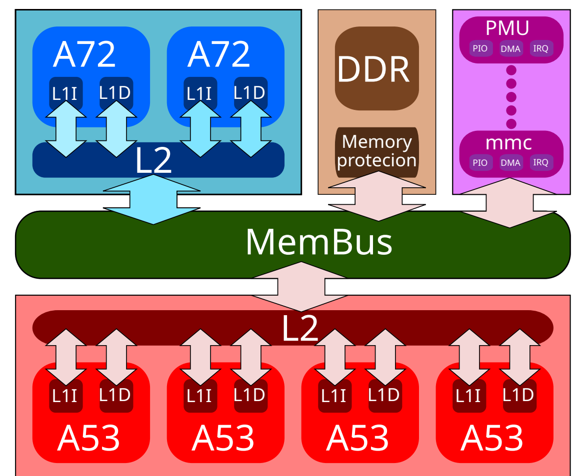


Figure 5.2: CPU architecture and cache hierarchy

The two clusters interface with the system bus using AXI Coherency Extension Protocol (ACE). They interface with the CCI500, which is responsible for cache coherency between the two clusters. The CCI500 then interfaces with all the devices using two network-on-chip (figure 5.3). Among all the devices, we see two DRAM controllers located on the main interconnects. These DRAM controllers, which support *DDR3/DDR3L/LPDDR3/LPDDR4*, have to be configured to use the installed RAM chips through internal registers. Consequently, the memory hierarchy of the *RK3399* is more complex than the *Vexpress* platform (mentioned in section 3.3.1.1) and its implementation in *gem5*. But as a starting point, to reproduce the *RK3399* in *gem5*, we can keep using the same CPU (in-order and out-of-order), caches, and memory model and then relying upon *Params* to configure them to be closer to *RK3399*'s expected results. Then, we can also leverage simpler models to explore *RK3399* and only simulate accurately what we need for our attack (typically the Cortex-A72 cluster as it contains fewer CPUs). Similarly, the *gem5* DRAM controller model can be configured to imitate timing (refresh, CAS, etc.) of JEDEC-compliant[3] RAM controllers. As we know how our RockPi4 configures the *RK3399* DRAM controller, we can just use its final configurations as our static DRAM model for our simulation. However, for our cache-timing endeavor, we can no longer ignore the specific ARM cache replacement policies. This is at this point that we added our **AutoLock** implementation in *gem5* in case it was needed to reproduce faithfully *RK3399*'s results (see section 3.3.1.1)

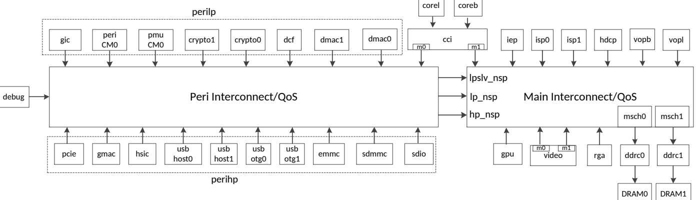


Figure 5.3: System bus structure of the *RK3399*, it feature two interconnects which links CPU with all the memories and devices.

5.2.2 *RK3399* boot process

The *RK3399* was also chosen because it can be configured to use OP-TEE. It contains a simple bootrom (noted `BootROM`) at address `0xffff0000`. This integrated `BootROM` can load the next bootloader steps from multiple sources. It checks them in this order: ① SPI, ② eMMC ,③ SD card, ④ USB development tools. Using the integrated eFUSE, it is possible to force the loaded bootloaders to be signed with a key contained in the same fuse.

The first boot step is loaded in the on-chip SRAM as DRAM has not yet been initialized. We use the boot process detailed on figure 5.4. It starts from the `BootROM` and uses bootloaders included on the SD card at specific block addresses indicated on figure 5.4. These bootloaders are:

1 **U-Boot TPL**: it configures the DRAM from the SRAM, it then returns to the `BootROM` which loads **U-Boot SPL** in the DRAM.

[3] JEDEC[JED58] issues the widely adopted standard for DRAM chip and controller

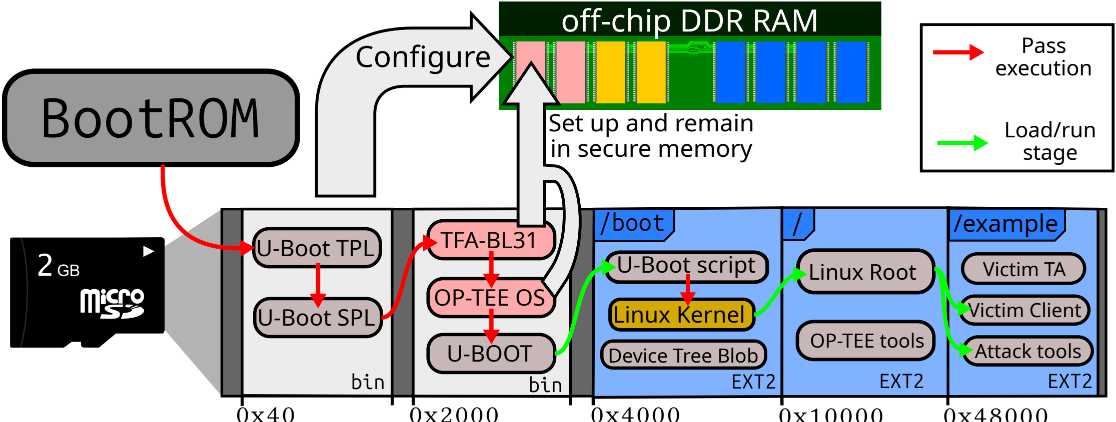


Figure 5.4: *U-Boot* assembled *RK3399* boot process. While the BootROM is embedded in the SoC, all bootloader stages are included in the SD card.

2 **U-Boot SPL**: it loads the steps 3,4 & 5 in DRAM and the SRAM.

3 **the TrustedFirmware-A BL31**: our secure monitor starts by configuring the device. It then dispatches OP-TEE and our last *U-Boot*. The *RK3399* only uses the *BL31 EL3 Runtime Firmware* from *TrustedFirmware-A* (mentioned in section 3.4.2).

4 **OP-TEE secure OS**: it boots, configuring its kernel and setting up the DRAM protection.

5 **U-Boot as bootloader**: It adds the memory node to the DTB using platform registers. Finally, it loads from an EXT2 partition, the Linux kernel, and its DTB into the DRAM.

6 **Linux**: it boots while interacting with the *TrustedFirmware-A* and OP-TEE using the OP-TEE driver.

As we see on figure 5.4, all the bootloader steps are included on the SD card which is programmed using a single disk image. Most bootloader binaries are directly written in the image without using a filesystem. The SD card also contains two EXT2 partitions, the `/boot` partition, which contains the Linux kernel and DTBs, and a Buildroot partition containing the system root (`/`). This Buildroot partition is similar to the one used by the *Vexpress* platform and contains the OP-TEE library and daemon.

This single disk image can be written on an SD card. Loaded in the SD card slot of the RockPi4 board, the *RK3399* boots as expected while printing debug information to the UART2. This UART is accessible through the PIN header present on the board. Linux automatically mounts the root partition that we made using Buildroot. When the boot is finished, we can interact with the Linux command line using the UART.

5.2.3 Security features

The RK3399 has multiple devices reserved for a *secure OS* running in TrustZone. As mentioned before, it contains a secure eFUSE (efuse1). They are used by the original integrated `BootROM` to verify the first boot-loaded stage in our scenario from the SD card. The RK3399 features a programmable access controller that can protect memory and devices to ensure they are only accessible from the secure world (EL1S or EL3). This programmable access controller is used to protect the Trusted-Firmware BL31, which remains the integrated SDRAM. It is also configured by OP-TEE to create a secure 32ăMB partition in the DRAM memory[4]. OP-TEE secure OS, and TAs reside in this memory region which can not be accessed from the Rich OS. This access protection which enforces TrustZone security properties is not necessarily present in OP-TEE-supported platforms. To configure these functionalities, OP-TEE, and the bootrom uses specific devices[5] only accessible from the secure world (EL1S or EL3). Indeed, OP-TEE reports that it is using memory protection:

```
D/TC:0 0 platform_secure_ddr_region:35 protecting region 1: 0x30000000-0x32000000
```

[4]Up to 8 DRAM regions (RGN0 to RGN7) can be defined with the RK3399 access controller.

[5]The PMUSGRF and SGRF features register to configure access protection and CPU wake up in EL3, see table A.7(c) for system peripheral name and definition.

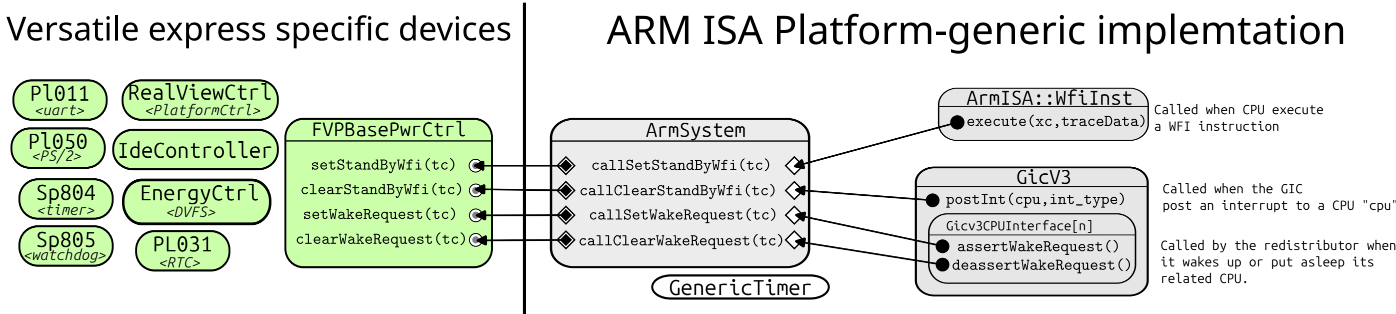


Figure 5.5: To support *Versatile express platforms (Vexpress)*, *gem5* ARM ISA implementation contains specific devices that behave like memory map IO. They can be connected to the system bus using ports in config files. However, to implement the `FVPBasePwrCtrl`, *gem5* integrates some of its functions directly into more general ARM ISA implementation in C++.

For example, the Raspberry Pi does not feature such protection, and OP-TEE Memory is accessible from Linux. On *RK3399*, a secure timer is also present. It is started by the *BL31* but stays unused.

5.3 PyDevices: fast prototyping with *gem5*

ARMv8-A has a wide variety of platforms with different devices, different memory maps, and different boot methods. On *gem5*, only the *Versatile express*-type platform (*Vexpress*) was implemented (described in section 3.3.1.1). This type of platform is mainly represented as a demonstration board from ARM and also virtual models like the *ARM FastModels*. The *BCM283X* for the Raspberry PIs and RK339X from Rockchip are completely different platform types and have different memory maps and booting processes. The *Vexpress* platform in *gem5* is implemented through specific devices provided as *SimObjects* (UART, WatchDog, etc..) and directly in assumptions, made in ARM ISA implementations (`ArmSystem` and *Vexpress* `PowerControler` are directly linked) and in generic ARM devices (GICs) implementations. This way the ARM implementation creates a direct link with other devices to follow the *Vexpress* specifications highlighted on figure 5.5.

Implementing a different *platform* would require writing a *SimObject* for each device and each modification would require a lengthy compilation. Taking this into consideration, we designed a fast prototyping interface in *gem5*: **PyDevices**.

5.3.1 PyDevices: programming model

PyDevices use the Python interpreter already integrated into *gem5* to implement the device and platform behavior. Thus, they use the *Python class* inside config files and its associated Python Object to not only hold parameters for the *Param Class* and *CcObject* but to also contain the method associated with the device. These methods are directly implemented in Python inside the config file. This way, they are loaded at runtime and not at compile time. This allows the fast prototyping of devices in *gem5*. To implement a platform-specific device, a *Python class* inheriting from PyDevices classes can be used. In that context, memory-mapped devices all inherit from the `BasicPioDevice` class which provides the base logic for a memory map device and the *Param* related to that concept: device address range, device memory latency, and a device port to connect the device to a bus. These parameters can be tuned to propose multiple instances of the same device at different addresses.

PyDevices also have at their disposal different sets of functionalities, each corresponding to one of three classes of *SimObjects* from which a Python config class can inherit:

- **PyPio**: These are the basic devices. They implement a memory-mapped IO device. They only require the implementation of two methods: `read` and `write`. They will be called respectively, when the device receives a read request or a write request.
- **PyDMA**: It includes PyPio functionalities, providing additional services. They implement DMA transfer as *CxxMethod*: `dmaRead` and `dmaWrite` methods can be used in Python to read and write from the physical memory.
- **PyInt**: It provides the ability to raise and cancel an interrupt using *CxxMethod*.

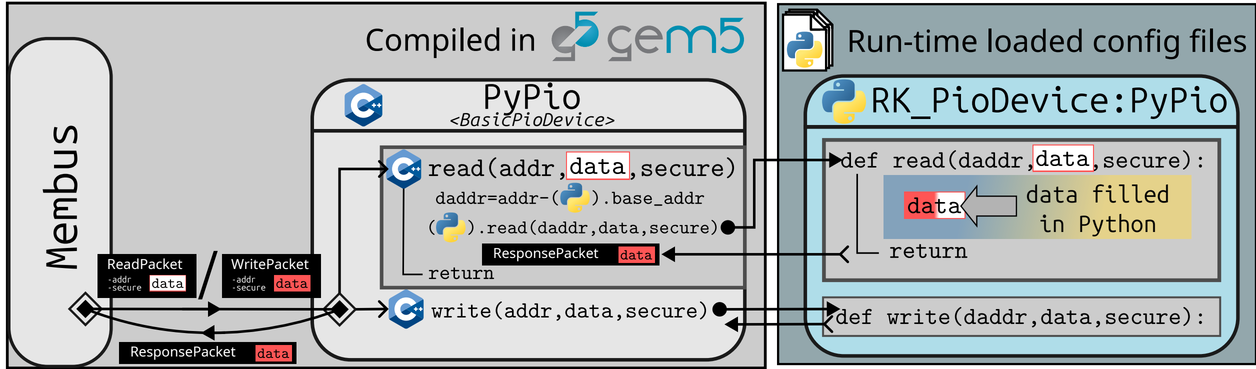


Figure 5.6: Representation of execution flow between C++ *gem5* code and Python config file code for *PyPio*.

These three classes are globally known as *PyDevices classes*. Thereby, PyDevices introduces a significant change from *CxxMethod* in *gem5* by allowing C++ to directly call config files-defined Python functions.

With the *pybind* API, mentioned in figure A.4 and figure A.5, it is possible to *override* the `read` and `write` method expected by the `BasicPioDevice` *CcObject*, redirecting their functions to the Python code in the config file which can then fully implement the device behavior in a runtime-defined manner. The execution flow of *PyPio* devices is illustrated on figure 5.6. Following *gem5* typical behavior, a packet addressed to the PyDevice arrives and is handled in the C++ implementation. In figure 5.6, this packet arrived from the membus to which the port of the *PyPio* object is connected. In the *PyPio* implementation, the `read` and `write` methods retrieves the Python *SimObject*. Then, they transform the packet and transfer it to the Python code in the config files for `read` and `write`. This way, the *CcObject* delegates the implementation of *read* and *write* responses to the Python code in the config files. This Python implementation is in charge of filling the data for *read* packet, as shown on figure 5.6. Since the device's detailed implementation is loaded at run time, it can be easily updated to polish the device implementation.

In addition, PyDevices also provide two special *Params*:

- `regs`: which maps a string to 64bit unsigned integer. It holds the value for all the device registers.
- `saved`: which maps a string to another string. It holds other values to be saved in *gem5* checkpoints.

These two parameters are taken as references to the Python value by the *CcObject*. This allows these parameters to be saved in checkpoints. This makes PyDevices fully checkpoint-able.

To implement certain devices, typically an SD memory controller, we have to implement a DMA interface for our *PyDevices*. As mentioned before, these functions are contained in the **PyDMA** *SimObject*: `dmaRead` and `dmaWrite`. However, because we wanted to keep the Python code for devices simple, we enforced the atomicity of DMA read and write in PyDevices. This allows DMA functions to be simple function calls that return the result of the memory access to be used in the same Python context (e.g. `data=dmaRead(addr)`). Otherwise, they would have required the use of callback functions for DMA responses (e.g. `dmaRead(addr,call_when_read_done)`).

Therefore, *timing* memory mode (section 2.2.2.4) can not be implemented in PyDMA. Thus, it uses *atomic* memory transaction when *gem5* is in *atomic* mode and *functional* memory transaction when in *timing* mode.

The implementation on figure 5.7 is an example using *PyPio* class. This `RK_efuse` can be instantiated in config files with `efuse0=RK_efuse(pio_addr=<device address>,pio_size=<device size>,secure=True)`. Indeed, multiple instances of the same device model can be created using a single implementation with different details passed as *Param* to the *CcObject* or just used as simple attributes for the Python implementation. In `RK_efuse` case, we have secure and unsecure *efuse* at different addresses.

5.3.2 Building a RockPi4 in *gem5*

With PyDevices and the rather complete TRM manual for the *RK3399*[Roc21], we can start implementing the necessary devices to boot our RockPi4 workload in *gem5*. This workload is made of two elements: our SD card image that we created and the integrated `BootROM` included in the *RK3399*. Thus, we had to extract this `BootROM` using a modified TPL to output its content to the UART2. We then designed new config files, to start reproducing the *RK3399* platform in *gem5* by integrating both already existing *SimObject* (RAM, CPU, GIC, etc.) and new PyDevices-implemented *SimObject*(UART, fuse, etc.).

rk_efuse.py

```python
class RK_efuse(PyPio):
  _REG_NAMES={
    0x0000: "EFUSE_CTRL" ,#efuse control register default:0x00000000
    0x0004: "EFUSE_DOUT" ,#efuse data out register default:0x00000000
    0x0008: "EFUSE_RF" ,#efuse redundancy bit used indicator register default:0x00000000
    0x0010: "EFUSE_JTAG_PASS" ,#Jtag password default:0x0cf7680a
    0x0014: "EFUSE_STROBE_FINISH_CTRL" ,#efuse strobe finish control register default:0x00009003
  }
  _REG_DEFAULT={
    "EFUSE_CTRL":0x0,"EFUSE_DOUT":0x0,"EFUSE_RF":0x0,
    "EFUSE_JTAG_PASS":0x0cf7680a,"EFUSE_STROB_FINISH_CTRL":0x00009003,}
  def __init__(self, amba_id=0x0,secure=False,**kwargs):
    super(RK_efuse,self).__init__(amba_id=amba_id,**kwargs)
    self.regs=self._REG_DEFAULT.copy()#passing the regs default value
    #this value is replaced if we load a checkpoint.
    self._secure=secure#saving if efuse device instance is secure
  def read(self,daddr,data,secure):
    if not daddr in self._REG_NAMES.keys():
      self.print(self,"error unknown register:",hex(daddr))
      return False
    reg=self._REG_NAMES[daddr]
    if reg=="EFUSE_DOUT":
      #Do something specific
    else:
      set_int(data,self.regs[reg])#classical "reading a register" behavior
      self.print_dbg_reg(data,daddr,reg)
    return True
```

Figure 5.7: Typical PyDevices implementation. A system can have multiple instances of the same devices with different settings.

Name	Offset	Size	Reset Value	Description
EFUSE_CTRL	0x0000	W	0x00000000	efuse control register
EFUSE_DOUT	0x0004	W	0x00000000	efuse data out register
EFUSE_RF	0x0008	W	0x00000000	efuse redundancy bit used indicator register
EFUSE_JTAG_PASS	0x0010	W	0x0cf7680a	Jtag password
EFUSE_STROBE _FINISH_CTRL	0x0014	W	0x00009003	efuse strobe finish control register

*Notes: Size: **B**- Byte (8 bits) access, **HW**- Half WORD (16 bits) access, **W**- WORD (32 bits) access*

Figure 5.8: Extract from the *RK3399* TRM[Roc21]: register description for the *efuse*

5.3.3 Retro engineering with PyDevices and *Ghidra*

Ghidra is a retro-engineering tool that provides an IDE to disassemble a compiled binary[Roh19].

Ghdira can connect to *GDB* to follow an execution to better understand how it works. It uses MI[6] to interact with an already-started *GDB*:

```
new-ui mi2 /dev/pts/1024
```

We can use the *RK3399* memory map described in the TRM (figure A.7(a)), to instantiate all the memory-mapped devices. We can then use their register definitions when they are described in the TRM. For example, the TRM extract on figure 5.8 lists all the registers for the integrated fuse module, called *efuse*. It has been used to build the *PyPio* example shown on figure 5.7, providing the value of `_REG_NAMES` and `_REG_DEFAULT`. This table format is kept through the TRM for other devices which allowed us to write dummy device implementations with register usage reporting. When this definition is not available, we can use a dummy device to fill the memory space. These definitions are runtime loaded and can be updated to implement `read` and `write` behavior for each register of devices. In addition, because we are writing the device implementations directly in Python, we can use the *GDB* API (see section 3.A.1) that we developed for config files. With this API we can:

- Stop execution when a specific device or memory map register is accessed.

[6] Machine Interface is a text-based protocol developed by *GDB* that allows a debugger to be used as a separate component of a larger system.

- Print in the *Ghidra* console the information we have about any memory access to a memory-mapped device (data, read-or-write, register name, etc.).

Thus, as the *Ghidra* console is connected to *GDB*, we can have information directly in *Ghidra* about memory-mapped registers and devices (figure A.9). This was a key element to retro-engineered part of the first stage of the bootrom. Indeed, using the *GDB* API we can stop the *Ghidra*-monitored execution when an unimplemented device is accessed while reporting which register was accessed. We can then upgrade the config files implementation for the needed registers (using the TRM and *Ghidra* execution context), adding new behavior to `read` and `write` functions. We can then verify that the new behavior allows the boot process to progress further by restarting the simulation without needing to recompile *gem5*. With this fast-prototyping process, we can implement new SoC in *gem5*.

5.3.3.1 Bootstraping until the OS

With our retro-engineering environment set up, it is now possible to progress in the boot phase, implementing devices when they are needed relying on the Rockchip TRM manual[Roc21]. In this subsection, we follow along the *RK3399* boot process to highlight issues we faced and what we implemented with PyDevices to overcome them. Unused devices only have dummy implementation in our virtual platforms. As mentioned in the manual, only the CPU0 (the first CPU of the A53) is running when the *RK3399* is powered on. Other CPUs stay powered off until Linux starts booting.

After multiple calls to the eFuse which do not block the simulation, the boot process goes through all the boot devices. As we are booting using an SD card, we only need to implement the SDMMC. Indeed, after unsuccessful interaction with the SPI and the eMMC as their implementations are incomplete, the `BootROM` tries to use the SDMMC to load the first boot image.

At this point, we implemented the SDMMC using the *PyDevices* in order to imitate the SD card. To do that, we kept the disk image which was written on the real SD card. Our SDMMC implementation recreate:

- SDMMC registers which are used to configure an interface to send and receive commands from the SD card. Our SDMMC implementation provides the proper response for the command using our disk image.
- Integrated FIFO which is used to pull data sequentially from the SD card.
- DMA access to automatically copy data from the SD card using the *PyDMA*. They are used to implement SDMMC DMA operation called: IDMAC[7].
- SD registers which are accessed using commands. They are used to detect the size and settings of the SD card. They are configured to be identical to the real platform if the disk-image perfectly fit in the SD card. Thus, we imitated a 1GB SD card.
- Interrupts which are needed for the SD-MMC to work properly in Linux.

With the SD card implemented, our boot process can progress until the Linux image boots. To follow this boot process, we use the UART2[8] accessible through a RS232 adapter on the real board and *GDB* (and a text log) for our simulation. However, since we are not using the configurable DRAM model, but a static model proposed by *gem5*, we have to completely ignore the DRAM initialization process. Avoiding this phase created an issue as it is used to detect the DRAM configuration. It is then kept in a device register[9] to be used by other boot stages. To circumvent this issue, we overrode the default value with one extracted from the real platform. Another issue faced in the boot process happens in the *TF-A BL31*, It sets a reset address, `RESET_ADDR`[10] which will be used for other CPUs which are still asleep. This is called *warm boot*. We had to modify *gem5* to allow this reset address to be changed after the simulation started[11]. We did not find any issues with OP-TEE secure OS in the boot. OP-TEE boots after the *TF-A BL31* finishes setting up the platform. OP-TEE does not use more devices than in previous phases. As we mentioned in section 5.2.3, OP-TEE configures its memory region to be secure using the SGRF.

Linux is loaded by the last *U-Boot* bootloader. After its initialization, it relocates itself to make spaces for the kernel image in the DRAM. It then initializes all the devices to prepare for bootloading Linux from the MMC. To Load Linux, *U-Boot* executes an integrated *U-Boot* command line using the *U-Boot* shell. We modified the default *U-Boot* configuration for the *RK3399* to load a *U-Boot script* that we placed in the `/boot` EXT2 partition. This script is in charge of loading Linux from the `/boot` partition and running it. Our script contains a customized Linux command line,

[7] Internal Direct Memory Access Controller from the Synopsysső DesignWareő Mobile Storage Host (SD/MMC controller) controller.
[8] This UART follows the *8250* specifications[L B24]
[9] `PMUGRF_OS_REG2` in the PMUGRF (see table A.7(c) for system peripheral name and definition.)
[10] This register is in the PMUSGRF (see table A.7(c)).
[11] Using *CxxMethod* we added in the `System` *SimObject*.

to specify the `root` partition UUID and enable specific scheduling features. All the other boot stage binaries are in their default *RK3399* configuration. When Linux has been loaded, it begins its own boot, configuring pagetable and detecting devices using the DTB from `/boot` partition. Until this point, only one CPU is needed and all the other CPUs have to stay suspended.

5.3.3.2 PyPowerState and Power Management Unit

At some point in Linux, other CPUs are launched using the PSCI interface which is called using the SMCCs as it is implemented in the secure monitor (*TF-A BL31*). The *TF-A BL31* uses the PMU (Power Management Unit) to start the secondary CPUs. Indeed, the PMU controls the power domains in the *RK3399*. When the power domain associated with a CPU is switched on, this CPU starts at an address set in the SGRF: `RESET_ADDR`. As the PMU interface is completely different from *Vexpress*, we had to implement dedicated power control mechanisms to change CPU behaviors in response to PyDevices register modifications. To control the power state of CPUs, *gem5* uses a `PowerState`, an integrated *SimObject* that controls the power status of the *SimObject* holding it (i.e. `sim_object.power_state`). To implement these power domains in Python, we created `PyPowerState`. They gave the Python environment methods to access and set `PowerState`. They also require a callback method to be implemented. This method is called by every `PowerState` update done in *gem5* C++ code in order to allow the `PyPowerState` to override it. The PMU in the *RK3399* provides registers to control power domains: for most power domains, we only track domain state but, for CPUs, we have to implement the effect of CPUs being switched on and switched off (figure A.7(c)).

For CPU power domains, we used *CxxMethod* in CPUs to start and resume execution when power domains are switched on and off. We specifically, implemented a `FORCED_OFF` *gem5* power state to prevent CPU from turning back on, when the power domains are manually switched off. When resuming execution, we also have to reset CPUs (which implies an ISA state reset). In that case, we use the `RESET_ADDR` set by *BL31* to enable warm boot.

Finally, the PMU is used to implement PSCI (Power State Coordination Interface)in the *BL31*: specific registers are used to enable CPU power domains to be switched off when the CPU encounters a WFI instruction (Waiting For Interrupt), and being switched back on when receiving an interruption from the GIC. As we can see in figure 5.5, on *gem5 Vexpress* platform, the GIC directly interacts with WFI instruction and the FVP power controller(*FVPBasePwrCtrl*) through the `ArmSystem`. We modified *gem5* to allow our Python implementations to react to these events and provide their own responses, With our modifications, *gem5* transmits these events to Python methods implemented in the `PyPowerState` associated with their respective CPUs:

- `setStandByWfi()` is called when the associated CPU enters WFI
- `clearStandByWfi()` is called when the associated CPU leaves WFI
- `setWakeRequest()` is called when the GIC wants to wake the associated CPU. it can return true to trigger a CPU reset as the CPU wakes up.
- `clearWakeRequest()` is called when the GIC wants to clear a pending wake request to the associated CPU.

With these in `PyPowerState`, we can implement the PMU and the CPU power domains. With our PMU simulation, the first CPU can wake up secondary CPUs during boot and all the CPUs can correctly respond to GIC wake requests, which allows them to support PSCI implementation in the *BL31*. CPUs can now sleep and wake up naturally which allows the boot to progress until the Linux shell command line (figure A.8).

5.3.4 *Rockchip-platform* environment

Now that we can boot the same disk image in both real-platform and simulation, we have to set up how to run programs compiled on our host machine and interact with them using our *GDB*-debug sessions. Indeed, we want to keep our previous use cases, as they were compatible with TEE-Time and were representative of what a real attacker could do against a client-TA scenario. But we also want to share the same disk image and scripts between the simulation and real platforms. To reconcile these two conflicting ideas, we choose to implement a dummy device called `m5_device`, using PyDevices. This device is located in an unused memory space (`0xFF17000`) of the *RK3399*. It allows us to distinguish between simulation and real platform, using shell command `devmem`. `devmem 0xFF17000 32` returns `0xdeadbeef` only in simulation, on the real platform it returns `0`. With this function, we modified our `init.d` script to perform some operations only in simulations:

- Taking the boot checkpoint, as we still want to avoid having to simulate the boot phase each time we want to run a new simulation. This checkpoint is still known as *BootPoint*.

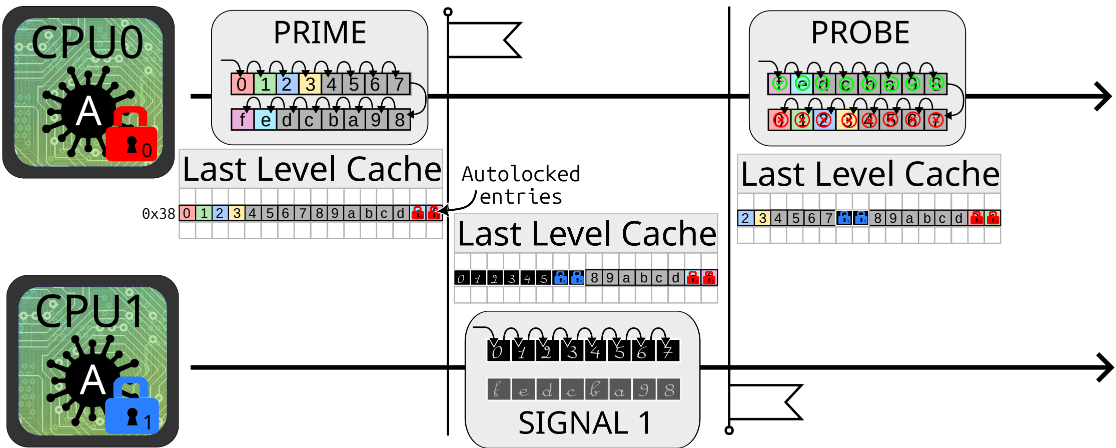


Figure 5.9: Covert channel primitives used to detect *AutoLock* on our RockPi4 and our rockchip platforms. This figure shows how this *signal primitives* works when *AutoLock* is active. CPU1 sends a 1 by priming 8 entries of its prime set. CPU0 checks if it receives a bit using Prime+Probe (with its own prime set).

- Importing our initial script using `m5 readfile`, which is not located in the disk image but on the host and thus can be changed between runs.

Indeed, we kept our Buildroot configuration that adds OP-TEE tools and `m5` executable. But, as we have only, one disk medium available, we have to put our attack scenario (attack, client, TA, scripts, etc.) on the same disk image as the root partition. We therefore created a partition that fills the rest of our SD card (1GB). This partition called `/example` on figure 5.4, is dedicated to our attack scenario and is only mounted after the boot checkpoint in order to avoid issues with modified filesystem nodes. To reproduce the same attack scenario, between the two platforms, we use a shell script file that performs our scenario when run. This attack scenario's script is executed differently between simulation and real platform:

- On the simulation platforms: our `init.d` uses `m5 readfile` to load a initial script. Since this initial script is loaded using `m5 readfile`, it can be different between runs without having to regenerate our *BootPoint*. This initial script is responsible for configuring and executing the attack scenario's script.
- On the real platform: our `init.d` automatically mounts the secondary partition and then waits in the Linux shell (figure A.8). The attack scenario's script has to be run manually.

This use case is the base of the **Rockchip-platform**. It combines virtual and physical platforms to build scenario compatible with both environments. With our *Rockchip-platform* a scenarior can run in RockPi4 and *gem5* without any modifications. Furthermore, we can still override attack and victim parameters in simulation settings to test different configurations.

With our *Rockchip-platform*, we can run demonstration programs to compare results between our simulated and real *RK3399*. Since we want to perform cache timing attacks, we need to tune our simulation to closely resemble our RockPi4 cache-wise. By running the same example on both simulation and real platform, we can thus tune the cache and CPUs *SimObject Params* (`tag_latency`, `data_latency`, `response_latency`, etc.) to have similar timing results between simulated and real platforms.

5.4 Using TEE-Time and Prime+Probe on the *Rockchip-platform*

To run our attack scenario, we chose to use the Cortex-A72. The attack runs on one CPU while the victim program runs on the other. We thus need to fine-tune the A72-L2 cache model and L1 cache models to correctly reproduce its behavior in our simulation platform.

5.4.1 Detecting cache configuration

To study the Cortex-A72 cache hierarchy, we use a demo *Prime+Probe* attack that demonstrates *a cache-covert channel* between two of its threads that share the same L2 cache in the Cortex-A72. using different transmitting primitives

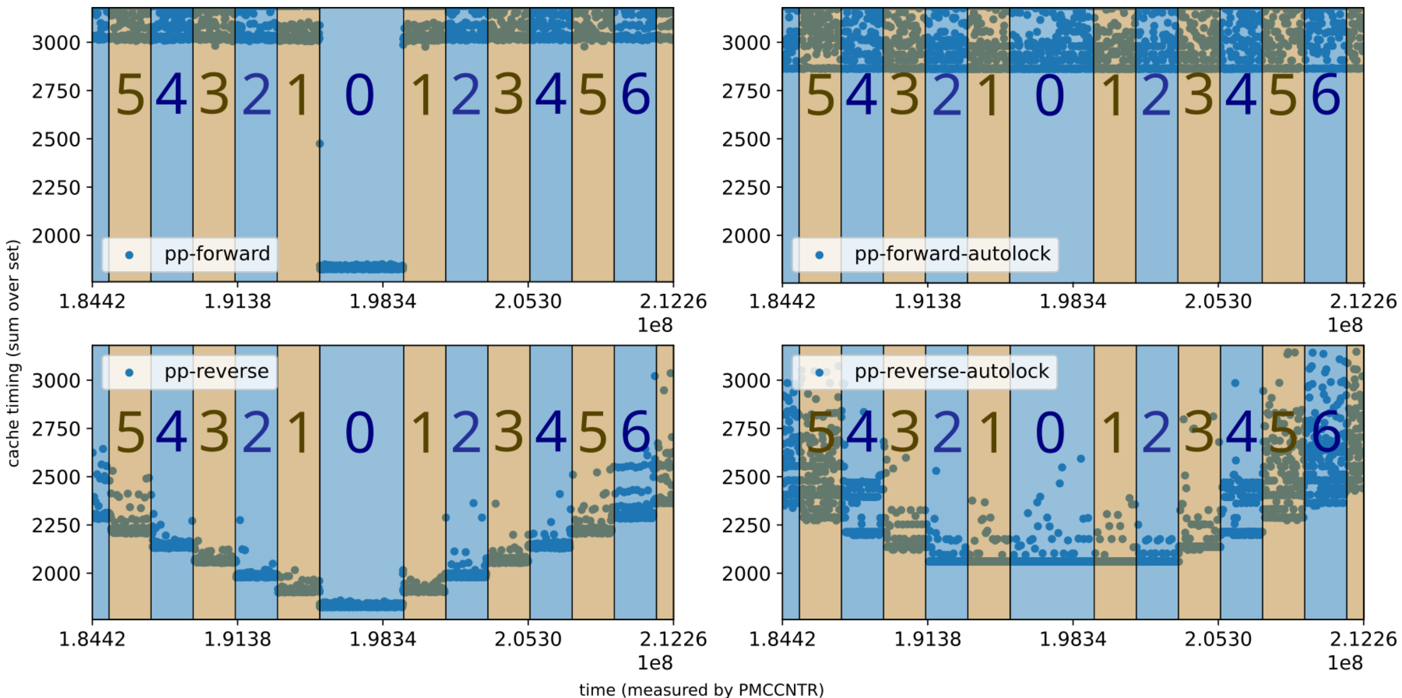


Figure 5.10: How *Prime+Probe* interacts with *AutoLock*: above are *Prime+Probe* forward and below are *Prime+Probe* reverse. On the left, without *AutoLock*, and on the right, with *AutoLock*. The victim uses cache occupancy, indicated as colored rectangles, to send a stair signal clearly visible on *pp-reverse*.

that fill the cache (figure 5.9), the sender thread communicates with the receiver thread. The receiver thread detects these cache states using specific *Prime+Probe* direction. This means that for the receiver, a 1 corresponds to a specific timing value for the prime set and a 0 to a different one. The sender primitives thus create different cache occupancy patterns, which are of course influenced by the actual cache replacement policy. Thus, we chose these primitives to show different behavior between when *AutoLock*(see section 3.3.1.1) is used and when it is not. On figure 5.9, we represented locked entries for each CPU. The *Probe* phase shows different results because of the blue padlocks line. They represent lines locked because of the CPU1 L1 cache on which the sender thread runs. We run the covert-channel demo in both simulation and real platforms, between the two CPUs in the Cortex-A72 cluster (figure A.10).

With these results, we confirm that the *RK3399* uses *AutoLock* in its Cortex-A72 cluster. Indeed, we see the two lines being locked on figure A.10. These lines cannot be evicted because of *AutoLock* since the sender CPU holds them in its L1D. Indeed as we mentioned in section 3.3.1.3, *AutoLock* prevents lines in a L2 cache from being evicted if they are held in a L1 (see figure 3.14). We have already implemented an *AutoLock* model that we can enable in our *gem5* platform. After enabling it and tuning the cache model *Params*, we got the result on the right of figure A.10.

Although we now have a simulation platform that produces similar cache timing results to our real RockPi4 platform, *AutoLock* modifies our assumption on the cache model with regard to *Prime+Probe*. This implication suggests that we review our TEE-Time configuration, under the light of *AutoLock*.

5.4.2 *AutoLock* and *Prime+Probe*

As explained in section 5.4.1, *AutoLock* prevents the eviction of lines already present in L1. This mechanism prevents some elements of the prime set from being *hit* because they can never evict the victim cache lines. This means that certain attack output values become less likely because some element of the prime set entry may be forced to be *misses*. We propose the figure 5.10 to visualize, that complements the figure 4.5, by comparing the different directions of probing, with and without *AutoLock*. On the bottom left, which corresponds to *Prime+Probe* reverse traces, each stair level corresponds to a $\sum T_{hit\text{-}miss}(o)$ (with o from 0 to 7) associated with a cache occupancy o. The same victim behavior produces the trace on the top left when using *Prime+Probe* forward, with only two value $\sum T_{hit}$, the lowest, and $\sum T_{miss}$, the highest.

When we enable *AutoLock*, we have the two plots on the right of figure 5.10. On them, because of *AutoLock*, the T_{hit} output and $T_{hit\text{-}miss}(o)$ for $o \leq 2$ are no longer distinguishable. This is caused by the 2-way cache L1. Consequently,

AutoLock hides some T_{hit} outputs from *Prime+Probe* forward. For this reason, to still be able to use *Prime+Probe* despite *AutoLock*, we have to use *Prime+Probe* reverse.

From these considerations, we conclude which configuration should be used for TEE-Time on the *RK3399*: *nhit VictimScan policy* and its associated attack, *Prime+Probe* reverse.

5.4.3 Pseudo-LRU: LRU implementation on real hardware

Real hardware platforms implement variations of the LRU cache replacement policy, called pseudo-LRU. In *gem5* LRU is implemented using using *"simulation tick"* timestamp to choose eviction victims. On the contrary, pseudo-LRUs on real hardware approximate the LRU behavior using simplifications that do not require timestamps and are also quicker to compute. These implementations still statistically behave like LRU. However, they do impact *Prime+Probe* by unexpectedly evicting prime set entries. This is visible in outputs, with entries being swapped and no longer being split between *miss* and *hit* (like in equation 4.14).

$$\{\underbrace{t_{hit}, .., t_{hit}}_{\text{assoc}-o}, \underbrace{t_{miss}, .., t_{miss}}_{o}\} \rightarrow \{t_{hit}, t_{miss}, t_{miss}, t_{hit}, t_{miss}, t_{hit}, ..., t_{hit}\} \quad (5.1)$$

This effect can be mitigated by using the sum of individual traces because it does not change with set entries permutations.

On figure 5.10, we see that $\sum T_{hit\text{-}miss}(o)$ acts as a minimum threshold for real measures. This minimum threshold is not affected by pseudo-LRU cache replacement policies. However, pseudo replacement policy can also sometimes evict more entries than expected, due to entry permutation causing self-eviction. In that case, the measured timing is higher. This means that, in this situation, the lower threshold is statistically more accurate than searching for an exact value (w.r.t noise). To verify that the Cortex-A72 L2 cache uses a pseudo-LRU, we can study our cover-channel demo, searching for disorganized timings that resemble what is shown on equation 5.1. This is clearly visible on figure A.10: timings on the left do not regroup in two block of *hits*(green) and *misses*(red), instead they are randomly distributed among all the prime set entries.

gem5 proposes a pseudo-LRU model called Tree-LRU(TLRU) or Tree-Pseudo-LRU (TPLRU). This implementation uses a binary search tree which is updated to point away from the last accessed entry. We can run our covert channel using *gem5* Tree-LRU, which produces the result on figure A.11. On this figure, we see that the *hits* and *misses* in the results from the real platform are not distributed in the same way as in *gem5* simulation. Although both of them have the same number of *hits* and *misses* and also show the scrambling behavior show on equation 5.1, they have differences in the way entries are scrambled. This difference can be due to two reasons: *gem5* TLRU differs from actual Cortex-A72 pseudo-LRU or both pseudo-LRU are similar but their statistical trajectory have diverged because of variations between simulation and real platforms. In both situations, using *gem5* TLRU will not benefit TEE-Time. However, pseudo-LRUs behave statistically like LRU[TOS10]. In the best-case scenario, using *gem5* TLRU does not give more information about the real platform pseudo-LRU behavior than using *gem5* perfect LRU.

For this reason, we chose to use *nhit_inclusive VictimScan* policy to take into account the pseudo-LRU behavior. However, we keep using LRU for our simulation platform cache replacement policy as using *gem5* TLRU will only come with drawbacks and fewer possibilities of generalization.

5.4.4 Running an attack on the *RK3399*

Running an attack on the *RK3399* is more complicated than on *gem5*, software constraints for our time measurement methods can no longer be ignored. As mentioned in section 3.3, the access to the ASM `PMCCNTR` is locked. To enable this register, we developed a small kernel module that exposes through *sysfs* module. With this module, the attack program has access to:

- The ASM `PMUSERENR` to enable the performance monitor (PMU) access in EL0.
- The ASM `CLIDR` and ASM `CSSIDR`[12] to access cache properties and configure the attack automatically.
- Uncached memory source device to be used for the result vector.

This kernel module, `attack_mod.ko`, still respects our threat model as we assumed that the attacker could compromise the OS but not the secure environment.

To run attacks efficiently on a realistic platform, we also use the following tricks:

[12] The ASM `CLIDR` (Cache Level ID Register) and ASM `CSSIDR` (Cache Size ID Register) are ARM system registers accessible through ASM `MSR` instructions (see section 3.3.1.1) that provides information about the cache architecture (size, associativity, level, etc.)

- Changing the scheduling policy to real-time(`SCHED_FIFO`). This guarantees an advantageous scheduling for our attack, giving it more execution time.
- Changing OS configuration for the real-time period to 1000 seconds (`echo 1000000000 > /proc/sys/kernel/sched_rt_period_us`) while disabling RT throttling: `echo -1 > /proc/sys/kernel/sched_rt_runtime_us` [13]
- Disabling idling for the attack CPU. This ensures that the attack CPU is not powered off by the scheduler. `echo 1 >/sys/devices/system/cpu/cpu<i>/cpuidle/state<n>/disable` can disable a specific idle state n.

5.5 A bridge between theory and real-world: attacking OP-TEE on a *RK3399*

As we are now able to run an attack on the *RK3399* and have a compatible simulation platform for TEE-Time to function, we can go back to our RSA attack (presented in section 4.6) and try porting it to our RockPi4. However, as we demonstrated in section 5.4.3 and section 5.4.1, the *RK3399* uses *AutoLock* and an unknown pseudo-LRU cache replacement policy. To account for that, we have to configure TEE-Time with the *nhit_inclusive VictimScan policy* and use its associated attack *Prime+Probe* reverse. Although, for our real platform attack, we kept our Linux client application, the same TEE system calls (from the *Cryptographic Operations API* mentioned in section 3.4.3) are also used when OP-TEE is registered as a cryptographic engine for OpenSSL. This is possible thanks to the `PKCS#11` pseudo-TA incorporated inside OP-TEE.

OP-TEE integration in openssl

```
openssl
OpenSSL> engine dynamic -pre SO_PATH:/usr/lib/engines-1.1/pkcs11.so -pre ID:pkcs11 \
-pre LIST_ADD:1 -pre LOAD -pre MODULE_PATH:/usr/lib/engines-1.1/libckteec.so.0.1.0
(dynamic) Dynamic engine loading support
[Success]: SO_PATH:/usr/lib/engines-1.1/pkcs11.so
[Success]: ID:pkcs11
[Success]: LIST_ADD:1
[Success]: LOAD
[Success]: MODULE_PATH:/usr/lib/engines-1.1/libckteec.so.0.1.0
Loaded: (pkcs11) pkcs11 engine
OpenSSL> genrsa -engine pkcs11 -out priv_key.pem 2048
engine "pkcs11" set.
Generating RSA private key, 2048 bit long modulus (2 primes)
....................................................................+++++
.......................................+++++
e is 65537 (0x010001)
```

Indeed, our client application reproduces a more general use case. In this context, our attack could scale to any program that uses libSSL to perform a RSA private key exponentiation on a *RK3399* SoC.

5.5.1 Instrumented scenario

To re-use our RSA-attack scenario in the *RK3399*, we have to update it to follow our template for *RK3399* attack scenarios, presented as *Rockchip-platform* in section 5.3.4. The *Rockchip-platform* requires to use the same script and disk image to:

- Perform the attack on the real *RK3399* without any intervention.
- Configure the scenario in the simulated environment using the *GDB* interface, changing attack and victim arguments and potentially disabling them.

Using our dummy m5 device (see section 5.3.4), we can enable specific functionalities for our attack scenario when it runs on a simulation platform. This way, when it runs on *gem5*, our attack can be configured by *GDB* and output its results on the host. This is passed as a `--m5` argument to the attack programs This argument allows the attack to use the `m5_writefile` to output the trace results on the host and not in the temporary disk image. Thus, the disk image is never altered between runs. To be at the same time configurable by TEE-Time and automatic on the real platform, we also have to integrate a default argument for the victim and the attacks into our attack scenario bash script. These arguments are overridden by `m5 env` environment variable in *gem5*.

[13] if RT throttle is disabled, the kernel should not output: `sched: RT throttling activated`

```
m5out/report.txt
Score ranking:
»»M->max_hit:(('0x38', 1), 146)
(1):('0x38', 1)
     score:0.9944598337950139
     hit_count:146
     top_addr:
        1@146=S#0x300d0e00:data + 39160 in section .bss
        2@0=0x53a60e00:UKN
»»S->max_hit:(('0x346', 1), 66)
(1):('0x346', 1)
     score:0.4925373134328358
     hit_count:66
     top_addr:
        1@66=S#0x300cd180:data + 23672 in section .bss
        2@66=NoMMU;:maybe_tag_buf + 40 in section .text
        3@66=S#0x300dd180:__heap1_start + 41112 in section .heap1
```

Figure 5.11: First *VictimScan* report for the *sec-sign* TA with *KEP*s from figure 4.16, using an improved *KEP toolbox* implication and using the *nhit_inclusive policy*.

5.5.2 Using TEE-Time to search for weaknesses

With our TEE-Time-compatible instrumented scenario, we can search for potential weaknesses that can be leveraged for an attack against the *sec-sign* TA, running on the RockPi4. TEE-Time is configured with the *nhit_inclusive VictimScan policy,* and we are thus using its associated attack *Prime+Probe* reverse. We can then run this *Prime+Probe* reverse attack on our real RockPi4 using the configuration verified by TEE-Time. The runtime comparison between TEE-Time on the simulated platform and the attack scenario on the RockPi4 can be found on table 5.2.

5.5.2.1 Finding good *KEPs* against *AutoLock*

AutoLock has changed how the victim behaves and how we can detect the *KEPs* shown on figure 4.16 that we used in our first RSA attack (see section 4.6). As our instrumented *sec-sign* scenario is now run on our new *RK3399* simulation platform, we have to go through the TEE-Time process from the start (see figure 4.4). *VictimScan* will try to find *KDS*, that we can use to detect our already defined-*KEPs* despite *AutoLock*. In that context, we are using *nhit_inclusive VictimScan policy*, contrary to section 4.6, to account for *AutoLock* and *RK3399* pseudo-LRU policy. Thus, the resulting *KDSes* are made of two elements: a cache index and a set's way occupancy.

With the KEPs on figure 4.16, *VictimScan* produces the results on figure 5.11. It proposes the following *nhit_inclusive KDS*:

- for [𝕄]: $(0x38, 1)$ with score 0.994459.
- for [𝕊]: $(0x346, 1)$ with score 0.49253731.

From the report on figure 5.11, we see that the [𝕊]-*KEP* can not be accurately detected using these *KEPs*: with a score of less than 0.5, [𝕊] can not be distinguished from [𝕄], (pl. see section 4.3.3) bor is obscured by other sources of cache activity.

Despite the low score, we can use *Attack Monitoring* to generate traces to have a better understanding of what the attacker sees. *Attack Monitoring* uses our two *KDS* to configure the *Prime+Probe* reverse attack. Considering the score on figure 5.11, we expect it to detect [𝕄], but not [𝕊].

Attack Monitoring produces the results on figure 5.12. It shows how the cache timings behave in the vicinity of each *KEP*. On figure 5.12, we see that the timing associated with [𝕊]-*KDS* (0x346) does not stay low when leaving its associated section. It seems because of *AutoLock*, an attacker cannot detect a single *squaring operation* and can only detect the start of a series of [𝕊]*KEPs*.

We can see this because [𝕊] cache line timing rises when entering [𝕊] section, but it does not return unless we enter [𝕄]-section.

We propose to use different *KEP* implementations using *KEP toolbox* to better model how the victim behaves:

- We can use a normal *scoped cache tracker*(see section 4.4.2.1) for the multiply.
- We can use a *backward scoped cache tracker* (see section 4.4.2.1) for the *squaring* phase, which is registered only if a square happens during the scoped sections. The scoped entry is placed at the end of the window while the

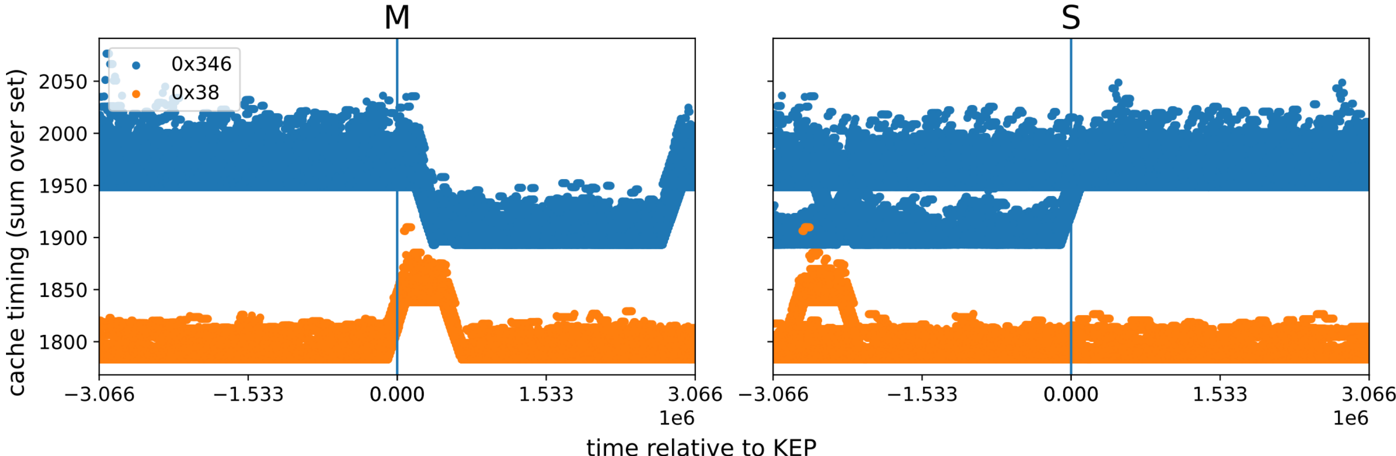


Figure 5.12: Zoomed in timing traces plotted relatively to *KEPs*. Two sets (cache set with index 0x38 and 0x346) are used in order to distinguish between the two *KEPs* ([𝕊] and [𝕄]). However, we see that we cannot detect [𝕊].

m5out/report.txt

```
Score ranking:
»»M->max_hit:(('0x38', 1), 148)
(1):('0x38', 1)
  score:1.0
  hit_count:148
  top_addr:
    1@148=S#0x300d0e00[S#0x300d0e00]:data + 39160 in section .bss
»»S->max_hit:(('0x346', 1), 66)
(1):('0x346', 3)
  score:1.0
  hit_count:66
  top_addr:
    1@66=NoMMU;[S#0x3008d180]:maybe_tag_buf + 40 in section .text
    2@66=S#0x300cd180[S#0x300cd180]:data + 23672 in section .bss
    3@66=S#0x300dd180[S#0x300dd180]:__heap1_start + 41112 in section .heap1
```

Figure 5.13: *VictimScan* report for the *sec-sign* TA with *KEP*s from figure 4.14(b) redefined, using the *nhit_inclusive policy*.

scope labeling *KEP* is placed after the `mbedtls_mpi_montmul` in with the [𝕊] section. This last one better represents the section behavior with a line only loaded the first time the section is used.

With these *KEP*s, we can detect:

- When we enter the multiply phase and leave the multiply phase.
- When we are in a squaring phase.

These improved *KEP*s produce the report on figure 5.13. It proposes the following *nhit_ inclusive*, *KDS*es:

- for [𝕄]: $(0\text{x}38, 1)$ with score 1.0.
- for [𝕊]: $(0\text{x}346, 3)$ with score 1.0.

Both have a 1.0 score, which guarantees that they can be detected and distinguished. However, with these points, we can only detect the end of windows. If we are out of a window section, we must be in a squaring section of the exponentiation function. We can still try to reconstruct the [𝕊] information by using the fact that the multiply window contains seven $(wsize + 1)$ `mbedtls_mpi_montmul` which all take the same time as they are done with the same modulo (see figure 4.14(b)). We call this *window measurement*, presented on figure 5.14. We compare the time difference between two of our [𝕄]-segments. These [𝕄]-segments are detected using the [𝕄]-related cache line helped by the [𝕊]-relate cache line. We know that among these time differences, there is at least one that only contains the window section and no extra squaring phase ([𝕊]). We know that because the first operation cannot be a [𝕊] and is necessarily a window.

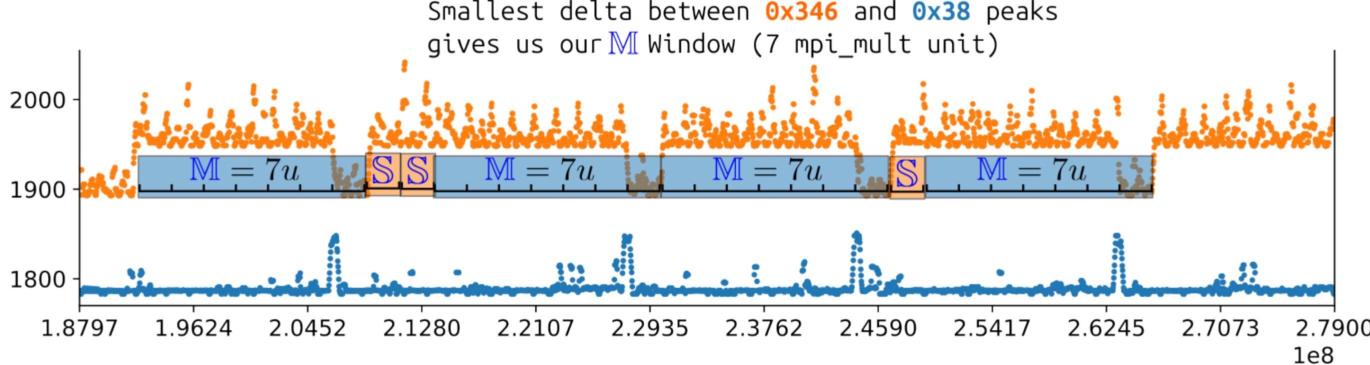


Figure 5.14: We can measure the time between peaks in 0x38 or 0x346. With this measurement, by comparing them with the minimum difference between two of these peaks, we can design a system of units to reconstruct the series of [𝕊]and [𝕄].

5.5.2.2 *Attack Monitoring* and real hardware results

We ran the same attack scenario on the RockPi4 and on the simulated platform using the configuration produced by TEE-Time. This scenario is run multiple times on the real hardware to compare different runs. We observe that attack results from the real platform are similar to the simulation ones in certain instances (figure 5.15) and different in others(figure 5.16) inside a single traces. When we compared different traces acquired from the real platform, we saw that different parts of the traces were identical to the simulation, while the others were just noise. These differences are due to the *gem5* simulation using perfect LRU while the real platform uses a pseudo-LRU (as mentioned in section 5.4.3). We confirmed this assumption by configuring our simulation to use *Tree-LRU* with *AutoLock* (middle trace on figure 5.16). The transient "stuck" timings appear in both *Tree-LRU* simulation traces (middle trace in figure 5.16) and real platform traces (bottom trace in figure 5.16). However, in detail, simulation *Tree-LRU* and real platform pseudo-LRU still behave differently, either because of randomness or small differences between model and reality.

	Run times(s)
Simulation on *gem5*	
Boot (only needed once)	2360.58
VictimScan	10603.99
Attack Monitoring	10006.69
Real platform	
Attack	1.080240
Export to SD card	5.394986

Table 5.2: Execution time for Rockchip platform comparing the simulated *gem5* model and the real platform.

We can remedy this issue by cumulating multiple real traces to reproduce the information contained in simulation traces. Using multiple traces is possible as OP-TEE uses *libtomcrypt*[lib23] with *libmbedTLS* only as *bignum* operation provider. *libtomcrypt* only implements base blinding. This attack would be more complex if OP-TEE used *mbedTLS* RSA implementation, which implements exponent blinding [UH23; KOU+23]. However, if we choose to do that, the short burst caused by set 0x38, which is used to detect [𝕄]will likely be lost by the averaging. For this reason, we solely rely on the 0x346 set to recover the window start points that correspond to our [𝕄] *KEPs*. This is possible because the [𝕊] *KDS* was chosen to distinguish [𝕊] from [𝕄] and in thus not present around [𝕄]. We call these points associated with the [𝕄] code segment: [𝕄]-points. With them, we can perform the window measurement on the real traces.

5.5.3 Extracting a key from real traces

After accumulating 50 traces, we fused them and then filtered them using a gaussian filter. To recover our [𝕄]-points from these traces, we used a peak detection algorithm, each [𝕄]-points corresponding to a peak in the 0x346 filtered

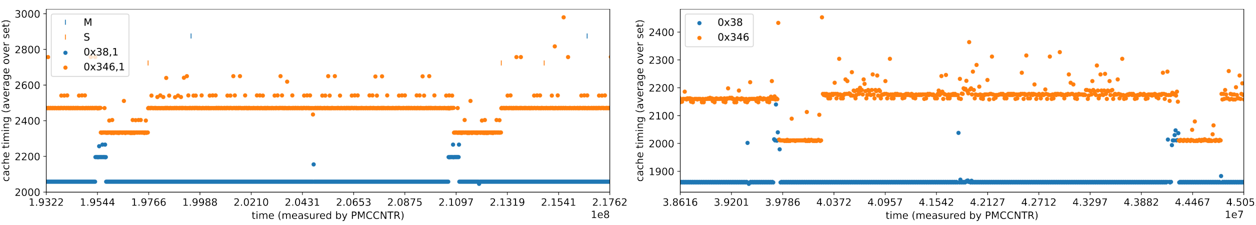


Figure 5.15: Comparison between simulation (left) and real hardware (right): centered around a similar pattern.

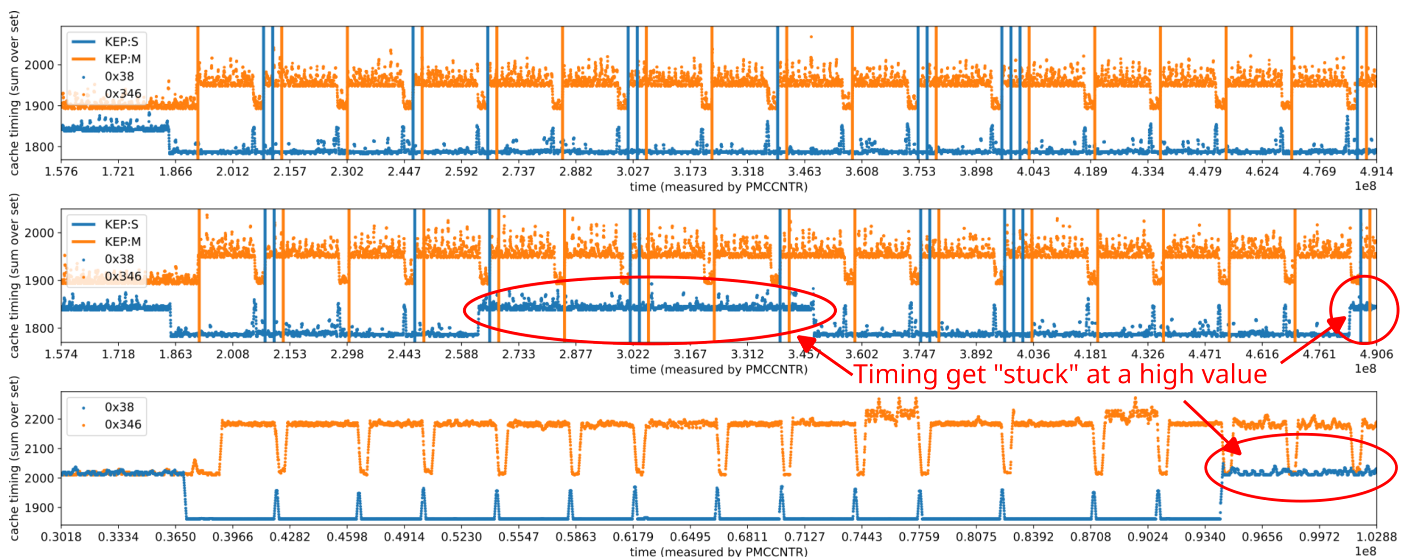


Figure 5.16: We compared traces between (from top to bottom): *gem5* simulation using LRU, *gem5* simulation using *Tree-LRU* and the real platform. We can see that there is the same behavior during which a prime set gets "stuck" in an occupied state between simulated *Tree-LRU* and the real platform.

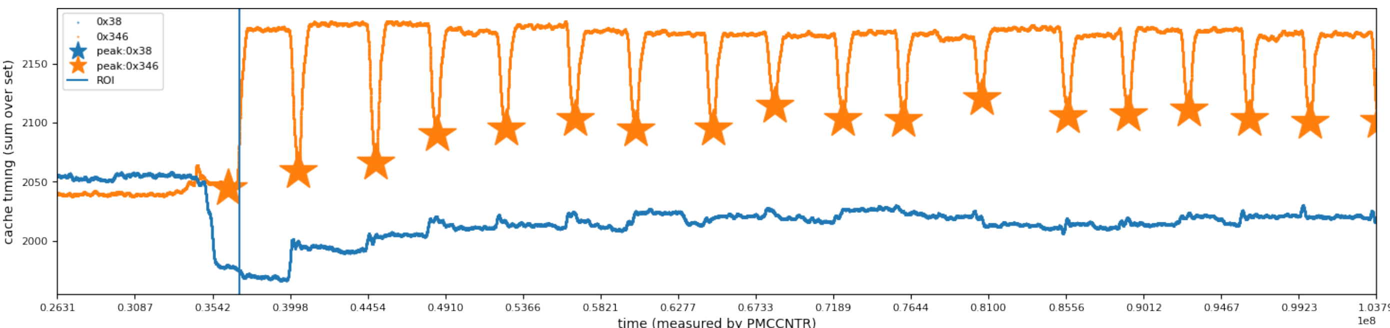


Figure 5.17: Accumulation of 50 real traces. These traces took 4 minutes to complete, with most of the time spent exporting data to the SD card. They have been fused and filtered with a gaussian filter. We used a peak detection algorithm to find the peaks associated with [$\mathbb{M}$], marked with stars.

trace. We automatically tune this algorithm knowing roughly the number of peaks in a trace: For a 1024 bits-private key, as our `mbedtls_mpi_exp_mod` function uses a 6 bit window, they cannot be more than $1024/6 \approxeq 171$ peaks. We facilitate this process using the `0x38` to determine a region of interest when it has a lower value. Using this algorithm on the traces on figure 5.17, we added the black line marking the region of interest and stars marking the peaks associated with [$\mathbb{M}$]-points. Each star represents a peak that will be used to perform the *window measurement*. From the peaks in this figure, we can retrieve the [$\mathbb{M}$]-points. With the [$\mathbb{M}$]-Points retrieved, we can now carry on *window measurement*:

- We determine the time length of a single window using the smallest difference between [$\mathbb{M}$]-points.
- With this single window, we find the length of a single Montgomery multiplication (`mbedtls_mpi_montmul`).
- We then try filling each time difference between [$\mathbb{M}$]-points with one window and then as much multiplication([$\mathbb{S}$]) as needed.
- We also treat specifically the difference between the last [$\mathbb{M}$]-point and the end of the region-of-interest to find the trailing multiplication.
- For each multiplication, we count a "[$\mathbb{S}$]"(possibly none) that will then be followed by a [$\mathbb{M}$].
- This makes our [$\mathbb{S}$][$\mathbb{M}$]-series, which we showed, can be used to recover a partial key (see figure 4.15). [$\mathbb{M}$] indicates 1 followed by 5 Xs (window_size -1), [$\mathbb{S}$] each indicates single 0 and trailing [$\mathbb{S}$]s (at the end of the exponent) each indicates single a X (either a 1 or a 0)

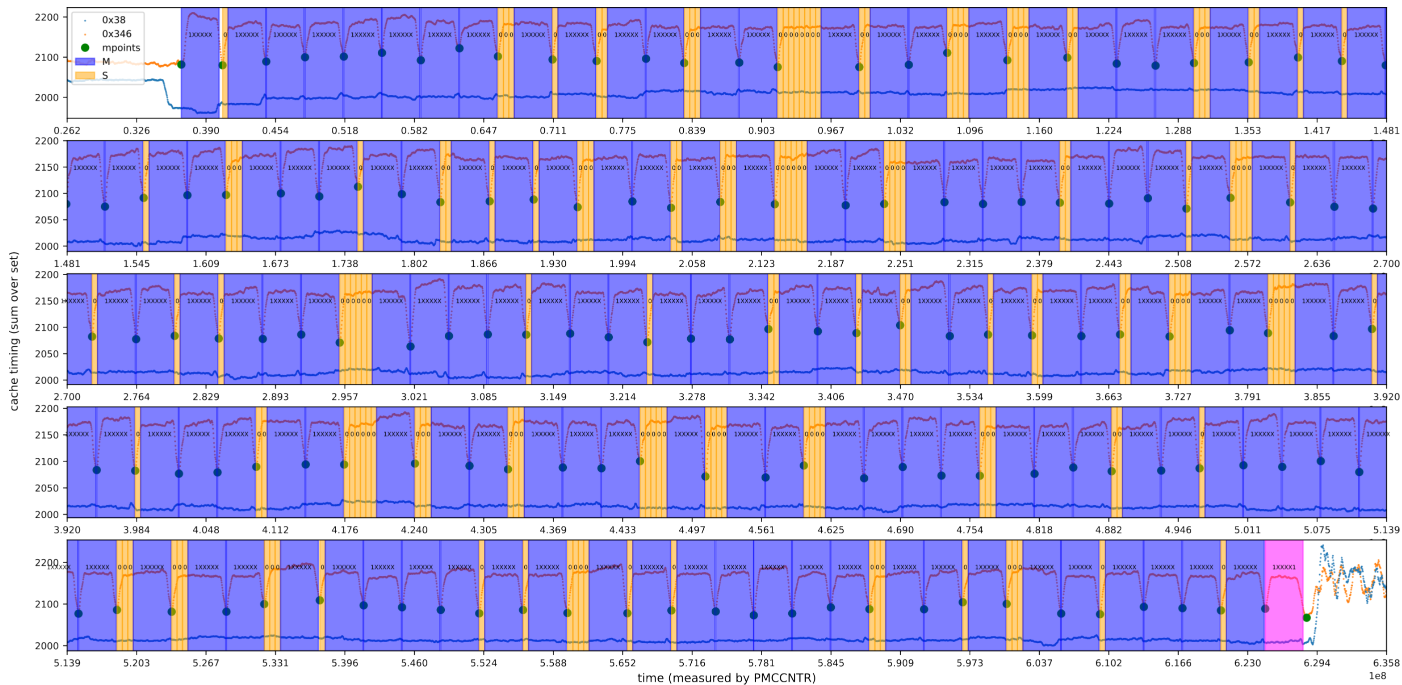


Figure 5.18: Attack on the real platform against the *sec-sign* TA: 1 and 0 are bits from the private key that we identified using the [𝕊][𝕄]-series; the X corresponds to bits that we do not know, and that can be either a 1 or a 0.

As the *sliding window exponentiation* skipped zeros at the start of the exponent, we know that all missing bits are leading zeros. We also know that D, the private exponent, cannot be even. Therefore, the last bit is necessary a 1 ($ED = 1 \ mod(p-1)(q-1)$ implies that D is necessary odd because E is odd.). We can overlay all this process on the traces, which gives us the full figure 5.18. In this figure, we can see the partially reconstructed key overlaid above its trace. Each blue rectangle corresponds to a [𝕄]-section, and each orange rectangle corresponds to a single [𝕊]-sections. The corresponding key bits are written on each of the rectangles. We performed the same operation for other keys and compared how many bits we recovered using our *window measurement* method (see table 5.3).

With these partial RSA keys, [UH23; MH20] and [KOU+23] suggest that we can use *Branch and Prune algorithm* to recover the rest of the keys. With this last phase, we would be able to extract a full RSA key from a state-of-the-art TEE implementation of a RSA signing application.

5.6 Conclusion

To build a new platform model from scratch, we had to create an efficient way of implementing devices without having to recompile *gem5* and with a simpler development approach. Having already leveraged config files' Python interface to implement key features of the *GDB* stub used in TEE-Time, we proposed PyDevices to leverage the same environment for peripheral devices. With this efficient approach, we explored the *RK3399* `BootROM` with *Ghidra*, implementing in *PyDevices* what was needed to progress. Going through the bootloaders until we reached the Linux user shell (figure A.8), we implemented the necessary devices, including power management and CPU wakeup mechanisms. With this simulation model, which can boot an unmodified SD card image, we developed a toolchain called *Rockchip-platform*, able to run the same disk image and programs on our *gem5* simulation platform and on our RockPi4 board. With this platform, we ran cache timing attack demos to tune our *gem5 RK3399* model by comparing its results with real hardware. Through

keys	[𝕊]	[𝕄]	total bits in [𝕊][𝕄]	known bits
key 1	145	146	1021	294
key 2	164	143	1022	310
key 3	134	148	1022	284

Table 5.3: Example of D reconstruction using [𝕊][𝕄]-series.

this testing, we confirmed that the Cortex A72 in the *RK3399* uses *AutoLock* with pseudo-LRU as its cache replacement policy. Taking this cache model into account, we propose a TEE-Time configuration able to deploy a *Prime+Probe* attack on the Cortex A72 in spite of *AutoLock*, with a *Prime+Probe* reverse attack. Using the *Rockchip-platform*, we ran TEE-Time against *sec-sign*, our secure signing scenario mentioned in section 3.4.5 and section 4.6. Using the *KEP* configuration used in section 4.6, TEE-Time proposed a *Prime+Probe* configuration able to retrieve a partial RSA key. Building upon this information presented by *Attack Monitor*, we proposed a method to extract from attack traces some bits of the RSA keys. Leveraging the *Rockchip-platform*, we ran the same attack configuration in the very same attack scenario on our RockPi4. By comparing attack traces between simulated and real hardware, we assessed the validity of our model with respect to this attack. With this consideration, we fused multiple real hardware traces to extract the partial key using the method mentioned earlier. This finally verifies TEE-Time methodologies and models against real hardware. But beyond this simple *RK3399* virtual platform, this use case demonstrates a methodology that leverages our *PyDevices* to retro-engineer secure platforms. In combination with TEE-Time, we could study the micro-architectural security of TEEs running on a secure platform without having access to a debugger.

Chapter 6

Conclusion and Perspectives

Contents

6.1 Introduction **101**
6.2 Overview of contribution **101**
6.2.1 Contributing to *gem5*: Trusted Execution Environment and *GDB* 101
6.2.2 Virtual Security Platform 101
6.2.3 TEE-Time tools 101
6.2.4 PyDevices: building the *Rockchip-platform* 102
6.2.5 Attacks against hash signing RSA scenarios with TEE 102
6.3 Future works **103**
6.4 Concluding remarks **103**

6.1 Introduction

Security operations performed in SoC are scrutinized by both attackers and system designers. These operations are protected by system and ISA primitives that enforce the isolation of these critical tasks from potentially malicious code running on the same hardware. While some weaknesses can be found directly in the software in vitro: user-after-free[Lee+15], buffer-overflow[One96], etc, weaknesses at the micro-architectural level that breach this isolation can only be found in vivo while running on the real system. Such weaknesses, like cache timing attacks or transient execution attacks, have been discovered using extensive knowledge of the SoC architecture and deep access to the hardware. Leveraging such weakness against a real application requires tight control over the execution environment and the hardware underlying it to ensure the stability of micro-architectural weaknesses effects. However, since the tool to study such vulnerabilities could be used to attack directly a critical operation, they are often blocked from accessing the most secure environment of a SoC. Thereby, in Trusted Execution Environments (TEE), security through obscurity extends to micro-architectural aspects. And although vulnerabilities have been found in commercial TEE [Rya19], such attacks methodology become hard to reproduce as the knowledge and setup that was used to deploy and carry on, this type of attack might not be available to the large public.

Considering these needs, I designed the TEE aspect of *Archisec Project Virtual Platform*, such as it could allow to study a *security through obscurity* platform, allowing the study of how TEE workloads interact with specific architecture when their access is barred on the real platform in order to discover vulnerabilities at the microarchitecture level.

6.2 Overview of contribution

This platform, built between multiple partners, is based on *gem5* with OP-TEE on ARM as our TEE of choice. But to correctly, not only simulate a TEE environment but also deploy the necessary tools to analyze attacks and weaknesses and ensure that their findings can scale to a real platform, I needed to improve *gem5* ARM implementation and create tools that can study the ARM microarchitecture interaction with the TEE environment. This *gem5* improvement and tools are the key contributions of this thesis, as they are at the center of the methodology in the articles I published (figure 6.1).

6.2.1 Contributing to *gem5*: Trusted Execution Environment and *GDB*

As *gem5* ARM ISA implementation is modeled after a *Vexpress* platform which is supported by OP-TEE, it only needed to completely support the security extension, to run OP-TEE. But, as this was not a typical use case of *gem5*, this aspect was incomplete. Therefore, I fixed this implementation until OP-TEE was able to fully boot on *gem5*. While improving *gem5*, I also fixed several aspects of its *GDB* implementation in order to ease my exploration of the microarchitecture. These contributions have been submitted to the *gem5* community and are now integrated into the *gem5* open-source project.

6.2.2 Virtual Security Platform

As I extensively used *GDB* to explore and analyze TEE workload as they run on *gem5*, to understand issues in *gem5* implementation of ARM ISA, I saw the possibilities which could be leveraged directly from *GDB* to access *gem5* model states. By implementing a suitable `monitor` command in *GDB* and creating API to respond to it from *gem5*, I opened new perspectives to use *GDB* and *gem5* in integrating debugging sessions, which could be programmed in a more expressive and accessible Python environment.

With this interactive debugging session, I built a way to use our *gem5* simulator as a *virtual security platform*, which provides API to develop micro-architectural security tools. With this API, tools can make micro-architectural effects visible from a source code point of view thanks to *GDB* source code visualization.

6.2.3 TEE-Time tools

With the *GDB-gem5* API and *GDB-instrumentation*, I proposed a tool that could study OP-TEE workload and search for cache timing weaknesses that could be leverage by a cross-core *Prime+Probe* attacks. This tool called *VictimScan* analyzes the correlation between key points in victim source code, directly linked with algorithmic knowledge called *Key Execution Point (KEP)*, and cache state, in order to find cache line and indices, which could be used to attack the TEE application. To verify *VictimScan* results, I designed a *Prime+Probe* attack which I monitored with another *GDB-Python* script to compare cache timing results measured by the attacks and victim execution point. This script

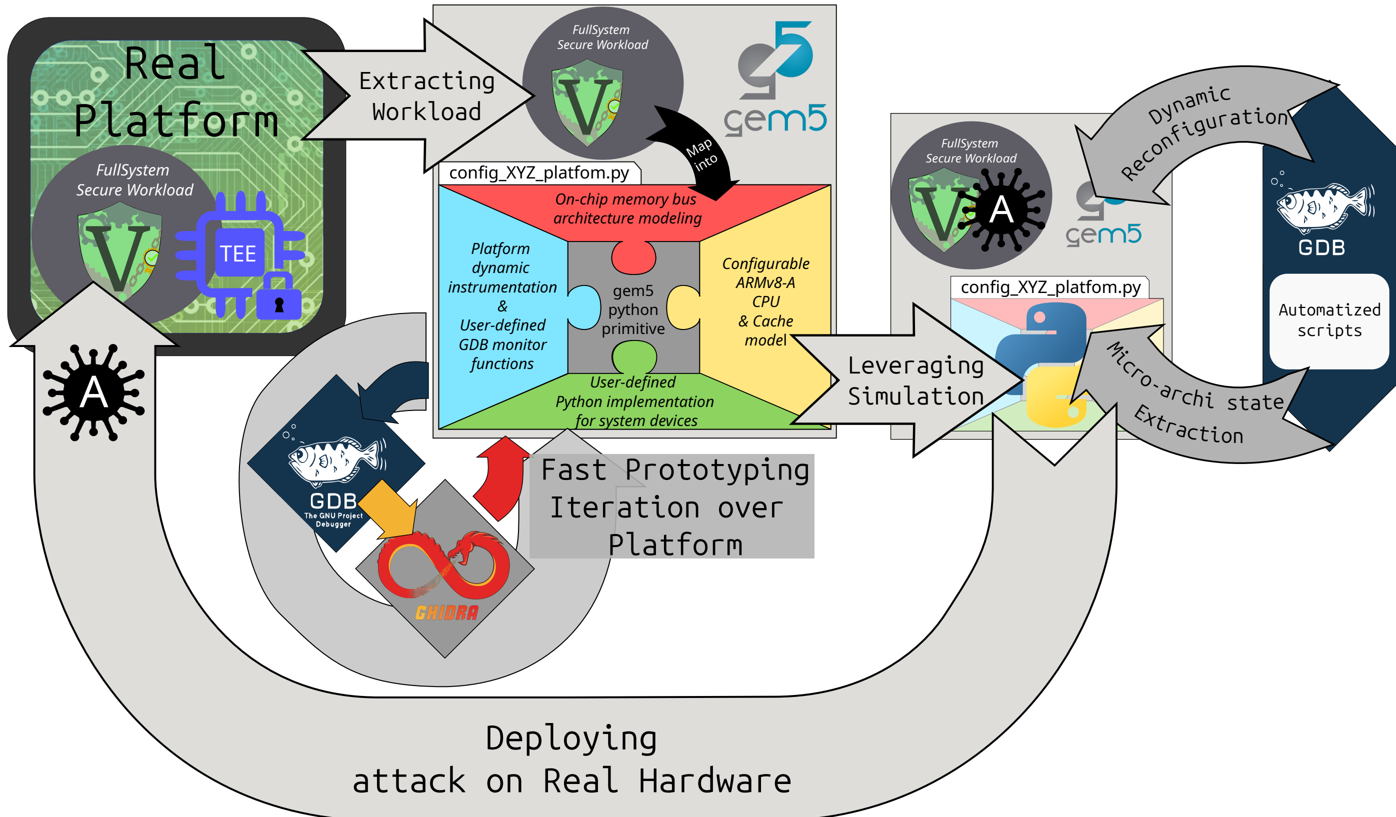


Figure 6.1: Overview of the methodology deployed in the thesis: Starting from a real platform, we extract a workload that uses a TEE to run it on *gem5*. Using *Ghidra* and *GDB*, we are able to improve *the config files* in *gem5* to boot and execute the workload. Now that we can simulate the workload, we use *automatized script* in *GDB* to study it and find vulnerabilities. We can leverage these vulnerabilities in attacks, which we evaluate in simulation. Finally, these attacks can be deployed on the real platform to conventionally verify the vulnerabilities.

verifies the correlation between KEP and attack results, confirming this attack could be used to detect them. These two tools form together *TEE-Time* which is presented as *A Dynamic Cache Timing Analysis Tool for Trusted Execution Environment*[FDC24b].

6.2.4 PyDevices: building the *Rockchip-platform*

To verify *TEE-Time* results, beyond the *Vexpress* ARM virtual platform, we had to model inside *gem5* a real OP-TEE-supported SoC. We chose the RockPi4, a development board from RADXA, and its *RK3399* from Rockchip. With an available TRM manual, we used *gem5* tools and methodologies to reproduce its cache and CPU topology inside *gem5*. To ease the integration of *RK3399* mandatory devices and accelerate their development cycle, I implemented a new way to write system devices in *gem5* that I called *PyDevices*. With PyDevices, system devices can be entirely written in Python, removing the need for recompiling *gem5* to add devices. These PyDevices implementations can also leverage the *GDB* API to communicate debug information to *GDB* or *Ghidra* in order to retro-engineer bootrom. With PyDevices, I develop a RockPi4 model in *gem5* capable of running the same SD card disk image as the real platform. With the *Rockchip-platform*, I can target both real and simulation platforms simultaneously.

6.2.5 Attacks against hash signing RSA scenarios with TEE

As OP-TEE RSA services use *mbedTLS bignum exponentiation* they are known to be attackable with *Prime+Probe*. However, such attacks can be complex to deploy as it requires knowing the exact set to attack which may depend between implementation and platform. This is exactly a TEE-Time use case: I used TEE-Time to analyses the security of an RSA hash-signing trusted application using OP-TEE crypto services. With the correct *KEP* in the sliding window exponentiation function from *mbedTLS*, TEE-Time proposed cache lines that can be used to perform an attack. With a

similar *KEP* configuration, we run a TEE-Time analysis in both our *Vexpress* platform and our Rockchip platform, which resulted in a validated correlation between attack timing traces and *KEPs*. As our Rockchip implementation includes an *AutoLock* model to imitate the cache replacement policy present on the real *RK3399*, it found different results. But as the Rockchip implementation was still attackable, we tried to use the same attack on our real platform. While a single run contained too much noise compared to the simulation to extract information. By combining 50 attack traces, we were able to extract all the information contained in the *KEPs*. Thereby, we retrieved a partial RSA private key with $\approx 30\%$ of the bits.

6.3 Future works

This methodology and its associated platform can be improved and extended in multiple ways:

- **Attacking RSA with exponent blinding**: As I mentioned in section 5.5, *mbedTLS* in *libtomcrypt* does not use exponent-blinding. This can be considered an obvious weakness that could be easily fixed by directly using *mbedTLS* without *libtomcrypt*. Although [KOU+23] demonstrated that such attacks are only possible if performed using a single trace, [UH23] proposed a method that could use multiple traces. In both situations, I could improve either the attack setup or post-processing to try to reproduce an attack against OP-TEE using a stronger cryptographic implementation on the *RK3399*.
- **Bootrom extraction and PyDevice**: Our Rockchip platform and attacks used a mostly open source workload with only the bootrom being closed-source. But the methodology I used to build the RK3399 model in *gem5*, such as it could execute a closed source bootrom. It could also be reproduced for a bigger bootrom, which would contain the TEE binary. As long as we can find a TRM manual that describes most of the memory map, we could use *Ghidra* to retro-engineer any bootrom and build a platform model in *gem5* using *PyDevices*. Typically, the *Nvidia Tegra X1* used in the Nintendo Switch has its Technical Reference Manual (TRM) widely available. Then, we would have to use *Ghidra* to find a cryptographic function to attack we would then be able to use TEE-Time to analyze this function and potentially build an attack that we would be able to run on the real platform for comparison. Thus, we would have an even more realistic approach to embedded system security.
- **RowHammer dynamic analysis**: As [Fra+22] added a *RowHammer*-supporting DRAM model to our virtual security platform, we could use said model to simulate *RowHammer* attacks. We could then deploy the same mechanism used in TEE-Time to detect possibilities of attack against cryptographic systems by tracking memory addresses of Keys and secrets through cryptographic algorithm execution and linking them with user-accessible DRAM rows which could be leveraged to perform *RowHammer*. This would allow automatizing *RowHammer* weaknesses research.
- **Transient execution attack tools similar to TEE-Time**: Transient execution attack relies on specific gadgets that have a peculiar speculative transient behavior. Researching such gadgets in binaries could be complex as they often depend on execution paths before reaching the gadget. In this context, *gem5* could be modified to track transient CPU pipeline states to detect promiscuous states and report them to *GDB* which could then organize and compile the results. We could then configure an attack with these results and, a la *Attack Monitoring*, control that transient execution happens as expected from *GDB*.
- **Software and Hardware countermeasure**: Now that TEE-Time has been verified with real hardware, we could use the report results as a metric to assess and compare different countermeasures both hardware, implementing them inside *gem5*, and software, using *gem5* and the real hardware as a demonstration platform.

6.4 Concluding remarks

In this thesis, we presented:

- A *Virtual security platform* that could be used to study in-vivo micro-architectural security against the state-of-the-art implementation of protected application: Trusted Executions Environment.
- *TEE-Time*, A methodology implemented in tool sets that connect to our *Virtual security platform* and automatically deduce and verify *Prime+Probe* weaknesses.

- A proof of concept, built upon new fast-prototyping tools, that demonstrates scalability of our virtual security platform findings by successfully predicting a cache timing attack against a TEE-protected RSA application running on a RockPi4.

Therefore, we demonstrated that, in spite of drawbacks when compared to a real platform, in terms of model accuracy and execution speed, a virtual platform built on *gem5* could be leveraged to study micro-architectural security. And considering limitations that are brought by Trusted Execution Environments to prevent debugging tools from accessing secure enclaves. Simulation platforms may, in that regard, be the most adequate for studying TEE security. We could hope that similar endeavors will shake the foundations of one of the last bastions of *security through obscurity*, ushering in a new era for more open secure enclaves and Trusted Execution Environments.

Bibliography

My publications

[For+21] Quentin Forcioli et al. "Virtual Platform to Analyze the Security of a System on Chip at Microarchitectural Level". In: *EuroS&PW 2021 - IEEE European Symposium on Security and Privacy Workshops*. Vienne, Austria, Sept. 2021, pp. 96–102. DOI: 10.1109/EuroSPW54576.2021.00017. URL: https://hal.archives-ouvertes.fr/hal-03353878.

[FDC23] Quentin Forcioli, Jean-Luc Danger, and Sumanta Chaudhuri. "A gem5 based Platform for Micro-Architectural Security Analysis". In: *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '23. Toronto, Canada: Association for Computing Machinery, 2023, pp. 91–99. ISBN: 9798400716232. DOI: 10.1145/3623652.3623674. URL: https://doi.org/10.1145/3623652.3623674.

[FDC24a] Quentin Forcioli, Jean-Luc Danger, and Sumanta Chaudhuri. "Defeating AutoLock: From Simulation to Real-World Cache-Timing Exploits against TrustZone." In: *In submission* (July 2024).

[FDC24b] Quentin Forcioli, Jean-Luc Danger, and Sumanta Chaudhuri. "TEE-Time: A Dynamic Cache Timing Analysis Tool for Trusted Execution Environments". In: *ISQED 2024: The 25th International Symposium on Quality Electronic Design* (Apr. 2024).

[Forne] Q. Forcioli. "ISCA 2022: GEM5 USERS' WORKSHOP". In: June 2022 [Online]. URL: https://www.gem5.org/events/isca-20225.

Other publications

[JED58] JEDEC. *Joint Electron Device Engineering Council*. 1958. URL: https://www.jedec.org/.

[One96] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996). URL: http://www.phrack.com/issues.html?issue=49&id=14.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. "On the importance of checking cryptographic protocols for faults". In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1997, pp. 37–51.

[Smi97] M.J.S. Smith. *Application-specific Integrated Circuits*. Addison-Wesley VLSI systems series. Addison-Wesley, 1997. ISBN: 9780201500226. URL: https://books.google.fr/books?id=3hxTAAAAMAAJ.

[Cow+98] Crispan Cowan et al. "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[MOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. URL: http://www.cacr.math.uwaterloo.ca/hac/.

[Swa01] Stuart Swan. "An introduction to system level modeling in SystemC 2.0". In: *Cadence Design Systems, Inc., draft report* (2001).

[Bel05] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46.

[Mar+05] Milo MK Martin et al. "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset". In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[Per05] Colin Percival. "Cache missing for fun and profit". In: *Proc. of BSDCan 2005*. 2005.

[Bin+06] Nathan L Binkert et al. "The M5 simulator: Modeling networked systems". In: *Ieee micro* 26.4 (2006), pp. 52–60.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on

Topics in Cryptology. CT-RSA'06. San Jose, CA: Springer-Verlag, 2006, pp. 1–20. ISBN: 3540310339. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805_1.

[Son+06] Xinyue Song et al. "A Qualitative Analysis of Privilege Escalation". In: *2006 IEEE International Conference on Information Reuse & Integration*. 2006, pp. 363–368. DOI: 10.1109/IRI.2006.252441.

[Hal+08] J. Alex Halderman et al. "Lest We Remember: Cold Boot Attacks on Encryption Keys". In: *17th USENIX Security Symposium (USENIX Security 08)*. San Jose, CA: USENIX Association, July 2008.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient cache attacks on AES, and countermeasures". In: *Journal of Cryptology* 23 (2010), pp. 37–71.

[Vah10] F. Vahid. *Digital Design with RTL Design, VHDL, and Verilog*. Wiley, 2010. ISBN: 9780470531082. URL: https://books.google.fr/books?id=-YayRpmjc20C.

[Dav+11] Lucas Davi et al. "Privilege escalation attacks on android". In: *Information Security: 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers 13*. Springer. 2011, pp. 346–360.

[DPM11] Loc Duflot, Yves-Alexis Perez, and Benjamin Morin. "What if You Can'T Trust Your Network Card?" In: *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*. RAID'11. Menlo Park, CA: Springer-Verlag, 2011, pp. 378–397. ISBN: 978-3-642-23643-3. DOI: 10.1007/978-3-642-23644-0_20. URL: http://dx.doi.org/10.1007/978-3-642-23644-0_20.

[ASS12] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. *JESD220A: Universal Flash Storage*. hhttps://www.jedec.org/sites/default/files/docs/JESD220A.pdf. 2012.

[Nah12] Hadi Nahari. *TLK: A FOSS Stack for Secure Hardware Tokens*. NVIDIA, 2012.

[Lad+13] Evangelos Ladakis et al. "You can type, but you cant hide: A stealthy GPU-based keylogger". In: *Proceedings of the 6th European Workshop on System Security (EuroSec)*. Citeseer. 2013.

[Mor+13] Nicolas Moro et al. "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller". In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 77–88. DOI: 10.1109/FDTC.2013.9.

[Sny13] Wilson Snyder. "Verilator: Open simulation-growing up". In: *DVClub Bristol* (2013).

[SB13] Patrick Stewin and Iurii Bystrov. "Understanding DMA Malware". In: *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA'12. Heraklion, Crete, Greece: Springer-Verlag, 2013, pp. 21–41. ISBN: 978-3-642-37299-5. DOI: 10.1007/978-3-642-37300-8_2. URL: http://dx.doi.org/10.1007/978-3-642-37300-8_2.

[Kim+14] Yoongu Kim et al. "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964. DOI: 10.1145/2678373.2665726. URL: https://doi.org/10.1145/2678373.2665726.

[MG14] Bojan Mihajlovi, eljko ili, and Warren J. Gross. "Dynamically Instrumenting the QEMU Emulator for Linux Process Trace Generation with the GDB Debugger". In: *ACM Trans. Embed. Comput. Syst.* 13.5s (Dec. 2014). ISSN: 1539-9087. DOI: 10.1145/2678022. URL: https://doi.org/10.1145/2678022.

[Ran+14] Mohammed Rangwala et al. "A taxonomy of privilege escalation attacks in Android applications". In: *International Journal of Security and Networks* 9.1 (2014), pp. 40–55.

[YF14] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: http://dl.acm.org/citation.cfm?id=2671225.2671271.

[Doy+15] Goran Doychev et al. "CacheAudit: A Tool for the Static Analysis of Cache Side Channels". In: *ACM Trans. Inf. Syst. Secur.* 18.1 (June 2015). ISSN: 1094-9224. DOI: 10.1145/2756550. URL: https://doi.org/10.1145/2756550.

[DUK15] CURTIS W. DUKES. *Committee on National Security Systems (CNSS) Glossary*. 2015. URL: https://rmf.org/wp-content/uploads/2017/10/CNSSI-4009.pdf.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[Lee+15] Byoungyoung Lee et al. “Preventing Use-after-free with Dangling Pointers Nullification.” In: *NDSS*. 2015.

[Les15] Stéphane Lescuyer. “ProvenCore: Towards a Verified Isolation Micro-Kernel.” In: *MILS@ HiPEAC*. 2015.

[Liu+15] F. Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[Ore+15] Yossef Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1406–1418. ISBN: 9781450338325. DOI: 10.1145/2810103.2813708. URL: https://doi.org/10.1145/2810103.2813708.

[PMB15] Dorottya Papp, Zhendong Ma, and Levente Buttyan. “Embedded systems security: Threats, vulnerabilities, and attack taxonomy”. In: *2015 13th Annual Conference on Privacy, Security and Trust (PST)*. 2015, pp. 145–152. DOI: 10.1109/PST.2015.7232966.

[Qua15] Qualcomm. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf. 2015.

[SAB15] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. “Trusted execution environment: what it is, and what it is not”. In: *2015 IEEE Trustcom/BigDataSE/Ispa*. Vol. 1. IEEE. 2015, pp. 57–64.

[Sam15] Samsung. https://docs.samsungknox.com/admin/whitepaper/kpe/samsung-knox.htm. 2015.

[SD15] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: *Black Hat* 15 (2015), p. 71.

[VPI15] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. “GPU-assisted Malware”. In: *Int. J. Inf. Secur.* 14.3 (June 2015), pp. 289–297. ISSN: 1615-5262. DOI: 10.1007/s10207-014-0262-9. URL: http://dx.doi.org/10.1007/s10207-014-0262-9.

[Alm+16] José Bacelar Almeida et al. “Verifying {Constant-Time} Implementations”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 53–70.

[And16] Android. https://source.android.com/docs/security/features/trusty/. 2016.

[Bos+16] Joppe W Bos et al. “Differential computation analysis: Hiding your white-box designs is not enough”. In: *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings 18*. Springer. 2016, pp. 215–236.

[CD16] Victor Costan and Srinivas Devadas. “Intel SGX explained”. In: *Cryptology ePrint Archive* (2016).

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A remote software-induced fault attack in javascript”. In: *DIMVA*. 2016.

[Gru+16a] Daniel Gruss et al. “Flush+ Flush: a fast and stealthy cache attack”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer. 2016, pp. 279–299.

[Gru+16b] Daniel Gruss et al. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *CCS*. 2016.

[Lip+16] Moritz Lipp et al. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 549–564. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[Nga+16] Bernard Ngabonziza et al. “Trustzone explained: Architectural features and use cases”. In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2016, pp. 445–451.

[OFl16] Colin O’Flynn. “Fault injection using crowbars on embedded systems”. In: *Cryptology ePrint Archive* (2016).

[Rei+16] Cezar Reinbrecht et al. “Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack”. In: *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*. 2016, pp. 1–6. DOI: 10.1109/SBCCI.2016.7724051.

[RQA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. “Sparse representation of implicit flows with applications to side-channel detection”. In: *Proceedings of the 25th International Conference on Compiler Construction*. 2016, pp. 110–120.

[Van+16] Victor Van Der Veen et al. "Drammer: Deterministic rowhammer attacks on mobile platforms". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 1675–1689.

[ZXZ16] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. "Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 858–870. ISBN: 9781450341394. DOI: 10.1145/2976749.2978360. URL: https://doi.org/10.1145/2976749.2978360.

[Ala+17] Murugappan Alagappan et al. "DFS covert channels on multi-core platforms". In: *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2017, pp. 1–6. DOI: 10.1109/VLSI-SoC.2017.8203469.

[Anc+17] Stéphanie Anceau et al. "Nanofocused X-ray beam to reprogram secure circuits". In: *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Springer. 2017, pp. 175–188.

[ARM17a] ARM. *Cortex A9 MPCore Accelerator Coherency Port*. Accessed: 2017-04-12. 2017. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407e/CACGGBCF.html.

[ARM17b] ARM. *SystemC Cycle Models User Guide Version 10.0*. 2017. URL: https://developer.arm.com/documentation/101124/1000.

[Ben17] Gal Beniamini. "Trust issues: Exploiting trustzone tees". In: *Google Project Zero Blog* (2017).

[Ber+17] Daniel J Bernstein et al. "Sliding right into disaster: Left-to-right sliding windows leak". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 555–576.

[Car17] Pierre Carru. "Attack ARM TrustZone using Rowhammer". In: *GreHack, 2017*. 2017.

[Cha17] Sumanta Chaudhuri. "Cache Timing Attacks from The SoCFPGA Coherency Port (Abstract Only)". In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2017.

[Che+17] Yue Chen et al. *Downgrade Attack on TrustZone*. 2017. arXiv: 1707.05082 [cs.CR].

[Gre+17] Marc Green et al. "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1075–1091. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/green.

[Ira+17] Gorka Irazoqui et al. *Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries*. 2017. DOI: 10.48550/ARXIV.1709.01552. URL: https://arxiv.org/abs/1709.01552.

[Jac+17] Nisha Jacob et al. "Compromising FPGA SoCs using malicious hardware blocks". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, pp. 1122–1127. DOI: 10.23919/DATE.2017.7927157.

[JRM17] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. "pybind11–Seamless operability between C++ 11 and Python". In: *URL: https://github. com/pybind/pybind11* (2017).

[Mau+17] Clémentine Maurice et al. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. Vol. 17. 2017, pp. 8–11.

[Men+17] Christian Menard et al. "System simulation with gem5 and SystemC: The keystone for full interoperability". In: *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2017, pp. 62–69. DOI: 10.1109/SAMOS.2017.8344612.

[Moh+17] Alian Mohammad et al. "dist-gem5: Distributed simulation of computer clusters". In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 153–162. DOI: 10.1109/ISPASS.2017.7975287.

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "Dude, is my code constant time?" In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1697–1702.

[Wan+17] Shuai Wang et al. "{CacheD}: Identifying {Cache-Based} timing channels in production software". In: *26th USENIX security symposium (USENIX security 17)*. 2017, pp. 235–252.

[Dob+18] Christoph Dobraunig et al. "SIFA: exploiting ineffective fault inductions on symmetric cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 547–572.

[LW18] Ben Lapid and Avishai Wool. "Navigating the samsung trustzone and cache-attacks on the keymaster trustlet". In: *European Symposium on Research in Computer Security*. Springer. 2018, pp. 175–196.

[LJ18] Bo Li and Bo Jiang. "Cache Attack on AES for Android Smartphone". In: *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*. ICCSP 2018. Guiyang, China: Association for Computing Machinery, 2018, pp. 138–143. ISBN: 9781450363617. DOI: 10.1145/3199478.3199488. URL: https://doi.org/10.1145/3199478.3199488.

[Lip+18] Moritz Lipp et al. "Meltdown". In: *USENIX Security*. 2018.

[Low18] Jason Lowe-Power. *gem5 Architecture Support*. 2018. URL: https://www.gem5.org/documentation/general_docs/architecture_support/s.

[Wu+18] Meng Wu et al. "Eliminating timing side-channel leaks using program repair". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 15–26. DOI: 10.1145/3213846.3213851. URL: https://doi.org/10.1145/3213846.3213851.

[ZS18] Mark Zhao and G. Edward Suh. "FPGA-Based Remote Power Side-Channel Attacks". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 229–244. DOI: 10.1109/SP.2018.00049.

[And19] Android. *Android Verified Boot*. https://source.android.com/docs/security/features/verifiedboot/avb. 2019.

[BY19] Michael Garrett Bechtel and Heechul Yun. "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention". In: *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*. Ed. by Björn B. Brandenburg. IEEE, 2019, pp. 357–367. DOI: 10.1109/RTAS.2019.00037. URL: https://doi.org/10.1109/RTAS.2019.00037.

[BBA19] E. M. Benhani, L. Bossuet, and A. Aubert. "The Security of ARM TrustZone in a FPGA-based SoC". In: *IEEE Transactions on Computers* (2019), pp. 1–1. ISSN: 0018-9340. DOI: 10.1109/TC.2019.2900235.

[Bro+19] Robert Brotzman et al. "CaSym: Cache aware symbolic execution for side channel detection and mitigation". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 505–521.

[Can+19] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX Security Symposium*. extended classification tree at https://transient.fail/. 2019.

[FFY19] Jacob Fustos, Farzad Farshchi, and Heechul Yun. "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks". In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 61. DOI: 10.1145/3316781.3317914. URL: https://doi.org/10.1145/3316781.3317914.

[Koc+19] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[Min+19] Marina Minkin et al. *Fallout: Reading Kernel Writes From User Space*. 2019. arXiv: 1905.12701 [cs.CR].

[NZ19] Stefan Nicula and Razvan Daniel Zota. "Exploiting stack-based buffer overflow using modern day techniques". In: *Procedia Computer Science* 160 (2019). The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops, pp. 9–14. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.09.437. URL: https://www.sciencedirect.com/science/article/pii/S1877050919316527.

[Roh19] Roman Rohleder. "Hands-On Ghidra - A Tutorial about the Software Reverse Engineering Framework". In: *Proceedings of the 3rd ACM Workshop on Software Protection*. SPRO'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 77–78. ISBN: 9781450368353. DOI: 10.1145/3338503.3357725. URL: https://doi.org/10.1145/3338503.3357725.

[Rya19] Keegan Ryan. "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 181–194. ISBN: 9781450367479. DOI: 10.1145/3319535.3354197. URL: https://doi.org/10.1145/3319535.3354197.

[Sch+19a] Stephan van Schaik et al. "RIDL: Rogue In-Flight Data Load". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 88–105. DOI: 10.1109/SP.2019.00087.

[Sch+19b] Michael Schwarz et al. *ZombieLoad: Cross-Privilege-Boundary Data Sampling*. 2019. arXiv: 1905.05726 [cs.CR].

[sof19] I hate software. *Reverse-engineering Samsung Exynos 9820 bootloader and TZ*. http://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html. 2019.

[Wei+19] Zane Weissman et al. "JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms". In: *CoRR* abs/1912.11523 (2019). arXiv: 1912.11523. URL: http://arxiv.org/abs/1912.11523.

[Wer+19] Mario Werner et al. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 675–692. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/werner.

[Yan+19] Mengjia Yan et al. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy (Corrigendum)". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, p. 1076. DOI: 10.1145/3352460.3361129. URL: https://doi.org/10.1145/3352460.3361129.

[Yu+19] Xiaodong Yu et al. "Comparative Measurement of Cache Configurations' Impacts on Cache Timing Side-Channel Attacks". In: *12th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2019, Santa Clara, CA, USA, August 12, 2019*. Ed. by Rob Jansen and Peter A. H. Peterson. USENIX Association, 2019. URL: https://www.usenix.org/conference/cset19/presentation/yu.

[AJ20] Sam Ainsworth and Timothy M. Jones. "MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State". In: *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 132–144. DOI: 10.1109/ISCA45697.2020.00022. URL: https://doi.org/10.1109/ISCA45697.2020.00022.

[DBR20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1021–1038.

[Gra+20] Ben Gras et al. "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures." In: *NDSS*. 2020.

[KDG20] Mohammad Nasim Imtiaz Khan, Asmit De, and Swaroop Ghosh. "Cache-Out: Leaking Cache Memory Using Hardware Trojan". In: *IEEE Trans. Very Large Scale Integr. Syst.* 28.6 (2020), pp. 1461–1470. DOI: 10.1109/TVLSI.2020.2982188. URL: https://doi.org/10.1109/TVLSI.2020.2982188.

[Low+20] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR].

[MH20] Gabrielle De Micheli and Nadia Heninger. *Recovering cryptographic keys from partial information, by example*. Cryptology ePrint Archive, Paper 2020/1506. https://eprint.iacr.org/2020/1506. 2020. URL: https://eprint.iacr.org/2020/1506.

[Sam20] Samsung. *Samsung TEEGRIS*. https://developer.samsung.com/teegris/overview.html. Accessed: September 22, 2020. 2020.

[YL20] Heedong Yang and Manhee Lee. "Demystifying ARM TrustZone TEE Client API Using OP-TEE". In: *The 9th International Conference on Smart Media and Applications*. SMA 2020. Jeju, Republic of Korea: Association for Computing Machinery, 2020, pp. 325–328. ISBN: 9781450389259. DOI: 10.1145/3426020.3426113. URL: https://doi.org/10.1145/3426020.3426113.

[ARM21a] ARM. *AMBA AXI and ACE Protocol Specification*. 2021. URL: https://documentation-service.arm.com/static/602a9df190ee6824a1e02b98?token=.

[ARM21b] ARM. *Fast Models Fixed Virtual Platforms (FVP) Reference Guide*. 2021. URL: https://developer.arm.com/documentation/100966/latest/.

[AM21] Pierre Ayoub and Clémentine Maurice. "Reproducing spectre attack with gem5: How to do it right?" In: *Proceedings of the 14th European Workshop on Systems Security*. 2021, pp. 15–20.

[Fra+21a] Loc France et al. "Implementing rowhammer memory corruption in the gem5 simulator". In: *2021 IEEE International Workshop on Rapid System Prototyping (RSP)*. IEEE. 2021, pp. 36–42.

[Fra+21b] Loc France et al. "Vulnerability assessment of the rowhammer attack using machine learning and the gem5 simulator-work in progress". In: *Proceedings of the 2021 ACM workshop on secure and trustworthy cyber-physical systems*. 2021, pp. 104–109.

[Kou+21] Zili Kou et al. “Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 979–984. DOI: 10.1109/DAC18074.2021.9586226.

[Lav+21] Corentin Lavaud et al. “Whispering devices: A survey on how side-channels lead to compromised information”. In: *Journal of Hardware and Systems Security* 5 (2021), pp. 143–168.

[lea21] GlobalPlatform leadership. *Global Platform*. https://globalplatform.org/. 2021.

[Per+21] Thomas Perianin et al. “End-to-end automated cache-timing attack driven by machine learning”. In: *Journal of Cryptographic Engineering* 11.2 (2021), pp. 135–146.

[Ren+21] Xida Ren et al. “I See Dead ţops: Leaking Secrets via Intel/AMD Micro-Op Caches”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 361–374. DOI: 10.1109/ISCA52012.2021.00036.

[Roc21] Rockchip. *Rockchip RK3399 Technical Reference Manual*. 2021. URL: https://rockchip.fr/Rockchip%20RK3399%20TRM%20V1.3%20Part1.pdf,%20https://rockchip.fr/Rockchip%20RK3399%20TRM%20V1.3%20Part2.pdf.

[Shu+21] Anatoly Shusterman et al. “Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality”. In: *IEEE Transactions on Dependable and Secure Computing* 18.5 (2021), pp. 2042–2060. DOI: 10.1109/TDSC.2020.2988369.

[Syn21] Synopsys. *Platform Architect*. 2021. URL: https://www.synopsys.com/cgi-bin/proto/pdfdla/docsdl/platform_architect_ds.pdf?file=platform_architect_ds.pdf/.

[AIM22] Paolo Amato, Niccolò Izzo, and Carlo Meijer. “Mobile Systems Secure State Management”. In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. 2022, pp. 564–571. DOI: 10.1109/DSD57027.2022.00081.

[Bak+22] Anubhab Baksi et al. “A Survey on Fault Attacks on Symmetric Key Cryptosystems”. In: *ACM Comput. Surv.* 55.4 (Nov. 2022). ISSN: 0360-0300. DOI: 10.1145/3530054. URL: https://doi.org/10.1145/3530054.

[BY22] Michael Bechtel and Heechul Yun. “Denial-of-Service Attacks on Shared Resources in Intels Integrated CPU-GPU Platforms”. In: *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*. 2022, pp. 1–9. DOI: 10.1109/ISORC52572.2022.9812711.

[Fra+22] Loc France et al. “Modeling Rowhammer in the gem5 simulator”. In: *CHES 2022-Conference on Cryptographic Hardware and Embedded Systems*. 2022.

[Gro+22] Mathieu Gross et al. “Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC”. In: *J. Cryptogr. Eng.* 12.2 (2022), pp. 181–196. DOI: 10.1007/s13389-021-00273-8. URL: https://doi.org/10.1007/s13389-021-00273-8.

[Pi22] Raspberry Pi. *BCM2837*. 2022. URL: https://github.com/raspberrypi/documentation/blob/develop/documentation/asciidoc/computers/processors/bcm2837.adoc.

[BGL23] Lilian Bossuet, Vincent Grosso, and Carlos Andres Lara-Nino. “Emulating Side Channel Attacks on gem5: lessons learned”. In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2023, pp. 287–295. DOI: 10.1109/EuroSPW59978.2023.00036.

[BL23a] Lilian Bossuet and Carlos Andres Lara-Nino. “Advanced Covert-Channels in Modern SoCs”. In: *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2023, pp. 80–88. DOI: 10.1109/HOST55118.2023.10133626.

[BL23b] Lilian Bossuet and Carlos Andres Lara-Nino. “Emulating Covert Data Transmission on Heterogeneous SoCs”. In: *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 2023, pp. 1–6. DOI: 10.1109/AsianHOST59942.2023.10409377.

[Dua+23] Guoyun Duan et al. “TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation”. In: *Future Generation Computer Systems* 144 (2023), pp. 192–204.

[Eng23] DENX Software Engineering. *U-boot*. https://www.denx.de/wiki/U-Boot. Accessed: 2023-16-01. 2023.

[FBL23] Anis Fellah-Touta, Lilian Bossuet, and Carlos Andres Lara-Nino. “Combined Internal Attacks on SoC-FPGAs: Breaking AES with Remote Power Analysis and Frequency-based Covert Channels”. In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2023, pp. 281–286. DOI: 10.1109/EuroSPW59978.2023.00035.

[Gei+23] Antoine Geimer et al. “A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 1690–1704.

[KOU+23] Zili KOU et al. “Cache Side-channel Attacks and Defenses of the Sliding Window Algorithm in TEEs”. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137116.

[lib23] Team libtom. *libtomcrypt*. https://github.com/libtom/libtomcrypt. 2023.

[Lin23a] Linaro. *Linaro*. https://www.linaro.org/. Accessed: 2023-16-01. 2023.

[Lin23b] Linaro. *Trusted-Firmware A: Secure Payload Dispatcher (SPD)*. https://trustedfirmware-a.readthedocs.io/en/latest/components/spd/index.html. Accessed: 2023-16-01. 2023.

[Lin23c] Linaro. *TrustedFirmware A*. https://www.trustedfirmware.org/projects/tf-a/. Accessed: 2023-16-01. 2023.

[UH23] Rei Ueno and Naofumi Homma. “How Secure is Exponent-blinded RSA–CRT with Sliding Window Exponentiation?” In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2023), pp. 241–269.

[Fou24] SCons Foundation. 2024. URL: https://scons.org/.

[L B24] 0:1 L Break Into Program. *8250 UART*. 2024. URL: http://www.breakintoprogram.co.uk/hardware/components/8250-uarts.

[Low24a] Jason Lowe-Power. *gem5 Memory system*. 2024. URL: https://www.gem5.org/documentation/general_docs/memory_system/.

[Low24b] Jason Lowe-Power. *Learning gem5*. https://gem5.googlesource.com/public/gem5-website/+/refs/heads/stable/_pages/documentation/learning_gem5/. Accessed: 2024-05-03. 2024.

Appendix A

Appendix

Contents

A.1 About gem5 . . . . **114**
A.2 About PyDevices **116**
A.3 About ARM **116**
A.4 About RK3399 . . . . **118**
A.4.1 Retro-engineering 118
A.4.2 Covert-channel results 120

A.1 About *gem5*

my_sim_object.cc

```
Event* event= new EventBaseType(this);//creating an event
schedule(event, latency);//scheduling said event in *latency* time quantum
```

Figure A.1: *gem5* event scheduling in *CcObject* implementation.

Sconscript

```
Import('*')
#Adding a new SimObject Python file
SimObject('MySimObject.py', sim_objects=[
    'MySimObject',])#declare all the SimObject present in the files
#Declare a source C++ file
Source('my_sim_object.cc')
#declare a debug flag named: 'MySimObject'
DebugFlag('MySimObject')
```

Figure A.2: Integrating a new *SimObject* in *gem5* build system: *Scons*

new	instruction	purpose	handling
	`m5_arm(address)`	Increment the *arm* statistic in the associated Workload Object	handled by *gem5*
	`m5_quiesce()`	Quiesce the calling core/thread	handled by *gem5*
	`m5_quiesce_ns(ns)`	Quiesce the calling core/thread for *ns* ns	handled by *gem5*
	`m5_quiesce_cycle(cycles)`	Quiesce the calling core/thread for *cycle* cycles	handled by *gem5*
	`m5_quiesce_time()`	Return the a CPU has been quiesced	handled by *gem5*
	`m5_rpns()`	Return *gem5* internal tick	handled by *gem5*
	`m5_wake_cpu(cpuid)`	Wake up a potentially suspended or quiesced CPU identified by *cpuid*	handled by *gem5*
	`m5_exit(ns_delay)`	Exit *gem5* after *ns_delay* ns	exit with `m5_exit instruction encountered`
	`m5_fail(ns_delay, code)`	Exit *gem5* *ns_delay* ns with a fail code	exit with `m5_fail instruction encountered` and the code
	`m5_sum(a,b,c,d,e,f)`	Return the sum of a, b, c, d, e, f	handled by *gem5*
	`m5_init_param(key_str1,key_str2)`	Access a `init_param` from *gem5* (generally set up by python)	handled by *gem5*
	`m5_checkpoint(ns_delay,ns_period)`	Create a periodic checkpoint after *ns_delay* ns with period *ns_period*	exit with `checkpoint`
	`m5_reset_stats(ns_delay,ns_period)`	Reset statistics after *ns_delay* ns regularly at a period of *ns_period* ns	handled by *gem5*
	`m5_dump_stats(ns_delay,ns_period)`	Dump statistics after *ns_delay* ns regularly at a period of *ns_period* ns	handled by *gem5*
	`m5_dump_reset_stats(ns_delay,ns_period)`	Dump and reset statistics after *ns_delay* ns regularly at a period of *ns_period* ns	handled by *gem5*
	`m5_read_file(buffer,len,offset)`	Write the content of a host file provided by the config file to the *buffer*	handled by *gem5*
	`m5_write_file(buffer,len,offset,filename)`	Write *buffer* to the file *filename* on the host	handled by *gem5*
	`m5_debug_break()`	Trigger a breakpoint in *gem5* if it is currently being debugged (running *gem5* in GDB)	handled by *gem5*
	`m5_switch_cpu()`	Switch CPU, changing the CPU model use	exit with `switchcpu`
	`m5_dist_toggle_sync()`	Toggle sync between different *gem5* process (used in *dist-gem5*[Moh+17])	handled by *gem5*
	`m5_add_symbol(addr,symbol)`	Add symbol *symbol* with address *addr* to *gem5* (this is used by *Exec* DebugFlag)	handled by *gem5*
	`m5_load_symbol()`	Load the symbol file provided in `symbolfile` `System` *SimObject Param* in *gem5*	handled by *gem5*
	`m5_panic()`	Panic the simulator, stopping the simulation	handled by *gem5*
	`m5_work_begin(workid,threadid)`	Start a statistic automatic separation and create a checkpoint	handled by *gem5* besides checkpoint
	`m5_work_end(workid,threadid)`	End a statistic automatic separation and create a checkpoint	handled by *gem5* besides checkpoint
	`m5_workload()`	Call a *gem5* event (which can be implemented in Python) associated with the workload)	user-defined in python
	`m5_env(buffer,len,varname)`	Recover in a buffer the string associated with the key *varname* in a config python dict	handled by *gem5*

Table A.1: List of all the *m5* instructions. *"Exit with"* indicates that the `m5.simulate()` function returns with an event that contains a given message and/or code. Checkpointing always uses `m5_checkpoint` handling which is exiting the simulation with the `m5_checkpoint` message. The `m5_env` instruction is a new *m5* instruction, we added in our *gem5* build.

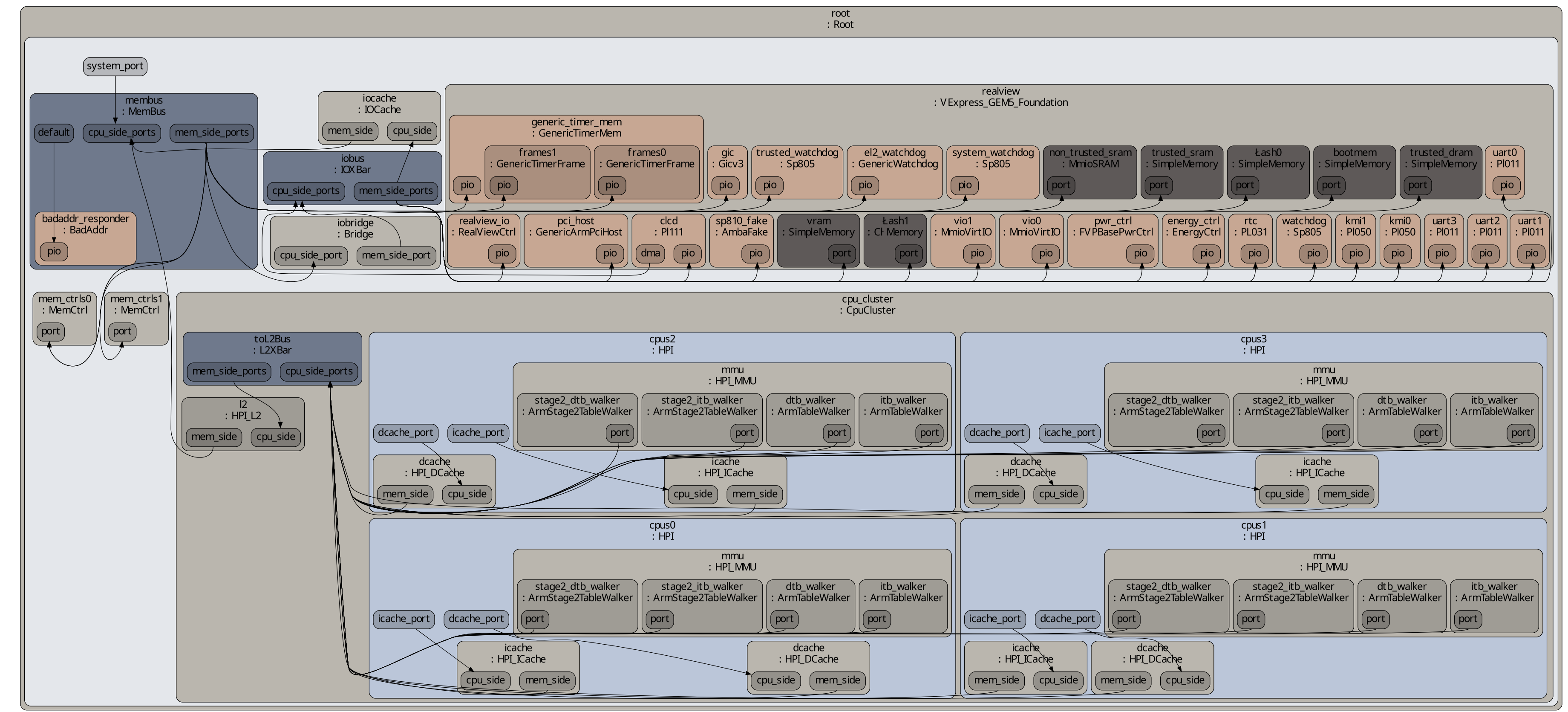


Figure A.3: *SimObject* tree and port: *gem5* can automatically generate a diagram to represent memory bus architecture. *SimObject* Tree is represented by box inclusion (children are included in parents) while port connections are represented as arrows (from `cpu_side` to `mem_side`). This one represents an ARM Versatile Express platform *SimObject*.

A.2 About PyDevices

sim_object.cc

```cpp
namespace py = pybind11;
SimObject::PyObj& SimObject::pyObj(){
    if(_pyObj!=nullptr)
        return *_pyObj;
    py::module_ m5 = py::module_::import("m5.object.SimObject");
    py::object obj=m5.attr("instanceDict").attr("__getitem__")(name());
    _pyObj=new py::object(std::move(obj));
    return *_pyObj;
    panic("_pyObj is not defined");
}
```

Figure A.4: How to get the Python object associated with the *CcObject* using pybind11

py_pio.cc

```cpp
namespace py = pybind11;
py::object& obj=pyObj();
py::object f=obj.attr("read");
PyData vec{std::vector<uint8_t>(pkt->getPtr<char>(),pkt->getPtr<char>()+pkt->getSize())};
//From Python read(self,daddr,data,secure)
bool ret=f(daddr,vec,pkt->isSecure()).cast<bool>();
```

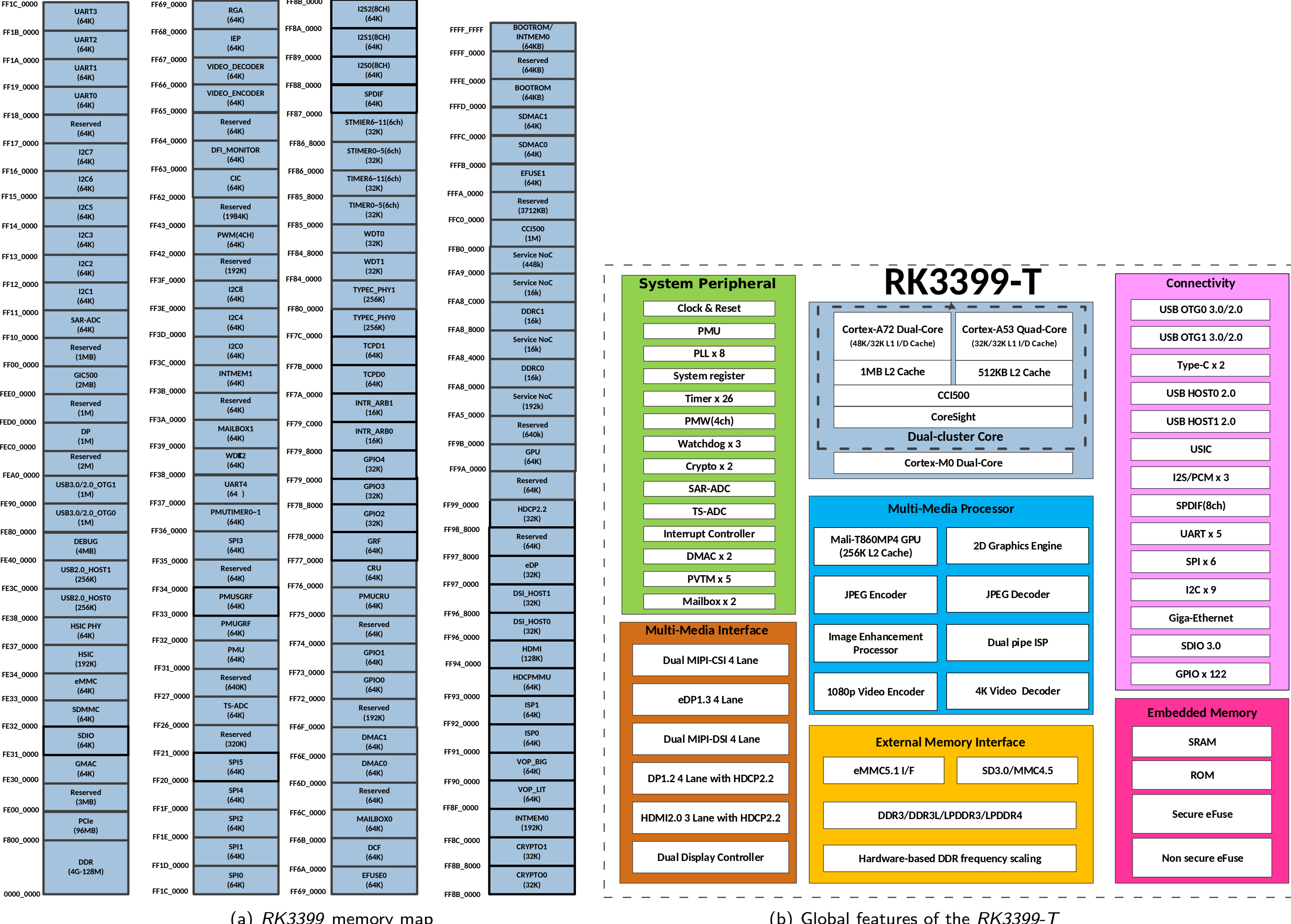
Figure A.5: Calling a Python method using A.4

A.3 About ARM

msr.S

```asm
        MRS x0, VBAR_EL1  // moving the content of VBAR_EL1 to x0
        MSR VBAR_EL1, x0  // moving the content of x0 to VBAR_EL1
```

Figure A.6: Accessing ARM system registers with `MSR` and `MRS`.



(a) *RK3399* memory map (b) Global features of the *RK3399-T*

System peripheral	
Abbr.	Definition
CRU PMUCRU	**Clock & Reset Unit** **Power Management Unit, Clock & Reset Unit** *The CRU is an APB slave module that is designed for generating all of the internal and system clocks, resets of chip. CRU generates system clock from PLL output clock or external clock source, and generates system reset from external power-on-reset, watchdog timer reset or software reset.*
GRF PMUGRF	**General Register Files** **Power Management Unit General Register Files** *The general register file will be used to do static set by software, which is composed of many registers for system control. The GRF is divided into two sections:* *GRF (used for general non-secure system) & PMUGRF (used for always-on system)* *The function of general register file is: IOMUX control, Control the state of GPIO in power-down mode, GPIO PAD pull down and pull up control, Used for common system control, & Used to record the system state*
SGRF PMUSGRF	**Secure General Register Files** **Power Management Unit Secure General Register Files** *These are the General registers only accessible through Secure memory transactions (in EL3 or EL1S).* *They control: DDR secure regions, peripheral security(block unsecure accessed to devices),* *DMA controller security related flags, other security functions, and* `RESET_ADDR` *registers.*
PMU	**Power Management Unit** *It contains registers controlling power domains. With its registers, power domains can be switched on and off.* *It is needed to implement PSCI using automatic power mode switch for CPU domains.*

(c) System peripheral in the *RK3399*, they contains memory mapped registers (see on figure A.7(a)). Their register's names are prefixed with their abbreviation

Figure A.7: Extracts from the *RK3399-T* TRM manuals

A.4 About *RK3399*

A.4.1 Retro-engineering

UART2 serial port

```
[    0.959803] EXT4-fs (mmcblk0p5): warning: mounting unchecked fs, running e2fsck is
    recommended
[    0.966967] EXT4-fs (mmcblk0p5): re-mounted 31ec842f-def1-467a-9b3b-0f9746868327. Quota
    mode: none.
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Initializing random number generator: OK
Saving random seed: [    7.201199] random: crng init done
OK
Set permissions on /dev/tee*: OK
Create/set permissions on /data/tee: OK
Starting tee-supplicant: Using device /dev/teepriv0.
D/TC:? 0 tee_ta_init_session_with_context:624 Re-open TA
    7011a688-ddde-4053-a5a9-7b3c4ddf13b0OK
8
D/TC:? 0 tee_ta_close_session:529 csess 0x300dff30 id 1
D/TC:? 0 tee_ta_close_session:548 Destroy session
Starting network: OK
devmem 0xFF170000 32: 0x00000000
Launching m5 test function
is_m5: false
We are not in gem5
mounting /dev/mmcblk0p6 on /mnt
[    7.449708] EXT4-fs (mmcblk0p6): warning: mounting unchecked fs, running e2fsck is
    recommended
[    7.464393] EXT4-fs (mmcblk0p6): mounted filesystem 90411e9d-c3db-436b-a537-3c777d969169
    without journal. Quota mode: none.
[   10.735720] platform ff320000.syscon:io-domains: deferred probe pending
[   10.736327] platform ff770000.syscon:io-domains: deferred probe pending
[   10.736915] platform cpufreq-dt: deferred probe pending
[   10.737411] platform sdio-pwrseq: deferred probe pending
Welcome to Buildroot, type root or test to login
buildroot login: root
#
```

Figure A.8: Image of the shell Linux command line interface

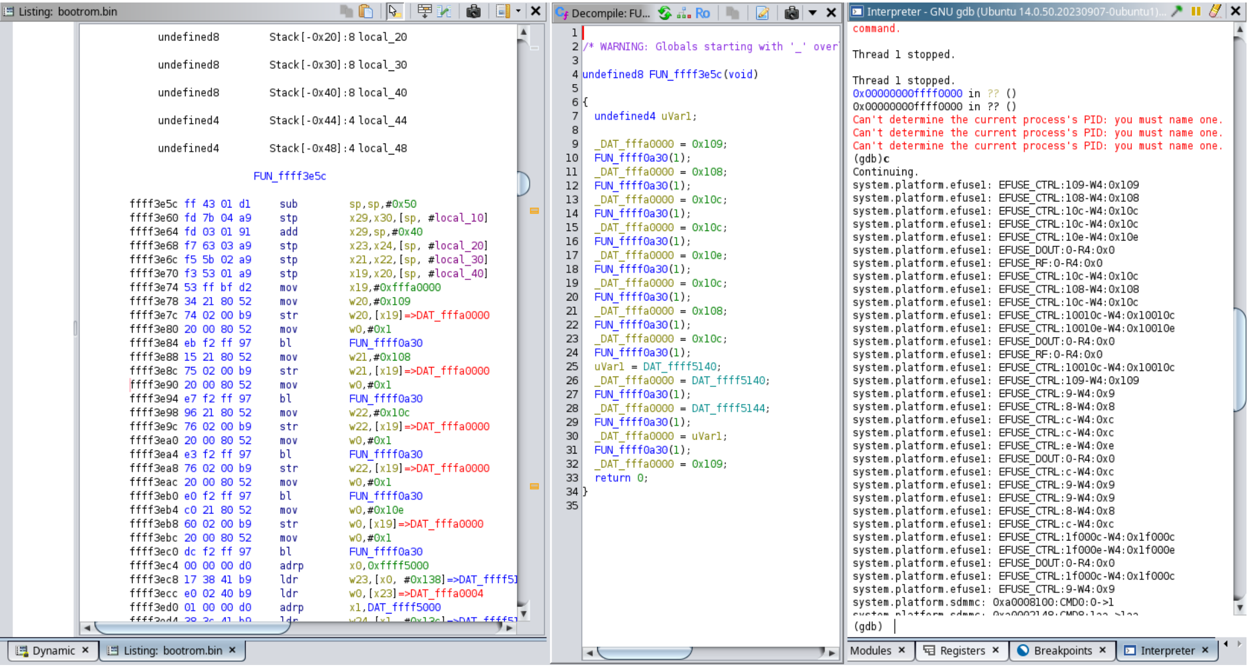


Figure A.9: *Ghidra* interface when it is connected to *gem5* featuring PyDevices: the integrated console displays messages from the PyDevices indicating which register was accessed.

A.4.2 Covert-channel results

AutoLock covert channel on the RockPi4

```
pass[16]=0
Result from VicC@fffff7600240:
        -startT=0x1a960e6, endT=0x1a972ae
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:211, [11]:97, [12]:236, [13]:97, [14]:97, [15]:97,
        -sum:1805, av:112, max:236
pass[17]=1
Result from VicC@fffff7600240:
        -startT=0x1aa6c76, endT=0x1aa814b
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:177,
        -[8]:97, [9]:236, [10]:302, [11]:225, [12]:193, [13]:283, [14]:200, [15]:194,
        -sum:2586, av:161, max:302
pass[18]=0
Result from VicC@fffff7600240:
        -startT=0x1ab7b4f, endT=0x1ab8d16
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:210, [11]:97, [12]:236, [13]:97, [14]:97, [15]:97,
        -sum:1804, av:112, max:236
pass[19]=1
Result from VicC@fffff7600240:
        -startT=0x1ad0cf0, endT=0x1ad21bd
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:177,
        -[8]:97, [9]:225, [10]:200, [11]:368, [12]:207, [13]:236, [14]:194, [15]:195,
        -sum:2578, av:161, max:368
pass[20]=0
Result from VicC@fffff7600240:
        -startT=0x1ae1c05, endT=0x1ae2dcc
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:210, [11]:97, [12]:236, [13]:97, [14]:97, [15]:97,
        -sum:1804, av:112, max:236
pass[21]=0
Result from VicC@fffff7600240:
        -startT=0x1af2590, endT=0x1af374d
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:211, [11]:97, [12]:225, [13]:97, [14]:97, [15]:97,
        -sum:1794, av:112, max:225
pass[22]=1
Result from VicC@fffff7600240:
        -startT=0x1b031d2, endT=0x1b04609
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:177,
        -[8]:97, [9]:223, [10]:200, [11]:236, [12]:193, [13]:236, [14]:194, [15]:193,
        -sum:2428, av:151, max:236
pass[23]=1
Result from VicC@fffff7600240:
        -startT=0x1b1431f, endT=0x1b1579b
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:188,
        -[8]:97, [9]:237, [10]:212, [11]:236, [12]:193, [13]:254, [14]:194, [15]:207,
        -sum:2497, av:156, max:254
pass[24]=0
Result from VicC@fffff7600240:
        -startT=0x1b25600, endT=0x1b267bc
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:210, [11]:97, [12]:225, [13]:97, [14]:97, [15]:97,
        -sum:1793, av:112, max:225
pass[25]=1
Result from VicC@fffff7600240:
        -startT=0x1b3624a, endT=0x1b376d8
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:188,
        -[8]:97, [9]:225, [10]:279, [11]:225, [12]:200, [13]:236, [14]:193, [15]:193,
        -sum:2515, av:157, max:279
pass[26]=1
Result from VicC@fffff7600240:
        -startT=0x1b47298, endT=0x1b4873a
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:180,
        -[8]:97, [9]:225, [10]:193, [11]:350, [12]:192, [13]:225, [14]:200, [15]:194,
        -sum:2535, av:158, max:350
```

AutoLock covert channel on the simulated RK3399 in gem5 using LRU+AutoLock

```
pass[16]=0
Result from VicC@fffff7a02100:
    -startT=0xba3d05f, endT=0xba3e311
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:93, [11]:93, [12]:93, [13]:93, [14]:210, [15]:210,
    -sum:1722, av:107, max:210
pass[17]=1
Result from VicC@fffff7a02100:
    -startT=0xba9a27a, endT=0xba9b7ea
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:209, [9]:210, [10]:210, [11]:210, [12]:210, [13]:210, [14]:211, [15]:210,
    -sum:2424, av:151, max:211
pass[18]=0
Result from VicC@fffff7a02100:
    -startT=0xbaf595e, endT=0xbaf6c11
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:93, [11]:93, [12]:93, [13]:93, [14]:210, [15]:211,
    -sum:1723, av:107, max:211
pass[19]=1
Result from VicC@fffff7a02100:
    -startT=0xbb53c03, endT=0xbb55174
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:210, [11]:210, [12]:210, [13]:210, [14]:211, [15]:210,
    -sum:2425, av:151, max:211
pass[20]=0
Result from VicC@fffff7a02100:
    -startT=0xbd0b11e, endT=0xbd0c3d0
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:93, [11]:93, [12]:93, [13]:93, [14]:210, [15]:210,
    -sum:1722, av:107, max:210
pass[21]=0
Result from VicC@fffff7a02100:
    -startT=0xbd68889, endT=0xbd69b3b
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:93, [11]:93, [12]:93, [13]:93, [14]:210, [15]:210,
    -sum:1722, av:107, max:210
pass[22]=1
Result from VicC@fffff7a02100:
    -startT=0xbdc6962, endT=0xbdc7ed4
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:210, [11]:211, [12]:210, [13]:210, [14]:211, [15]:210,
    -sum:2426, av:151, max:211
pass[23]=1
Result from VicC@fffff7a02100:
    -startT=0xbe25a51, endT=0xbe26fc3
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:210, [11]:210, [12]:211, [13]:210, [14]:210, [15]:211,
    -sum:2426, av:151, max:211
pass[24]=0
Result from VicC@fffff7a02100:
    -startT=0xbe83fbc, endT=0xbe8526e
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:93, [11]:93, [12]:93, [13]:93, [14]:210, [15]:210,
    -sum:1722, av:107, max:210
pass[25]=1
Result from VicC@fffff7a02100:
    -startT=0xbee16f2, endT=0xbee2c64
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:210, [11]:210, [12]:211, [13]:210, [14]:210, [15]:211,
    -sum:2426, av:151, max:211
pass[26]=1
Result from VicC@fffff7a02100:
    -startT=0xbf3ff89, endT=0xbf414fb
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:211, [11]:210, [12]:210, [13]:211, [14]:210, [15]:210,
    -sum:2426, av:151, max:211
```

Figure A.10: Comparison between results on the RockPi4 and in simulation after it was tuned: the sender occupies 8 ways to send a 1 and 0 to send a 0

AutoLock covert channel on the RockPi4

```
pass[16]=0
Result from VicC@fffff7600240:
        -startT=0x1a960e6, endT=0x1a972ae
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:211, [11]:97, [12]:236, [13]:97, [14]:97, [15]:97,
        -sum:1805, av:112, max:236
pass[17]=1
Result from VicC@fffff7600240:
        -startT=0x1aa6c76, endT=0x1aa814b
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:177,
        -[8]:97, [9]:236, [10]:302, [11]:225, [12]:193, [13]:283, [14]:200, [15]:194,
        -sum:2586, av:161, max:302
pass[18]=0
Result from VicC@fffff7600240:
        -startT=0x1ab7b4f, endT=0x1ab8d16
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:210, [11]:97, [12]:236, [13]:97, [14]:97, [15]:97,
        -sum:1804, av:112, max:236
pass[19]=1
Result from VicC@fffff7600240:
        -startT=0x1ad0cf0, endT=0x1ad21bd
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:177,
        -[8]:97, [9]:225, [10]:200, [11]:368, [12]:207, [13]:236, [14]:194, [15]:195,
        -sum:2578, av:161, max:368
pass[20]=0
Result from VicC@fffff7600240:
        -startT=0x1ae1c05, endT=0x1ae2dcc
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:210, [11]:97, [12]:236, [13]:97, [14]:97, [15]:97,
        -sum:1804, av:112, max:236
pass[21]=0
Result from VicC@fffff7600240:
        -startT=0x1af2590, endT=0x1af374d
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:211, [11]:97, [12]:225, [13]:97, [14]:97, [15]:97,
        -sum:1794, av:112, max:225
pass[22]=1
Result from VicC@fffff7600240:
        -startT=0x1b031d2, endT=0x1b04609
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:177,
        -[8]:97, [9]:223, [10]:200, [11]:236, [12]:193, [13]:236, [14]:194, [15]:193,
        -sum:2428, av:151, max:236
pass[23]=1
Result from VicC@fffff7600240:
        -startT=0x1b1431f, endT=0x1b1579b
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:188,
        -[8]:97, [9]:237, [10]:212, [11]:236, [12]:193, [13]:254, [14]:194, [15]:207,
        -sum:2497, av:156, max:254
pass[24]=0
Result from VicC@fffff7600240:
        -startT=0x1b25600, endT=0x1b267bc
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:97,
        -[8]:97, [9]:97, [10]:210, [11]:97, [12]:225, [13]:97, [14]:97, [15]:97,
        -sum:1793, av:112, max:225
pass[25]=1
Result from VicC@fffff7600240:
        -startT=0x1b3624a, endT=0x1b376d8
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:188,
        -[8]:97, [9]:225, [10]:279, [11]:225, [12]:200, [13]:236, [14]:193, [15]:193,
        -sum:2515, av:157, max:279
pass[26]=1
Result from VicC@fffff7600240:
        -startT=0x1b47298, endT=0x1b4873a
        -[0]:97, [1]:97, [2]:97, [3]:97, [4]:97, [5]:97, [6]:97, [7]:180,
        -[8]:97, [9]:225, [10]:193, [11]:350, [12]:192, [13]:225, [14]:200, [15]:194,
        -sum:2535, av:158, max:350
```

AutoLock covert channel on the simulated RK3399 in gem5 using LRU+AutoLock

```
pass[16]=0
Result from VicC@fffff7a02100:
    -startT=0xba474f0, endT=0xba487a3
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:211, [11]:210, [12]:93, [13]:93, [14]:93, [15]:93,
    -sum:1723, av:107, max:211
pass[17]=1
Result from VicC@fffff7a02100:
    -startT=0xbaa21bc, endT=0xbaa372f
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:211, [9]:210, [10]:211, [11]:210, [12]:210, [13]:211, [14]:210, [15]:210,
    -sum:2427, av:151, max:211
pass[18]=0
Result from VicC@fffff7a02100:
    -startT=0xbb007a7, endT=0xbb01a5a
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:210, [11]:211, [12]:93, [13]:93, [14]:93, [15]:93,
    -sum:1723, av:107, max:211
pass[19]=1
Result from VicC@fffff7a02100:
    -startT=0xbb5ea1e, endT=0xbb5ff90
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:211, [11]:210, [12]:210, [13]:211, [14]:210, [15]:210,
    -sum:2426, av:151, max:211
pass[20]=0
Result from VicC@fffff7a02100:
    -startT=0xbd125d9, endT=0xbd1388d
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:211, [11]:211, [12]:93, [13]:93, [14]:93, [15]:93,
    -sum:1724, av:107, max:211
pass[21]=0
Result from VicC@fffff7a02100:
    -startT=0xbd7032e, endT=0xbd715e0
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:210, [11]:210, [12]:93, [13]:93, [14]:93, [15]:93,
    -sum:1722, av:107, max:210
pass[22]=1
Result from VicC@fffff7a02100:
    -startT=0xbdce986, endT=0xbdcfef8
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:210, [11]:210, [12]:211, [13]:210, [14]:210, [15]:211,
    -sum:2426, av:151, max:211
pass[23]=1
Result from VicC@fffff7a02100:
    -startT=0xbe2d929, endT=0xbe2ee9c
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:211, [10]:210, [11]:210, [12]:211, [13]:210, [14]:211, [15]:210,
    -sum:2427, av:151, max:211
pass[24]=0
Result from VicC@fffff7a02100:
    -startT=0xbe89004, endT=0xbe8a2b6
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:93, [9]:93, [10]:210, [11]:210, [12]:93, [13]:93, [14]:93, [15]:93,
    -sum:1722, av:107, max:210
pass[25]=1
Result from VicC@fffff7a02100:
    -startT=0xbee6e00, endT=0xbee8372
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:210, [9]:210, [10]:210, [11]:210, [12]:211, [13]:210, [14]:210, [15]:211,
    -sum:2426, av:151, max:211
pass[26]=1
Result from VicC@fffff7a02100:
    -startT=0xbf45d6d, endT=0xbf472e0
    -[0]:93, [1]:93, [2]:93, [3]:93, [4]:93, [5]:93, [6]:93, [7]:93,
    -[8]:211, [9]:210, [10]:211, [11]:210, [12]:210, [13]:211, [14]:210, [15]:210,
    -sum:2427, av:151, max:211
```

Figure A.11: Comparison between results on the RockPi4 and in simulation after it was tuned using *gem5* Tree-LRU implementation: the sender occupies 8 ways to send a 1 and 0 to send a 0

List of Figures

1.1 Overview of Trusted Execution Environment typical use cases. 9
1.2 Scope of the Archisec project 11
1.3 Overview of the Archisec platform, instrumentation tools, and simulation capabilities 11

2.1 RTL simulator: when input is updated, all the logical gates that depend on it are computed, w.r.t clock-edge sensibility. Intermediate signal are stored for future computation, as any signal update is generally considered atomic. 14
2.2 Behavioral simulator: component with different model types, exchange messages. 14
2.3 Emulator: an emulator only reproduces the functional effect of the ISA and system components. Aspects of systems that are platform dependant (performances, randomness, etc.) are not reproduced 15
2.4 Simplified representation of *gem5* event queue (*gem5* only use the queue 0) 17
2.5 Image of the tree of *SimObject* and *Params* that is created by config files in *gem5* 18
2.6 Typical config file for *gem5* written in Python. 18
2.7 Overview of CPU models in *gem5*. They represent typical CPU architecture. Their pipelines tick following a clock which is modeled using regular events 19
2.8 Atomic model: two memory transactions through ports between *SimObjects1* and *SimObjects2*. A *SimObjects1* event is processed, which requires sending a packet to *SimObjects2*. 20
2.9 Timing model: two memory transactions through ports between *SimObjects1* and *SimObjects2*, the second one get back-pressured. 21
2.10 methodology comparison between article; Ours is *(C)* 23
2.11 Execution privilege: using MMU isolation, multiple levels of application can run on a single system. 24
2.12 Embedded attack scenarios: An attacker tries to attack a program running in an embedded system (**V**). It can run an attack program (**A**) on the target or use a physical medium (black arrows) to attack the victim (**V**) in order to bypass its interfaces (puzzle pieces around the victim). These constitute the victim attack surface. 24
2.13 Execution privileges: Different execution privileges control access to system elements. Each creates isolation layers using hardware mechanisms. The tenant software in higher privilege levels can grant access to their elements using an API that restricts what interactions are allowed. An attacker, wanting to have unrestricted access to a system element, will try to bypass hardware mechanisms or find vulnerabilities in the access API. 25
2.14 Representation of a side-channel: an attack program and a victim program use the same shared resource. 27
2.15 Comparison between *Flush+Reload* and *Prime+Probe*: We can see they exploit different cache set-up . . 28
2.16 A weak implementation can leak information in the cache through a variety of means: data access, instruction fetch, MMU table walk, or hardware prefetcher. 28
2.18 Typical *Spectre* Gadget 30
2.17 *Spectre* attack representation 30
2.19 *RowHammer* effect and DRAM structure 31

3.1 Structure of Chapter 3: white circles represent the key contributions for each of the subject axis. 34
3.2 Minimalist *SimObject* definition and declaration in *gem5* 36
3.4 Sequence of executed instructions when the *DebugFlag:Exec* is set 37
3.3 Enabling and disabling *DebugFlag* from config files 37
3.6 Internal logic for gem5 to restore a checkpoint from `ckpt_dir`. 38
3.5 Extract from a config file which handles the simulation loop. It follows the building and configuration of the system model and its instantiation. 38
3.7 How to use `cxxmethod` decorator to implement Python callable C++ methods 39
3.8 Example of *GDB* monitor call handling in *gem5* config files 39
3.9 A *GDB-stub* in a SoC or in *gem5* connected to *GDB* 39
3.10 GDB monitor message reception and handling in gem5 config files. 40
3.11 *GDB* monitor command: sending and receiving message during the simulation. 40
3.12 Simulation scripting: we use *GDB* to modified simulation parameters on-the-flight, the binary is never modified. 40
3.13 Typical use case for the interactive debugging session between *GDB* and *gem5* 41

3.14 *AutoLock*: This cache replacement policy prevents eviction of L2 lines that are still present in a cache L1. This lock is set up when a cache L1 receive a *miss* response from the L2. This lock on the L2 line is opened when all the L1 lines that lock a L2 line are evicted. 43
3.15 Representation of how a *Flush+Reload* works. The attacker knows that the victim took the branch because it detected the victim loaded the magenta gear. 44
3.16 Our prime set uses a double-linked list. Its elements are allocated in such a way that they have all the same cache index (here is `0x38`). If enough are allocated, they fill out all the possible ways for their index. 45
3.17 With the structure on figure 3.16, we can *probe* the set while traversing it. Timing measures for a prime entry are directly stored in it 45
3.18 How direction of probing control self-eviction: after a victim accessed two lines, different *probe* directions produce different results. 46
3.19 Comparison of the timing result for the cache timing attack between a Raspberry Pi 3B+(left) and *gem5*(right) 46
3.20 Result obtained with a baremetal attack (on the left) and a fast treatment to make result more visible on the right 47
3.21 Success of the attack when multiple CPUs are generating noise 48
3.22 ARMv8-A Exception Levels 48
3.23 TrustZone memory model: secure labeling is propagated along the memory hierarchy. 49
3.24 explanation of the AxPROT signal from the AXI4 norm [ARM21a] 49
3.25 Representation of the trusted boot process for *Vexpress* platforms 50
3.26 Node to add to DTB which declare *SMC* as a way to access OP-TEE 50
3.27 OP-TEE programming model: how TA communicates with Linux client applications 51
3.28 Structure of a Trusted Application (TA): User object files and libraries are bundled in a signed container. It provides the element to verify its content and load it in OP-TEE. 51
3.29 Opening a TA session and launching command from Linux 53
3.30 `xtest` allows to install TA at run time. `/mnt/ta` is a folder containing the TA to install 54
3.31 Implementation of the *sec-store* TA in OP-TEE: It tests reading and writing from secure storage. It thus uses the `tee-supplicant` to access the REE filesystem without exposing the encryption key. 55
3.32 Implementation of the *sec-sign* TA in OP-TEE: the TA prepares a buffer for the client. In this buffer, the client loads a message, which is then sent back to the TA to be signed using a RSA key never exposed to the client. 56
3.33 *gem5* platform configuration to emulate 3rd Party IPs. 56
3.34 Different timing modes for *gem5* taken from [Low24b]. The atomic mode closely resembles the SystemC Loosely Timed(LT), and the timing mode resembles the approximately timed(AT). 57
3.35 GIC architecture in *gem5* and ARMv8-A: each CPU as an interface which communicates the GICv3. There are only two types of interrupts FIQ and IRQ. The GIC distributes interrupts to the interface and controls to which CPU an interrupt will be routed and what type of CPU interrupts will be used. 59
3.36 Gadget to measure access time: using memory barriers (`DSB SY`) and instruction barriers `ISB`, the execution time of a single `LDR` instruction is measured. 60

4.1 Illustration of the *VictimScan* component in TEE-Time. Using simulator access to cache dump, *VictimScan* evaluates vulnerabilities using the source code. 62
4.2 Cross-core *Prime+Probe* attacks: an attack running in CPU1 is trying to attack the TEE in CPU0. 63
4.3 An attacker uses a cache timing attack to determine which branch a victim takes. To do that, it has to choose the right line to detect only an operation that happens in that branch. 64
4.4 Overview of TEE-Time: With this process we use our simulation platform (*gem5+GDB*) to craft an attack that we can test on real hardware. 65
4.5 *Prime+Probe* directions: above are *Prime+Probe* forward and below are *Prime+Probe* reverse. The victim uses cache occupancy, indicated as colored rectangles, to send a stair signal clearly visible on *pp-reverse*. . 68
4.6 `m5 env`: our new *m5* instruction to load environment variable at runtime 69
4.7 Structure of *VictimScan*: running in *GDB*; *VictimScan* programs breakpoint on *KEP* in *gem5*. From these breakpoints, *VictimScan* extracts raw cache data which are formatted using *VictimScan policy*. These formatted dumps, now made of a set of *KDS*, can be presented to the ranking algorithm. These *KDS* are processed to produce the *VictimScan* report 70
4.8 Classifying cache data on whether the victim function takes a branch using *Punctual Key Execution Point* 71

4.9 Classifying cache data on whether a victim function takes branch when the leaking computation happens after the branch. . . . 72
4.10 Classifying cache data on whether a victim takes branch when the leaking computation happens before the branch was taken . . . 72
4.11 Typical report from *VictimScan*, showing the *KEPs* classes, and the associated *KDS* with their scores; ranked in decreasing order. Each *KDS* also specifies the corresponding address in the binary, e.g. instructions from `.text` section, or variables in the heap. . . . 74
4.12 Cache timing traces for the simple example, the bottom figure being the top zoomed. The X-axis is the time. The moments when execution reaches a *KEP* are indicated with vertical lines. Prime set timings are shown with colored dots, with their Y-value corresponding to the sum access time for the prime set ($\sum T$). . . . 75
4.13 Zoomed in timing traces plotted relatively to *KEPs*. The attack traces for `0x210` sense a signal for both ♥ and ♠. The attack traces for `0x268` sense a signal for only ♠. The black vertical lines mark the moment the *KEP* was triggered and the timings are plotted relative to this moment. . . . 76
4.14 Algorithm and implementation from *mbedTLS* for the sliding window exponentiation algorithm from [MOV01]. . . . 77
4.15 S are square operation and M are multiply operation . . . 77
4.16 These are the *Key Execution Points*, we use to scan `mbedtls_mpi_exp_mod`. They are all scoped key execution points defined by the highlighted sections. . . . 78
4.17 TEE-Time report generated for *mbedTLS* and the *KEP* specified in the code extract. `UKN` implies that the cache line belongs to code outside *GDB* knowledge (e.g Linux kernel) . . . 79
4.18 Cache timing traces for the mbedTLS attack, the bottom figure being the top zoomed. The X-axis is the time. The moments when execution reaches a *KEP*, are indicated with ticks. *Prime+Probe* timings are shown with colored dots, with their Y-value corresponding to the sum access time for the prime set ($\sum T$). . . . 79
4.19 Zoomed in timing traces plotted relatively to *KEPs*. The black vertical lines mark the moment a *KEP* was triggered. Timings are plotted relative to this moment. . . . 80

5.1 Image of the RockPi4 C plus. It has GPIO pins like the Raspberry Pi, which can be used to access a UART. 82
5.2 CPU architecture and cache hierarchy . . . 83
5.3 System bus structure of the *RK3399*, it feature two interconnects which links CPU with all the memories and devices. . . . 83
5.4 *U-Boot* assembled *RK3399* boot process. While the `BootROM` is embedded in the SoC, all bootloader stages are included in the SD card. . . . 84
5.5 To support *Versatile express platforms (Vexpress)*, *gem5* ARM ISA implementation contains specific devices that behave like memory map IO. They can be connected to the system bus using ports in config files. However, to implement the `FVPBasePwrCtrl`, *gem5* integrates some of its functions directly into more general ARM ISA implementation in C++. . . . 85
5.6 Representation of execution flow between C++ *gem5* code and Python config file code for *PyPio*. . . . 86
5.7 Typical PyDevices implementation. A system can have multiple instances of the same devices with different settings. . . . 87
5.8 Extract from the *RK3399* TRM[Roc21]: register description for the *efuse* . . . 87
5.9 Covert channel primitives used to detect *AutoLock* on our RockPi4 and our rockchip platforms. This figure shows how this *signal primitives* works when *AutoLock* is active. CPU1 sends a 1 by priming 8 entries of its prime set. CPU0 checks if it receives a bit using Prime+Probe (with its own prime set). . . . 90
5.10 How *Prime+Probe* interacts with *AutoLock*: above are *Prime+Probe* forward and below are *Prime+Probe* reverse. On the left, without *AutoLock*, and on the right, with *AutoLock*. The victim uses cache occupancy, indicated as colored rectangles, to send a stair signal clearly visible on *pp-reverse*. . . . 91
5.11 First *VictimScan* report for the *sec-sign* TA with *KEP*s from figure 4.16, using an improved *KEP toolbox* implication and using the *nhit_inclusive policy*. . . . 94
5.12 Zoomed in timing traces plotted relatively to *KEPs*. Two sets (cache set with index `0x38` and `0x346`) are used in order to distinguish between the two *KEPs* ([𝕊] and [𝕄]). However, we see that we cannot detect [𝕊]. . . . 95
5.13 *VictimScan* report for the *sec-sign* TA with *KEP*s from figure 4.14(b) redefined, using the *nhit_inclusive policy*. . . . 95
5.14 We can measure the time between peaks in `0x38` or `0x346`. With this measurement, by comparing them with the minimum difference between two of these peaks, we can design a system of units to reconstruct the series of [𝕊]and [𝕄]. . . . 96
5.15 Comparison between simulation (left) and real hardware (right): centered around a similar pattern. . . . 96

5.16 We compared traces between (from top to bottom): *gem5* simulation using LRU, *gem5* simulation using *Tree-LRU* and the real platform. We can see that there is the same behavior during which a prime set gets "stuck" in an occupied state between simulated *Tree-LRU* and the real platform. 97
5.17 Accumulation of 50 real traces. These traces took 4 minutes to complete, with most of the time spent exporting data to the SD card. They have been fused and filtered with a gaussian filter. We used a peak detection algorithm to find the peaks associated with [𝕄], marked with stars. 97
5.18 Attack on the real platform against the *sec-sign* TA: 1 and 0 are bits from the private key that we identified using the [𝕊][𝕄]-series; the X corresponds to bits that we do not know, and that can be either a 1 or a 0. 98

6.1 Overview of the methodology deployed in the thesis: Starting from a real platform, we extract a workload that uses a TEE to run it on *gem5*. Using *Ghidra* and *GDB*, we are able to improve *the config files* in *gem5* to boot and execute the workload. Now that we can simulate the workload, we use *automatized script* in *GDB* to study it and find vulnerabilities. We can leverage these vulnerabilities in attacks, which we evaluate in simulation. Finally, these attacks can be deployed on the real platform to conventionally verify the vulnerabilities. 102

A.1 *gem5* event scheduling in *CcObject* implementation. 114
A.2 Integrating a new *SimObject* in *gem5* build system: *Scons* 114
A.3 *SimObject* tree and port: *gem5* can automatically generate a diagram to represent memory bus architecture. *SimObject* Tree is represented by box inclusion (children are included in parents) while port connections are represented as arrows (from `cpu_side` to `mem_side`). This one represents an ARM Versatile Express platform *SimObject*. 115
A.4 How to get the Python object associated with the *CcObject* using pybind11 116
A.5 Calling a Python method using A.4 116
A.6 Accessing ARM system registers with `MSR` and `MRS`. 116
A.7 Extracts from the *RK3399-T* TRM manuals 117
A.8 Image of the shell Linux command line interface 118
A.9 *Ghidra* interface when it is connected to *gem5* featuring PyDevices: the integrated console displays messages from the PyDevices indicating which register was accessed. 119
A.10 Comparison between results on the RockPi4 and in simulation after it was tuned: the sender occupies 8 ways to send a 1 and 0 to send a 0 120
A.11 Comparison between results on the RockPi4 and in simulation after it was tuned using *gem5* Tree-LRU implementation: the sender occupies 8 ways to send a 1 and 0 to send a 0 121

List of Tables

2.1 Comparison between simulators and the real platform as a reference. We can see that *gem5* is a suitable tool for our use case. 16

3.1 Simulation Configuration and Runtime. Times measured by *gem5*. No *GDB* acceleration is used in these runs. We run our examples on a *Intel(R) Xeon(R) Gold 6128* with 256GB of DDR4 54

4.1 Correspondence between *VictimScan policy* and *Prime+Probe* 69
4.2 Simulation runtime. Times measured by *gem5*. When using *GDB*, the acceleration methodology is used. We run our examples on a *Intel(R) Xeon(R) Gold 6128* with 256GB of DDR4. 76

5.1 RK3399-T: CPU and cache information gathered from ARM and Rockchip TRM documentation. 83
5.2 Execution time for Rockchip platform comparing the simulated *gem5* model and the real platform. 96
5.3 Example of D reconstruction using [$\mathbb{S}$][$\mathbb{M}$]-series. 98

A.1 List of all the *m5* instructions. *"Exit with"* indicates that the `m5.simulate()` function returns with an event that contains a given message and/or code. Checkpointing always uses `m5_checkpoint` handling which is exiting the simulation with the `m5_checkpoint` message. The `m5_env` instruction is a new *m5* instruction, we added in our *gem5* build. 114

Titre: Modélisation des micro-architectures pour la sécurité avec la plate-forme *gem5*

Mots clés: Cybersécurité, Système-sur-Puce (SoC), Attaques Micro-Architecturales, Environnement d'exécution de confiance (TEE), Retro-ingénierie, Platforme virtuelle

Résumé: Les systèmes embarqués sont la cible d'une grande variété d'attaques, tant au niveau logiciel que matériel. Parmi celles-ci, les attaques micro-architecturales sont particulièrement difficiles à étudier. En effet, en tirant parti des comportements spécifiques des systèmes sur puce (System-on-Chip (SoC)), ces attaques permettent à un attaquant de prendre le contrôle d'un système ou de ressources protégées, en contournant les mécanismes d'isolation entre processus. Ces attaques peuvent cibler toutes les parties d'un SoC : CPU, caches, mémoire, accélérateurs (FPGA, GPU,), interfaces, etc. L'environnement d'exécution de confiance (TEE), au cur de la sécurité des SoC modernes, impliqué dans la sécurisation d'applications bancaire, est lui aussi la cible d'attaques micro-architecturales. Dans cet thèse, j'adopte une approche basée la simulation pour la sécurité: au travers d'une plate-forme virtuelle basée sur gem5, je reproduis et étudie les attaques micro-architecturales contre les SoCs. Pour ce faire, j'ai amélioré le support de gem5 pour les TEEs, rendant possible l'utilisation d'un TEE open-source (OP-TEE) et le débogueur GDB présent dans gem5 pour permettre ainsi l'étude des scénarios d'attaque, tirant partie du simulateur. Avec cette interface, j'ai créé TEE-Time, un outil qui analyse les faiblesses cache-timing. Grâce à TEE-Time, j'ai trouvé des vulnérabilités dans des implémentations cryptographiques standard de RSA dans OP-TEE. Je les ais validées par des attaques cache-timing simulées avec gem5. Pour étendre ces attaques à un système réel, j'ai développé une plate-forme virtuelle reproduisant la carte RockPi4. Pour simuler son SoC RK3399 designé par Rockchip, j'ai développé les PyDevices des outils de prototypage-rapide utilisant l'interface Python de gem5. A travers la simulation d'attaque cache, j'ai découvert que le RK3399 utilisait AutoLock, un protocole de cache spécifique à ARM. En incorporant AutoLock dans gem5, j'ai simulé un scénario d'attaque ciblant le RSA d'OP-TEE sur le RK3399. En exécutant cette même attaque sans aucune modification sur un RockPi4, j'ai réussi à faire fuir en moyenne $\sim$30% des bits de la clé RSA, faisant ainsi le lien entre attaques cache et leur exploitation dans un vrai système.

Title: Modeling of micro-architecture for security with *gem5*

Keywords: Cybersecurity, System-On-Chip, Micro-Architectural Attacks, Trusted Execution Environnement, Retro-engineering, Virtual Platform

Abstract: Embedded systems are the target of a wide variety of attacks, both software and hardware level. Microarchitectural attacks are particularly difficult to study. By taking advantage of the specific behaviors of systems-on-a-chip, these attacks enable an attacker to take control of a system or protected resources, bypassing process isolation mechanisms. These attacks can target all element in an SoC: CPU, caches, memory, accelerators (FPGA, GPU), interfaces, etc. The Trusted Execution Environment (TEE), key element of SoC security and involved in securing banking applications, is also the target of micro-architectural attacks. In this thesis, I adopt a simulation-based approach to security: through a virtual platform based on gem5, I reproduce and study micro-architectural attacks against TEEs. To achieve this, I improved gem5's support for TEEs, allowing the use of an open-source TEE (OP-TEE) I also augmented the GDB debugger present in gem5 to allow the study of attack scenarios, leveraging the simulator environment. With this interface, I created TEE-Time, a tool to analyze cache-timing weaknesses. Thanks to TEE-Time, I found vulnerabilities in standard RSA implementations in OP-TEE, I validated this vulnerabilities with cache timing attacks simulated using my virtual platform. To further validate these attacks on a real system, I developed a virtual platform reproducing the RockPi4 board. To simulate the Rockchip RK3399 SoC on the RockPi4, I developed PyDevices fast-prototyping tools for system devices using gem5's Python interface. Through cache timing simulation, I discovered that the RK3399 uses AutoLock, an ARM-specific cache protocol. Compiling AutoLock into gem5, I ran my attack scenario targeting OP-TEE's RSA implementation on the RK3399 simulation. By executing this same attack without any modification on a RockPi4, I managed to leak an average of 30% of the RSA key bits, thus making the link between cache attacks and their exploitation in a real system.

Institut Polytechnique de Paris
91120 Palaiseau, France