University of Versailles Saint-Quentin-en-Yvelines (UVSQ)

Institute of Sciences and Techniques of Yvelines (ISTY)

Master 2 Internship at CNRS/IRISA

Simulating Transient Execution Attacks with *gem5*

Internship Supervisors
Clémentine MAURICE
Annelie HEUSER

Referent Teacher
Louis GOUBIN

Author
Pierre AYOUB

Internship
Rennes, from 01/04/2020 to 31/09/2020

Acknowledgments

As an internship is not a single-person adventure, I must thank the people who made this experience possible.

First of all, I would like to thank my two internship supervisors in particular, Clémentine Maurice and Annelie Heuser, who supported me throughout this period of time. They monitored the progress of my work over the weeks and took their some of their time to give me a couple of valuable advice. Their presence allowed me to improve my working methods while letting me explore the ideas I wanted to develop. I would like to give a special acknowledgment to Clémentine, who placed her trust in me before knowing me and starting to work with me.

I would also like to thank two of my teachers, Louis Goubin and Michaël Quisquater, for their instruction, their remarkable management of the master's degree during my academic year and the time they spend to discuss my future with me. I definitely could not thank all the teachers I have admired during the year, otherwise the list would be too long.

I wish to thank my colleagues at the laboratory too, even though I have not seen them much because of the COVID, for the little shared moments and interesting discussions.

Finally, I would like to conclude this by acknowledging all those who, in one way or the other, have contributed to this internship or to the development of this report.

Mots clés : Sécurité, microarchitecture, matériel, simulation, prédiction de branchement, mémoire cache, processeur, attaque « transient », *Spectre*, *Flush+Reload*, *gem5*

Résumé

Des informations classées « secret défense » aux informations médicales, des grands organismes nationaux aux petites entreprises, la sécurité numérique est devenue un champ d'expertise et de recherche essentielle dans le processus de protection de l'information depuis quelques dizaines d'années. À l'heure actuelle, en s'appuyant sur d'élégants modèles mathématiques, des outils logiciels aux implémentations élaborées et du matériel à la pointe de la technologie des semi-conducteurs, l'ensemble des acteurs traitant de l'information à protéger est en mesure de répondre à cette contrainte à différents niveaux de sécurité. Cependant, le domaine de la sécurité fait face à de nombreux défis à relever dans un futur proche, afin de pouvoir continuer à garantir ce qu'il prétend garantir. Là où les cryptosystèmes et les logiciels sont mis au défi par des chercheurs en cryptographie et en sécurité depuis un certains temps, l'architecture matérielle était un aspect parfois négligé voir oublié des recherches avant les 6 dernières années : mais aujourd'hui, le monde académique et industriel s'en est emparé. S'ajoute à cette asymétrie entre les considérations logicielle et matérielle un autre défi pour cette dernière, celui de la reproductibilité. En effet, par nature, la recherche d'attaque et de défense sur le matériel est complexe à mettre en œuvre ainsi qu'à reproduire dans le temps.

Ainsi, c'est dans ce contexte que s'inscrivent de nombreuses recherches sur la sécurité matérielle, aussi bien académiques qu'industrielles, depuis ces quelques dernières années. Cependant, la question de la reproductibilité n'est que très peu abordée dans la littérature. L'équipe *EMSEC* au sein de l'*IRISA*, une Unité Mixte de Recherche du *CNRS*, s'intéresse aux questions relatives à la cryptographie et la sécurité matérielle. Plus particulièrement, le travail effectué durant mon stage au sein de cette équipe a pour objectif de faire avancer l'état de l'art sur la simulation d'une certaine classe d'attaque sur le matériel, en mettant en avant la reproductibilité.

Keywords : Security, microarchitecture, hardware, simulation, branch prediction, cache memory, processor, "transient" attack, *Spectre*, *Flush+Reload*, *gem5*

Abstract

From military "top secret" classified to medical information, from the largest national organisms to the smallest companies, computer security has become an essential expertise and research field in the process of protecting information over the past decades. Nowadays, based on elegant mathematical models, carefully crafted software implementation and hardware at the cutting edge of semi-conductor technology, the group of actors dealing with sensitive information is able to satisfy the security requirement at different levels. However, the area of computer security is faced with numerous challenges in a close future to continue to guarantee the security it claims to provide. Where cryptosystems and software are tested by cryptography researchers or software engineers since many years, hardware architecture was often a neglected or forgotten point of security before the past 6 years: but today, both academic and industrial worlds have taken over this subject. On top of this asymmetry between software and hardware considerations, the latter faces another challenge: reproducibility. Indeed, by essence, attack and defense research on hardware is complex to implement as well as to reproduce in time.

Thus, it is in this context that much research on hardware security, both academic and industrial, have been carried out over the past few years. Nonetheless, the question of reproducibility is barely studied in the literature. The *EMSEC* team within the *IRISA* laboratory, being formally a "Mixed Research Unit" of the *CNRS*, is interested in cryptography and hardware security related questions. More specifically, the work achieved during this internship aims to complete the state-of-the-art knowledge on simulation of a particular class of hardware attacks, focusing on reproducibility.

Contents

List of Figures

List of Tables

Glossary

ARM The brand name of a processor designer company. *ARM* produces its processors at a conceptual level, then sells them to a founder company to build them. The name also designate the brand *RISC ISA*. 4, 6, 20, 21, 23, 25, 28, 29, 37–40, 42, 43, 45, 46, 51, 52, 55, 56, 62, 63, 66

Backdoor A functionality or a vulnerability, respectively depending if it is voluntary or not, which allows an unrestricted access by bypassing the usual security check (for, e.g., authentication) in a piece of software or hardware. 2

Basic block A sequential set of instructions, delimited at the beginning by a jump target and at the end by a branching instruction. 20

Branch Predictor A logical unit in the *CPU* which is used to predict – i.e., to guess – if the next branch in the future executed code will be taken or not taken, based on various heuristics. 6, 12

High-Performance Computing A domain of computer science which is devoted to high-performance computers, which are enormous machines with thousands of cores dedicated to the simulation or resolution of huge problem – e.g., astronomy or cosmology, energy, transports. High-performance computing can refer to all sub-fields that are used to increase performance of these computers, simultaneously theoretical – e.g., linear algebra – and applied one – e.g., parallel programming –. 6, 20, 27, 46

Instruction Set Architecture Abstract model of how a processing unit works. It defines basics instructions, memory and arithmetic operations, and all that is required to have a complete machine. For exemple, *x86* is the most widespread *ISA*. 9, 13, 20, *Acronyms:* ISA

Interrupt A temporary break into the program execution flow to execute another piece of code, causing by another software or the hardware, which can be synchronous or asynchronous. 39

LISP An old family of programming languages being both imperative and functional. Often considered as extremely mathematically elegant, it is a language of choice for artificial intelligence research. 5

Micro-operation In a CISC architecture, or in some RISC, regular assembly instructions are divided into multiple micro-operation before to be executed. It allows a finer-grained control into instruction execution and dependency analysis. 21

Microarchitecture In a central processor, it corresponds to the physical implementation of a specific *ISA*. It can be extended to any hardware on how the low-level software specification is physically implemented and wired. 2

MSR A special register related to the microarchitecture of the processor, unlike the general purpose registers – e.g. of a general purpose register, EAX is available into all processors that implement the *x86 ISA*. *MSR* are used to control some low-level functionalities related to debugging, profiling, performance measurement and error control. A software that use a register like this will be very likely to be specific to one particular microarchitecture. In order to read or write into one of these registers, the programmer must use the MRS or MSR instructions for the *ARM* architecture, respectively. 40, *Acronyms:* MSR

Outlier In a data series, an outlier is an entry which differ significantly from the others. It can be due to a bias, hazard, an error, or even be normal in rare cases. It is to the statistician or the experimenter to determine their origin. 45, 48, 53

Page In the context of virtual memory – where each program has its own address space –, a page is a chunk of memory. Each page is a contiguous range of addresses which is mapped onto physical memory, but not all pages have to be in physical memory at the same time. When a program references a part of its address space, the hardware performs the necessary mapping on the fly. 40

PMC A *MSR* specialized into the counting of events related to the processor performance. Every modern processors possessed a limited number of *PMC*. An event is a fact related to the hardware which happened at a given time. Counting an event could be, for example, counting the number of cache misses or issued instructions. 39, 62, *Acronyms:* PMC

PMU A specialized hardware module that holds *PMC* – generally between 2 and 8 – and used to program them to count the desired events. Every *PMU* is able to count a huge number of events, but only few can be monitored at the same time – depending on of the number of *PMC*. If we want to monitor a number of events larger than the number of *PMC*, the *PMU* will have to use multiplexing and extrapolation techniques in order to approximate the value of all the events. 39, 55, 62, 63, *Acronyms:* PMU

Prefetcher A hardware or software unit charged of loading memory ahead of usage time by identifying patterns in memory accesses. 40

R A programming language dedicated to statistics and data analysis. It has a strong community composed of computer scientists and mathematicians, which makes a complete ecosystem around it for data science. 5

Scheduler The operating system component in charge of scheduling. Scheduling corresponds to the allocation of execution time window to the processes on the system. A scheduler can use a lot of different algorithms, depending on its goal – minimizing latency, execution time, maximizing fairness – or its architectures – single core, multiple cores. 39

Simultaneous Multithreading Technology to improve *CPU* performance, virtually allowing more than one thread to execute on one physical core. The *Intel* implementation is commonly known as *Hyper-Threading (HT)*. 2, *Acronyms:* SMT

sysfs It is a pseudo file system which is used to expose variables from kernel space to user space. The variable exportation is achieved by a directory hierarchy where each file corresponds to one variable, some of them being read-only and the others being configurable by writing into them. 39

Turing-complete It corresponds to a system that is able to simulate any Turing machine thanks to its arithmetic and memory operations. 2

x86 The more wide-spread and popular *CISC ISA* in the world for computers, developed by *Intel and originating from the famous 8086 processor – back in the end of the 70s. 20, 28, 31, 37–39, 43, 45, 46*

Introduction

COULD we find some points in common between computer security and the atmosphere of the Earth? Both of them are constantly around us, are essential – in a different way – for our lives, but the vast majority of people never thought about them. Essentially, almost every service we use every day is more or less tight with computer security or cryptography. We can think about bank services and online shopping, web-based administrative services, telecommunications, social networks and so on. Moreover, we could also suppose how will be the relationship between the future and digital security with smart devices, autonomous cars, connected cities and the Internet of Things. All of these technologies have a strong connection: they would not be usable and reliable without security, cryptography and privacy, which are major topics of our societies evolution in the past decades.

Until now, we have talked about computer security, privacy and cryptography. To be precise, we need to carefully define them. Cryptography is the science of protecting information by making it unintelligible for someone that is not intended to see it. Modern cryptography is based on advanced mathematical theories that have been studied for around a few dozens years and applied on digital information, while this field finds its roots into simple techniques already used during the antiquity to protect sensible messages. Privacy as a field is very large and corresponds to both theoretical and practicals aspects. It aims to study and design privacy-aware technologies, protocols and systems. There are multiple approaches to privacy: of course the technical one, but also an ethical and legal one. Finally, computer security is the applied field that studies the security of computers, networks and systems while developing attacks and defenses. We understand that these three specialties are tightly intertwined today.

Computer security emerges from the "hacker" world built in the 60s at the *Massachusetts Institute of Technology (MIT)*. This term was used to qualify students who liked to experiment with computer technologies, not only applied to security but in general with novel techniques. In the 80s, mainstream media start to publish about some famous hackers – sometimes called *crackers* – which committed some crimes, like *Kevin Mitnick* for its intrusion into the database of the *American Department of Defense*. Then, the 90s was the decade of hackers clubs and, with Internet – the successor of ARPAnet –, a large spread of information about this world in both mentality and skills. To quote a major hacker and open source figure about hackers:

> Hackers built the Internet. Hackers made the Unix operating system what it is today. Hackers run Usenet. Hackers make the World Wide Web work. – *Eric Raymond*

Today, the computer security field is composed of numerous actors. All private companies have computer security experts to protect their digital information system. Groups of security experts, called *Computer Emergency Response Team (CERT)*, are mobilized by companies or governmental agencies to strive against computer attacks and malwares. Every government has intelligence agencies that gather information, mainly military and economical one, but they do not share a lot with the computer security community. Last but not least, the academic world continues to contribute to computer security in a more institutional way.

Computer security has many applications, like virology, systems, reverse engineering, networks, forensic, software and data. The specialization that interest us is hardware security. The latter is not especially widely spread when people though about security, but it is just as important as other fields. Indeed, when a hardware constructor claims that hardware cannot be compromised, that it is completely isolated from software, or that the implementation of a specific cryptographic algorithm is secure, we can see in the literature that this is not always true. Hardware security has multiple approaches and targets. One of them, for example, is the use of electromagnetic radiation caused by electronics circuits. From this source of leakage, we can perform different kinds of attacks, an example is to reconstruct the image displayed on a computer screen from the leakage of the cable, today known as *Van Eck Phreaking* [Van85]. The use of electromagnetic emanation is known since a long time by intelligence agencies: the NSA has multiple standards of attacks and defenses since the end of the 70s, known as *TEMPEST* [NSA82]. Hardware security mostly targets embedded devices, which are ubiquitous. An example is to analyze the firmware of some digital circuits, even modifying them to implement an embedded backdoor in a computer component, demonstrated by Zaddach et al. in a hard-drive firmware [Zad+13]. Finally, the specialization of hardware security that we study in this report is the security of the microarchitecture. This domain of computer security studies and exploits components microarchitecture to develop attacks and defenses, targeting both hardware or software implementations of security check or cipher algorithms. A lot of additional mechanisms are implemented into our *CPUs* – not essential to have a Turing-complete machine – for compatibility, performance or security reasons. This sometimes generates security issues, more and more often due to their complexity. One illustration of a performance mechanism exploitation is *PortSmash* developed by Aldaya et al., which exploits *SMT* to gather some information on the execution of a target software that leads to cryptographic key leakage [Ald+19]. This kind of work demonstrates how we can attack some software with another piece of software by knowing how a performance optimization is physically implemented.

Microarchitectural attacks exploit slight details of the electrical and logical architecture of the processor. Thus, hardware security is by essence more complex to perform than other kinds of security research: tiny details of microarchitecture can be different from two same models of components, hardware can be expensive or no longer available, compatibility of researches can be very reduced across hardware evolution in time or various hardware families. Both developing novel methods and reproducing past researches are affected by these observations. Simulation allows to programmatically run tasks, with fixed parameters or with different set of parameters. This allows to perform automatically a lot of reproducible tests with different data. It is in this context that the simulation of the microarchitecture can help, improve or even solve these problems in the future. This kind of hardware simulation has been used for one or two decades, but is not so conventional because it faces many challenges of accuracy and speed. Applying simulation to microarchitectural attack would be a real advantage to solve the problems cited above. However, to be able to simulate these attacks, the simulation models should be very accurate for tiny details (*e.g.*, cycle-accurate). This internship aims to explore the possibility to use this kind of simulation with a simulator named *gem5*, and to apply it to a specific class of microarchitectural attacks.

This report is organized as follows. First, we are going to introduce the internship context by presenting the organization, the goals and the working methods. Then, we will explain all the needed background to understand the internship, taking topics from various fields – security, architecture and simulation. Next, we will cover all the setup needed to start the main work of the internship. Finally, we will detail our contributions and discuss them.

1 Internship Context

THIS 5th year internship aims at delve into a branch of a security that sounds interesting for our future career and our specials interests. In this case, it is about integrate a research laboratory to explore novels topics which does not have a lot of prior research on them and needs a multidisciplinary background – security, architecture, cryptography –.

In this chapter, we will look at the context in which the internship was achieved. Firstly, we will present the organization taken as a whole – since it is not specialized at all into computer security – and the laboratory. Then, we will focus on the working environment and topics which are not directly related to computer security, nevertheless important to research in general. Finally, we will discuss the outline of the internship achievement.

1.1 Organization and Laboratory Presentation

The *National Centre for Scientific Research (CNRS)* is the major national research center in France, also being one of the top as the world and European scale. With more than 80 years of existence, it is divided into dozens of institutions – often in partnership with universities, engineering schools, or other research laboratories – which leads to more than 1000 research units and 31 000 researchers exploring all domains of knowledge, including humanities and social sciences, natural and formal sciences. Moreover, the center is famous for its number of researcher laureate to the Nobel Prize and Fields Medal.

The *Research Institute of Computer Science and Random Systems (IRISA)* is the largest French computer science laboratory, founded in 1975 and issued from a collaboration between eight institutions, of which we find among them the *CNRS*, the *National Institute of Applied Sciences (INSA)* and the *National Institute for Research in Computer Science and Automation (INRIA)*. It is composed of more than 800 researchers, which covers both applied and theoretical computer science.

The internship will take place into a team associated to the *CNRS*, inside the *IRISA* laboratory.

1.1.1 Activities

The *CNRS* develop its activities around all domains of science. Here, we are interested in its work in computer science. Both theoretical and applied computer science are carried out by researchers, with an emphasis – at least in their communications – in new fields, like quantum cryptography, or multidisciplinary topics, like the computer science contribution to the future of the bio-technologies. By being a public and general research center, the organization is not enclosed into one field or commercial sector.

The *IRISA* operate in many fields of computer science: networks, systems, architecture, language, signal processing, artificial intelligence, security and cryptography. The computer security department is interested in the entire information processing chain, from hardware to networks and cryptography. Also, some related subjects like privacy or security policy are studied. Finally, the laboratory aims at apply its research to significant problems of our society, like environment and ecology, transports, energy and culture.

1.1.2 Geographical Situation

The head office of the *CNRS* is implemented into Paris, since it is a world-wide organism and has laboratories across the French frontier. Head office of the *IRISA* is located in Rennes, and its laboratories are implemented on 3 geographical sites: Lannion, Vannes, Rennes. In theory, the internship should have been in the latter, but due to the COVID-19 pandemic, the majority of the internship has been done in remote working. In Rennes, the *IRISA* laboratory is located on the Beaulieu campus, founded during the 60s and specialized into natural and formal sciences. On this campus, we can find the Rennes 1 University and the *INSA*, which makes it a scientific pole for the region.

1.2 Working Environment

The laboratory does not differ from others on many aspects. We are destined to talk to people from multiple disciplines, which is very instructive especially when you need help on a new field. We will describe the team in which the internship have taken place, as well as available hardware and environment. For the working methodology, we have large freedom of choice and it is very dependent on the supervisor. The chosen one for the internship will be detailed in Section 1.2.3.

1.2.1 Team Composition

The team associated to the *CNRS* that hosted me is called *EMSEC*, which stands for "Embedded Security and Cryptography". The acronym perfectly represents the activities of the team, which addresses questions related to cryptography, formal methods, and security of software and hardware systems. This internship clearly fit into the last category. Members of *EMSEC* are responsible for the organization of some security events in the academic community, like *Gildas Avoine* who leads both *EMSEC* and the *GDR* security group. The latter cover all topics from cryptography to security, and coordinate some events to stimulate the research and federate different actors. The two supervisors of this internship are *Clémentine Maurice* – permanent member of the *EMSEC* team – and *Annelie Heuser* – permanent member of the *TAMIS* team –, both being *CNRS* full-time researchers. *C. Maurice* works on low-level hardware security by software attacks, which is fully-related to this internship, while *A. Heuser* works on vulnerability and malware analysis both with a formal and applied approach. Every week, a meeting was scheduled to take stock about the internship's progress.

1.2.2 Available Hardware

There were two requirements to be able to work properly: a powerful enough machine to run the simulation – which is quite processor intensive – and a board under the *ARM* architecture to develop and run the attacks. For the first requirement, a decent desktop computer will be fine – no need for a node of a high-performance cluster. The second one has been fulfilled by buying a *Raspberry Pi 4*, a tiny fully functional computer embedded in a single card. Its configuration is described in Table 1.1, while the details of the processor will be covered in Section 2.1.3. This one is used as a representative [1] member of processors used in some devices like smartphones. Finally, to have a bit more of diversity into our future experiments, a node of a national cluster, *Grid5000*, was available for us – with an *ARMv8-based Marvell ThunderX2 99xx* processor composed of 32 cores (256 hardware threads), used as a representative [1] member of *ARM* high-performance processors.

1.2.3 Reproducible Research

Reproducibility is the capacity for an experiment to be repeated, with the same results, by another team of researchers across time. Reproducibility is of major importance for a scientific study, otherwise,

[1]We use it as a representative member because we did not have a complete panel of processors to experiment with. Therefore, it is not statistically representative.

Table 1.1: Hardware description of the *Raspberry Pi*.

Component	Model	Properties
Raspberry Pi	v4 Model B	
Processor	ARM Cortex-A72 (ARMv8) SoC Broadcom BCM2711	Quad-core, 64-bit, 1.5 GHz
Memory	–	4GB LPDDR4-3200 SDRAM
Storage	Kingstone Micro SD HC I	32 GB
Network	–	IEEE 802.11ac, Gigabit Ethernet, Bluetooth

the chain of knowledge building will be broken. Almost every philosopher of science would agree about this statement, while even the preeminent *Karl Popper* – one of, if not the major epistemologist of this century – consider reproducibility as a pair criterion for a study to be considered as scientific, with his well-known falsifiability criterion [Pop34]. Moreover, sciences must cope with a reproducibility crisis since two decades, especially medicine and social sciences like psychology [Ioa05]. Indeed, more or less studies – depending on the field considered – are not reproducible or in contradiction between each other. This is a multi-factorial effect, and its analysis is a matter of *meta-science*: the scientific study of the scientific process. This is why I have chosen to follow a *MOOC* on reproducible research, made by the *INRIA* and *CNRS* researchers [PLH]. The goal of this course is to give us both methodologies and tools to conduct research in a more reproducible way. First, a large overview of the evolution of scientific research methods at an individual scale – mainly note taking and organization – is presented. Then, they introduce the concept of *computational document*: a digital document where everything that is needed to reproduce the considered study is embedded into it, that means data, code, results, comments and redaction. We have the choice between three "modern" – the oldest was created in 1976 – tools to accomplish the class:

1. *RStudio*, created by *Joseph J. Allaire* ;
2. *Jupyter*, developed by the *Jupyter Project* nonprofit organization ;
3. *Emacs with Org-mode*, created by the famous hacker and free-software advocate *Richard M. Stallman*.

The first one is an *IDE* for the *R* language, the second a framework especially built "to support interactive data science and scientific computing across all programming languages", the last one is a powerful text editor developed in *C* and *LISP*. The chosen one for this internship is *Emacs with Org-mode*, since it is the more extensible – any *LISP* program can be executed inside *Emacs* – and flexible one – nearly every programming language is supported by *Org-mode* –, but also the more difficult to use – since it is quite old, modern conventions are not used inside the editor –. Concretely, *Org-mode* is an *Emacs* extension which brings a *meta-programming language* into the editor. This language allows, in a single file or document, to use and combine different common programming languages – *e.g.*, *Bash* and *Python* – and orchestrates data exchanges between them. For this internship, the advantage of using this is evident: the person that will work on the same subject will be able to see *all* my results, and replicate them if needed into his research. The downside of this is simple: since *Emacs* and *Org-mode* are very complete tools, they can be difficult to use and it has for effect to slow down the research process at the beginning.

1.3 Internship Execution

The internship subject not being taught during my master's year, I had only a few personal knowledge about hardware security and microprocessor simulation before the beginning. Thus, before to respond to the problem statement of the internship, I had to become aware of the state-of-the-art knowledge in both fields. This is what I called the preliminary work. Then only, I spent my time developing and experimenting attacks and simulation in order to respond to the issue of the internship.

1.3.1 Preliminary Work

The first subject that catch my time is the class of attacks studied into this internship (explained in Section 2.3.2). There is a *lot* of readings about this class of attacks. Of course there is some research papers, but also technical guides on the Internet, and a plethora of technical news – from security companies or medias with a security expert. We have to understand that when this kind of attacks were discovered few years ago, it has generated a massive excitement around it. Thus, there is a lot of confusion on the Internet about the taxonomy of these attacks, because in the end, they create a very deep tree of classification. Of course, all of this will be demystified later in the report. Moreover, during the first month, I was able to experiment with some related attacks. For example, I used a tool called *Mastik* – developed by *Yuval Yarom*, a well-known researcher in this area – to perform various micro-architectural attacks on the cache, hence obtain an *RSA* key during a *GnuPG* encryption [Yar].

The second crucial topic to learn before to start the main work was the simulation part. *gem5* has some documentation and even an official guide, but unfortunately, they are quite both more or less outdated and incomplete. Having said that, this did not stop me to simulate some simple systems and to learn *gem5* basics. For the most of it, learning *gem5* is achieved by online guides and practice. There is a bunch of papers which use this simulator, but they are very specialized on a particular question and related to high-performance computing for the majority of them.

Aside from these two subjects, I was also able to perform an extensive review on some *CPU* optimization mechanisms, like the branch predictor. To accomplish this, both online university courses and research papers have been used. It was very instructive, the only pitfall is that, retrospectively, it should have been done shorter.

Finally, we can recall that there was a certain amount of time spent into the setup of a convenient reproducible research environment (as described in Section 1.2.3). Moreover, since I will continue in a Ph.D. thesis next year, all of this methodology will definitively be very useful.

1.3.2 Main Work

The main work has been divided into three parts (excluding the report writing), evolving at the same time with my findings and encountered difficulties. It would have been complex to plan all the internship at the beginning since the subject is not well-known.

The very first time slice of the internship has been devoted to set up all required tools, then study and use the already existing implementations of considered attacks. The setup was quite long, since there was a lot of imperatives. The first one have been to install the Raspberry Pi with a 64-bit kernel and compilation tools, which was not as easy as we could think. Then, we needed to compile and make *gem5* work, which can be tricky under some conditions. Concerning the attacks, a lot of implementations of them are available on the Internet, unfortunately most of them are developed and built for the *x86* architecture. In our case, we needed an *ARM* implementation. Reference implementations were not working on our hardware, and after a long period of time trying to find out why, we gave up and decided to implement our own attack. In addition to a working implementation, we needed one with specific behavior and possibilities to fit our requirements, as explained in Section 4.1.2.

The second part was the attack implementation. It was not easy and longer than thought before. The main point is that this kind of attacks are very unstable: a small perturbation in the hardware usage or operating system load can, sometimes, completely change the final results of one run [2]. Thus, we needed an implementation especially constructed to remove these irregularities in the results, otherwise comparing different runs would be useless.

Finally, the third work item was the simulation with *gem5*. The first goal was to build a platform – based on our *Raspberry Pi* and its processor – capable of reproduce some attacks that works on our real hardware. Then, the second step was to study how the system is implemented into *gem5* in order to understand how we can modify the simulated system to improve the accuracy of the simulation. Ultimately, it would have been very interesting to extend our study to others class of

[2]This behavior can remind us a chaotic system in physics, where "a butterfly's wing can produce a tornado".

attacks. Unfortunately, it has not been done because the implementation of the attack on our real hardware have taken too much time, and same time cannot be taken for each kind of attacks.

2 Background

MULTIDISCIPLINARY problems need multidisciplinary background. As we said before, this internship is at the crossroads between computer architecture and security. To fully understand all the issues and challenges, we need to study these two domains. Thus, we will first dive into the microarchitecture of a modern processor. Obviously, this report cannot cover all the microarchitectural components of a *CPU*, consequently only microarchitecture basics and relevant units will be explained. Then, we will dig into the simulation of the microarchitecture, its feasibility and all of its challenges. Finally, we will see how all of this knowledge can be used to attack cryptographic implementations, or the memory of other software from a simple one without any special privilege.

2.1 Microarchitecture

The architecture of a processor is defined by its *ISA*, which is the interface between the hardware and the software – we are talking about low-level software, though. The same *ISA* could have multiple implementations. The implementation of an *ISA* is called the microarchitecture of a processor.

At the very beginning of computers, processors were quite simple. They executed instructions, one by one, in the program order. The execution was straightforward, meaning that each step of the instruction processing was performed sequentially. However, this simple architecture was very inefficient compared to today's processors. Indeed, the old processors already had fewer hardware resources compared to new ones, because the transistor count was very *small* – thousands transistors in a 70's processor, billions in a today's processor –, but also because nearly no optimization techniques were implemented. Today, an incredibly high number of techniques is deployed in our processors, mainly to gain performance and reduce power consumption. We will describe some of these techniques, which are useful to our understanding of microarchitectural attack and simulation.

Pipeline. The *pipeline* is of one of the most fundamental optimization mechanism. Instead of viewing the execution of an instruction as a single operation, we divide this execution into many steps, called *pipeline stages* [TA13]. An instruction execution can be divided into dozens stages – for the most complex pipeline –, each stage relying on one different hardware unit. Thus, this mechanism allows *instruction-level parallelism*, with all stages able to run in parallel, the only dependency being that a stage n needs the value of the stage $n-1$. Figure 2.1 shows a five-stage *pipeline*, function of time. Each Sx is a pipeline stage. The boxes represent an instruction execution, the number inside being the instruction identifier. We can see that each instruction flows through the different *stages* of the *pipeline*, deeper when time passes. For example, at time 3, we have the *operands* of the first instruction being *fetched*, the *decoding* of the second instruction and the *fetch* of the third one, all in parallel. A processor capable of executing more than one instruction per cycle is called *superscalar* [SS95], often with multiples execution units inside the $S4$ box of Figure 2.1, allowing to fully execute instructions in parallel. Other techniques are used to increase pipeline usage inside *superscalar* processors, like Out-of-Order (OoO) execution allowing to re-order instructions at runtime and execute them more

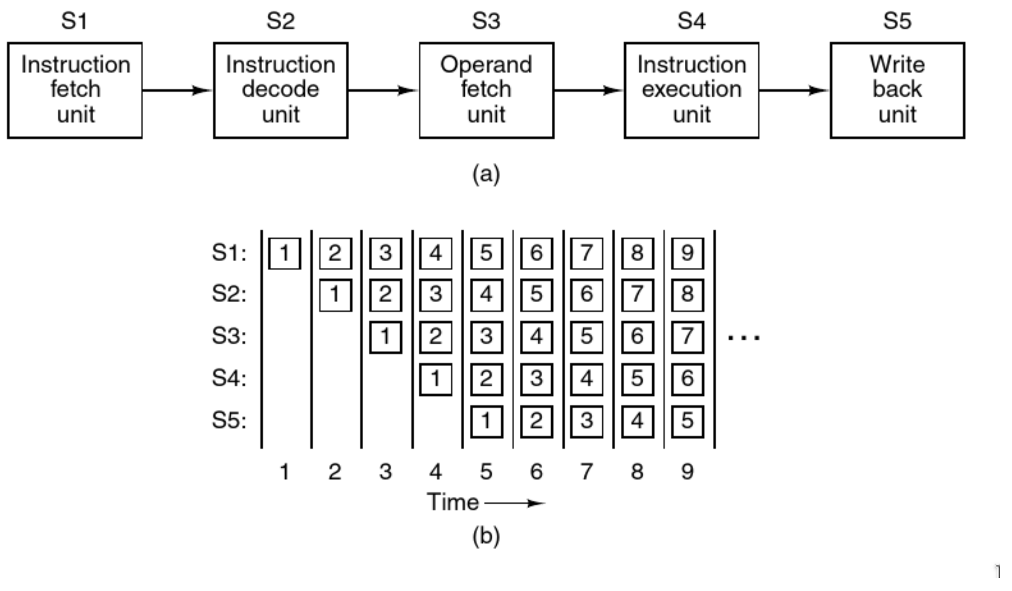


Figure 2.1: A five-stage pipeline [TA13].

efficiently.

Many optimizations aim at reduce the penalty of a *pipeline stall*, that means a pause into the execution of instruction in the pipeline due to an event with a high latency – a memory access, a costly computation, a dependency to resolve, and so on. This is the case for the caches, used to reduce the latency of memory accesses, and the branch prediction which implements *speculative execution*, which aims at keeping the pipeline fully loaded at all time.

2.1.1 Caches

Since processors are much faster than memories, the latency when loading a word from the main memory is too high to be acceptable. The solution is to organize the memory into a hierarchy, with smaller but faster memories close to the processor, and larger but slower memories further away. A cache memory is a small, faster, and expensive memory [Smi82] [TA13] – on the contrary of the main memory which is larger, slower and economic. Cache memory is the kind of memory which is close to the processor, which aims at keeping more often used data close and quickly accessible. When the processor needs to load a memory word, it will first check the caches. If it finds the required data, it will be called a *cache hit*. Otherwise, we call this a *cache miss* and the processor will check the main memory, further away in the hierarchy. Even the cache memory is organized in a hierarchy, often between 2 and 4 levels – denoted *L1* to *L4*. The first level, *L1*, is connected to the processor load/store unit and fetch unit. The Last-Level Cache (LLC) – which could be the *L2*, the *L3* or the *L4*, depending on the processor – is the largest but the slower one, connected to the main memory. A *cache line* is the smaller unit to load or flush data from the cache, often equal to 64 consecutive bytes. A lot of properties, architectures and protocols could be used to govern caches. We will briefly describe some of them, being exploited during microarchitectural attacks or necessary to understand how a basic cache works.

Granularity. The first level of cache, the *L1*, is often separated in two: one instruction cache and one data cache. This is called a *banked* or *splited* cache. This architecture allows to load a value for an operand and an instruction in parallel [TA13]. The other levels are called *unified* caches, because both instructions and data resides in the same cache.

Coherency. The cache coherency aims at giving a coherent view of the memory for all cores and all processors of one system. In practice, it requires that writing into one cache should be replicated in another caches to be coherent. The simplest form of coherency is the *snooping-based* one, where all cores share a common bus and the *snooping unit* spy memory writes. When a write is detected from one core, other cores invalidate the data in their own cache. The most known protocols used for cache coherency are *MSI*, *MESI*, and *MOESI* [SBL04].

Accessibility. A cache can be accessible from only one core, then it is called a *private* cache, or from multiple core of the processor, then it is called a *shared* cache. In multi-core processors, the LLC is often *shared* between cores to maintain *coherency* between the lower level of caches.

Inclusion. Caches are generally *inclusive*, which means that a level n of cache contains all the data and instructions of level $n-1$, in addition to its own exclusive content since it is larger. If a cache does not include the lower level content, it is called *exclusive* cache.

Mapping. Caches are smaller than main memories, thus, we have to use a technique to map a data of the main memory to a place in the cache. Obviously, since the cache is a subset of main memory, collisions are going to happen – which produce bad performance. We will briefly present techniques used to map a data into the cache.

Direct-mapped Each memory word can be stored in exactly one place in the cache. This is not used today, since it produces bad performance when two addresses compete for one cache line.

n-way set-associative Each memory word can be stored in n places. The cache is divided into n sets. Finding the right number of sets is a question of trade-off between complexity and collision.

Fully associative Each memory word can be stored anywhere in the cache. It can work only in small cache, otherwise the lookup penalty would be too high.

Indexing and Tagging. The *index* determines where the data of main memory goes into the cache – or into a set –, while the *tag* is used to perform a lookup when the processor search for a value in the cache – both for a load or a write. Each line contained in the cache contains a *tag*, which is used to identify the data of the main memory contained by the current cache line [TA13]. This *tag* can be computed in a various number of way, but the simplest one is to directly use a portion of the memory address. The same choices arrives to compute the *index*. When using a portion of a memory address, we have the choice to use the Virtal Address (VA) or the Physical Address (PA). It leads to the following choices:

Physically Indexed (PI) / Physically Tagged (PT) The cache could use the physical address of a memory location to compute the index or the tag.

Virtually Indexed (VI) / Virtually Tagged (VT) The cache could use the virtual address of a memory location to compute the index or the tag. Only using virtual addresses can produces *aliases* and *homonyms* problems, because a different physical address could translate in the same virtual addresses, and *vice-versa*. An *aliases* arrives when several VA correspond to the same PA, thus the same physical location can be duplicated in the cache and a memory access could access to an old copy of the data. An *homonyms* arrives when a given VA corresponds to several PA, then a memory access could access the wrong data.

These 4 choices can be combined. Typically, virtually indexed caches are on-chip while physically indexed caches are off-chip. We often found:

Physically Indexed and Physically Tagged (PIPT) This is the most simple construction, it avoids completely problems of address *aliasing* and *homonyms*, and is more flexible to transmit data between cores. However, for each cache lookup, a translation has to been made which induce a penalty. Most modern processors uses generally a *PIPT* LLC [Gru+16].

Virtually Indexed and Physically Tagged (VIPT) It is quicker because it does not involve the Memory Management Unit (MMU) for all memory operations. It avoids *homonyms* and some *aliasing* problems like *VIVT* caches. Most modern processors uses generally a *VIPT L1I* cache.

Table 2.1: Types of branches.

Type	Example	Direction	Possibility	Resolved
Conditional	`if`	Unknown	2	Execution
Unconditional	`goto`	Taken	1	Decode
Call	`call`	Taken	1	Decode
Return	`ret`	Taken	∞	Execution
Indirect	`ptr()`	Taken	∞	Execution

Policies. A number of policies governs the cache usage, regarding the nature of the operation.

Replacement In a set-associative cache, we have to choose which cache line has to be evicted when the cache is full. Some simple algorithms could be used, like *randomly* choose the value to evict or use the *Least Recently Used (LRU)* policy, where the oldest used value is evicted. Other algorithms exists [ZPL01].

Writing When a value is wrote in the cache, we can either write this value in memory immediately – this is a *write-through* – or write it later, *e.g.*, when the value will be evicted – this is the *write-back*. The latter is preferred to save useless stores that are expensive.

Prefetching When a memory value is loaded, the processor can speculatively load values the are close to this loaded value – this is done by the *prefetcher*. This is based on *spatial locality*, which means that closest values in the memory are likely to be used together. This is the most simple form of *prefetching*, while a lot of techniques and heuristics exists [Mit16a].

2.1.2 Branch Prediction

The final important concept of the micro-architecture needed to understand the core of the attacks studied during this internship is the branch predictor component, which implements *speculative execution*. In this section, we will learn some basics and more advanced internals of branch prediction, enough to understand the *gem5* branch predictor's implementation and the impact on micro-architectural attacks. Many sources have been synthesized here [Mit16b] [Smi98] [Mut18] [Lee]. This review might be extensive compared to the required knowledge to understand most basics attacks. However, it can be useful if we study, in the future, more advanced attack scenarios.

The branch predictor is an internal component of the processor, which tries to guess the outcome of a branch instruction. If the next encountered branch is a conditional one, the predictor has to guess if the branch will be *taken* or *not taken*, even before the evaluation of the branch condition. If the branch is an unconditional one, the predictor has to guess where the branch will lead the execution flow. There are different types of branches that could cause a pipeline stall, presented in Table 2.1. We can see that different types of branches could produce a variety of different scenario. Often, different kinds of branches will need different requirements in the branch predictor implementation.

From what we have seen, we can distinguish 3 interrogations that have to be predicted during the fetch stage of an instruction:

Instruction prediction Whether the instruction is a branch or not.

Branch prediction Guess whether a conditional branch is or isn't taken.

Branch target prediction Guess the target address of a taken branch before the computation by the execution unit.

Historically, we can separate two paradigms of prediction:

Static prediction This prediction is done at compile time. For example, we can decide to always apply the *taken* or the *not taken* strategy to all branches. There is also some basics heuristics like *Backward Taken, Forward Not-Taken (BTFN)*, which is adapted to branches that handle loops. We can also perform *profile-based* prediction, with a first run to profile the program execution path for a given input. There is also the *program-based* approach, which consists in static analysis of the generated code and uses more advanced heuristics, regarding the nature of the instructions [BL93]. Finally, there is *programmer-based* prediction, where the *ISA* attach semantics to branches about preferred direction by programmer's pragmas – *likely*, *unlikely*. For example, in the *Motorola MC88110*, we have `bne0` which means *preferred taken* and `beq0` which means *preferred not taken*. Static prediction is quite old and since it has no apparent impact on micro-architectural attacks or in *gem5*, it will not be studied further here.

Dynamic prediction This prediction is done at runtime. The processor guesses the future program behavior based on past execution. It can look at the *temporal correlation*, which means that a branch previously taken is likely to be taken again. It can also look at *spatial correlation*, where we can detect some preferred execution path. All the following algorithms fall into this paradigm. Into the dynamic branch prediction family, there are two different goals when designing a branch predictor:

General predictor Concentrate on designing branch predictors that work best on all types of branches. The future discussed algorithms falls into this category.

Specific predictor Try to resolve a specific branch prediction problem, like predicting inner loop branches, end loop branches, return branches, etc. Usually, these specific branch predictors are used as *side-predictors* with the general one. They will not be studied here.

Algorithms. We will talk about algorithms used in branch prediction units. Before, we have to understand an essential concept to really appreciate the following algorithms. A *saturating counter* is a short sequence of bits – generally, between 2 to 4 bits – incremented when a branch is taken, and decremented when a branch is not taken. Usually, the uppermost bit of the counter is used as the prediction (if it is set to 1, then it indicates that the branch was previously taken at least $\frac{1}{2}$ time). The following algorithms are presented in a tree-like fashion, to clearly distinguish them. The complexity of presented algorithms will increase progressively.

1. **Single-Level**

   The following algorithms have the particularity to have a *single-level index* mechanisms to predict the outcome of a branch.

   (a) **Last Time (1-bit Counter)**

   This is a very simple algorithm. We store *one bit* indicating whether the branch was *taken the last time or not*, and we base our prediction on the value of this single bit. This bit can be stored in the *Branch Target Buffer (BTB)* – detailed later, a structure which stores target addresses of encountered branches. This algorithm is not very efficient.

   The detailed protocol is the following. We compare the *tag* of the branch – one key that we want to use for identifying the branch, for example, its address – with the tag entry in the *BTB*. If it is equal, then we look at the single counter bit. If it is set to 1, then we take the *BTB* address entry as the next Program Counter (PC) value. Otherwise, we just increment the PC by one.

   (b) **Bimodal (2-bit Counter)**

   This is a simple algorithm, which is an improvement of the *Last Time* predictor [1]. The goal is to add what's called *hysteresis* to our state machine (the states being the different counter values), to solve the problem that the single-bit algorithm changes his prediction too quickly. To do this, we store a *two-bit counter* per branche. One bit provides the prediction, and the other provides the *hysteresis*. The counter can also be part of the *BTB*. The counter uses *saturating arithmetic*, which is an arithmetic in a fixed range, *i.e.*, with a maximum and a minimum value. We get approximately 85-90% accuracy.

[1] It should not be confused with the *Bi-Mode* predictor, described later.

2. **Two-Level**

   The *Two-Level Predictor* (also called *Correlation Predictor*) idea is originally presented in 1991 [YP91], enhanced in 1992 [YP92]. The name come from the idea of using a two-level indirection when predicting the outcome of a branch.

   This is one essential concept introduced with this paper. A *history pattern* is a sequence of n-bit indicating, for each bit, the *taken* or *not taken* direction for the last n executed branches in fetch order. The buffer that hold history pattern(s) is called the *Branch History Register (BHR)*.

Paradigms. This *Two-Level* scheme considers two paradigms for doing branch prediction. They are presented below.

(a) **Global Branch Prediction**

It works by considering all branches history. This can be useful to identify patterns in the program and to predict a branch outcome which is dependent of previous branches. For example, if we have a `if (a == b)` branch which follows two taken branches being `if (a == 0)` and `if (b == 0)`, then we know that the current branch has to be taken. This scenario cannot be identified with the local predictor – presented below, but the global predictor is able to.

More precisely, we want to make a prediction for the current branch based on the outcome of the current branch the last time the same global history of branches was encountered. To do this, we have to keep track of the global history (*taken* / *not taken*) for a fixed number of branches. The buffer that holds the global history is called the *Global History Register (GHR)*, which is a shift register – basically, a global *BHR*. Then, we just have to use the *GHR* value to index a table that records the outcome for each *GHR* values seen in the past – *e.g.*, with a 2-bit counter per index: this is the *Pattern History Table (PHT)*. These mechanisms explained the *Global Two-Level*, called like this because of the indirection of one table indexing another and the global paradigm.

(b) **Local Branch Prediction**

It works by considering only the history of a particular branch. For example, if we have a `if (a == 0)`, only the previous history (*taken* or *not taken*) for this branch – *i.e.*, this address – will be considered.

This mechanism is the analog of the *Global prediction*, but instead of considering a branch history for all branches, we store one *history pattern* per branch. The history register – the architectural structure containing the history pattern – is selected based on the address of the branch – *i.e.*, the value of the PC when fetching the branch.

(c) **Taxonomy**

The literature about two-level predictors is plentiful. Figure 2.2 describes all strategies that could be used in a *two-level* branch predictor. These strategies are classified as the following:

i. *BHR* can be global (G), per set of branches (S), or per branch (P) ;
ii. *PHT* counters can be adaptive (A) or static (S) ;
iii. *PHT* can be global (g), per set of branches (s), or per branch (p).

Usually, between all of these strategies, the *per set of branches* one is chosen because it is more accurate than the *global* one and less costly than the *per branch* one. This choice is exploited in some version of *Spectre* – the attack we will discover in Section 2.3.2 – to perform what is called an *out-of-place* training, which exploits *branch aliases*.

Bias-Free. These algorithms are *two-level* ones, part of a family with the goal of reducing the *bias*. A *biased branch* is one with a probability of outcome which is superior or inferior to 1/2. For example, a loop condition or an error handling branch is biased, because it is not taken and taken very rarely, respectively.

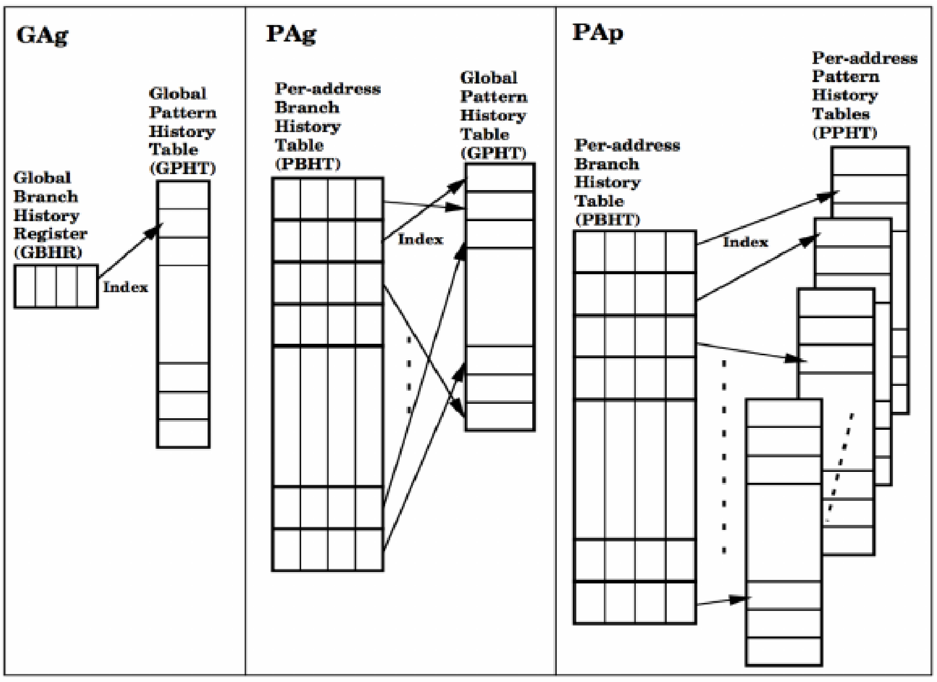

GAg
Global Pattern History Table (GPHT)
Global Branch History Register (GBHR)
Index
PAg
Per-address Branch History Table (PBHT)
Global Pattern History Table (GPHT)
Index
PAp
Per-address Pattern History Tables (PPHT)
Per-address Branch History Table (PBHT)
Index

Figure 2.2: Taxonomy of two-level predictors paradigms [Mut18].

(a) **Agree**
This *Agree Predictor* [Spr+97] tries to reduce the *inherent bias* of a branch. A lot of branches are *taken* or *not taken* most of the time, thus the bias in inherent to their nature. To have a better exploitation of this property, to reduce the bias and the interference in the *PHT*, this predictor uses another structure to store some *bias hints*, being little indicators which are used in the balance of the final prediction. It can be used with both a local and a global approach.

(b) **Bi-Mode**
This branch predictor aims to reduce the bias and the interference in the *PHT* by separating *mostly-taken* and *mostly-not-taken* branches into *two different PHTs* [LCM97].

Interference-Free. These algorithms are *two-level* ones, part of a family with the goal of reducing *interference*, also called *aliasing*. Since the number of branch predictor entry is limited, multiples branches with different behaviors could share the same branch predictor entry, resulting in significant degradation of the prediction accuracy.

(a) **GShare**
This one is a little improvement over the *Global prediction*. The idea is to add a bit *more context* while using the global history. The improvement works by *hashing together* (*e.g.* by a XOR) the *branch address* (from the PC) and the *global history pattern* (from the *GHR*) and use this value to index the *PHT*. A good consequence of using hashed index is to reduce *PHT* interference.

(b) **GSkew**
This predictor [MSU97] [Sez] has for goal to reduce the interference in the *PHT*. It achieves this by using *multiples PHTs*, *e.g.*, 3, and *indexing* each one with a *different hash* function. Then, we have a *majority voting* system that chooses between the different outcomes. The idea is that branches that interfere in one table are unlikely to do so in another table.

(c) **YAGS**
This predictor [EM98] aims to reduce *PHT* interference by maintaining a *small cache* containing all *branches that interfere* with other ones. It is then used to not mispredict them

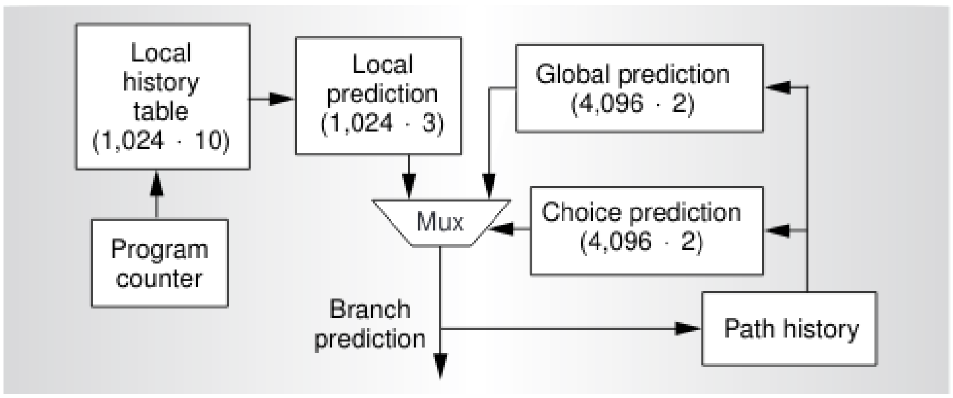

Local history table (1,024 · 10)
Local prediction (1,024 · 3)
Global prediction (4,096 · 2)
Program counter
Mux
Choice prediction (4,096 · 2)
Branch prediction
Path history

Figure 2.3: Illustration of the *Tournament Branch Predictor* [Kes99].

again.

Hybrid. These algorithms are *two-level* ones, part of a family with the goal of combining both *global* and *local* paradigms.

(a) **Tournament**

This branch predictor algorithm (also called *Hybrid Two-Level Prediction*) [McF93], is used since the *Alpha 21264* [Kes99] processor launched in the year 1996.

All branches, depending on their nature, can be predicted better with either *local* or *global* correlation techniques. Since this algorithm implements both techniques but we should only have one final decision, it uses a *chooser* (called *choice predictor*) which chooses between the *local* or *global* predictor for each branch. The following numbers are given as an implementation illustration, from the *Alpha* processor. The *chooser* is a 4096-entry table of 2-bit *saturating prediction counters*. It is indexed by the *path history* (a 12-bit *history pattern* (4096 possibilities)) and dynamically selects either the *local* or *global* predictor for each branch invocation. The processor has to train the prediction counters to reflect the correct choice between the two. Since it is dynamic, for one branch invocation, both predictors can be used during the program's life.

Figure 2.3 shows how the processor combine local history prediction, on the left, with global history prediction and a choice predictor, on the right.

The *local predictor* consists of a two-level table which records the history of individual branches encountered. The first level, the *local history table*, is a 1024-entry table of 10-bit history pattern (same kind of the path history) indexed by the PC (*i.e.*, the branch address). The second level, named *local prediction*, is a 1024-entry table, each entry being a 3-bit saturating counter which indicates if the branch is taken or not.

The *global predictor* is a single-level 4096-entry table indexed by the path history (as the choice predictor), named *global prediction*, each entry being a 2-bit saturating counter.

The *key idea* of this scheme is to *combine* both local and global predictors with a two-stage prediction (history pattern then saturating counters), helped with a multiplexer controlled by the choice predictor. The number of tables, of depth, of entries and of counter size is a try to determine how to obtain the better trade-off between storage, lookup and hardware complexity, finally performance for this algorithm.

3. **Advanced**

These algorithms are the most advanced ones. They can use statistical techniques, machine learning concepts, or be very complex in term of implementation. But in return, they have the best accuracy.

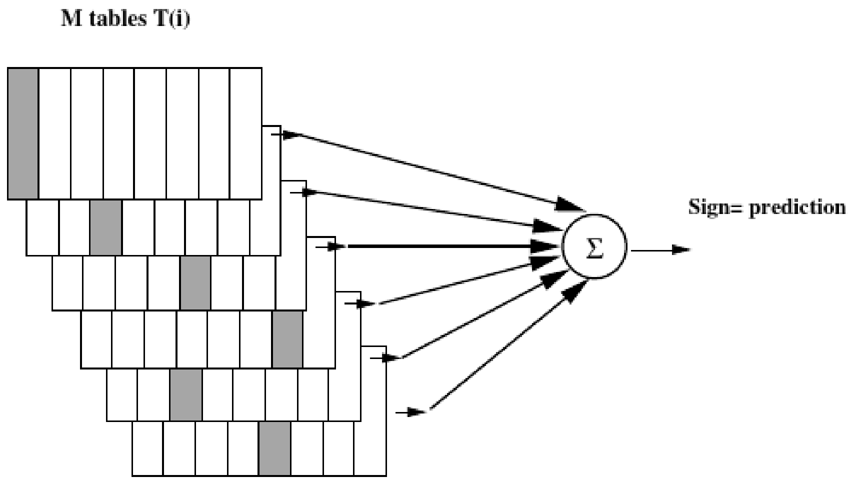


Figure 2.4: Illustration of the *O-GEHL* predictor [Sez05].

(a) **Perceptron**

This is a machine learning based branch predictor [JL01]. The key idea is to *learn the correlation* between branch *history* and branch *outcome*, assigning *weights* to correlations and compute prediction based on the history input. We can see the *perceptron* as a *binary classifier* (in our case) which is able to learn from a linear function, as presented in Equation 2.1.

$$y = w_0 + \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

Where y the binary outcome of the branch, w_0 the bias between how outcome is dependent on branch history, x_i an element of the pattern history input vector, and w_i the weight associated with this last element.

A *training function* is used to learn the weight that reproduces as faithfully as possible the linear relation, if it exists, between the history pattern and the branch outcome.

(b) **O-GEHL (Optimized GEometric History Length)**

This predictor [Sez05] uses *different tables* – between 4 and 12 – with *different history lengths*. This allows capturing both recent and old correlations only for branches that need it.

The prediction in each table is stored as a *signed saturated counter*. The final prediction is the sign of the *sum* of all the prediction for each table, just as the perceptron predictor previously seen. The tables are indexed by the *hash* of the branch *history* and the branch *address*. The length of a branch history $L(i)$ is computed as a *geometric series* approximation, as shown by Equation 2.2. For example, with $\alpha = 2$ and $L(1) = 2$, we obtain the following series: $L = \{0, 2, 4, 8, 16, 32, 64, 128\}$. Figure 2.4 gives a visual representation of the general algorithm.

$$L(i) = \alpha^{i-1} L(1) \tag{2.2}$$

(c) **TAGE (TAgged GEometric)**

The key observation of this predictor, like the previous *O-GEHL*, is that 1) different branches require different history lengths to have the better prediction accuracy and 2) that an adder tree is a very effective approach [Sez11] [SM06]. From these observations, we want to have multiple *PHTs* indexed with *GHRs* having *different history lengths*, and intelligently allocate *PHT* entries to different branches. Moreover, this predictor relies on *partially tagged* components, meaning that used tags – to identify branches – are composed of a limited

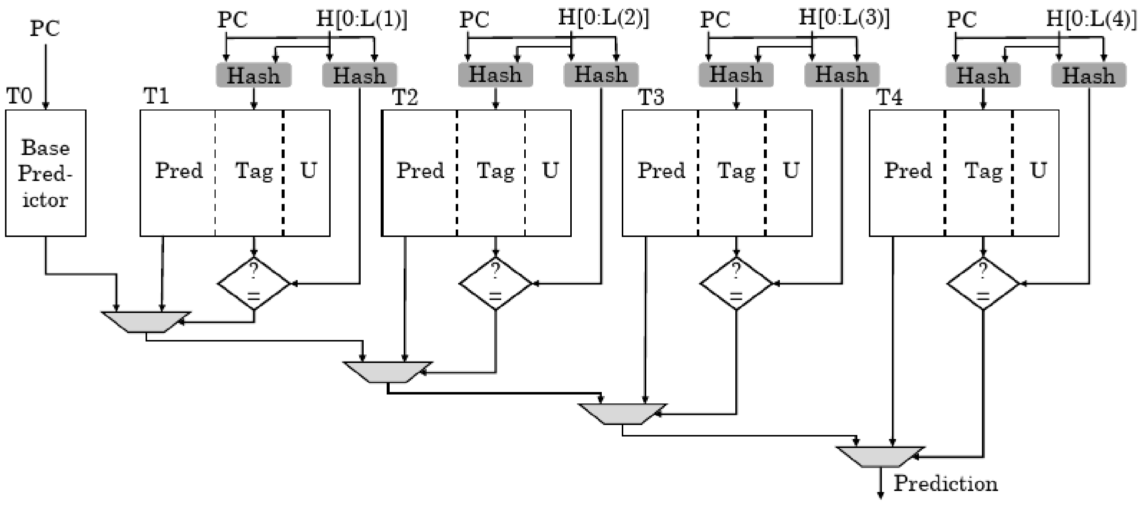


Figure 2.5: Representation of the *TAGE* predictor [Mit16b].

number of bits which is a trade-off between a waste of storage and the number of collision between multiple branches.

Figure 2.5 gives an overview of the predictor. In this picture, we can see a 5-component TAGE predictor. The *base predictor* can be, *e.g.*, a simple PC-indexed 2-bit counter (*bi-modal*) table. This one is backed with several *tagged predictor*, which are indexed by different history with an *increasing length* – which form a geometric series.

More formally, we consider $T0$ as the *base predictor* and *multiple partially-tagged components* denoted Ti with $(1 \leq i \leq M)$, and M the number of components $(4 \leq M \leq 12)$. A Ti predictor is indexed with a global *history length* evolving as an approximation of a geometric series, described in Equation 2.3. A Ti predictor is composed of a *signed prediction counter*, a *tag*, and a *useful counter* (denoted as U).

$$L(i) = \lfloor(\alpha^{i-1} * L(1) + 0.5)\rfloor \tag{2.3}$$

The *prediction process* is the following. All the predictors – base and tagged – are accessed in parallel. For all *tagged predictors* with a *hit* access – the hash matches the tag –, the prediction of the *one with the longest history* is used as the final prediction. Otherwise, the prediction of the base predictor is used.

The *update* process is the following. The predictions from predictors that produce a hit while being not used as the final prediction are called *alternate predictions*. When an *alternate prediction* is different from a *final prediction*, the U *counter* of the *final prediction* is increased or decreased by 1 if the *final prediction* was correct or incorrect, respectively. For the *pred counter*, it is incremented or decremented for the *final prediction* regarding the correct outcome. Finally, when an incorrect prediction happened on Ti with $i < M$ – in other words, which have not the longest history –, then a *new entry* is allocated in a predictor Tk with $i < k < M$.

All the steps described above are synthesis, there is a lot of details both on the algorithmic and implementation side in the papers.

(d) **TAGE-SC-L**

This predictor is basically a *TAGE* predictor with a *statistical corrector* (SC) and a *loop predictor* (L) [Sez14] [Sez16]. The *SC* is used to predict a class of statistically biased branches, *e.g.*, branches that are not correlated with the branch history pattern but have a bias towards a direction.

Structures. Now that we know all the major different algorithms, we will present some structures used in different branch prediction schemes.

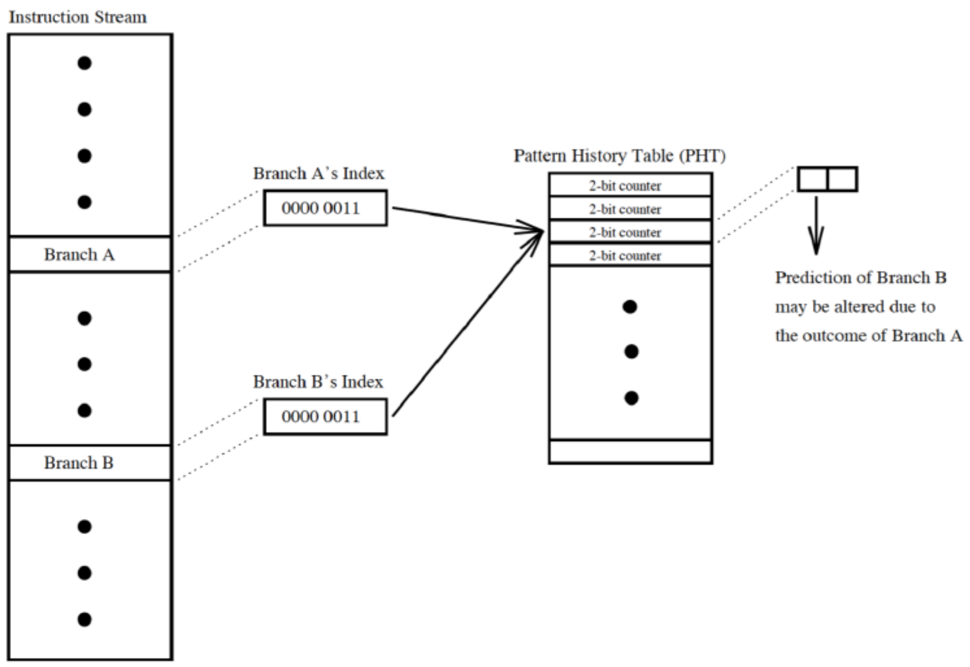


Figure 2.6: Possible implementation of a *PHT* [Mut18].

1. **Branch Target Buffer (BTB)**

   This *buffer* (a small associative memory, also called *Branch Target Address Cache*) is used by the *branch target predictor* [LS83]. In is most basic implementation, it stores pairs composed of a source address – a branch, which corresponds to the PC when fetched – and a destination/target address – where branch redirected the control flow.

   The key idea is to store target addresses that remain the same. For a conditional direct branch, this is true across dynamic instances but with 2 possibilities – the first being the next sequential instruction, and the other is the one stored in the *BTB*. We note that we have 100% success rate for static branches – unconditional, call, return –, since these target addresses are fixed.

   The *BTB* can resolve 2 of the 3 interrogations to be predicted:

   (a) Of course, it resolves the way to find the target address (3), being its main purpose ;

   (b) It resolves also the question about the nature of the instruction (1). In fact, if the PC value has an entry in the *BTB*, then it points to a branch instruction ; if there is none, then it points to probably not a branch.

2. **Pattern History Table (PHT)**

   This table is an array, indexed by a bit sequence identifying a scenario – a *history pattern* –, where each entry stores the prediction of the outcome for the considered branch.

   Concretely, one possible implementation is represented in Figure 2.6, where we can also observe an interference between two branches. In this picture, we have two different branches in our instruction stream. This *PHT* is typically used when using a *two-level predictor*, with the first-level – called index in this image – being a history pattern and the second being a saturating counter stored inside the *PHT*. This figure also illustrates one problem with this approach, called *branch interference* – or *aliasing* –, caused by two different branches having the same history pattern, then reading and modifying – wrongly – the same *PHT* entry.

   In the context of a *single-level predictor*, the table that stores the counters is called *Branch History Table (BHT)* – might be stored directly into the *BTB*. This is exactly the same structure, but it is indexed by an address of a branch and not by a history pattern.

3. **Return Address Stack (RAS)**

   This is a small buffer organized as a stack. For each encountered call, the return address – which is the value of the PC summed to the size of the instruction – is pushed onto the Return Address Stack (RAS). Then, the return address can be popped from the RAS for each return instruction encountered. It lowers the latency to start fetching the next instruction after a return branch.

4. **Trace Cache**

   This is an additional *instruction cache* [RBS96] which tries to maximize the chances that the next sequential instruction is the next instruction to be executed, after the predictor answered to the question "{"which instruction should we fetch next?}, particularly in superscalar processors where multiple instructions are fetched in one cycle. It has the same types of characteristics as a typical cache – associativity, indexing, hit/miss latency, and so on.

   The goal of this cache is to store *execution traces* of previously decoded/executed instructions, allowing inferring which instructions should be fetched next based on the previous traces for one instruction. It allows making appear contiguous instructions that are otherwise non-contiguous – *i.e.*, spreads into multiple basic blocks –, finally increasing the instruction fetch bandwidth and lowering the latency by directly providing instructions to the decoder.

   If there is a hit in the *trace cache*, the instruction cache is bypassed. A hit on this trace cache has two requirements:

   (a) The fetch address matches a trace tag in the trace cache ;

   (b) The branch prediction matches the branch flag.

   Since it does not influence the outcome of the branch predictor and exists "just" for the speedup, it is not directly related to the kind of attacks we studied. Nonetheless, knowing what is this structure and its impact could be useful.

2.1.3 *ARM*

ARM is a brand which develops an *ISA* under the same name. The latter fall under the *RISC* characterization of an *ISA*, which implies a more simple instruction set than a *CISC ISA*, both of in terms of implementation's complexity and instruction operation's complexity. Today's *ARM* processors are mainly used in embedded systems and mobile phones, because they have a high energy efficiency. However, recently, *ARM* processors started to be competitive in the high-performance computing domain. *ARM* develops the design of their processors, however, the photolithography process – the operation of applying an image to a substrate, here the silicon – is left for founders companies, like *Broadcom* for our processor.

A lot of families and versions of the architecture exist. The architecture goes from the version 1 – *ARMv1* – to the current version 8. For each version since the 6th, the architecture is divided into three profiles: 1) *A* for application, optimized for general applications 2) *R* for real-time, built for safety-critical environment 3) *M* for micro-controller, designed for embedded systems.

We will concentrate ourselves, in this report, on the *ARMv8-A* architecture. Introduced in 2011, this one brings the 64-bit to the *ARM* architecture, called *AArch64*, with the new 64-bit instruction set called *A64*. A complete presentation of the *ARM* instruction set or architecture would be out of scope of this report. Nonetheless, we will present some assembly snippets required to understand the kind of implementation needed by our attacks.

ARM Assembly. When performing cache attacks, fine-grained control operations are needed to control the cache, the memory, and having an accurate time measure. In *ARM*, there is not an easy `rdtsc` instruction like in *x86* to measure the elapsed cycles from a certain point. Workarounds are described later in the report. Below are detailed the operations that we need to do in order to perform an attack [20]:

Table 2.2: General information about the *ARM Cortex-A72* processor and the *Raspberry Pi v4* main memory.

Component	Parameter	Value
Processor	Cores	4
	Frequency	1.5GHz
	Voltage	1.2V
Memory	Size	4GB
	Type	SDRAM
	Standard	LPDDR4
	Frequency	2400MHz
	Voltage	1.1V

Flush an address from the cache We can use the `DC CIVAC` instruction to clean and invalidate the data cache by a specified address to the point of coherency [2]. After flushing an instruction, we have to imperatively use – in this order – a memory barrier and an instruction barrier. It acts as an ordering instruction which ensures that when the following instructions will be really executed, the cache flush ended correctly. Not doing that is a common error when working with the micro-architecture, where the hardware optimizations can play tricks on us.

Instruction barrier The `ISB` instruction ensures that all instructions that comes after the function in program order are fetched from the cache or memory after the function has completed.

Memory barrier The `DSB SY` instruction ensures that memory accesses that occur before the function have completed before the completion of the function.

By combining these 3 operations in a various ways, we will be able to implement our micro-architectural attack on our *ARM* processor.

ARM Cortex-A72. The *ARM Cortex-A72* that we got with the *Raspberry Pi* was extensively studied through the *ARM* manual [16] during the internship. Until the end of this chapter, we will detail the microarchitecture of our processor in order to understand what we would like to model with *gem5* and what is important in our kind of attack.

Table 2.2 gives some very high-level information about our processor and the main memory of the *Raspberry Pi*. The latter is the only exception which is not contained in the processor, nonetheless it has to be modeled with *gem5* too. The values of this table are supposed to be self-explanatory.

Figure 2.7 gives a large overview of the processor's architecture. We can observe the "Miscellaneous" block on this scheme, which is used by the hardware and processor's developers, that can be modeled thank to the "platform" system of *gem5* – which will be described later. We discover the level 2 memory system, which handles the communications and their coherency between different cores. This part could be precisely controlled and protocols could be accurately modeled in the simulator, however it will not be deeply studied in this report. Since the level 2 memory system is quite common on a large class of processor, it is not so interesting. The most interesting part here is the core architecture. We can directly see how the branch predictor, at the left, is in interaction with the instruction cache and hence, the fetch stage of the pipeline. This is a crucial detail to have a successful attack. We can also observe some optimization stages, like the "rename" block which handles register renaming.

Pipeline. Figure 2.8 gives an overview of the processor's pipeline, while Figure 2.9 gives a detailed view of all stages' components. Instructions are first fetched, then decoded into internal micro-operations. From there, the micro-operations proceed through register renaming and dispatch stages.

[2]The point at which all processors that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory.

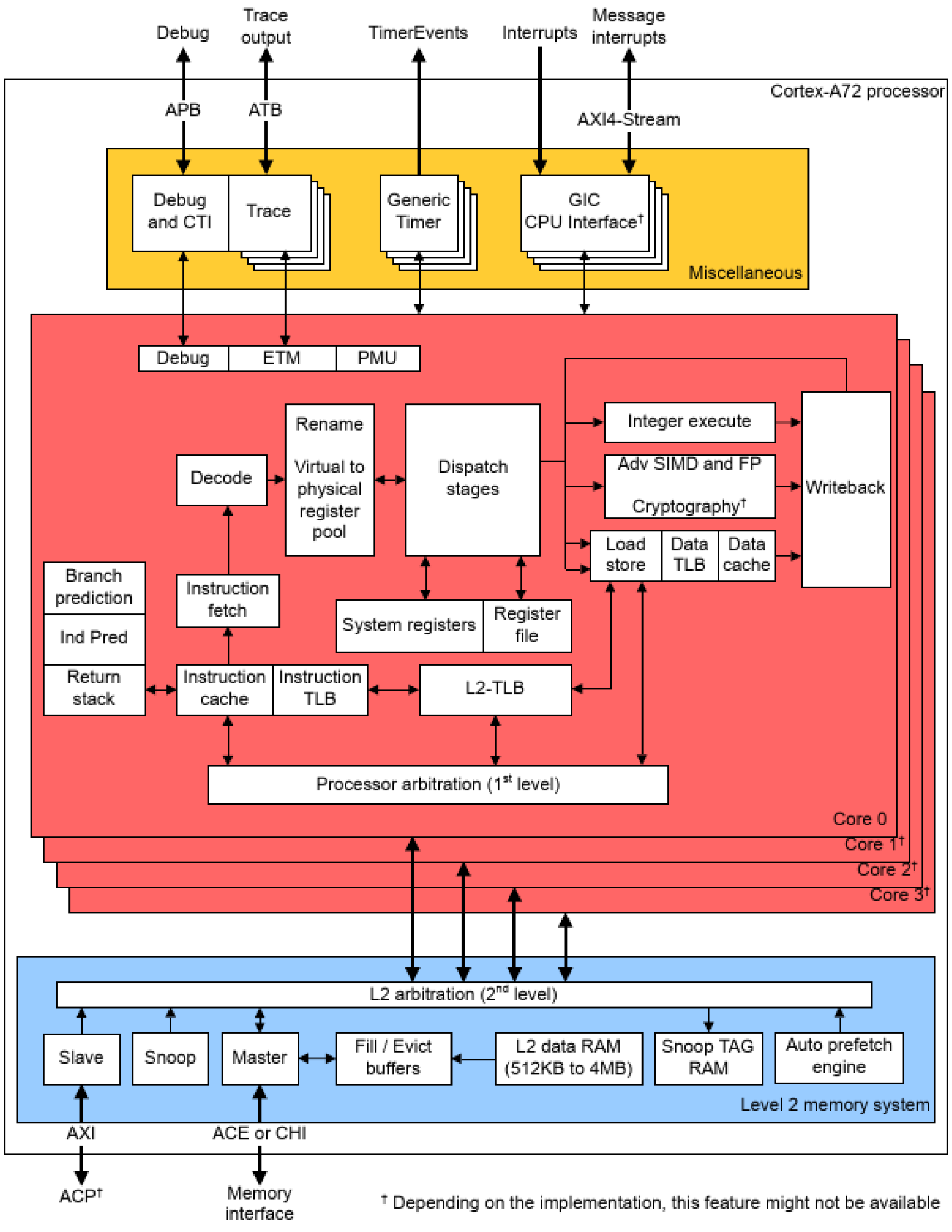


Figure 2.7: Overview of the *ARM Cortex-A72* architecture [16].

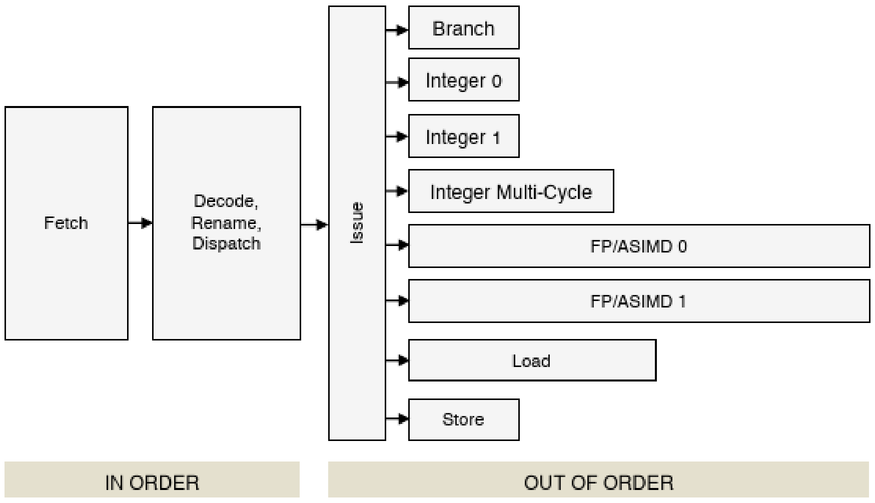


Figure 2.8: Overview of the *ARM Cortex-A72* pipeline [16].

Once dispatched, micro-operations waits for their operands and issue out-of-order to one of eight execution pipelines. Each execution pipeline can accept and complete one micro-operation per cycle. The detailed view gives interesting information that will be, in the future, reported in our simulator. Size entries are given for the caches and the branch predictor's components. We observe how the *decode* stage is able to communicate with the branch predictor, indeed, when a branch is decoded, the decode unit instruct the branch predictor to predict its outcome. Then, the branch predictor's prediction will be issued to the instruction cache, in order to load the predicted instructions into the fetch stage.

Our processor core architecture and pipeline fulfilled the following properties:

Superscalar Thank to the issue unit of the pipeline, its queues and the parallelism of the execution units, at most 8 micro-operations could be executed in one cycle.

Variable-length Regarding the execution unit and the nature of the executed instruction, the number of stages can be variable – hence, the latency of the execution of one instruction can be variable.

Out-of-order Thank to the combination of the dispatch, issue, write-back and the retirement buffer units, instructions could be re-ordered at runtime and executed out-of-order if they are not inter-dependent.

Branch Predictor (BP). We will detail the architecture and the strategy of the branch predictor of this processor, since it is a critical component for our attack – indeed, it is the one that is attacked. This branch predictor is a *two-level global history-based* one, with some *side-predictors* components. All needed background has been seen in Section 2.1.2.

As a general rule, the branch prediction hardware predicts all branch instructions regardless of the addressing mode, including conditional branches, unconditional branches, indirect branches, and branches that switch between *ARM* and Thumb states. Below, a detailed description of the components of our branch predictor:

1. **Dynamic Predictor**

   The *two-level dynamic predictor* is composed of a first stage, which is called *Global History Buffer (GHB)*. Basically, it is a *GHR* which holds a *global history pattern* of all branches. Since it is a processor destined to hardware constrained in term of resources, it makes sense to use only the global paradigm which is less expensive. The *BTB* identifies branches and provides targets for direct branches, and is tagged by all the memory space information [3] to be unique in a virtual

[3]Virtual Address (VA), Address Space Identifier (ASID), Virtual Machine Identifier (VMID), Security, Exception Level (EL).

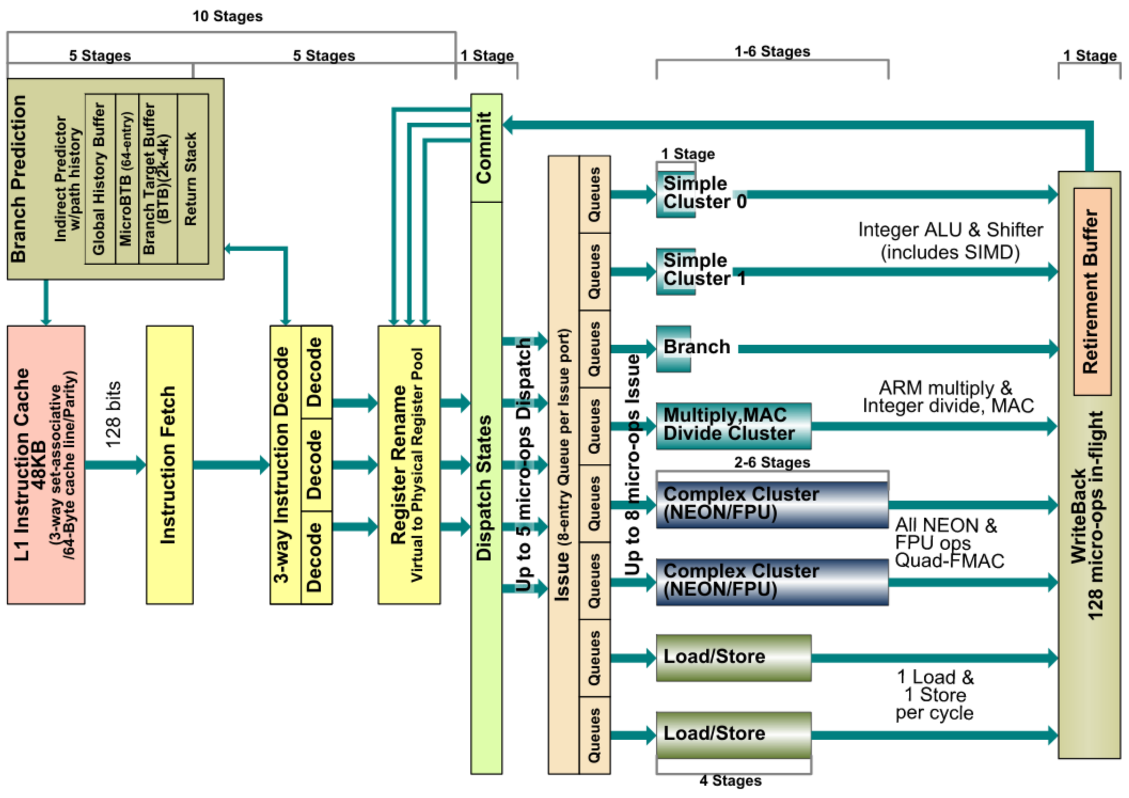

10 Stages
5 Stages
5 Stages
1 Stage
1-6 Stages
1 Stage
Branch Prediction
Indirect Predictor w/path history
Global History Buffer
MicroBTB (64-entry)
Branch Target Buffer (BTB)(2k-4k)
Return Stack
L1 Instruction Cache 48KB (3-way set-associative /64-Byte cache line/Parity)
128 bits
Instruction Fetch
3-way Instruction Decode
Decode
Decode
Decode
Register Rename Virtual to Physical Register Pool
Commit
Dispatch States
Up to 5 micro-ops Dispatch
Issue (8-entry Queue per Issue port)
Queues
Up to 8 micro-ops Issue
1 Stage
Simple Cluster 0
Simple Cluster 1
Integer ALU & Shifter (includes SIMD)
Branch
Multiply,MAC Divide Cluster
ARM multiply & Integer divide, MAC
2-6 Stages
Complex Cluster (NEON/FPU)
Complex Cluster (NEON/FPU)
All NEON & FPU ops Quad-FMAC
Load/Store
Load/Store
1 Load & 1 Store per cycle
4 Stages
Retirement Buffer
WriteBack 128 micro-ops in-flight

Figure 2.9: Details of the *ARM Cortex-A72* pipeline [16].

Table 2.3: Static predictor strategy.

Branches types	Instructions	Action
Direct unconditional	`B immediate`	Predicted *taken*
Direct unconditional call-type	`BL{,X} immediate`	1) Predicted *taken*
		2) Preferred return address is pushed on the RAS
Unconditional return-type	`ret`	1) Predicted *taken*
		2) Target is popped from the RAS

memory address space. This property could make it more difficult to abuse it. It is composed of 2048 to 4096 entries, while there is also a faster but smaller *micro-BTB* of 64 entries. Finally, there is not information about the *PHT* used in this processor. The only information that we know about it is that it is either global or per set of branches, thank to our experiments [4].

In the manual, we can read “All predictions are checked at branch resolution time to ensure that a legal branch is resolved.”. This claim said that, at an architectural level, an illegal branch will never be executed. However, in the micro-architectural domain, this is not true. This is the core of the *Spectre* attack described in Section 2.3.2.

2. **Static Predictor**

   This predictor is useful to accelerate cold startup of code, since branches must be resolved at least one time to be predicted by the *dynamic predictor*. The strategy of the static predictor is described into Table 2.3, which could be implemented in the simulator, but it should not have a lot of impact on our attack.

3. **Indirect Predictor**

[4]In fact, a *Spectre-PHT-{SA,CA}-OOP* is able to mistrained the branch predictor, which imply that the same *PHT* entry was used to predict more than one branch during the attack.

Table 2.4: Configuration of the instruction and data caches.

	Level 1 Instruction (*L1I*)	Level 1 Data (*L1D*)	Level 2 (*L2*)
Accessibility	Private	Private	Shared
Granularity	Banked	Banked	Unified
Total Size	48K	32K	1MB
Cache Line Size	64B	64B	64B
Mapping	3-way set associative	2-way set associative	16-way set associative
Replacement policy	LRU	LRU	Pseudo-Random
Writing policy			Write-back
Indexing	PIPT	PIPT	PIPT
Prefetching	Yes (1)	Yes (2)	Yes (2)
Inclusion property			Inclusive with *L1D* (3)

(1) Speculative fetch by the 1st level of processor arbitration, with no guarantee of execution.
(2) Prefeteched by the load/store unit.
(3) And duplicate L1D tag RAM for handling snoop requests.

This predictor predicts indirect branches that are not *return-type* instructions. It augments each branch address with an additional state that predicts the target address. For conditional indirect branches, the additional state is only used for branches that are predicted *taken* by the conditional branch predictor.

4. **Return Stack**

   This predictor stores the address and the *ARM* state of the instruction after a *function-call* type branch instruction. The address is equal to the *Link Register* value stored in `X30`. An instruction causing a return stack push could be either `BL <immediate>` or `BLR <register>`.

Hardware Prefetcher. The hardware prefetcher aims at loading data in the cache memory ahead of their request by a load instruction. The goal is to hide the main memory latency. This component is important for micro-architectural covert-channel because it could be the source of measurements error. Thus, it has to be carefully considered.

This one targets the *L1D* and *L2* caches for both load and store accesses. It handles prefetch generation for instruction fetch and *TBW* descriptor accesses. However, it is limited to prefetch within the *4KB* page of current request. This limit is particularly important to verify when we perform an attack on the cache memory.

Caches. In this subsection, we will cover the architecture of the caches used by our processor. A few parameters are critical for the attack, however, they are globally important in term of accuracy for the simulated system – since the cache hierarchy is one of the most important improvement for processor performance. Table 2.4 gives the hardware configuration of our caches, both for level 1 (*L1D* and *L1I*) (data and instruction) and level 2 (*L2*). From an attacker point of view, two parameters are really important here. The first one is the accessibility, since a shared cache – here, the *L2* – enable cross-cores attacks. The second one is the indexing method, which influence heavily what an attacker would be able to do with a virtual address. The following points described cache component or related unit.

1. **Level 2 (*L2*)**

   Since the second level of cache memory is shared between cores, it is often more complex than the first level. The architecture of this level is represented in Figure 2.10. From this diagram, we see that the *tag array* – which contains a collection of *tags*, where a *tag* specifies the identity of the data in the main memory of which the entry is a copy – is divided into multiple *tag banks* to enable up to two simultaneous requests. Then, *tag bank* are divided into multiple *data banks*

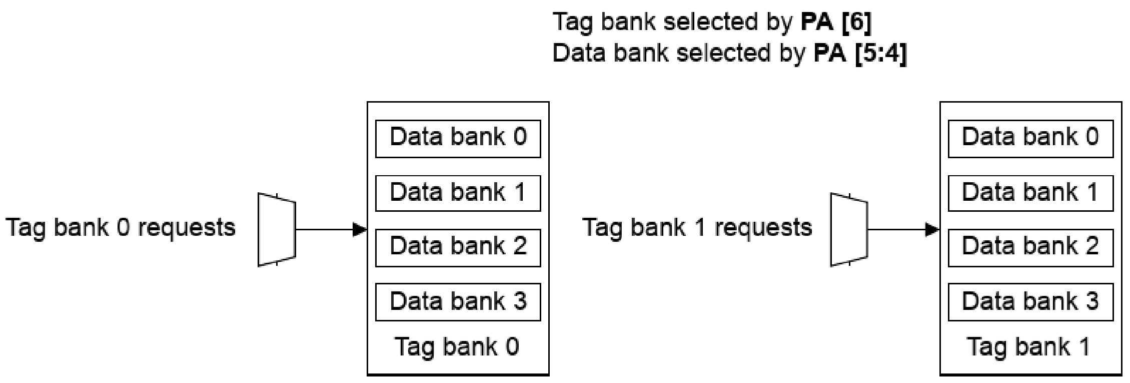


Figure 2.10: Architecture of the *L2* cache banks [16].

Table 2.5: Configuration of the *TLB*.

	L1I micro-*TLB*	*L1D* micro-*TLB*	*l2* main-*TLB*
Accessibility	Private	Private	Private
Granularity	Banked	Banked	Unified
Entries	48	32	1024
Mapping	Fully associative	Fully associative	4-way set associative
Page Sizes	4KB, 64KB, 1MB	4KB, 64KB, 1MB	4KB, 64KB, 1MB, 16MB
Hit latency	1 cycle	1 cycle	Variable number of cycles

(*i.e.*, 4 data banks) to allow streaming access – which means that the memory can serve the data with a constant bit rate above a certain threshold. For an accurate simulation in term of performance, this architecture has to be modeled in the simulator.

2. **Snoop Control Unit (SCU)**

   The *SCU* maintains coherency between the individual data caches in the processor. It contains buffers that can handle direct cache-to-cache transfers between cores without having to read or write any data to the external memory system. It uses hybrid *Modified Exclusive Shared Invalid (MESI)* and *Modified Owned Exclusive Shared Invalid (MOESI)* protocols to maintain coherency between the individual *L1* data cache and the *L2* cache.

MMU. The *MMU* is the processor's component charged of translating virtual memory addresses to physical addresses. Moreover, this unit performs some security and coherency check. The *MMU* holds the following components:

1. **Translation Lookaside Buffer (TLB)**

   The *Translation Lookaside Buffer (TLB)* is a component which caches recently used translations. This is the first looked place before to perform a virtual to physical address translation. It contains a global indicator, an Address Space Identifier (ASID), in order to permit context switches without *TLB* flushes. Its hardware configuration is given in Table 2.5.

2. **Table Walk Cache (TBW Cache)**

   The *table walk unit*, which contains logic that reads the *translation tables* from memory, performs memory accesses. Like regular memory accesses, they can be cached to speed-up future access. This is the role of the *table walk cache* [BCR10]. It stores intermediate levels of translation table entries required during a table walk, issued after a miss in the *TLB*. Its hardware configuration is given in the table 2.6 [Sch+17].

Table 2.6: Configuration of the Table Walk Caches.

Parameter	Value
Type	Translation cache
Entries	64
Mapping	4-way set associative
Granularity	Unified

2.2 Simulation

Simulation corresponds to the process of running a software on a computer in order to reproduce a phenomenon. The computers used are generally super-computers with thousands of cores, but a simple simulation can also be run on a personal computer. In this report, we will use the term simulation in a very precise way, explained later. This chapter is separated in two parts. The first one gives a little background on what is computer simulation, and the kind of simulation we use. The second one is concentrated on the simulator we used during our internship, *gem5*.

2.2.1 Background

The idea of using computers to simulate real-world events is almost as old as computers themselves. We can mention the simulation of munition trajectories during the *Second World War*, including the atomic bomb. However, the democratization and the massive usage of simulation is far more recent. Indeed, a simulation often required a large amount of hardware resources – in terms of computational power and memory space – to be accurate enough so that it can be useful for complex system. Almost any physical event can be simulated, from galaxy movements to molecular interactions, through meteorological system evolution. These are the kind of simulation used in high-performance computing.

In our field of research, the system that we simulate is a computer itself – we simulate computers with computers. Simulation should not be confused with virtualization. While virtualization – in his most popular form – aims at creating a virtual machine which is a complete virtual hardware environment destined to run another operating system in it, the use case is to run a software inside it for some reason – isolation, compatibility, etc. In our case, we do not want to run a software in a virtual machine to use it, but run a software on a simulated computer as accurately as possible in terms of micro-architectural behavior. This is what we could call micro-architectural simulation.

Usage. Now that we understand the kind of simulation we studied, we will illustrate how it can be useful. How the precise simulation of the micro-architecture of a processor – or an entire system – could be interesting? Hardware research and development is difficult and very expensive. Unlike software engineering, to evaluate a hardware system, you have to implement it in the real world, which can be quite costly. This cost could be partially eliminated by simulate a new system instead of implementing it. This system we talk about could be, for example, a performance optimization mechanism or an energy saver functionality. These are common usages of micro-architectural simulation. The latter could also be used to simulate micro-architectural attacks and defenses, which is a quite recent practice. Indeed, there are very few papers that fall into the security category.

Constraints. Micro-architectural simulation is not an ideal system: it comes with important constraints. When talking about computer simulation, we can distinguish three simulation levels:

Functional The simulation is concentrated on the result of one function. If the real function and the simulated function have the same result, no matter the method used, then the simulation is accurate.

Timing Here, we concentrate the simulation on the time taken for one functionality. If the real and simulated functionalities have the same execution time – the time could be relative to the simulator and be different from the real time –, then the functionality is accurate.

Cycle-accuracy This is the more complex simulation, where an accurate functionality has to take exactly – in a perfect simulation – the same number of cycle of the real functionality. The consequence is that all the micro-architectural logic has to be reproduced – approximated – in the simulator, which is a very complex task while today's processors are composed of billions of transistors.

Another constraint is the time taken by a simulation. Indeed, as they are very complex, simulate a simple function can sometimes take hours. Consequently, a good workflow for the experiments has to be set up.

2.2.2 A Cycle-Accurate Simulator: *gem5*

gem5 is a state-of-the-art cycle-accurate computer simulator [Bin+11] [Low+20]. With a lot of publications into conferences or scientific papers, it is used both in academic research and in industry for research and development of new hardware technologies [Lowe]. *gem5* is a fusion between two previous simulators in 2011: *M5* and *GEMS*. The governance of the project follows an open-source model, where a proportion of the users are also contributors. The project is hosted by *Google*. As a state-of-the-art project, it can be difficult to obtain some help. *gem5* has an official documentation [Lowb] and an old one [Lowc]. Moreover, a user can ask for help or report a bug on the official mailing list [Lowd] or in the *StackOverflow* section for *gem5*.

This simulator has a plentiful number of functionalities, like taking a snapshot of the system and restoring it later. It can run a simulation into multiple modes, regarding the needs of the user, to find the best trade off between accuracy and taken time. A large number of architectures is supported by the simulator, including *x86* and *ARM*. Numerous kinds of processors are available, in order to simulate a variety of systems, from embedded devices to high-performance computers.

Architecture. *gem5* comes with his sources, but also with some external helper programs, configuration examples, and bootloaders. The simulator is made of two parts: the *C++* core and the *Python* interface. The *C++* core is where the logic of the components is programmed. Each interaction, computation, and transmission is modeled in *C++*. For the *Python* interface, it is linked to the *C++* objects in order to interconnect these elements in a simple way. Each *C++* object ends up in a *Python* equivalent object, and the values of *Python* attributes in a system script are passed to the *C++* objects during the instantiation of the system. To summarize, we have two sides of *gem5*: the first one is where objects' logic are defined in *C++*, the second one is where objects' ports are tied up to connect them, in *Python*. Indeed, each object has "ports" to send or receive packets: this is the main connection paradigm of *gem5* objects.

Modeling. To model a system, we use the simple syntax and the Object Oriented Programming (OOP) paradigm of *Python* to connect components with the others and to configure them, to finally end up with our desired system. It can be very easy and quick for a simple system, but some objects are deeply customizable – latencies, number of registers, and others small micro-architectural properties.

Every object in *gem5* inherit from the `SimObject` class. Then, a deep hierarchy of object is built on top of the `SimObject` interface. The hierarchy of processors will be described and tested in Section 3.3. Memory declines in multiple flavors, from the main memory to the smallest caches. Buses are used to connect memories and computational units. For the caches, two systems exists. The first one is called "classical caches" in *gem5* terminology, where real-world coherency protocols are approximated with simpler heuristics. This one is much faster than the second system, however it is less accurate. The second one is the *Ruby* system, which allows modeling caches and their protocols in a very precise way by specifying the protocols directly with the *Ruby* language. Since this system is much slower, and we do not need this accuracy in our research, we will not cover it in this report. Finally, tiny components as controllers or functional units of the processor can also be simulated, even created directly in *C++*.

Several modes exists in *gem5*. The first choice is the memory mode:

Atomic In this mode, every memory request is served instantaneously. There is no contention, no delay or latencies. Obviously, this mode is not suited for real experiments. The intended use case is for debugging or testing.

Timing In this one, timing measurements can be performed. The latencies and contentions are simulated. The accuracy of the memory subsystem will be determined by how your memory components, mainly caches, will be configured.

Another choice, this one being more complex, arrives when you want to simulate a system:

System-call Emulation (SE) In this mode, we only gives a binary to *gem5* and the system executes this binary only. Every system call made by the binary is intercepted by *gem5* and emulated by his core. If a system call is not implemented, it could simply ignore it in the best scenario, or crash the system in the worst scenario. This mode is faster than the second one, because only one process is executed. However, it is less representative of what happened in a real system with an operating system running. Nonetheless, it is a good first step when you want to model an architecture or perform a quick experiment, but when it comes to high accuracy, the second mode is well suited.

Full-system Simulation (FS) This mode intends to run a full and real operating system directly on *gem5*. The system calls no longer need to be implemented by *gem5*, the operating system takes care of their execution. Only the instruction set has to be implemented into *gem5*. To use this mode, we give to *gem5* a kernel image – *e.g.*, *Linux* – and a disk image. The disk image must contain a complete operating system – *e.g.*, *Ubuntu* –, with our binary or binaries to run.

Bare metal This mode is used to run directly assembly code on a complete system. It is intended to model bare metal systems used in real-life, sometimes in critical embedded systems. In this report, we will not investigate this mode since the attack we studied does not target this kind of system, at least in our implementation.

FS Modeling. Modeling a system in FS mode is more complex than in SE mode. In fact, an SE system can be viewed as a subset of an FS system. We will not detail all tiny components that need to be configured to run a FS mode simulation. However, we can briefly describe the platform system of *gem5*. This is an object used by other platforms as a parent, while each child being a platform for one specific architecture. Every architecture can be simulated by a few platforms representing them. A platform is modeled by *gem5*'s developers, relying on real hardware's specification. For example, for the ARM architecture, we can use the *RealView* platform, based on *ARM Versatile Express* real boards. The platform aims at modeling some architectural components except the processor, *e.g.*, interrupts controller, memory mapping, input-output, and so on. We have not found any documentation about this system, we have had to understand it by reading the *gem5*'s source code.

2.3 Security

This chapter will be devoted to the application of the previous knowledge to security. Indeed, two kinds of attacks are able to exploits these hardware properties. We will describe them chronologically.

2.3.1 Side-Channel Attacks

The side channel attack idea takes its roots in a very classical scenario. Let's imagine the following: a criminal wants to steal what is inside a protected vault. To achieve this, he can turn the lock until finding the right position for a certain number of digit. However, there is too many possibilities to be achieved by testing all of them. Lucky he is, he pay attention to the fact that when the lock is moving on the right position for one digit, the sound emitted is not the same as the one when it is not on the right position. Thank to this information – a side-channel information about the locking system – he will be able to open the vault without knowing the key before.

From this imaged scenario, we are able to understand that a side channel attack is an attack against a security system using additional information, these originating from the interaction between the security system and its environment. This is why it is a “side-channel”, and not a “main-channel”: these additional information are leaked in an unattended way.

Background. The term side-channel attack originate from the cryptanalysis field, from a paper of Kocher et al. [Koc96] in 1996. This paper presented a completely new technique, and has opened a wide new field of research. This kind of attack is considered as important today, since it is not trivially resolved despite the years of research. Moreover, no matter the mathematical robustness of a cryptosystem, it could be potentially broken by a side-channel attack.

Since the discovery of side-channels, various ways of gaining information were discovered. We classify them as the following [ZF05]:

Passive This kind of side-channel attack is achieved by gaining information about the cryptosystem being computed on the attacked hardware without modification of the latter. This kind of attack is much more probable in real-life, and is a real threat to security systems. Note that this kind of attack is classified by the nature of the gained information.

> **Acoustic** Listening to the sound emitted – hence acoustic – by a piece of hardware could allow an attacker to break a cryptosystem. This corresponds to the illustration of the lock-picker and the vault, as the difference that the sound emitted by electrical components is not exploitable without recording and signal processing techniques, since it is in a domain like ultrasonic sound [GST17].
>
> **Electrical** There is no electrical component with a perfectly stable usage of electrical power. Moreover, electrical consumption is often dependent of the nature of executed computations. Hence, measuring the electrical consumption could give side-channel information [KJJ99].
>
> **Electromagnetic** In the same way as the electrical consumption, the electromagnetic emission of a piece of hardware is often correlated with the nature of executed computations. Receiving, measuring and processing them is a powerful vector of side-channels attack [Cam+18].
>
> **Timing** Timing information refers to the time taken for a given computation. In fact, depending on the value of some variables, the critical computation could take more or less time. If we can measure the time of the computation, then we could deduce the values of the variables [Koc96]. This is the kind of side-channel that interest us in the rest of this report, but not performed in the same way and against the same target as the original paper.

Invasive This kind of side-channel attack is achieved by modifying the hardware that is attacked. The modification is generally not trivial, and could be sometimes destructive. Consequently, this category is less probable to be usable in real-life. Note that this kind of attack is classified by the manner used to gain the information.

> **Probe** A probing attack corresponds to the use of specialized probes, inserted directly inside the attacked hardware to be able to gain information about the computations. While a probing attack is invasive, it is a passive attack.

Active This kind of side-channel attack is achieved by modifying actively the computations of the hardware. Note that this kind of attack is classified by the manner used to gain the information.

> **Fault** A fault attack consists of injecting a fault, *i.e.*, causing a computational error, during the computation that is attacked, *e.g.*, a cipher operation. Because of the injected fault, a cryptosystem like *DES* could leak some information, then used as a side-channel information [BS97]. The fault can be caused by an invasive or not invasive process.

A Cache Timing Attack: Flush+Reload. Now that we understand caches and timing side-channel attack, we can describe the *Flush+Reload* attack [YF14]. This attack is a *software-based microarchitectural side-channel attack*. Indeed, the method used to gain the information is only through a software program, hence it does not involve physical measurements. The target of this attack is the cache of our processor, thus a micro-architectural component. Finally, the attack uses a leaked timing information to gain knowledge on a secret, which could be a cipher key. Cache attacks were already known at the publication date of this attack. However, this one was very important in term of efficiency and feasibility compared to the others.

Explanation. The attack is based on four observations:

1. An assembly instruction – `clflush` on x86 – is available to evict a cache line based on a virtual address, and another instruction is available to precisely measure a time slice – `rdtsc` on x86. These instructions are widely used to perform non-security related operations.
2. The time taken to load a variable is clearly distinguishable between a cache hit and a cache miss. This is the result of the optimization of the memory hierarchy.
3. The LLC is often shared between cores and is inclusive with lower levels of caches. This is an architectural design choice for coherency between caches.
4. Some mechanisms – page sharing and coalescing – allows any process to share memory pages with another one. This is the result of an important memory space optimization.

These observations have the following implications:

1. We are able to know which address is cached or not cached based on timing information (consequence of 1 and 2) and able to evict an address from the cache (consequence of 1).
2. No matter the core running a process, if the latter flush an address from the cache, it will be flushed for all level of caches of all cores (consequence of 3).
3. A malicious process can share the memory containing the instructions of another attacked process. Indeed, the attacked process being stored on the disk, the malicious process is able to map the code segment of the attacked process in its own memory. Due to 4, the memory pages containing the instructions will be shared between the two processes.

From the implications, we have our attack scenario. An attacker on a machine, without any privilege, could create a process that will map an attacked process in its own memory. The attacker process, in the simpler form of the attack, has to locate interesting instructions of the victim process. Then, the attacker process will loop indefinitely over the following cycle:

- Flush the interesting instructions from the cache.
- Wait a certain period of time for victim execution.
- Measure the time taken to reload victim instructions from the memory.

By doing this, the attacker process will be able to determine which instructions the victim has executed during the period of time. Indeed, if we measure a fast reload, *i.e.*, a cache hit, it means that the victim has reloaded and – probably – executed these instructions during the period of time. This scenario allows a spying process trace the execution of another process.

Properties. The particularity of this attack is the following:

High resolution This attack is able to spy at the cache line granularity, often of 64 bytes. Indeed, the used instruction to flush an address from the cache flushes the entire cache line containing the data of the address.

Low noise This attack, when performed, does not suffer from a lot of noise, unlike some previous cache attacks. Consequently, this increase accuracy.

Cross-core Unlike previous attacks, this attack does not need to have the spying process on the same core that executes the attacked process. Consequently, this makes this attack much more usable in the real world.

Applications. In the original paper, the authors were able to recover a *GnuPG RSA* keys by spying at the *GnuPG* process during an encryption [YF14]. The attack has been performed as a *timing side-channel* attack against the *square-and-multiply* modular exponentiation algorithm. Various applications have been made since. Hornby successfully applied this attack to everyday application, *e.g.*, by spying the web-browsing of a user [Hor16]. Gruss et al. created an automated heuristic to attack applications without any prior knowledge, *e.g.*, being able to distinguish keystroke of the user, that means acting like a keylogger [GSM15].

Alternatives. Finally, note that there is a lot of cache attacks discovered, past and before *Flush+Reload*. Nonetheless, we can see that the publication of the latter resulted in a lot of new work – a revival – on cache side-channel attacks.

Prime+Probe [OST06] It occupies specific cache sets, then letting the victim program being scheduled, it determines which cache sets are still occupied or not – and thus, trace the victim process.

Evict+Time [OST06] It measures the execution time of the victim process, then evict a specific cache set by any technique, finally it measures the execution time of the victim program again. It determines if the victim executes the evicted portion or not.

Evict+Reload [GSM15] Like *Flush+Reload*, but instead of using the `clflush` instruction, the eviction is done by accessing physically congruent addresses.

Flush+Flush [Gru+16] Like *Flush+Reload*, but the attack only relies on the execution time of the flush instruction, which depends on whether the data is cached or not.

Collide+Probe [Lip+20b] It exploits µ-Tag – computed by the cache way predictor on *AMD* – collisions of addresses in order to spy the memory accesses of a victim process on the same logical core.

Load+Reload [Lip+20b] It exploits the predictor's behavior for aliased address – virtual addresses mapping to the same physical address – in *AMD* processors – to evict an address from the cache with a single load instruction.

2.3.2 Transient Execution Attacks

A microarchitectural attack is an attack which targets the micro-architecture of a component. Here, we will look at processor microarchitectural attack. Today's processors are incredibly complex and contains numerous mechanisms to optimize performance and energy consumption. Some attackers can successfully use their knowledge of microarchitectural components to attack software running on the processor. We have already talked about *cache attacks*, mainly with *Flush+Reload*. Other processor mechanisms are targeted and exploited, *e.g.*, DVFS-like mechanisms used to control the *voltage* of the processor [Mur+20] or *hyper-threading* and concurrent accesses to the TLB [Gra+18]. In this chapter, we will talk about a class of microarchitectural publicly unveiled in 2018: *transient execution attacks*.

Terminology. Before to dig into this kind of attacks, we have to define what unites them. As we know, to optimize performance, today's processors try to keep the pipeline fully loaded at all time. In order to do this, it can speculate on which instructions will be executed in the future, it can assert that no faults will be raised, and so on. When a speculation, an assertion or something goes wrong during the execution of one instruction inside the pipeline, the latter is flushed. Instructions that were inside just before the flush were partially or completely executed, but never committed to the architecture – they are *squashed*. A *transient instruction* refers to an instruction partially or completely executed by the processor which can affect its micro-architectural state, leaving its architectural state without any trace of its execution. An execution of a *transient instruction* – the latter not being *transient* by nature, but by its execution context – is called a *transient execution*. These attacks are not *side-channel* ones – despite this is often confounded on the Internet –, because they do not gain information through an accidental leakage about a system to break it. Instead, they use a particular knowledge to encode an information, *via* a transient execution, into the micro-architectural state of the processor. Then, the attacker will be able to recover the secret through the use of a *covert-channel*. A *covert-channel* is a channel of communication used by two agents – here, two processes – in order to communicate without being allowed to [Ge+16]. This kind of channel could be seen as a particular case of side-channel, where the attacker control both the sender and the receiver [Can+19]. Thus, techniques used in *side-channel attacks* on the *cache* – like *Flush+Reload* – can be used as *covert-channel* too.

Background. Since 2018, many transient attacks were discovered [Can+19]. They have received a large coverage by generalist and traditional media, which is quite rare. However, this has led to a lot of confusion in the terminology and taxonomy of these attacks – between other bad circumstances. During the first months, *Meltdown* and *Spectre* were names given to these vulnerabilities. Then, we realized that there were not isolated vulnerabilities, but a complete class of vulnerabilities [CKG20]. Before to detail the transient attack we are interested in, we will briefly present the families of transient execution attacks, as in Figure 2.11:

Meltdown [Lip+18] [Lip+20a] This kind of attack exploits transient execution after a faulting instruction, causing a pipeline flush. In fact, the vulnerability allows performing computations on a result from a faulting instruction, bypassing hardware protection. *Meltdown* could exploit both in-order and out-of-order pipelines. Nonetheless, here transient execution are allowed due to a processor bug, *i.e.*, an unexpected behavior.

Spectre [Koc+18] [Koc+20] This kind of attacks exploits transient execution after a misprediction from the processor concerning future executed instructions. In fact, the vulnerability allows bypassing software defined security. This class of vulnerability is particularly complex to fix, because the cause of the transient instruction is not a bug: it is precisely what the processor is supposed to do, in order to optimize performance.

Load-Value Injection (LVI) [Bul+20] Being very new, this attack is not on the figure presented above. While *Meltdown* and *Spectre* directly leak the data from the victim process, *LVI* inject attacker's data – through a transient execution – into the victim process in order to leak the targeted data.

A Speculative Execution Attack: Spectre. *Spectre* is an attack which exploits *speculative execution* to execute *instructions transiently*. Particularly, the branch predictor's architecture is exploited to make it executes unauthorized instructions transiently – except for the *v4 - STL* of *Spectre*, described below. The general scenario is the following. The unauthorized instructions will modify the micro-architectural state in a way that is dependent of the targeted value – the secret that we want to recover. When the transient execution has been done, the attacker will be able to probe the micro-architectural domain thank to the covert-channel, find the modification, and infer the targeted value. The covert-channel could be of any types, however, *Spectre* is frequently exploited with cache covert-channel for their efficiency, particularly *Flush+Reload*. This is the method which will be described.

Taxonomy. This is a brief presentation of *Spectre*-type attacks. Note that a lot of names refers to the same class, as explained above.

v1 - PHT (Pattern History Table) (Bounds Check Bypass) [Koc+18] This version of *Spectre* exploits the PHT, which is the structure used to predict the outcome of a conditional branch, as explained in Section 2.1.2. By mistraining the branch predictor, in this case targeting the BTB, *Spectre* will make the processor executes speculatively a branch which should not be executed, because of a malicious condition.

v2 - BTB (Branch Target Buffer) (Branch Target Injection) [Koc+18] This one exploits the BTB, which is used to predict the target address of a branch, as explained above. By mistraining the branch predictor, in this case targeting the BTB, *Spectre* will make the processor executes speculatively a piece of code maliciously designated as the target of a branch.

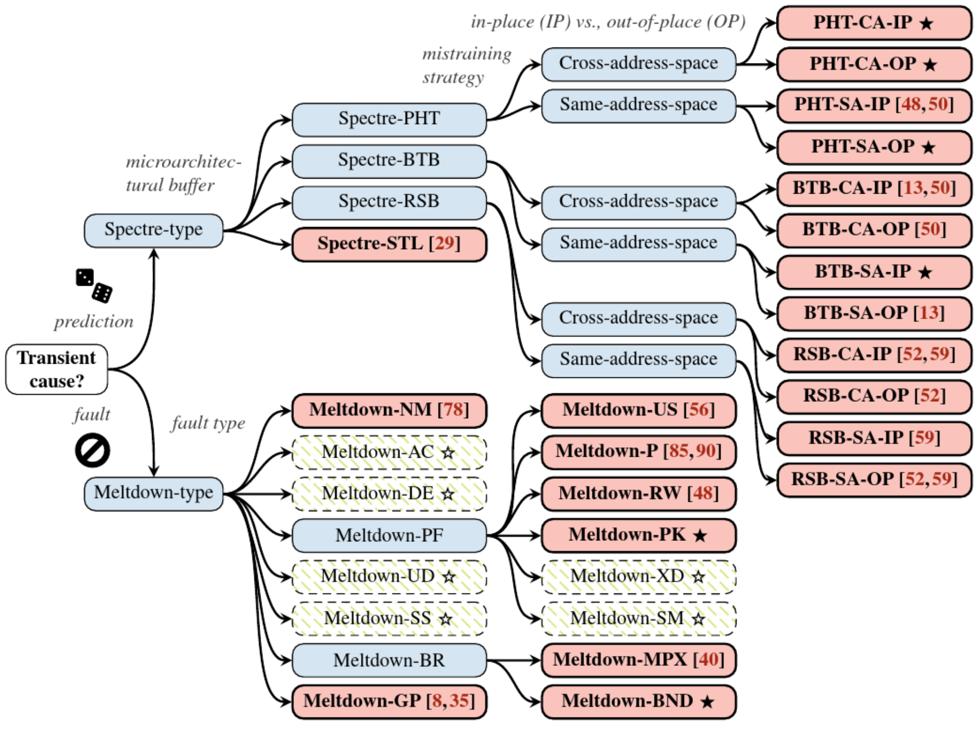
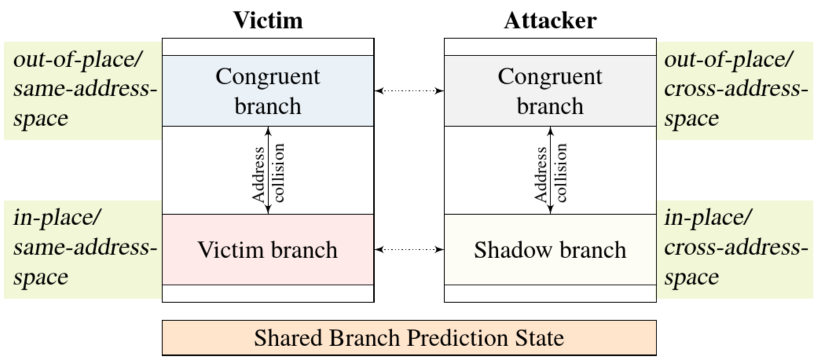


Figure 2.11: Taxonomy of *Transient Attacks* [Can+19].



Victim
Attacker
out-of-place/ same-address-space
in-place/ same-address-space
out-of-place/ cross-address-space
in-place/ cross-address-space
Congruent branch
Victim branch
Shadow branch
Address collision
Shared Branch Prediction State

Figure 2.12: Strategies for *Spectre* Training [Can+19].

v3 - RSB (Return Stack Buffer) (Return Address Injection) (ret2spec) [MR18] [Kor+18] It exploits the RSB, used to predict return address after a function. By mistraining the branch predictor, in this case the RSB, *Spectre* will make the processor executes speculatively a piece of code maliciously designated as the return of a function.

v4 - STL (Store To Load) (Speculative Store Bypass, SSB) [Hor18] It exploits *store-to-load forwarding*, an optimization mechanism that allow to forward a value in the store buffer – not already written to the memory, but waiting for it – to a load that uses the same address. This version of *Spectre* does not exploit *control flow speculative execution*, but *data flow speculation* instead, where the processor could speculate that a *load* instruction could be executed speculatively if there is none *store* on the same memory address.

Except for the *v4 - STL* version, all *Spectre* versions use a preparatory phase where the branch predictor is poisoned. Illustrated by Figure 2.12, this strategy declines in 4 flavors:

Same-Address Space (SA) In-Place (IP) In the same process of the victim, the attacker executes the victim branch with valid input to poison it.

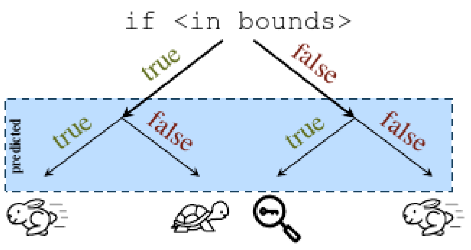


Figure 2.13: Scenarios of a *Spectre-PHT* [Koc+18].

Same-Address Space (SA) Out-of-Place (OOP) In the same process of the victim, the attacker executed a *congruent* branch – *i.e.*, a branch with an address such as it will modify the state of the victim branch – to the victim branch with valid input to poison it. It exploits branch predictor's *aliasing* or *interference*, as explained in Section 2.1.2.

Cross-Address Space (CA) In-Place (IP) In another process regarding the victim, the attacker executes a *shadow* branch – *i.e.*, with the same virtual address that the victim branch – with valid input to poison it.

Cross-Address Space (CA) Out-of-Place (OOP) In another process regarding the victim, the attacker executed a *congruent* branch to the victim branch with valid input to poison it.

Explanation. While we have experimented with a few types of *Spectre* attacks during the internship, we have mainly concentrated our efforts on the *PHT* version. This is the version that we will describe. The goal of the *Spectre-PHT* is to produce the *false then true* pattern showed in Figure 2.13. Consider the following code :

```
if (x < array1_size)
    y = array2[array1[x]];
```

The `if` is the *attacked branch*, contained in a *victim function*. The index of `array1`, `x`, is an argument of the function. The *training phase* of *Spectre* will execute this function a lot of time with a valid index for the first array. This will have for effect that the PHT's saturating counters will be incremented at their maximum. Then, during an *attack run*, the attacker will send a malicious index for the first array in the *victim function*, such as the sum of the first array's address and the index gives the targeted secret address, *i.e.*, `array1 + x = &secret`. During the attack run, the processor will execute transiently the `array1[x]` memory access, which corresponds to a load of the secret. For now, nothing helpful for the attacker. If the transient window is long enough, *i.e.*, the load, computation and comparison of the condition are long enough, the access of the second array `array2[secret]` will be also executed transiently. This memory access is the *encoding phase*, where the victim function plays the *sender* role in the covert channel. When the processor will finish the computation of the condition of the attacked branch, all memory accesses will be rolled back at an architectural level. Then, the last part of the attack arrives with the covert-channel, here *Flush+Reload*, where the attacker will be the *receiver* – this is a *recover phase*. The attacker will load every value of the second array and measure the time taken by the load – supposing that before the attack, the attacker has flushed all values of the second array from the cache. When the attacker will detect an element of the array being cached, it will know that it has been cached during the transient execution. Thus, the index of the cached element is equal to the secret value, consequently the attacker has gained the knowledge of the targeted secret. This scenario allows understanding *Spectre-PHT* principle – in the real-world, the implementation is *quite* more complex, but these details will be left out here.

Applicability. As we could imagine, the exploitation of a *Spectre* attack is not trivial. However, some powerful scenarios are tested with *Spectre*, the most impressive one being an attack in a cloud.

Indeed, it is theoretically feasible to steal data of another process in a cloud environment, where the attacker and the victim processes run on the same processor, while being completely isolated at the operating system level. This vulnerability is present since more than a decade in our processors. Since it is the direct consequences of an important optimization mechanism, it can't be simply disabled by turning off the branch predictor – the cost will be too heavy. However, as far as we know, no public exploitation of *Spectre* in the real world is known yet [Apv18] – while *Meltdown* has surely been exploited in the past.

Countermeasures. A variety of state-of-the-art countermeasures have been developed for the two last years. The problem with them, currently, is that none of them cover all the *Spectre* variants on all architectures with an acceptable performance cost. The first kind of countermeasure is deployed case-by-case, in the *Linux* kernel, with special instructions to prevent speculation before critical branches. Since it has to be done manually for each branch, it is very unlikely to cover every critical branches and being sustainable in the long term. Wang et al. [Wan+18] proposed *oo7*, an automated way of finding *Spectre* vulnerable code by control flow extraction, taint analysis and address analysis. The previous kind of countermeasures are *software-based*. Other countermeasures are *hardware-based*, like Barber et al. [Bar+19] who proposed *SpecShield*, a micro-architectural framework allowing isolating transient instructions until determined as safe. Other are *hybrid-based* solution which mix hardware and software cooperation, like Fustos et al. [FFY19] who proposed *SpecGuard*, allowing the developers marking a memory region as *non-speculative* – when containing a secret, a key, etc. – thank to an operating system *Application Programming Interface (API)*, and then used a microarchitecture extension to prevent speculative execution for these marked regions.

3 Setup

Now that all the required – and beyond – knowledge has been studied, we can get our hands dirty by diving into the setup of all required tools in order to achieve the internship. Three main topics must be discussed here. The first one is the setup of the *Raspberry Pi*, the *ARM* board that we used. Usually, this step should have been very quick and easy, nevertheless we will see that it was not the case. The second one is the setup and first experiments with a *Spectre* implementation. Again, it should have been very easy... if we were under an *x86* board. For the *ARM* architecture, finding a reliable implementation can quickly become very tricky. Finally, the third topic is the setup of *gem5*. This one is a state-of-the-art project, therefore like most of them, the lack of documentation and help can make simple operations considerably harder.

3.1 Raspberry Pi

The recent *ARM* processors vulnerable to *Spectre* – with advanced hardware optimization mechanisms – are based on the *ARMv8* architecture [1]. As we know (presented in Section 2.1.3), the *ARMv8* is a 64-bit architecture, therefore we need a 64-bit operating system – which implies generally a 64-bit kernel, user land, compilation tools. The challenge was to meet these requirements. Table 3.1 provides a summary of possible operating system for the *Raspberry Pi*.

1. The first tested operating system was the official and recommended *Raspbian*, which exists only – for the stable and official version – in 32-bit flavor. The installation of the system was quite easy, and it boots correctly. Unfortunately, since it is a complete 32-bit operating system, we have faced the fact – which showed a lack of anticipation here – that this system is not able to run our binaries, mainly our *Spectre* attack which is a 64-bit compiled *ARMv8* code. Therefore, this system was not suitable for our work.

2. The second one was a developer version of *Raspian*, which was attractive because it was very easy to upgrade from the official 32-bit version to this one. The upgrade installs the latest version of *Raspbian* with a 64-bit kernel but preserves the original user land, which allows us to run 64-bit *ARMv8* binaries. This setup was used for a few weeks, until we needed to do some experiments with kernel modules. In order to do this, we needed a 64-bit compilation tool chain (compiler and a debugger) and the kernel sources for the latest 64-bit version. These last requirements were too complicated to meet with a mixed 32/64-bit system, and we decided to switch for a full 64-bit system. Therefore, this system was not completely suitable for our work.

3. The third experimented system was the – future – official *Raspbian* in the 64-bit flavor, currently in beta testing phase. This system would have been perfect, but after the installation, the only result we got is an infinite hang during the boot process.

[1]As far as we know, there is some *Spectre* implementation for *ARMv7*. Nonetheless, they are destined to be executed on *ARMv8* processors running in *ARMv7* compatibility mode. Moreover, they are very inefficient.

Table 3.1: Tested Operating Systems.

No.	Operating System	Kernel – Userland	Status	Results
1	Raspbian	32-bit – 32-bit	Stable v2020.05.27	★★☆
2	Raspbian	64-bit – 32-bit	Developer Rolling-Release	★★☆
3	Raspbian	64-bit – 64-bit	Beta v2020.05.27	★☆☆
4	Ubuntu Server	64-bit – 64-bit	Stable v20.04.1 LTS	★☆☆
5	Kali Linux	64-bit – 64-bit	Stable v2020.2a	★★★
6	Manjaro	64-bit – 64-bit	Stable v20.0	☆☆☆

☆☆☆ Untested ; ★☆☆ Do not boot ★★☆ Boot but unsuitable ; ★★★ Boot and suitable

4. We had to find another suitable system. The *Ubuntu Server* for *Raspberry Pi* seemed to be a good candidate, since it is a full 64-bit operating system. But instead of the boot hand, we got an infinite boot loop. Therefore, this one and the preceding one were not usable at all on our hardware.

5. Finally, we tested a headless version of *Kali Linux*, a penetration testing oriented distribution. The only properties we were interested in was a light runtime environment and the full 64-bit distribution, with compilation tool chain and kernel sources. After the installation, this operating system revealed that it worked and was absolutely suitable for our work.

The code used for the installation is available in Appendix A.1.

3.2 *Spectre*

Our *Raspberry Pi* being successfully installed and usable, we were able to start experimenting with *Spectre*. At the beginning of the internship, we thought that we could just use an existing implementation of *Spectre* and that all would be fine. It was an assumption a bit too optimistic.

There is a *lot* of implementations of *Spectre* for the *x86* architecture, the main paper providing the reference one [Koc+19]. When it comes to *ARM*, the reference implementation is the *IAIK* one [Can+] – also called the *TransientFail* one, from the name of the website. This repository – which is bound to Canella et al.'s taxonomy of all *Spectre* and *Meltdown* attacks [Can+19] – contains all known variants of *Spectre* and *Meltdown* for both the *x86* and the *ARM* architecture.

The first steps are quite straightforward: download, read, compile, execute. When it works directly – like on our *x86* desktop computer –, there is nothing much to do. But when you do not have the expected output, a large number of possibilities appears to improve the results.

In this chapter, we will not describe in details our *Spectre* output and our experiments. Instead, we will briefly describe our output and list tried methods with the hope of getting something better. It will be useful if someone encounters the same problem and give an overview of all parameters that can impact – or not – a transient attack.

The output is the following. *Spectre* is supposed to recover a secret string without accessing to it. When the attack is successful, the recovered string – which is the output – must match the secret string. In our case, it was not the case by default. In very rare cases, the attack was successful, but for the majority of cases, the attack was only able to recover the first letters of the secret string, and the rest of the output was wrong. Finally, sometimes, the attack did not work at all.

To simulate an attack on our simulator and evaluate the accuracy of our simulation, we needed to have something stable and reproducible on our real system. Therefore, we wanted to improve the efficiency and/or the stability of the *Spectre* implementation. From here, many techniques can be tried.

Cache miss threshold The *cache miss threshold* is the number of cycles at which we distinguish the latency of a cache hit from a cache miss. This number was automatically computed by the implementation, and therefore could be wrong [2]. Thanks to the code developed for the *Cache Template Attack* [GSM15] – a generic cache attack using *Flush+Reload* to automatically detect and exploit cache leakage of an application without a prior knowledge of it –, we can obtain the distribution of latencies for cache hits and cache misses. Therefore, we can be sure to configure the adapted cache hit threshold. Unfortunately, it did not help.

Pinning *Pinning* is the fact of putting a process on a particular processor core and preventing the scheduler of the operating system to change the execution core of the process [3]. By default, executed processes switch from one core to another during their life-time. Pinning the *Spectre* process can improve the efficiency of the attack, because if one process performing a cache attack is switched to another core, depending on the processor micro-architecture and the attacked level of cache, the targeted cache can be changed without the process being noticed. Unfortunately, it did not help in our case.

Timing method The *timing method* is the technique used to measure the latency between two memory accesses – to detect a cache hit or a cache miss. By default, the *IAIK* implementation uses the `clock_gettime()` function, a *POSIX* standardized function which measures the clock with a resolution of nanosecond. This is the easiest method. Two alternatives are proposed by the authors:

1. Using a special register – a *PMC* – of the processor [4]. Like the `tsc` register in the *x86* architecture – which is a processor's register incremented by 1 at each cycle – accessible with the `rdtsc` assembly instruction, there is the `PMCCNTR_EL0` register in the *ARMv8* architecture which has the same functionality and which is contained in the *PMU*. To be accessed, the latter needs a kernel module which allow the user land to read its *PMC*. Therefore, we need `root` permissions – which is completely unrealistic, since it is supposed to be an attack without any special permissions. Nonetheless, it provides better results – because of the higher accuracy in time measurements – but does not solve all *Spectre* instability.

2. Using the *Linux*-specific `perf_event` interface [5]. The latter is an *API* which provides access to the *PMU* of the processor in a abstracted and convenient way, from the user land and without `root` permissions. In theory, this method should give a cycle-accurate timer (better than the default nanosecond accurate timer). In practice, with our system and our processor, we supposed that the overhead of using this kernel interface was too heavy: it was worse than the default technique, while the previous was better.

Interupts According to the authors of the implementation, causing a lot of interrupts [6] in parallel to the *Spectre* attack can improve its efficiency. We do not really know why, but it did not help anyway.

Mitigations Since there is no hardware protection to defeat *Spectre* yet – which seems to be the best way, as far as we know –, *Linux* kernel's developers have implemented some software mitigations to defeat a *Spectre* attack against the kernel. These mitigations should not interfere with our attack in theory, since it is contained in one user space process. But, it does not cost anything to try to disable them. The verification of enabled mitigations can be achieved by a simple lookup into some files exposed by *sysfs* [7]. In our case, a mitigation called `__user pointer sanitization` was enabled. This mitigation consists in disabling speculative execution for some sensitive pointers that are passed to the kernel by the user during a system call, a pointer which can be malicious – an attempt of a

[2]In this particular implementation, the threshold is contained in the `CACHE_MISS` variable.
[3]On *Unix* systems, pinning can be achieved using the `taskset` command.
[4]To use this method, you'll have to set the `ARM_CLOCK_SOURCE` variable to `ARM_TIMER` and use a special kernel module, illustrated in Appendix A.2.
[5]To use this method, you'll have to set the `ARM_CLOCK_SOURCE` variable to `ARM_PERF` and add a `perf_init()` call in your `main.c` file (undocumented by the authors).
[6]On *Unix* systems, interrupts can be generated with the `stress -i 1 -d 1` command
[7]`/sys/devices/system/cpu/vulnerabilities/spectre_v{1,2}`

speculative bound check bypass [Tor]. Unfortunately, despite all our efforts, it seems that it is not possible to disable this mitigation. The conclusion is that it did not help.

Page size *Spectre* implementation tries to spread its probe array over multiples pages in order to defeat the cache prefetcher, since a lot of them do not look further than the page size in memory. Indeed, if a prefetcher loads an element of the probe array, then the deduction of a transient execution access based on the latency of the probing access will be wrong. Different page size are supported, depending on the kernel and the processor architecture. Often, a page has a size of 4 kilobytes. A functionality called *Huge Page* allows the usage of pages larger than 1 megabyte or 1 gigabyte, and can be enabled transparently. In our case, we have verified that the functionality was supported [8] and forced it to be disabled [9]. However, it did not change the *Spectre* behavior.

Prefetcher The cache prefetcher has been a suspect for the unexplained results that we got. In theory, the cache prefetcher can be disabled in the *ARM* architecture by setting a bit of a *MSR* [10]. However, doing this requires special requirements: kernel privilege and a special processor state. It cannot be easily disabled.

Source modification A lot of attempts has been made to improve *Spectre* results by modifying its source code. It can be modified at different levels: training phase, exploitation phrase, timing probe phase, secret reconstruction phase. However, regardless the kind of modification, a slight change in the source code can considerably change the output – but without any apparent reason. Despite a lot of attempts, we could not identify the root cause of the erratic behavior of this *Spectre* implementation.

Frequency All modern processors adapt their frequencies to the current workload in real time, which corresponds to *frequency scaling* – often referenced as *Dynamic Voltage Frequency Scaling (DVFS)*. This is mainly due to energy consumption constraints. As a result, if the frequency changes between two latency measurements, the latter could become irrelevant if the frequency value is correlated to the latency. We have set the frequency to its maximum and disabled the frequency scaling [11], but like other techniques, it did not improve *Spectre* results.

Compiler version Sometimes, the same compiler but two different versions can produce very different machine code from the same source code. The reason lies – generally – into all the optimization mechanisms that are implemented into every modern compiler. We have learned that the original *Spectre* implementation was compiled with *GCC* version 7. Instead of our version 9, we have tried *GCC* version 7 and version 8. It did not help.

Compiler optimizations We have seen that compiler versions can greatly modify its output. Therefore, it is the same for compilation flags, which would completely change the outcome of the attack. By default, the *Spectre* implementation was compiled at the optimization for code size (`-Os`). It corresponds to the optimization level that worked best for the authors during the implementation. However, it does not work well on our system. If we try no optimization (`-O0`), the *Spectre* attack no longer works. On the other hand, if we set a high and aggressive level of optimization (`-O1`, `-O2`, `-O3`), the implementation works perfectly. It is, however, not always the case when adding small changes to the source code. This is the pitfall of optimization flags for security research: the generated code is less understandable and less stable across different source code versions and across compiler versions.

[8] Verification of huge page support can be achieved with these two commands on a Raspberry Pi: `sudo modprobe configs && zcat /proc/config.gz | grep -i k_page`.

[9] With a *Linux kernel*, its sufficient to add `transparent_hugepage=never` to the `/boot/cmdline.txt` file to disable it explicitly.

[10] For the L1D and L2 caches on an *ARMv8 CPU*, the prefetcher can be disabled by setting the bit n°56 of the `CPUACTLR_EL1` register (`S3_1_c15_c2_0` in assembly) to 1.

[11] On *Linux*, it can be achieved with `root` permissions by issuing the following command: `cpupower frequency-set -g performance`.

A lot of techniques have been presented here. For the attentive reader, it will not be necessary to recall that all of these have, more or less, failed to improve *Spectre* stability and/or efficiency. This outcome is specific to our hardware and our *Spectre* implementation, indeed, one of the above tips can clearly change everything for another situation. In our case, the conclusion is that:

1. The core of the attack works on our hardware ;

2. Stability issues must reside in the source code of the implementation.

During our experiments, we tried to see what caused the inconsistent results of the attack. The final conclusion is that it seems to came from both the branch predictor, which is sometimes not tricked by *Spectre*, and the cache covert channel, which is sometimes too noisy.

3.3 Gem5

Set up *gem5* is not as difficult, unless you have any documentation and no tips in case of error. Fortunately, we provide some tips and detailed instructions in Appendix 3.3. The *gem5* project is hosted by *Google*, we just need to clone the repository from their server. Then, the build of the simulator is completely automated by *SCons*, which is a software construction tool written in *Python* – competing with, *e.g.*, *Rake* or *Ninja*. This software uses the flexibility and the power of *Python* to express complex build rules, completely replacing the *GNU Build System* (*make*, *autoconf*, etc.). When nothing goes wrong, a simple command is sufficient to compile *gem5*.

The longer and most interesting part is to understand how *gem5* works and behaves, by both reading the documentation and performing some experiments. The principles of *gem5* have already been described in Section 2.2.2. Here, we will present some conclusions – verified by experimentation – applied to our particular case of transient attacks simulation.

Before we start making a real *gem5* system, we had to understand which components were more suitable to fit in our work. The main choice, at the beginning, was the processor. We wanted to know which processor was able to simulate which kind of attack. The *Spectre* attack is very dependent on the type of processor used in *gem5*, since it rely on the behavior of the branch predictor to execute transient instruction and on cache behavior to retrieve the secret information. Consequently, we needed, at least, a speculative execution mechanism and a branch predictor to execute the transient instruction with the malicious data access, and a L1 cache to have a timing difference when accessing a data. We will present the 4 major base processors that we could use in *gem5*.

`AtomicSimpleCPU` [12] is the default and most simple processor in *gem5*. It executes a configurable number of instruction per cycle [13] and use the `atomic` memory mode – which deliver data in an atomic cycle. It is not suitable for real simulation nor transient attacks. In practice, it is used for debugging or some special functionalities that do not require a detailed processor.

`TimingSimpleCPU` [14] is the slightly more detailed version of the default processor type in *gem5*. It executes each instruction in a single clock cycle, except for memory instructions which use `timing` memory mode, *i.e.*, they are handled in the memory subsystem (caches, buses, controllers). In this processor, we have no speculative execution at all, thus it is not suitable for transient attacks.

`MinorCPU` [15] is an in-order processor. The main addition to the previous one is the 4-stage pipeline: 1) fetch line 2) decode line into macro-ops 3) decode macro-ops into micro-ops 4) execute. This processor is the first to expose many parameters directly in *Python*, contrary to the previous ones. This processor can be used to perform detailed timing measurements. It is able to use a branch predictor, with `TournamentBP` set by default.

[12]Defined in `src/cpu/simple/AtomicSimpleCPU.py` from the *gem5* source tree.
[13]Defined by the `width` parameter in the source file
[14]Defined in `src/cpu/simple/TimingSimpleCPU.py`
[15]Defined in `src/cpu/minor/MinorCPU.py`

`DerivO3CPU` [16] is an out-of-order processor that requires a separate instruction and data L1 caches. Like the `MinorCPU`, it has a branch predictor set to `TournamentBP` by default. It is heavily customizable and is the more representative of an advanced modern processor. This model can be used to simulate transient attacks.

There is also some out-of-tree – regarding the *gem5* source code – processors that are built on the previous ones. We can mention the `HPI` [17], which is a high-performance in-order processor [Ash17] that aims to be representative of a modern in-order – unlike our *ARM Cortex-A72* – *ARMv8-A* processor. It is based on the `MinorCPU` class. Like the previous one, it has a tuned `TournamentBP` branch predictor. Since it is an in-order processor, we prefer the `DerivO3CPU` over this one regarding the importance of out-of-order in most of today's high-performance processors.

[16] Defined in `src/cpu/o3/O3CPU.py`
[17] Defined in `configs/common/cores/arm/HPI.py`

4 Contributions

We have now all the necessary background to understand all the challenges faced and decisions made during this internship. The goal of the internship is to set up an accurate *gem5* system to be able to reproduce these attacks as faithfully as possible. The first step was the setup of all needed tools, described in Section 3. In this chapter, we will cover why and how it was necessary to start by experimenting with *Spectre*, in order to develop our own version to fit our requirements. When the latter was ready to be exploited, we focused our efforts on the realization of the *gem5* system.

Note that all experiments are fully scripted and highly automated. Every data, result and code is stored and available on request.

4.1 *Spectre*

Public *ARM* implementation of *Spectre* are not common. Indeed, there exist various *x86* implementations, but when it comes to *ARM*, there is only a single reference implementation – the *IAIK* one [Can+] – and a few personal implementations on the Internet, found on several blogs or *GitHub* – that are more or less similar. Consequently, if someone has special requirements for his *Spectre* implementation, it is very likely that they will have to implement it themselves.

Moreover, we have seen in Section 3.2 that the reference implementation of the *IAIK* team did not really work under our *Raspberry Pi*. We will see that despite all our experiments and our tries to obtain better results, the only way to have a stable and useful attack was to implement our own. After the description of our experiments, our designs choices and our implementation, we will present the results of our binary and its capabilities.

4.1.1 Experiments

We have seen (in Section 3.2) that a lot of efforts have been made to improve the *Spectre* behavior on our system, which was not stable nor predictable. Thus, in order to implement our own version, some choices were important and required experiments. The most important choice was the timing method. Some of them have already been presented in Section 3.2, since they were available in the *IAIK* reference implementation [Can+]. Another method is possible, which increases the implementation's complexity but worth it: using a *counter thread*.

Counter thread The idea behind this concept is very simple: we create a dedicated thread running in parallel on the machine, which increments a counter indefinitely. The downside is that it requires a multi-core processor – to obtain a good temporal resolution – and increase the code complexity. The advantage is that it seems, regarding our own experiments, to have the best resolution as a timing method, since it gives the best results with *Spectre* on *ARM*. The counter thread has been tested with an implementation found on *GitHub* [V-E], which will be called the *V-E-O* implementation later in this report.

Table 4.1: Timing methods usable for *Spectre*.

Method	Efficiency (*Cortex-A72*)	Efficiency (*ThunderX2*)	Usability	Conclusion
POSIX counter thread	★★★	★★★	★★☆	Perfect for efficiency
POSIX `clock_gettime()`	★★☆	☆☆☆	★★★	Perfect for simplicity
ARM PMU direct access	★★☆	–	☆☆☆	Not usable in real life
Linux `perf_event`	☆☆☆	–	★☆☆	Not worth it

Table 4.1 summarizes all the timing methods tested during the internship, sorted by efficiency. The efficiency was tested on two processors, a low-end one (*ARM Cortex-A72*) and a high-end one (*ARM Marvell ThunderX2 99xx*). The complexity represents both the amount of code needed to have a working method and the amount of knowledge needed to set up the method. Two interesting choices emerge from this:

***POSIX* counter thread** This is the best choice for *Spectre* efficiency, since in our experiments on both our low-end processor and our high-end processor, the accuracy was equal to 100% – in other words, the attack worked perfectly all the time. However, this method requires a multi-core processor and the knowledge of using the *POSIX thread API* (`pthread`).

***POSIX* `clock_gettime()`** This is the best choice for simplicity, and sometimes the best trade-off between simplicity and efficiency. While accuracy was around 95% – for the median – on our low-end processor, the cost of the solution is a simple *POSIX* function call – which means a cost equal to zero for a usual *C* programmer. Unfortunately, this method was not working on our high-end processor.

Branch predictor training. We have also few experiments concerning the training methods of the branch predictor. It is a fascinating subject, since the training phase must adapt itself to the branch predictor architecture – it can even take advantage of some branch predictor knowledge. However, we do not have interesting results about it – our efforts was concentrated on the timing methods, because it was the part with the worst efficiency. Nonetheless, the training currently used works quite well, and is the following. We perform a customizable number of loops – typically, 30 or 100 – executing the victim function with the attacked branch. Every five training runs, we perform an attack run with the malicious array index.

4.1.2 Faced Challenges and Design Choices

The issues with the reference implementation on our platform were manifold:

Instabilities The accuracy of the attack could vary from 0% to 100% in a non-uniform distribution.

Pathological Sometimes, the wrong output clearly followed a pattern, but the micro-architectural reason was unclear.

Unadapted The reference implementations are just basics *PoCs*, which means that they are quite simple. In fact, it was the case for the *IAIK* implementation. It does not provide a lot of customization options or useful research use cases, but it does a lot of esoteric operations with the `printf()` function, for example. Consequently, this can impact the efficiency of the attack on some system. Moreover, this makes the implementation very hard to use when we want to automatize experiments.

From these observations, we defined several criteria for our own implementation:

Table 4.2: Metrics in our *Spectre* implementation.

Metric	Method	Meaning
Nb. of byte to guess	`strlen()`	Challenge size
Nb. of correct guess	Hamming distance	Result accuracy
Sum of score of all guess	`score_sum += results[i]`	Result difficulty
Nb. of elapsed cycles	`rdtsc()`	Result speed
Nb. of cache misses	`perf_event` with `PERF_COUNT_HW_CACHE_MISSES`	Microarchitectural state
Nb. of mispredicted branches	`perf_event` with `PERF_COUNT_HW_BRANCH_MISSES`	Microarchitectural state

1. **Stable results** The accuracy of the attack should both be high and, most importantly, comprised between a reasonable interval – without the unavoidable outlier measures. This point is essential to have a good point of comparison between the runs on the real hardware and on the simulated system.

2. **No advanced compiler flags** We do not want side effects of compiler flags to affect the accuracy of the attack. Indeed, the compiler can sometimes really increase the efficiency and/or the bandwidth of the attack. However, it reduces the reproducibility between different architectures or compiler versions. Optimizations are not bad in essence, but they should be used manually, or at least, very carefully.

3. **Usable both on *gem5* and a real *ARM* system** Both *ARM* and *gem5* add constraints on our attack. On *gem5*, some instructions or processor functionalities are not implemented, which implies to limit ourselves or to find a workaround.

4. **Customizable at runtime** To be useful for attack development and parameters exploration – in order to get the best efficiency –, it would be great to have some attack parameters being modifiable at runtime.

5. **Simple output** Unlike the other *PoCs* found on the Internet, we want to have a simple output which will be both easy to parse and informative.

6. **Metrics** In order to have an informative output useful for our research, we needed to obtain some metrics about one run of the attack. They will be used both for attack research – understand the attack behavior – and attack simulation – compare a run between the real hardware and the simulated system. Table 4.2 describes the chosen metrics, the methods used to get them and their meanings.

4.1.3 Attack Development

In this section, we will describe the different steps of the attack's development. Our own implementation used the code base from the original attack code – the simple `spectre_poc.c`, presented in the reference paper [Koc+19]. We can already tell that our final code is very far from this simple one. We used this one for two reasons:

1. It was quite simple and without exotic customization,

2. The heuristic used to decide, for a guessed byte, the probability of being the right guess is simple and efficient. This heuristic is still used in our final version.

However, this *Spectre* attack is intended to be run on a *x86* processor. Thus, considering the work made by the *IAIK* team to write *ARM* assembly instead of *x86* one [Can+] [1], we have integrated

[1]The goal was to replace these *x86* instructions: `clflush`, `rdtsc` and `mfence`. It ends with a lot of *ARM* instructions, like `dsb ish`, `dsb sy`, `isb`, `dc civac`, and others.

Table 4.3: Example of output data of our *Spectre*.

total bytes	correct bytes	score sum	elapsed cycles	cache misses	branch mispredicted
40	40	22199	3730822749	10335053	565859
40	40	21787	3662569510	10144720	544939
40	40	5098	860236145	2373912	125350
...	...	...	...	...	...

and modified it – and also, wrote a lot of documentation and explanation, based mainly on our own experiments and the *ARMv8-A architecture manual* [20] – into the original `spectre_poc.c`.

At this point, we had a new *Spectre* implementation which was a mix between the original one for *x86* and the *IAIK* one for *ARM*. This new implementation could be executed on *ARM*, but the attack was not working at that time. To make it work, we had to imperatively modify the code as follows:

- Make the computation of the attacked branch condition longer – using a division – to increase the transient execution window [2];
- Ensure that some data were flushed at the right moment [3];
- Perform manually some common compiler optimizations – loop unrolling, invariant code motion, fission/fusion;
- Inline some functions. We give a special point to this one, which is a very important optimization for the efficiency of the attack – indeed, it allows to unload the *RSB* and hence the branch predictor, and makes the code faster – which is important for an attack happening during a few instruction's execution.

An interesting operation is to use the `-fopt-info` flag of *GCC* [Sta20], which enriched compiler's output with a summary of its own optimization decisions during the compilation of one binary. Thanks to this, we can then choose ourselves which optimization we want to implement or not – based on the impact on *Spectre* efficiency.

Then, another development phase arrived. At this step, we have fulfilled our first, second and third requirements (stable, no exotic compiler flags, usable both on *gem5* and on *ARM*) that we described in Section 4.1.2. The others needed a carefully engineered phase, where for each written functionality, we had to check if the efficiency of the attack had not decreased. Since a tiny change in the source code can sometimes lead to a broken attack, it was not easy to both allow attack parameters to be changed at runtime and add support for metrics counting during the attack. After this long step, we fulfilled our fourth, fifth and sixth requirements (customizable, simple output, metrics). The method used to get the number of mispredicted branches and caches misses was to exploit the *Linux* `perf_event` *API*, briefly presented in Section 3.2. This was not difficult to use since we were familiar with this interface, often used in high-performance computing. However, using it at the same time as performing a transient attack can be tricky.

Note that the timing method used in our own implementation is the same as the one used in the *IAIK* reference implementation, which is the *POSIX* `clock_gettime()` function to approximate elapsed time in nanoseconds. Previously (in Table 4.1), we have seen that the counter thread is a better choice than the latter – if you both have a multi-core machine and know how to use a multi-threading library, which is our case. Unfortunately, we knew that the counter thread was better in our case only after the finalization of the implementation. Consequently, due to time constraints, we have not changed the implementation.

[2]The time slice where instructions are executed in the transient domain, *i.e.*, executed speculatively then discarded – when the speculation was wrong, like in *Spectre*.

[3]A flush instruction, if no precaution is taken, can be delayed or moved into the introduction flow both by the compiler – statically, at compilation time – and the processor – dynamically, at runtime.

4.1.4 Results

Our implementation being finalized, we can present the results of this development. Some usage examples are given in Appendix B.1. After an execution on our *Raspberry Pi* with a fixed set of parameters, we end up with a table (4.3) with a lot of value, each row being one run. Below a description of the data:

`total bytes` Represents the number of bytes to guess;

`correct bytes` Represents the number of bytes correctly guessed by *Spectre*;

`score sum` Represents the number of iterations needed to obtain the current result, which is strongly correlated with the amount of noise – in term of cache and branch predictor usage;

`elapsed cycles` Represents the execution speed of the attack;

`cache misses` **and** `branch mispredicted` The number of cache misses and mispredicted branches during the attack will be used to compare against the *gem5* run as a micro-architectural accuracy hint.

These raw data are not very useful as is. Consequently, we wrote a little *Python* script to process then analyze the data. To achieve this, we used four *Python* libraries – that I did not know how to use before – to do various data acquisition, basics statistical analysis and visualization: *NumPy*, *SciPy*, *Pandas* and *Matplotlib*. We will briefly describe the script, what it does and show its output – but we will leave out the *Python* details.

Data processing. Some data are directly useful, others needs to be represented by a proportion, and finally some of them needs a more careful attention. For example, the number of correct bytes can be simply used as a percentage. When it comes to the number of cache misses and the number of mispredicted branches, it is useful to have the raw value for the real hardware / simulated system comparison. However, when we want to visualize the correlation between, say, the number of mispredicted branches and the accuracy of the attack, more steps are needed. Indeed, the raw number of mispredicted branches is strongly correlated to the number of elapsed cycles. Since the different runs have a different number of elapsed cycles – with sometimes a huge difference –, a ratio has to be computed if we want to compare the number of mispredicted branches across different runs.

Let b be the column vector of the number of correct bytes for each run from our table, c the column vector of the number of elapsed cycles and m the column vector of the number of mispredicted branches. Then, we compute for each element of the vectors indexed by i:

$$m_{\text{ratio}_i} = \frac{m_i}{c_i} \tag{4.1}$$

$$m_{\text{norm}_i} = \frac{m_{\text{ratio}_i} - \min(m_{\text{ratio}})}{\max(m_{\text{ratio}}) - \min(m_{\text{ratio}})} \tag{4.2}$$

$$b_{\text{norm}_i} = \frac{b_i - \min(b)}{\max(b) - \min(b)} \tag{4.3}$$

In other words, we use the ratio between mispredicted branches and time taken to perform the attack in order to compare the number of mispredicted branches between different runs (Equation 4.1), and we normalize values on a same scale ($[0,1]$) to be able to compare mispredicted branches and attack accuracy between different runs (Equations 4.2 and 4.3). Now, we are able to compute the correlation we talked about above. To do so, we compute a simple *Pearson's* linear correlation coefficient ρ between b_{norm} and m_{norm} such as:

$$\rho = \frac{\operatorname{cov}(b_{\text{norm}}, m_{\text{norm}})}{\sigma_{b_{\text{norm}}}\sigma_{m_{\text{norm}}}}, \tag{4.4}$$

where cov is the covariance and σ the standard deviation. Finally, we can easily compute a linear regression line in *Python* to visualize the correlation on the plot.

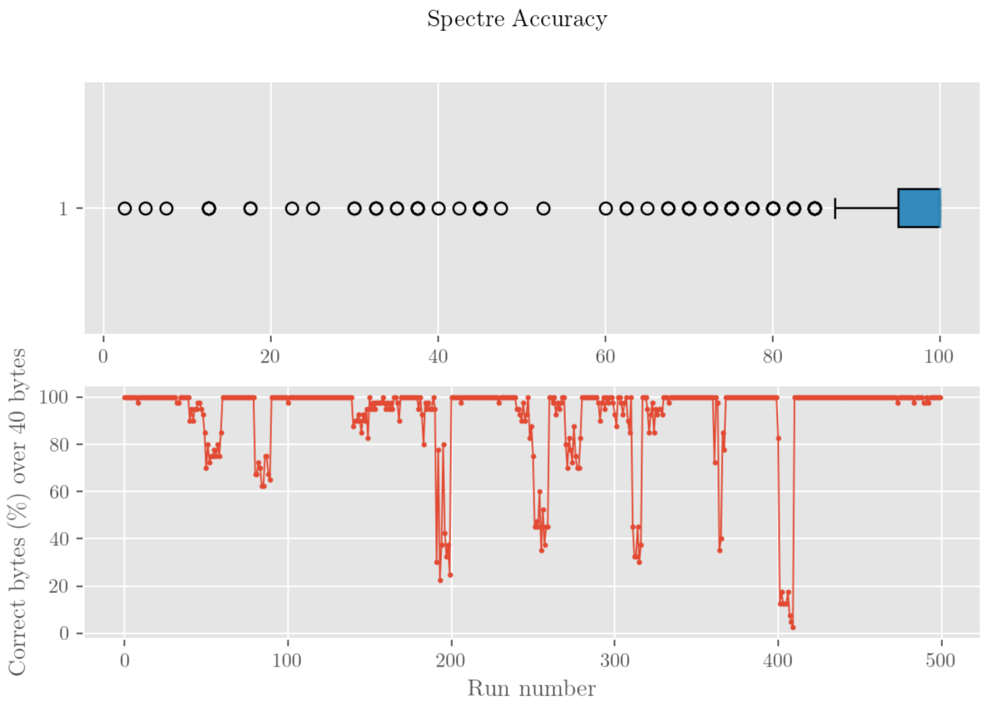


Figure 4.1: Percentage of correctly recovered bytes.

Interpretation. We will present some results of the script. All plots are not relevant, only few of them will be covered here. The bottom of Figure 4.1 is a sequence plot of the accuracy for all runs during our experiment, while the top is a box plot [4] of the sequence. We see that, statistically, the accuracy is very high and always close to 100%. Sometimes, we could observe huge drops in term of accuracy, but they are few enough to be considered as outliers. Figure 4.2 follows the same pattern as the previous one, except that the considered data is the number of iteration (`score sum` in Table 4.3) needed to get the result. This plot can be used to decide, for example, which *Spectre* implementation is faster than another and hence allows a better bandwidth – for an equal accuracy, of course. Finally, Figure 4.3 represent the number of mispredicted branches by the number of correct guessed bytes. On the plot, we can observe the regression line which gives a hint on the correlation. This plot – or kind of plot, since we could plot the number of cache misses by the number of correct guessed bytes, for example – could give a hint about the reason of the efficiency or inefficiency of an implementation.

We can deduce a few conclusions from this plot:

- The *Pearson's* correlation coefficient is positive ($r > 0$ on the plot), which means that we have a clear correlation between the accuracy of the attack and the number of mispredicted branches. Indeed, if we do not consider the group of points at the left (explained below), there is a strong correlation.
- We observe that, to have an accuracy superior to 80%, there is a minimum threshold of mispredicted branches that is required.
- Some low accuracy results – at the left of the plot – are unexplained by the number of mispredicted branches. We can suppose that it is due to another micro-architectural noise, for example the cache usage.

Finally, before to dig into the *gem5* system, Table 4.4 describes which implementation works on which hardware. We recall that our implementation (*IRISA*) and the *TransientFail* implementation use the `clock_gettime()` function as timing method, while the *V-E-O* implementation uses a counter thread.

[4]Which display the minimum, the maximum, the median (Q2, 50th percentile), the first quartile (Q1, 25th percentile), the third quartile (Q3, 75th percentile).

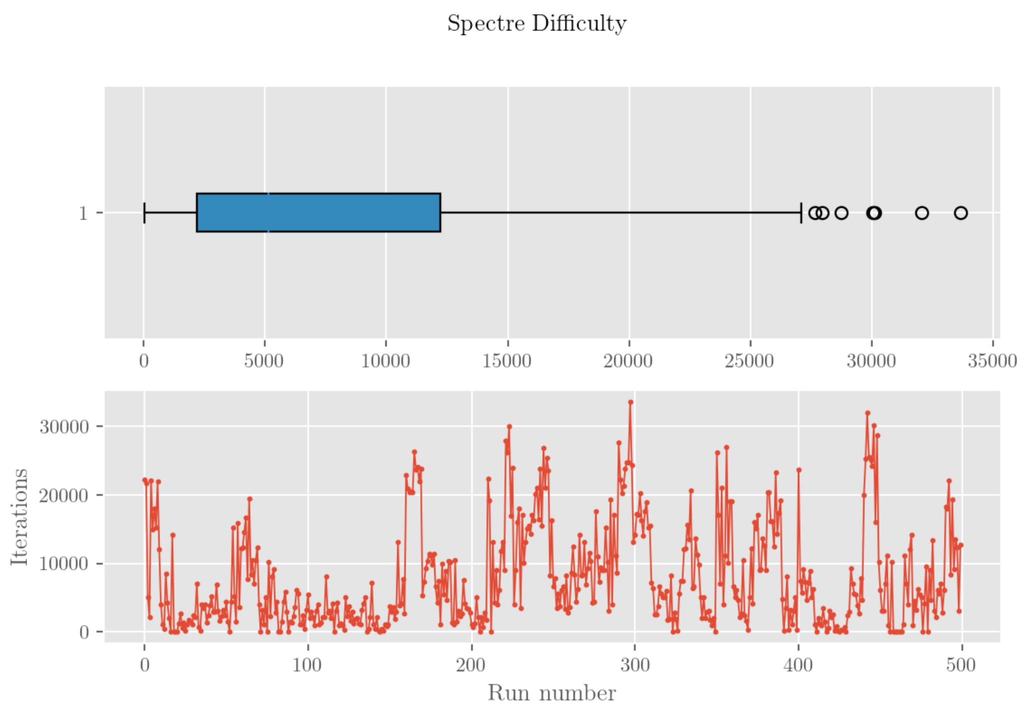


Figure 4.2: Number of iterations to get the actual result of Table 4.1.

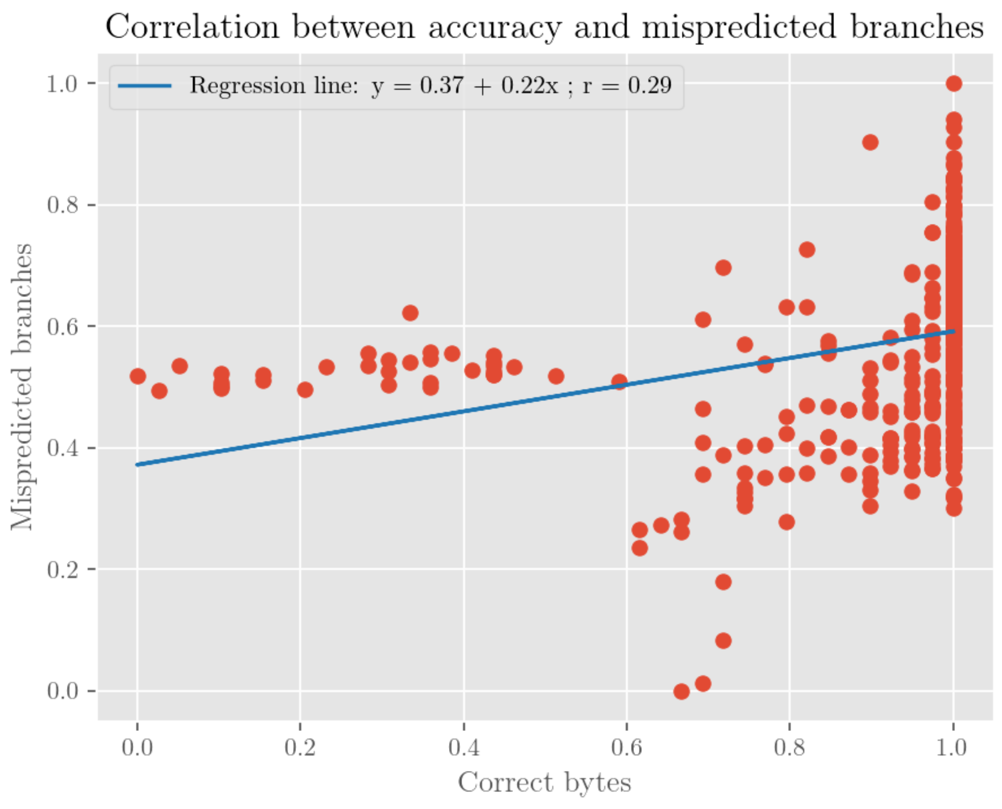


Figure 4.3: Comparison of correctly recovered bytes and mispredicted branches.

Table 4.4: Taxonomy of tested *ARM* implementations and tested hardware.

Implementation \ Target	*Raspberry Pi v4* (tab. 1.1)	*Grid5000* (sec. 1.2.2)
IRISA	Stable (95%)	Does not work (0%)
TransientFail [Can+]	Completely unstable	Does not work (0%)
V-E-O [V-E]	Perfect (100%)	Perfect (100%)

4.2 *gem5*

The final goal of the internship being to study the simulation of a transient attack, development using *gem5* is a necessary step. A contribution to *gem5* – maybe not to the project itself, but a development based on *gem5* in the large meaning – can be of two kinds. The first one is a core modification – or a completely new object –, which means a modification of the *C++* code of *gem5*, where all the logic of simulated components is computed. The second one is a configuration modification, which means a modification of the *Python* code that models a system – or the creation of a new one. We understand that the first type is way more complicated, critical, longer and tedious than the second one – but it can be necessary. The nature of the second one can vary a lot, from a very simple modification of a predefined system to the creation of a complete whole system of several hundreds lines of code.

Some predefined *gem5* systems are available into the source repository. They are kept mainly for two reasons:

1. They are used in automated tests;
2. They are given as *gem5* usage examples.

These systems are enough to perform very basic experiments – for example, testing the validity of a binary, a command or a predefined high-level configuration like the base class of the processor. When it comes to perform more complex experiments with low-level customization – caches microarchitecture, branch predictor, etc. –, you have to build your own script defining a system – from scratch, or based on predefined systems.

Because we fall into the category of “more complex experiments”, we will describe how we have written our own system configuration. First, we will present the experiments that we have done about *Spectre* on *gem5*. Then, we will cover the design choices of our system. Finally, we will detail the system development and finally, present the final result.

4.2.1 First Experiments

Before we dig into the details or our own system, we observed that we were able to obtain a more complex behavior of *Spectre* on a *gem5* system than the simple “works / does not work” outcome. The simulator being mostly deterministic and used with a simple system at the beginning, we could reasonably expect the simple behavior mentioned above.

A lot of debugging techniques exist to debug *gem5* itself or the code running under *gem5*. Describing all of them would be out of the scope of this report, but we will demonstrate one of them – maybe one of the more impressive ones. *gem5* integrates a functionality allowing to see the pipeline of a simulated processor. Thanks to some external programs like *Konata* [Shi], we can visualize all the instructions that have flowed into the pipeline. For each of them, we can identify the cycle of each processing step, and even see if an instruction have been fetched then flushed or speculatively executed. This last point is particularly interesting to us, since it is the main point of the *Spectre* attack after the branch predictor misprediction.

With *Konata*, we observed multiple *Spectre* scenarios, detailed in the Appendix C.1. The *Spectre* used here was the *PHT-SA-IP* version of the *TransientFail* repository. The system was our own system at a very basic state – with more or less no customization compared to the *gem5* base objects –, with a `DerivO3CPU` running in system-call emulation mode. Some observations have been made, but two distinct cases were observed: the attack works as expected, and the mistraining of the branch predictor

Table 4.5: Summary of detailed processors of *gem5*.

Processor	Inherit	Specialization	Cache Prefetcher	OoO	Pipeline	Branch Prediction
`MinorCPU`	`BaseCPU`	General	None	No	4-stage	`TournamentBP`
`HPI`	`MinorCPU`	ARMv8-A	`StridePrefetcher`	No	4-stage	`TournamentBP`
`DerivO3CPU`	`FullO3CPU`	General	None	Yes	7-stage	`TournamentBP`

is not successful, which leads to no transient instruction executed. The first case allows us to have a better understanding on both how the attack works and how *gem5* works, since we can see *gem5* internals – when reading the logs of executed instructions – and the *Spectre* micro-architectural effect – when visualizing it with *Konata*. The second case is useful to understand why *Spectre* could fail, and identify one from the multiple possible reasons.

4.2.2 Design Choices

For our system, there were two kinds of design choices. The first kind is a functional one. Indeed, using *gem5* for research can be extremely inconvenient if the used system is not carefully crafted for a specific type of workflow in mind. Some functional requirements will be presented:

Handle both *System-call Emulation (SE)* and *Full-system Simulation (FS)* modes Depending on the tested binary, on the required accuracy, on the time left for the experiment and a lot of other factors, we could want one mode or the other. The majority of predefined *gem5* systems only handle one mode, but not both. Our system script was designed to be able to use both modes.

Fast-forwarding ability When using the *FS* mode, boot-up a system or make the system reach a specific state can be *very* long – hours, at least – if we use a detailed processor, like the `DerivO3CPU`. It is unpractical to wait a time as long at every boot of a system before an experiment. This is where the *fast-forwarding*, also called *switchover* or *sampling*, is very useful. This functionally, if implemented into the system script, allows to boot-up a system or perform specific task on it with a fast processor, *i.e.*, not detailed processor. Then, when the system is in the desired state for the experiment, a snapshot of the system is taken. Later, we can restore the snapshot but with the detailed and slow processor to perform the experiment.

Handy experiments scripts When using the *FS* mode, we provide a kernel and a disk image to *gem5* for running the system. However, passing data *via* a disk image can quickly become inconvenient and slow. Thus, our script was designed to handle disk and experiments management easily, with a set of external script to handle disk images.

Now that we have exposed the functional requirements, we can discuss the architectural ones. In Table 4.5, we summarize interesting and detailed processors of *gem5* that we could use for our system. Note that the cache prefetcher and the branch predictor are only default settings that can thus be easily changed. It is not the case for the OoO execution handling or the pipeline stages. Note also that the pipeline of these processors largely differs, the one of the `DerivO3CPU` being far more representative of today's processors:

4-stage Fetch, Decode Micro/Macro, Execute;

7-stage Fetch, Decode, Rename, Issue/Execute/Writeback, Commit.

We decided to use the `DerivO3CPU` as a base processor for our simulation based on the work of Endo et al. [ECC14] who simulated an *ARM Cortex-A8* and an *ARM Cortex-A9* processors with *gem5* and an `O3CPU`, and Gutierrez *et al.* [Gut+14] who simulated an *ARM Cortex-A15* processor with *gem5* and an `O3CPU` in a survey of errors categorization using *gem5*, as well as the *gem5* documentation [Lowb] [Lowc]. This processor should be less user-friendly at customizing it than the `MinorCPU`, but also more accurate since we have an OoO execution and a deeper pipeline, closest to the pipeline of our real processor (as seen in Section 2.1.3).

We have analyzed *gem5*'s implementation of branch predictors. All implemented algorithms are covered in background Section 2.1.2. Under the *gem5* source tree [5], we can find: a *Single-Level Bimodal 2-Bit Counter*, a *Two-Level Global History-Based Bi-Mode*, a *Two-Level Hybrid Tournament*, a *LTAGE* and its derivatives – *TAGE-SC-L* –, and finally a *Perceptron-based* algorithm. By default, we also have a *BTB*, a *RAS*, and an *indirect predictor* which are implemented and used. Note that we have not found any implementation of a *static predictor*. As *side-predictor*, we also have an optional *loop predictor*. To match as closely as possible our *ARM Cortex-A72* – which uses a *Simple Two-Level Global History-Based* predictor –, we have chosen to use the *gem5*'s *Bi-Mode* predictor, accompanied by its *RAS* and *indirect predictor*. Despite the fact that the *Bi-Mode* predictor separate *mostly-taken* and *mostly-not-taken* branches into two different PHTs, the *Spectre* attack trains the attacked branch as a *mostly-taken* branch. Consequently, this branch should only be predicted from the *mostly-taken* PHT and, hence, our *Bi-Mode* predictor should act mostly like the branch predictor of our real processor.

4.2.3 System Development

The system and the processor were two important points during the development. The modifications to the latter was inspired by the `HPI` and the `O3_ARM_v7a` [6] processors, when some micro-architectural data for the *ARM Cortex-A72* processor were missing from the *ARM Cortex-A72* manual [16]. Our system development was inspired by the `starter_se` [7] and the `starter_fs` [8] systems, which are *SE* and *FS* mode systems respectively, made by *ARM* developers.

Bringing the *FS* mode into our system script has been a very complex task. Some documentation is available on how to use a predefined script in *FS* mode, but when it comes to the development of a script like this, there is no documentation at all. Thus, the development was a constant trial and error cycle, with interleaved phases of reading the *gem5* source code and pure development. Consequently, our system contains a lot of comments and documentation on how to write a *FS* mode script. It could be interesting to write a guide on how to do it, but it would be out of scope of this report. Instead, we will just describe, from a high-level point of view, how to create a *FS* mode script:

1. In our *Python* script, we get the system disk image path, kernel image path and arguments.

2. Our system must inherit from the `ArmSystem` class. Since we want to use both SE and FS mode in the same script, with have to use dynamic inheritance here. The `ArmSystem` class define high level attributes for the ARM architecture, like security, virtualization and cryptographic extensions, *GIC* address, exception level, and other ARMv8 specific parameters.

3. Our system must use the *RealView VExpress* platform, hence containing a child object with the `VExpress_GEM5_V1` object. This platform defines the memory mapping, configure the interrupts, declare the *GIC*, and a lot of low-level ARMv8 specific parameters. This platform contains the majority of on-chip and off-chip devices and memories.

4. To control the system when it will be booted up, we have to connect to it a *gem5* terminal and a *VNC* server. They are represented by *gem5* objects, and used to communicate with the host system.

5. Unlike SE mode where our processor load the binary as a workload, here, this is our system which has to load the kernel as a workload.

6. We have to load the system disk image as a *VirtIO PCI* block device, which is a virtual block device accessible inside the *gem5* guest system mapped to a real file on the host system. We have to do the same with the workload image, containing our binary to run on the system for the experiment.

[5] In the `src/cpu/pred` directory.
[6] Found in the `configs/common/cores/arm/O3_ARM_v7a.py` file from the *gem5* source tree.
[7] Found in the `configs/example/arm/starter_se.py` file.
[8] Found in the `configs/example/arm/starter_fs.py` file.

Table 4.6: Taxonomy of tested *Spectre-PHT* implementations on *gem5*.

Implementation \ Target	*gem5* FS `O3CPU` `BiModeBP`
IRISA	Stable (55%)
TransientFail [Can+]	Completely unstable
V-E-O [V-E]	Perfect (100%)

7. We have to connect all on-chip and off-chip Input-Output (IO) devices and memories to the system. It requires a *gem5* bridge, used to interconnect different types of buses.

8. Finally, we have to set up the boot loader to boot the system and the *Linux* kernel with its command-line parameters. We do not forget to declare the *DTB* file, a hardware configuration file passed to the kernel.

During the development, we also had to face some *gem5* bugs. In this case, our system seemed to hang past a certain point, without further information. After some investigation and thanks to *Konata* , we found out that *gem5* was skipping some instructions in an unintended way, which made the system fall into an infinite loop. As a result, we had one *Spectre* binary which worked on our real system, but did not work on *gem5*.

Our final system works under three modes: *SE* mode with a detailed and slow processor, *FS* mode with a simple and fast processor and *FS* mode with a detailed and slow processor. The scheme of the system in his latest described mode is presented in Figure 4.4. In this figure, we can see the different components of the system. Each component is labeled by its name at the top while its class is displayed just under. The most important component here is the `cpu_cluster` object of the class `ARM_A72_Cluster`, which is the package of our processor and holds inter-core components as cores themselves. Each core is represented by a `cpux` object, here, a 4-core processor. We can distinguish the two levels of caches, as the *TLB*, the table walker and the walker cache. The second most important component is the platform, here the `VExpress_GEM5_V1` under the `realview` object. This object contains on-chip and off-chip memories and devices, required to simulate an entire system. We observe some controllers (power, energy, PCI, interrupts, LCD), important memories (SRAM, flash, boot) and some other components. Finally, the whole system is tied up by some memory buses and interconnection bridges. The usage of our system is presented in Appendix B.2.

4.2.4 Results

In this section, we will present the results we obtained with our *Spectre* and other implementations on our *gem5* system. However, the results are not complete. First, some adjustments will be done after the writing of the report but before the end of the internship. Secondly, experiments with *gem5* take a very long time, thus some results are in progress but not ready yet.

Table 4.6 describes the results of the different implementations on our system. We have used the 3 implementations that we described before in our experiments, in Section 4.1.4. From this table, we can distinguish several behaviors regarding the implementation:

IRISA For our implementation, the results are stable, which is one of our requirements. However, our implementation is not as efficient on *gem5* as on our real *Raspberry Pi* – or rather, *gem5* is not so faithful regarding our *Raspberry Pi*. Indeed, we only approximately recover 55% of the secret's bytes. The efficiency that we got on *gem5* is in the middle compared to the results on our low-end and high-end real *ARM* processors. Figure 4.5 shows our *Spectre* accuracy on *gem5*, where we can see that *gem5* is able to reproduce some instabilities of the real hardware, but not as much. There is a lot less of outliers, and also a lot less of runs – indeed, a *Spectre* run on *gem5* can take nearly 1 hour.

TransientFail [Can+] An interesting result with the *TransientFail* / *IAIK* implementation is that its inconsistency and pathological behavior is also reproduced on *gem5*. It would have been

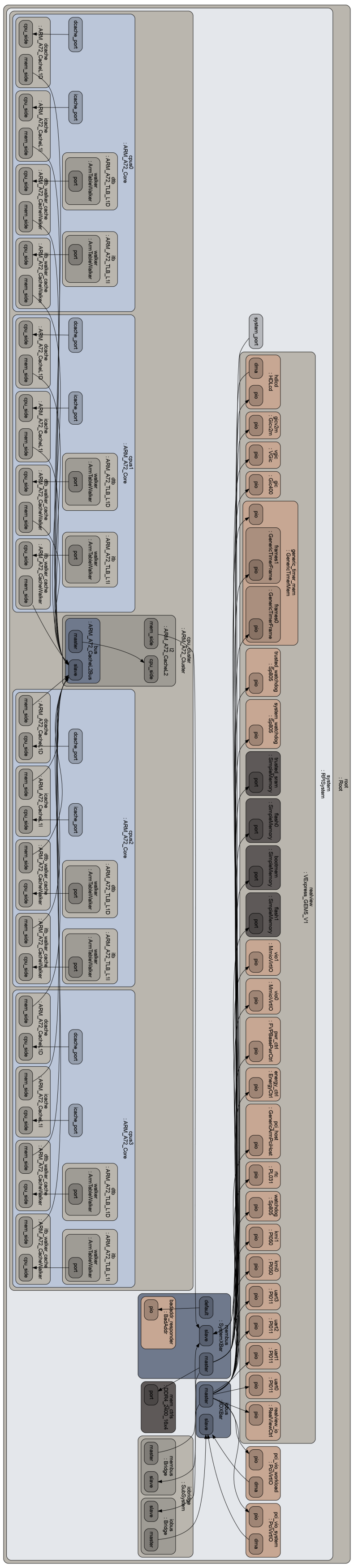


Figure 4.4: Scheme of the final *gem5* configuration in full-system simulation.

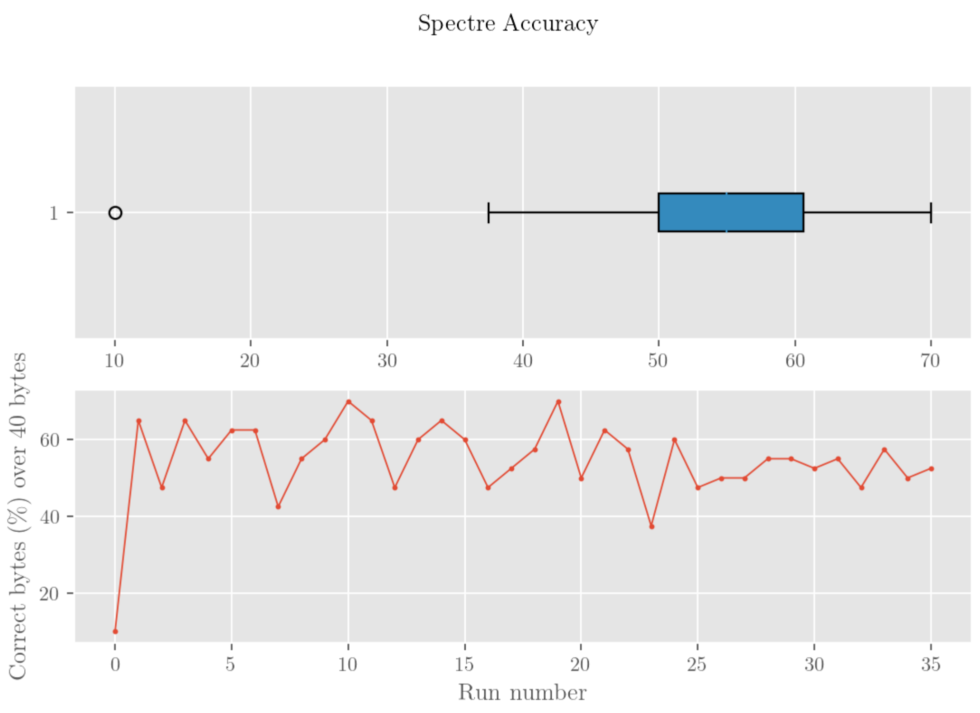


Figure 4.5: Percentage of correctly recovered bytes on *gem5* of our *Spectre* implementation.

interesting to understand the micro-architectural reason of this behavior with *gem5*, but our experiments have not successfully highlighted the root cause.

V-E-O [V-E] This implementation with the *counter thread* is as efficient on *gem5* than our real hardware, for both our high-end and low-end ARM processors.

However, we were not able to get micro-architectural counters – the ones we get with `perf_event`, the kernel API – with *gem5*. Thus, we do not have our previous plots for the *gem5* system, except for *accuracy* and *difficulty* metrics. We know that the ARM PMU is implemented in *gem5*, but unfortunately, in the time allowed, we have not been able to make it work – the only resources we had was the *gem5* source code, without documentation nor examples. At the time of writing, we think we are on the right way to make it work – maybe before the end of the internship, we will be able to compare the micro-architectural characteristics between our runs on the real hardware and the ones on *gem5*.

As a conclusion for these results, we see that *gem5* is able to reproduce the *Spectre* attack, even with different branch predictors – not all experiments were reported in the report. We can increase the accuracy of the attack by making a correct *gem5* configuration, however, it is not a perfect fit with our real hardware – which is excepted, micro-architectural simulation being so complex. Nonetheless, *gem5* demonstrate great capacities to simulate some interesting behavior also observed on the real hardware and to get closer of our real hardware under certain conditions. This is promising for future work and investigations.

4.3 Discussion

Our work is a preliminary attempt in the simulation of transient attacks with *gem5*. Due to the lack of documentation on the simulator and the difficulty to obtain a stable attack on our real hardware, the *gem5* part of the internship isn't as advanced as we initially wanted. Nonetheless, this preliminary work is important to have a clear picture of what can be done and of the future challenges. Moreover, there are many interesting avenues for future work:

- Expand the *Spectre* binary with new metrics. Indeed, currently only 6 metrics have been added. Others can be easily added to cover others use cases. For example, to measure the efficiency of the attack, we could compute a bandwidth and compare this metrics across implementations or systems.

- Expand the *Spectre* binary with new core attacks. During our development, we tried to make our implementation modular enough to add other *Spectre* variants. This will allow, from the same consistent assembly and metrics code base, measure and reproduce different kind of attacks. Later, it could be used to create a taxonomy of reproducible attacks on *ARM* real hardware and/or *gem5*.
- Expand the processor of the *gem5* system with detailed micro-architectural information about the branch predictor. Currently, we do not have enough information about the branch predictor of the *Cortex-A72* to have a *gem5* system which would be extensively accurate.
- From the previous point, we could reverse-engineer or identify the branch predictor automatically, based on some micro-benchmarks for example. Then, we could use an abstract model to represent it and later reproduce it automatically in *gem5*. This is however a more long-term research direction.

4.4 Limitations

Despite all our efforts, our work suffers from some limitations. We think that the main point to discuss is the generalization. As we worked on a (near) single piece of hardware, it is possible that our *Spectre* implementation would not be so usable on others systems. This remains to be tested. On the other side, our *gem5* is built for the *Cortex-A72* in mind. It is possible that, for it too, it would be a bit too specific. Consequently, our work remains to be extended at a wider scale.

Conclusion

FACING to the development of new kinds of vulnerabilities, computer security researchers have to develop new methods to study and reproduce them. One type of attack targets the micro-architecture of the computer's microprocessor. The work realized under this internship aims to address these initial questions on this kind of attack. The main work is a preliminary one, being a complex subject and not well-studied yet in the research community. During this internship, we have first developed a new implementation of the *Spectre* attack, in order to obtain an attack more efficient, stable, useful and reproducible. Then, we have tried to build a system on a cycle-accurate simulator, *gem5*, designed to simulate this transient attack. At this time, the work on the *gem5* system is not finished yet – the end of the internship will be completely devoted to it. We also described possible future research directions.

I wish to say how this internship brought me many skills and opportunities. Firstly, being in Rennes and in a new laboratory, with a complete new team that I did not know before was very rewarding. Secondly, this field is a good one to use and learn multidisciplinary knowledge. Furthermore, during the internship, I have been in contact with other security researchers to discuss their implementation of *Spectre*. Also, I have been able to report some bugs or make some suggestions to *gem5* and *Konata* developers. Finally, starting a Ph.D. thesis next year, this internship will be extremely valuable in term of research methodology and organization.

Appendices

A Setup

A.1 Installation of the *Raspberry Pi*

This is how to install the v2020.2a of *Kali Linux* on a *Raspberry Pi*.

1. First, download and extract the operating system image:

```
zip="kali.xz"
img="kali.img"
wget -O "$zip" "https://images.offensive-security.com/arm-images\
/kali-linux-2020.2a-rpi3-nexmon-64.img.xz"
xzcat "$zip" > "$img"
rm -f "$zip"
```

2. Connect the *MicroSD* card – without mounting it, in our case we unmount it – and locate its block device name, then "burn" the image on the *MicroSD* card.

```
# Identify the block device name.
lsblk
umount /dev/sdd1
sudo dd bs=4M if="$img" of=/dev/sdd status=progress conv=fsync
rm -f "$img"
```

3. Finally, plug the *SD* card and the power cable into the *Raspberry Pi*. Find it is *IP* address with `nmap -sn 192.168.1.0/24` – or use the `ZeroConf` service if available – and then connect to it, like this:

```
ip="192.168.1.5"
ssh kali@$ip
```

4. We can then update our system and install useful software:

```
sudo apt-get update && sudo apt-get upgrade
sudo apt-get -y install tree dfc hwloc inxi stress locate linux-cpupower git \
     gdb rsync htop
```

5. Allow connections to the root user by setting it a password:

```
sudo su root
passwd
toor
toor
exit
```

6. Finally, copy our *SSH* key on the *Raspberry Pi* to easily and quickly connect to it next time:

```
exit
ssh-copy-id kali@$ip
ssh-copy-id root@$ip
```

7. Note that:

   - We can log in with username `kali` and password `kali`.
   - *SSH* is enabled by default.
   - Kernel headers and sources are already provided.

A.2 Build a Kernel Module for Accessing the *ARM PMU*

The *ARM PMU* can be accessed by two methods:

- **Indirect access** by the *Linux* kernel `perf_event` interface, which provides a standardized way of accessing to the most common *PMCs* across different *ARM* architecture. The downside is the *API* overhead.

- **Direct access** by reading the *PMC* directly with assembly code embedded into a *C* program (in our case).

Direct access is interesting during the development and debugging phase of some attacks. Since it requires a kernel module – hence, `root` permissions –, it is not usable in a real life. The `PMCCNTR_EL0` is an interesting counter, since it gives a very precise time resolution (cycle-accurate). A lot of interesting counters are accessible by the *PMU*.

To read the `PMCCNTR_EL0`, the *C* code is as simple as that:

```
asm volatile("MRS %0, PMCCNTR_EL0" : "=r"(result));
```

However, if you execute this instruction without any precaution, you will face an `Illegal instruction` exception and the software will be aborted – even if the exception level required is `EL0`, which corresponds to the user space.

This is a simple explanation of what is a kernel module, from a reference book [SBP09]:

> What exactly is a kernel module? Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

This is simplest kernel module, to illustrate how it works:

```c
#include <linux/module.h>
#include <linux/kernel.h>

int hello_init(void)
{
    pr_alert("Hello World!\n");
    return 0;
}
void hello_exit(void)
{
    pr_alert("Goodbye World!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Accompanied by its Makefile:

```
obj-m := hello.o
```

Compiled like this:

```
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

And finally executed by inserting it:

```
sudo insmod hello.ko
```

We can see with `dmesg | tail -1` that the `Hello World!` message has appeared in the kernel logs. What is happened here is that the kernel – the `kmod` daemon which handle kernel modules –, with higher privilege than any user, executed our own code on demand. This can be exploited – in an intended way – to control processors features.

In order to access to the *ARM PMU*, a kernel module – written by a computer enthusiast – explained on this blog [Sunb] and available on *GitHub* [Suna] can be used. The detailed explanation of this kernel module goes beyond of the scope of this report, but some hints can be given.

From the *ARM Architecture Manual* [20] (Section D13.4.17, page 3714), the `PMUSERENR_EL0` register "Enables or disables EL0 access to the Performance Monitors". The mapping of the register – which bit corresponds to which functionally – is described here. From the manual, we understand that three bits must be set to 1: `ER`, `CR`, `EN`. These bits enable, respectively, access to the event counter, to the cycle counter, and to the previously specified registers from user land (which is `EL0` in *ARM* terminology).

In the kernel module presented above, this corresponds to this snippet:

```c
#define ARMV8_PMUSERENR_EN_EL0  (1 << 0) /*  EL0 access enable */
#define ARMV8_PMUSERENR_CR      (1 << 2) /*  Cycle counter read enable */
#define ARMV8_PMUSERENR_ER      (1 << 3) /*  Event counter read enable */

[...]

/*  Enable user-mode access to counters. */
asm volatile("MSR PMUSERENR_EL0, %0" : :
             "r"((u64)ARMV8_PMUSERENR_EN_EL0|ARMV8_PMUSERENR_ER|ARMV8_PMUSERENR_CR));
```

Conceptually, it is not more complicated than that. In practice, you will also have to reset the counters, start and stop them, and so on. But the principle is the same: write what you want into the appropriate bit of the appropriate register at the appropriate time. Note that the code for enabling access to the *PMU* needs to be executed on each core – the *Linux* kernel provides facilities to do that.

A.3 Installation of *gem5*

For a full detailed procedure, complete list of possibilities and requirements, we can check the official documentation [Lowa]. We will clone and install *gem5* from the *Google* repository:

```
git clone https://gem5.googlesource.com/public/gem5
```

Install some recommended packages to obtain a speed-up during the simulation:

```
sudo apt-get install libgoogle-perftools-dev
```

By default, we compile it in `opt` mode which is the best default, for the `ARM` architecture, using `8` cores and *Python* 3. *Python 2* was supported until recently, but since it arrives at his end of life, trying to use it is a bad idea. If you need to debug *gem5*, use `debug` instead of `opt` mode. If you need a fast simulation, you can use `fast` mode instead, but you will not able to get any debugging output from *gem5* itself and runtime error checking.

```
cd gem5
scons PYTHON_CONFIG=python3-config build/ARM/gem5.opt -j 8
```

If your build fails because of some warnings, comment the `'-Werror',` line in the `SConstruct` file:

```
# Treat warnings as errors but white list some warnings that we
# want to allow (e.g., deprecation warnings).
main.Append(CCFLAGS=['-Wno-error=deprecated-declarations',
                     '-Wno-error=deprecated',
                     #'-Werror',
                    ])
```

If your build fails because `/usr/bin/env: 'python': No such file or directory`, install `python3` and/or create a symbolic link like this:

```
sudo ln -s /usr/bin/python3 /usr/bin/python
```

B Development

B.1 Usage of *Spectre*

In this appendix, we will present how our *Spectre* binary can be used. First, we can display the help to have the relevant information:

```
code/poc/spectre_arm/spectre --help
```

```
Usage: spectre [OPTION...]
Spectre -- A Spectre implementation useful for research

  -c, --cache_threshold=NUMBER   Cache threshold separating hit and miss
                             (default: automatically computed)
  -l, --loops=NUMBER         Number of loops (training and attack) per attempts
                             (default: 30)
  -m, --meta=NUMBER          Number of meta-repetition of Spectre (default: 1)
  -q, -s, --quiet, --silent  Don't produce the header for csv
  -t, --tries=NUMBER         Number of attempts to guess a secret byte
                             (default: 999)
  -v, --verbose              Produce verbose output
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Report bugs to <pierre.ayoub@irisa.fr>.
```

Here is a demonstration of a simple and convenient usage. Note that the following code can be wrapped in a `for` loop, in order to perform meta-meta repetition – this will increase the accuracy of the data treatments in the *Python* script.

```
code/poc/spectre_arm/spectre -m 10 -l 100 -t 999
```

This command runs 10 *Spectre* attacks, with 999 tries to guess each byte, and with a number of 100 training loops per byte per try – with a proportion of 5 training runs per attack run. The total number of call of the `victim_function()` – containing the attacked branch – can be computed as meta $*$ tries $*$ loop, here `999000`.

Here is a representation of the file organization:

```
asm.c
asm.h
main.c
Makefile
perf.c
perf.h
spectre
spectre_pht_sa_ip.c
spectre_pht_sa_ip.h
util.c
util.h
```

`asm.{c,h}` contains the assembly *ARMv8* code. `main.c` contains the general algorithm handling experiments, arguments and metrics. `perf.{c,h}` contains the code managing the source of the metrics with `perf_event`. `spectre` is the final binary. `spectre_pht_sa_ip.{c,h}` is the implementation of the *PHT-SA-IP* version of the *Spectre* core. Finally, `util.{c,h}` contains miscellaneous functions.

B.2 Usage of *gem5*

This appendix will briefly present the usage of our *gem5* system. Like the *Spectre* binary, the system is self-documented:

```
gem5.opt RPIv4.py --help
```

```
usage: RPIv4.py [-h] [-v] [--num-cores NUM_CORES] [--se] [--fs]
                [--fs-kernel FS_KERNEL] [--fs-disk-image FS_DISK_IMAGE]
                [--fs-workload-image FS_WORKLOAD_IMAGE]
                [--fs-restore FS_RESTORE]
                [se-command [se-command ...]]

Raspberry Pi 4 Model B Rev. 1.1 - Syscall emulation & Full-system simulation
Script based on a real Raspberry Pi system. It is shipped with a "reproduced"
ARM Cortex-A72 CPU. The intended use is security research. It can be used both
in system-call emulation or full-system simulation. For the full-system
simulation mode only, first boot your system and create a checkpoint where the
used CPU will be the atomic one. Only then, restore you system from your
checkpoint, where the CPU used will be the detailed one. When passing
filenames in arguments of the script, please be sure that your M5_PATH
environment variable is set accordingly.

positional arguments:
  se-command            Command(s) to run (multiples commands are assigned to
                        a dedicated core)

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Print detailed information of what is done
  --num-cores NUM_CORES
                        Number of CPU cores (default = 1)
  --se                  Enable system-call emulation (must provide 'command'
                        positional arguments)
  --fs                  Enable full-system emulation (must provide '--fs-
                        kernel' and '--fs-disk-image' options)
  --fs-kernel FS_KERNEL
                        Filename of the Linux kernel to use in full-system
                        emulation (searched under '$M5_PATH/binaries'
                        directory)
  --fs-disk-image FS_DISK_IMAGE
                        Filename of the disk image containing the system to
                        instantiate in full-system emulation
  --fs-workload-image FS_WORKLOAD_IMAGE
                        Filename of the disk image containing the workload to
                        mount in full-system emulation
  --fs-restore FS_RESTORE
                        Path to a folder created by "m5 checkpoint" command to
                        use for restoration
```

This is a simple usage example of running a *Spectre* in *SE* mode :

```
gem5.opt RPIv4.py -v --se "spectre_arm/spectre -l 30 -t 50 -m 10"
```

This is a more advanced example usage of running a *Spectre* in *FS* mode .

- Define useful functions and variables:

```
# Kernel and disk images are respectively automatically found in
# "$M5_PATH/{binaries,disks}" directories.

# Full system disk image.
gem5_disk="linaro-minimal-aarch64.img"
# Workload disk image.
gem5_workload="workload.img"
# Linux kernel image.
gem5_kernel="vmlinux.arm64"

# Boot the full-system simulation mode with a fast CPU.
gem5_fs_boot() {
    gem5.opt -q -d fs_mp_boot RPIv4.py -v --num-cores=4 --fs \
             --fs-kernel=$gem5_kernel --fs-disk-image=$gem5_disk \
             --fs-workload-image=$gem5_workload
}

# Print the path of the first restore point under the current directory.
gem5_fs_find_restore() {
    find . -name "cpt.*"
}

# Restore the full-system simulation mode with a detailed CPU.
gem5_fs_restore() {
    gem5.opt -q -d fs_mp_restore RPIv4.py -v --num-cores=4 --fs \
             --fs-kernel=$gem5_kernel --fs-disk-image=$gem5_disk \
             --fs-workload-image=$gem5_workload \
             --fs-restore=$(gem5_fs_find_restore)
}
```

- Boot the full-system with a fast processor:

```
gem5_fs_boot 2>/dev/null &
```

- Connect to it, wait that the boot process complete (dozens of minutes, you will reach a prompt at a certain point), and create a checkpoint:

```
gem5/util/term/m5term localhost 3456
m5 checkpoint
```

- We can then end our full-system simulation:

```
kill %gem5_fs_boot
```

- Boot the full-system with the detailed and slow processor:

```
gem5_fs_restore 2>/dev/null &
```

- Connect to it, wait that the restore process complete (few minutes, you will reach a prompt at a certain point), and mount the workload disk – a simple `ext4` disk image, prepared with one of our script (not covered here):

```
gem5/util/term/m5term localhost 3456
mkdir -p workload
mount /dev/vdb1 ./workload
```

- We can finally issue our *Spectre* run:

```
gem5/util/term/m5term localhost 3456
cd workload
./spectre
```

C Experiments

C.1 Visualize *Spectre* on *gem5* with *Konata*

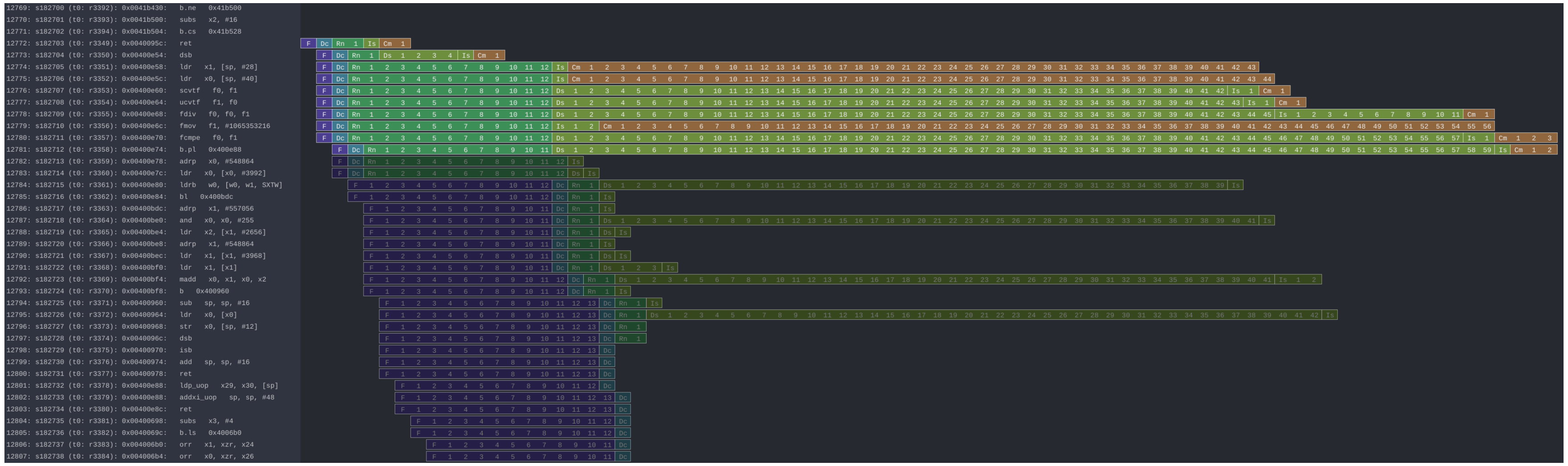
The context of this appendix is explained in Section 4.2.1. We propose an observation of a successful *Spectre* attack on a *gem5* system, with *Konata* , by describing Figure C.1.

Just before, we will briefly describe what information are displayed on a *Konata* output. Each line corresponds to one instruction. The screen is separate in two: information about the instructions at the left, and a visual representation of the instructions execution at the right. On the foremost left, the line number is displayed, then, a unique number identifying the instruction in the pipeline . There is unimportant text between a pair of parentheses, and finally, the address of the instruction – in the binary – with its assembly equivalent. On the visual representation at the right, we can see how has evolved the instruction inside the pipeline. A colorful instruction is a completely executed one, whereas a gray-shaded instruction is one which was not architecturally executed – but for the microarchitecture, the instruction was more or less executed, depending on when it was flushed away from the pipeline . The different stage of execution in the pipeline are distinguished by the used color on the line, also corresponding to a mnemonic [1]. Then, at each cycle and for each step, a counter incremented one by one is displayed, representing the latency of the pipeline stage for this instruction.

On the figure, we can see that the first instruction executed, the `ret` at address `0x40095c`, is the return of the `flush()` *C* function – which is charged of flushing the operand used later in the attacked branch – into the `access_array()` function – which is the victim function with the potentially malicious array access. The speculative execution happened just after the instruction with the `b.pl 0x400e88` mnemonic, which is the attacked branch. The transient read in the array which will be probed during the *Flush+Reload* phase is the `ldr x0, [x0]` at address `0x400964`, which is completed but never committed to the architectural state. Unfortunately from a security perspective, this transient read has modified the microarchitectural state of the processor, leading to the indirect leak of data. We know which assembly instruction corresponds to which *C* statement by some preliminary steps before the *Konata* analysis, where we have disassembled and analyzed the generated binary.

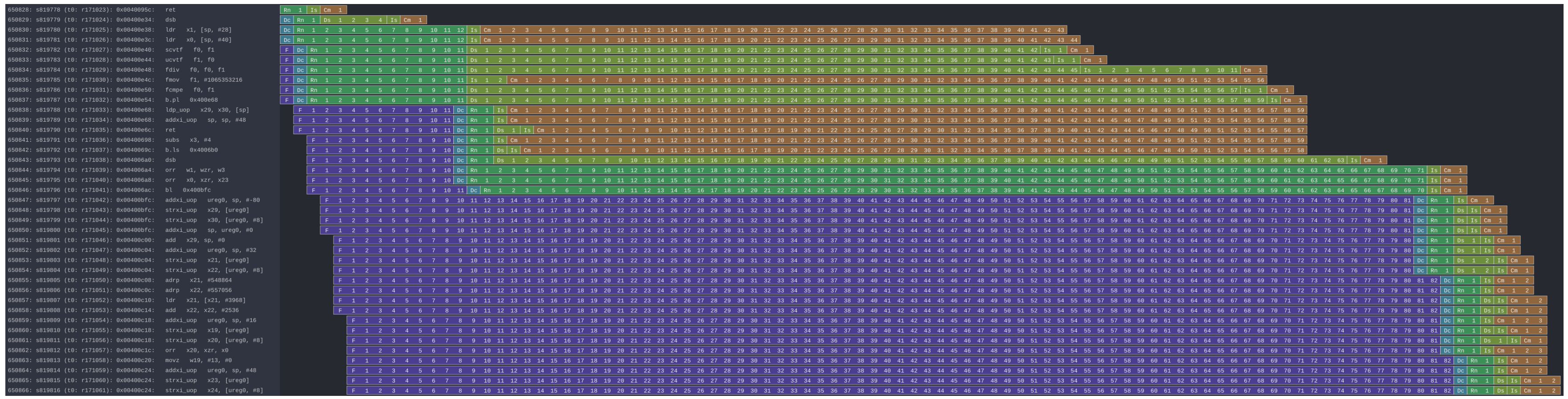
Another interesting observation with *Konata* is when *Spectre* does not defeat the branch predictor in the intended way. It is shown in Figure C.2. On this figure, we can see that the branch `b.pl 0x400e88` in `access_array()` is predicted *not taken* while it is in reality *not taken* – because it is the attack, thus causing a malicious read if it is *taken*. We saw then that the executed code corresponds to the `cache_decode_pretty()` function, which try to read the cached bytes in the probing array – unfortunately, it will only find the bytes used in the training phase since there was no transient execution. This case allows us to identify the microarchitectural root cause of a *Spectre* fail: “Is the noise on the caches? Is the branch predictor?”. Here, it is clearly the branch predictor which has resisted to the *Spectre* training phase.

[1] F : Instruction fetch ; Dc : Instruction Decode ; Rn : Rename ; Ds : Dispatch ; Is : Issue ; Cm : Completion of execution ; End of Cm stage : Retire.



```
12769: s182700 (t0: r3392): 0x0041b430:   b.ne   0x41b500
12770: s182701 (t0: r3393): 0x0041b500:   subs   x2, #16
12771: s182702 (t0: r3394): 0x0041b504:   b.cs   0x41b528
12772: s182703 (t0: r3349): 0x0040095c:   ret
12773: s182704 (t0: r3350): 0x00400e54:   dsb
12774: s182705 (t0: r3351): 0x00400e58:   ldr   x1, [sp, #28]
12775: s182706 (t0: r3352): 0x00400e5c:   ldr   x0, [sp, #40]
12776: s182707 (t0: r3353): 0x00400e60:   scvtf   f0, f1
12777: s182708 (t0: r3354): 0x00400e64:   ucvtf   f1, f0
12778: s182709 (t0: r3355): 0x00400e68:   fdiv   f0, f0, f1
12779: s182710 (t0: r3356): 0x00400e6c:   fmov   f1, #1065353216
12780: s182711 (t0: r3357): 0x00400e70:   fcmpe   f0, f1
12781: s182712 (t0: r3358): 0x00400e74:   b.pl   0x400e88
12782: s182713 (t0: r3359): 0x00400e78:   adrp   x0, #548864
12783: s182714 (t0: r3360): 0x00400e7c:   ldr   x0, [x0, #3992]
12784: s182715 (t0: r3361): 0x00400e80:   ldrb   w0, [w0, w1, SXTW]
12785: s182716 (t0: r3362): 0x00400e84:   bl   0x400bdc
12786: s182717 (t0: r3363): 0x00400bdc:   adrp   x1, #557056
12787: s182718 (t0: r3364): 0x00400be0:   and   x0, x0, #255
12788: s182719 (t0: r3365): 0x00400be4:   ldr   x2, [x1, #2656]
12789: s182720 (t0: r3366): 0x00400be8:   adrp   x1, #548864
12790: s182721 (t0: r3367): 0x00400bec:   ldr   x1, [x1, #3968]
12791: s182722 (t0: r3368): 0x00400bf0:   ldr   x1, [x1]
12792: s182723 (t0: r3369): 0x00400bf4:   madd   x0, x1, x0, x2
12793: s182724 (t0: r3370): 0x00400bf8:   b   0x400960
12794: s182725 (t0: r3371): 0x00400960:   sub   sp, sp, #16
12795: s182726 (t0: r3372): 0x00400964:   ldr   x0, [x0]
12796: s182727 (t0: r3373): 0x00400968:   str   x0, [sp, #12]
12797: s182728 (t0: r3374): 0x0040096c:   dsb
12798: s182729 (t0: r3375): 0x00400970:   isb
12799: s182730 (t0: r3376): 0x00400974:   add   sp, sp, #16
12800: s182731 (t0: r3377): 0x00400978:   ret
12801: s182732 (t0: r3378): 0x00400e88:   ldp_uop   x29, x30, [sp]
12802: s182733 (t0: r3379): 0x00400e88:   addxi_uop   sp, sp, #48
12803: s182734 (t0: r3380): 0x00400e8c:   ret
12804: s182735 (t0: r3381): 0x00400698:   subs   x3, #4
12805: s182736 (t0: r3382): 0x0040069c:   b.ls   0x4006b0
12806: s182737 (t0: r3383): 0x004006b0:   orr   x1, xzr, x24
12807: s182738 (t0: r3384): 0x004006b4:   orr   x0, xzr, x26
```

Figure C.1: *Konata* observation of a successful *Spectre* attack.



```
650828: s819778 (t0: r171023): 0x0040095c:   ret
650829: s819779 (t0: r171024): 0x00400e34:   dsb
650830: s819780 (t0: r171025): 0x00400e38:   ldr   x1, [sp, #28]
650831: s819781 (t0: r171026): 0x00400e3c:   ldr   x0, [sp, #40]
650832: s819782 (t0: r171027): 0x00400e40:   scvtf   f0, f1
650833: s819783 (t0: r171028): 0x00400e44:   ucvtf   f1, f0
650834: s819784 (t0: r171029): 0x00400e48:   fdiv   f0, f0, f1
650835: s819785 (t0: r171030): 0x00400e4c:   fmov   f1, #1065353216
650836: s819786 (t0: r171031): 0x00400e50:   fcmpe   f0, f1
650837: s819787 (t0: r171032): 0x00400e54:   b.pl   0x400e68
650838: s819788 (t0: r171033): 0x00400e68:   ldp_uop   x29, x30, [sp]
650839: s819789 (t0: r171034): 0x00400e68:   addxi_uop   sp, sp, #48
650840: s819790 (t0: r171035): 0x00400e6c:   ret
650841: s819791 (t0: r171036): 0x00400698:   subs   x3, #4
650842: s819792 (t0: r171037): 0x0040069c:   b.ls   0x4006b0
650843: s819793 (t0: r171038): 0x004006a0:   dsb
650844: s819794 (t0: r171039): 0x004006a4:   orr   w1, wzr, w3
650845: s819795 (t0: r171040): 0x004006a8:   orr   x0, xzr, x23
650846: s819796 (t0: r171041): 0x004006ac:   bl   0x400bfc
650847: s819797 (t0: r171042): 0x00400bfc:   addxi_uop   ureg0, sp, #-80
650848: s819798 (t0: r171043): 0x00400bfc:   strxi_uop   x29, [ureg0]
650849: s819799 (t0: r171044): 0x00400bfc:   strxi_uop   x30, [ureg0, #8]
650850: s819800 (t0: r171045): 0x00400bfc:   addxi_uop   sp, ureg0, #0
650851: s819801 (t0: r171046): 0x00400c00:   add   x29, sp, #0
650852: s819802 (t0: r171047): 0x00400c04:   addxi_uop   ureg0, sp, #32
650853: s819803 (t0: r171048): 0x00400c04:   strxi_uop   x21, [ureg0]
650854: s819804 (t0: r171049): 0x00400c04:   strxi_uop   x22, [ureg0, #8]
650855: s819805 (t0: r171050): 0x00400c08:   adrp   x21, #548864
650856: s819806 (t0: r171051): 0x00400c0c:   adrp   x22, #557056
650857: s819807 (t0: r171052): 0x00400c10:   ldr   x21, [x21, #3968]
650858: s819808 (t0: r171053): 0x00400c14:   add   x22, x22, #2536
650859: s819809 (t0: r171054): 0x00400c18:   addxi_uop   ureg0, sp, #16
650860: s819810 (t0: r171055): 0x00400c18:   strxi_uop   x19, [ureg0]
650861: s819811 (t0: r171056): 0x00400c18:   strxi_uop   x20, [ureg0, #8]
650862: s819812 (t0: r171057): 0x00400c1c:   orr   x20, xzr, x0
650863: s819813 (t0: r171058): 0x00400c20:   movz   w19, #13, #0
650864: s819814 (t0: r171059): 0x00400c24:   addxi_uop   ureg0, sp, #48
650865: s819815 (t0: r171060): 0x00400c24:   strxi_uop   x23, [ureg0]
650866: s819816 (t0: r171061): 0x00400c24:   strxi_uop   x24, [ureg0, #8]
```

Figure C.2: *Konata* observation of a defeated *Spectre* attack.

D Guides

D.1 Troubleshooting *gem5*

As *gem5* could be very complicated to debug sometimes, here is a compilation of different problems that a developer could encounter. This section is mainly intended to be used as a cheat sheet by a future *gem5* developer, or overfly by a reader which wants to know what *gem5* development could look like.

AttributeError: Can't resolve proxy 'any' of type 'XXX' from 'XXX'.

1. **Goal**

   - This is the error we will troubleshoot:

   ```
   AttributeError: Can't resolve proxy 'any' of type 'ArmSystem'
                   from 'system.realview.generic_timer'
   ```

2. **Explanation**

   - The *proxy parameter* in *gem5* is a *Python* helper mechanism which is used to handle, affect and verify parameters passed to a `SimObject`. This mechanism is implemented in the `src/python/m5/proxy.py` file.
   - A *special proxy paramater* is a *proxy parameter* which have a dedicated class into the `proxy.py` file. Consider this *special proxy parameter* (`Parent.any`):

   ```
   system = Param.System(Parent.any, "The system the object is part of")
   ```

   - This is its special implementation:

   ```
   class AnyProxy(BaseProxy):
       def find(self, obj):
           return obj.find_any(self._pdesc.ptype)

       def path(self):
           return 'any'
   ```

   - Which means: "We will affect to the `system` attribute of the current object any object of type `System` found into the parent object". It allows to affect a precise type of variable without knowing its name in the parent object.

3. **Resolution**

   - To resolve our problem, we have to find the *special proxy parameter*:
     - `system` inherit from the `System` class (`System.py`) ;
     - `system.realview` is of `VExpress_GEM5_V1` class (`RealView.py`) ;
     - `system.realview.generic_timer` is of `GenericTimer` class (`GenericTimer.py`).
   - In the `GenericTimer` class, we can find the *special proxy parameter* mentioned in the message:

   ```
   system = Param.ArmSystem(Parent.any, "system")
   ```

   - This parameter search for an `ArmSystem` in the parent (`VExpress_GEM5_V1`).
   - The `VExpress_GEM5_V1` class has a `system` attribute which is our `system` object here.
   - Therefore, our `GenericTimer` will find a `System` object but not the specialized `ArmSystem` object, which produce the error of matching type.
   - Finally, to resolve the *proxy* error, we have to change our `system` object to an `ArmSystem` object – or an object which inherit from the `ArmSystem` – instead of a `System` object.

gem5 has encountered a segmentation fault!.

1. **Goal**

   - Troubleshoot this kind of, is not understand, cryptic error:

   ```
   gem5 has encountered a segmentation fault!

   --- BEGIN LIBC BACKTRACE ---
   /opt/Gem5/build/ARM/gem5.opt(+0xd14cc9)[0x5579a2c41cc9]
   /opt/Gem5/build/ARM/gem5.opt(+0xd2781f)[0x5579a2c5481f]
   /lib/x86_64-linux-gnu/libpthread.so.0(+0x14140)[0x7f766cdc8140]
   /opt/Gem5/build/ARM/gem5.opt(+0x134d5d4)[0x5579a327a5d4]
   [...]
   /opt/Gem5/build/ARM/gem5.opt(+0x586ebe)[0x5579a24b3ebe]
   /lib/x86_64-linux-gnu/libpython3.8.so.1.0(+0xa1a78)[0x7f766ce77a78]
   /lib/x86_64-linux-gnu/libpython3.8.so.1.0(_PyObject_MakeTpCall+0xa7)
       [0x7f766ce78817]
   [...]
   /lib/x86_64-linux-gnu/libpython3.8.so.1.0(+0x7dc4d)[0x7f766ce53c4d]
   --- END LIBC BACKTRACE ---
   ```

2. **Explanation**

   - This error is produced by the *gem5* binary itself, by its *C++* code base.
   - Often caused by a `NULL` pointer being dereferenced in *gem5*, possibly caused by:
     - A parameter that is asserted to be set, but in fact, it is not.
     - A port that this asserted to be linked, but in fact, it is not.

3. **Resolution**

   - The best practice to do here is to use *gdb*.
   - Ideally, you should use the `gem5.debug` binary:

```
$ gdb $GEM5/build/ARM/gem5.opt
$ run --debug-break=2000 -d /tmp $GEM5_SCRIPTS/RPIv4.py -v --fs \
    --fs-kernel=$gem5_kernel --fs-disk-image=$gem5_disk
```

- Optionally, if you can to debug the binary just before that the `segfault` happened, use trial and error strategy to refine your `--debug-break` tick start in order to arrive where you want to go in the execution state.
- At some point, you will arrive at your segfault. Below is the relevant information output by our *gdb* to resolve the bug:

```
Program received signal SIGSEGV, Segmentation fault.
0x00005555568a15d4 in ArmSystem::ArmSystem (this=0x5555595cfb00,
                                             p=0x555558cba1a0)
    at build/ARM/arch/arm/system.cc:77
77                  _resetAddr = workload->getEntry();
```

```
$rsp : 0x00007fffffffc6c0  →  0x00007ffff50c6398  →  0x0000000000000000
$rbp : 0x00005555595cfb00  →  0x0000555557e10020  →  0x0000555556d1fd70
    →  <ArmSystem::~ArmSystem()+0> lea rax, [rip+0x10f02a9]
    # 0x555557e10020 <_ZTV9ArmSystem+16>
$rsi : 0x0000555557f3e0a0  →  0x0000555558f53140  →  0x0000555558f53120
    →  0x000055555961c540  →  0x000055555961c560  →  0x000055555961c580
    →  0x000055555961c5a0  →  0x000055555961c5c0
$rdi : 0x0
$rip : 0x00005555568a15d4  →
    <ArmSystem::ArmSystem(ArmSystemParams*)+276> mov rax, QWORD PTR [rdi]
```

```
  0x5555568a15c4 <ArmSystem::ArmSystem(ArmSystemParams*)+260>
    cmp BYTE PTR [rbx+0x144], 0x0
  0x5555568a15cb <ArmSystem::ArmSystem(ArmSystemParams*)+267>
    je  0x5555568a1648 <ArmSystem::ArmSystem(ArmSystemParams*)+392>
  0x5555568a15cd <ArmSystem::ArmSystem(ArmSystemParams*)+269>
    mov rdi, QWORD PTR [rbp+0x190]
→ 0x5555568a15d4 <ArmSystem::ArmSystem(ArmSystemParams*)+276>
    mov rax, QWORD PTR [rdi]
```

```
    72                _havePAN(p->have_pan),
    73                semihosting(p->semihosting),
    74                multiProc(p->multi_proc)
    75          {
    76              if (p->auto_reset_addr) {
→   77                  _resetAddr = workload->getEntry();
```

- We have just found the source of the `SEFGAULT`:
  - `workload->getEntry();` dereference `workload` pointer to call the `getEntry()` function.
  - `mov rax, QWORD PTR [rdi]` is the pointer dereference in assembly.
  - The `rdi` register is set to `0x0` – equivalent to `NULL` in *C*.
  - This leads to the segmentation fault. Hence, our workload is not really passed to our `ArmSystem` object. In fact, our workload was linked to the wrong `SimObject` by inadvertence.

fatal: XXX.

1. **Goal**

   - Troubleshoot this kind of error:

   ```
   fatal: Must specify at least one workload!
   ```

2. **Explanation**

   - This error is generated in the *C++* source code of *gem5*, by its own error handling mechanism.
   - The cause of the error is self-explanatory. What has to be understand is why *gem5* does not find a workload if we actually think that the workload is correctly passed.

3. **Resolution**

   - The best practice is to search for the error (without the error-level keyword) in the source code:

   ```
   ack "Must specify at least one workload" $GEM5/src
   ```

   ```
   /opt/Gem5/src/cpu/o3/deriv.cc:47:
       fatal("Must specify at least one workload!");
   ```

   - We can then search, in the source code, the source of the error:

   ```
   sed -n '35,54'p /opt/Gem5/src/cpu/o3/deriv.cc
   ```

   ```
   DerivO3CPU *
   DerivO3CPUParams::create()
   {
       ThreadID actual_num_threads;
       if (FullSystem) {
           // Full-system only supports a single thread for the moment.
           actual_num_threads = 1;
       } else {
           if (workload.size() > numThreads) {
               fatal("Workload Size (%i) > Max Supported Threads (%i) on This
                     CPU", workload.size(), numThreads);
           } else if (workload.size() == 0) {
               fatal("Must specify at least one workload!");
           }

           // In non-full-system mode, we infer the number of threads from
           // the workload if it's not explicitly specified.
           actual_num_threads =
               (numThreads >= workload.size()) ? numThreads : workload.size();
       }
   ```

   - Here, we can understand that the `O3CPU` have taken the first `else` path, when he should have taken the first `if` (because we are in *FS* mode). Then, the processor searched for a workload linked on it, but there is none because, again, we are in *FS* mode – and the workload has to be linked to the system –, therefore producing the fatal error.
   - To fix this particular error, you have to set `full_system=True` variable of the `Root` object.

XXX port of XXX not connected to anything!.

1. **Goal**

   - Troubleshoot this kind of error:

   ```
   panic: Pio port of system.realview.generic_timer_mem not
          connected to anything!
   ```

2. **Explanation**

   - This error is generated in the *C++* source code of *gem5*, by its error handling mechanism.
   - The reason is clear: the setup of one `SimObject`'s port is badly programmed or forgotten.

3. **Resolution**

   - The linkage of this port should perhaps have been done directly by you, or by an helper function already provided by *gem5*.
   - To distinguish between these two ways, search in the source code the implementation of concerned object (here, `system.realview.generic_timer_mem`). Understand its function, its ports usage, and so one.
   - One trick that can help a lot is the generated `config.dot.pdf`, which give a graphical representation of the system (with links between `SimObject`).

Kernel panic - not syncing: VFS: Unable to mount root fs.

1. **Goal**

   - Troubleshoot this *kernel panic*:

   ```
   [    0.224367] List of all partitions:
   [    0.224394] fe00         1048320 vda
   [    0.224397]  driver: virtio_blk
   [    0.224440]   fe01         1048288 vda1 00000000-01
   [    0.224441]
   [    0.224480] No filesystem could mount root, tried:
   [    0.224481]  ext3
   [    0.224510]  ext4
   [    0.224524]  ext2
   [    0.224537]  squashfs
   [    0.224551]  vfat
   [    0.224566]  fuseblk
   [    0.224579]
   [    0.224606] Kernel panic - not syncing: VFS:
                   Unable to mount root fs on unknown-block(254,0)
   [    0.224656] CPU: 0 PID: 1 Comm: swapper/0 Not tainted 4.18.0+ #1
   [    0.224692] Hardware name: V2P-CA15 (DT)
   [    0.224717] Call trace:
   [    0.224741]  dump_backtrace+0x0/0x1c0
   [    0.224765]  show_stack+0x14/0x20
   [    0.224790]  dump_stack+0x8c/0xac
   [    0.224812]  panic+0x130/0x288
   [    0.224836]  mount_block_root+0x22c/0x294
   [    0.224861]  mount_root+0x140/0x174
   ```

```
[    0.224884]  prepare_namespace+0x138/0x180
[    0.224910]  kernel_init_freeable+0x1c0/0x1e0
[    0.224939]  kernel_init+0x10/0x108
[    0.224961]  ret_from_fork+0x10/0x18
[    0.224987] Kernel Offset: disabled
[    0.225009] CPU features: 0x21c06492
[    0.225032] Memory Limit: 2048 MB
[    0.225056] ---[ end Kernel panic - not syncing: VFS: Unable to mount
                    root fs on unknown-block(254,0) ]---
```

2. **Explanation**

- This error is generated by the *Linux* kernel, in a *FS* mode system.
- We can see, from the error:
  - The kernel recognize the *VirtIO* block device, which means that this driver is correctly loaded.
  - The kernel tried the `ext` file system, which means that the file systems are correctly loaded.
  - The kernel detect a `vda1` partition.

3. **Resolution**

- The problem lying into the specification of the root partition, on the kernel command line. In the full-system simulation script, we have to correctly set the root partition, like this:

```
# Linux kernel boot command flags.
kernel_cmd = [
    <...>
    # Tell Linux where to find the root disk image.
    "root=/dev/vda1",
    <...>
]
system.workload.command_line = " ".join(kernel_cmd)
```

- Do not forget to replace `<...>` with other correct options.
- Before our modification, the *VirtIO* block device was specified (`/dev/vda`). However, the kernel wants a partition (`/dev/vda1`), not a block device.

Bibliography

Proceedings

[Bar+19] Kristin Barber et al. "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels". In: *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 2019, pp. 151–164. doi: 10.1109/PACT.2019.00020. url: https://doi.org/10.1109/PACT.2019.00020.

[BS97] Eli Biham and Adi Shamir. "Differential fault analysis of secret key cryptosystems". In: *Advances in Cryptology — CRYPTO '97*. Ed. by Burton S. Kaliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525. isbn: 978-3-540-69528-8. doi: 10.1007/BFb0052259.

[Bul+20] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 54–72. doi: 10.1109/SP40000.2020.00089. url: https://doi.org/10.1109/SP40000.2020.00089.

[Cam+18] Giovanni Camurati et al. "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers". In: *Proceedings of the 25th ACM conference on Computer and communications security (CCS)*. CCS '18. Toronto, Canada: ACM, 2018.

[Can+19] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *Proceedings of the 28th USENIX on Security Symposium (USENIX Security 19)*. SEC'19. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 249–266. isbn: 9781939133069. url: https://www.usenix.org/system/files/sec19-canella.pdf.

[CKG20] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. "The Evolution of Transient-Execution Attacks". In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. GLSVLSI '20. Virtual Event, China: Association for Computing Machinery, 2020, pp. 163–168. isbn: 9781450379441. doi: 10.1145/3386263.3407583. url: https://doi.org/10.1145/3386263.3407583.

[ECC14] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. "Micro-Architectural Simulation of In-Order and Out-Of-Order ARM Microprocessors with Gem5". In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. SAMOS XIV. CEA LIST. Nov. 2014. doi: 10.1109/SAMOS.2014.6893220. url: https://www.researchgate.net/profile/Damien_Courousse/publication/266403326_Micro-architectural_Simulation_of_In-order_and_Out-of-order_ARM_Microprocessors_with_gem5/links/54327cea0cf225bddcc7a652/Micro-architectural-Simulation-of-In-order-and-Out-of-order-ARM-Microprocessors-with-gem5.pdf.

[EM98] A. N. Eden and T. Mudge. "The YAGS Branch Prediction Scheme". In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 31. Dallas, Texas, USA: IEEE Computer Society Press, 1998, pp. 69–77. ISBN: 1581130163.

[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.

[Gra+18] Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks". In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, pp. 955–972. ISBN: 9781931971461.

[Gru+16] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez. Cham: Springer International Publishing, 2016, pp. 279–299. ISBN: 978-3-319-40667-1. DOI: 10.1007/978-3-319-40667-1_14.

[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)*. Graz University of Technology. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-gruss.pdf.

[Gut+14] Anthony Gutierrez et al. "Sources of Error in Full-System Simulation". In: *International Symposium on Performance Analysis of Systems and Software*. ISPASS. University of Michigan and ARM Research. IEEE, Mar. 2014. DOI: 10.1109/ISPASS.2014.6844457. URL: https://tnm.engin.umich.edu/wp-content/uploads/sites/353/2017/12/2014.03.Sources-of-error-in-full-system-simulation.pdf.

[Hor16] T. Hornby. "Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD". In: BlackHat '16. 2016.

[JL01] Daniel A. Jiménez and Calvin Lin. "Dynamic Branch Prediction with Perceptrons". In: *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. HPCA '01. USA: IEEE Computer Society, 2001, p. 197. ISBN: 0769510191.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9.

[Koc+19] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. S&P. 2019. URL: https://spectreattack.com/spectre.pdf.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '96. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 104–113. ISBN: 3540615121. DOI: 10.1007/3-540-68697-5_9.

[Kor+18] Esmaeil Mohammadian Koruyeh et al. "Spectre Returns, Speculation Attacks using the Return Stack Buffer". In: *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. Ed. by Christian Rossow and Yves Younan. USENIX Association, 2018. URL: https://www.usenix.org/conference/woot18/presentation/koruyeh.

[LCM97] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. "The Bi-Mode Branch Predictor". In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 30. Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 4–13. ISBN: 0818679778.

[Lip+18] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[Lip+20b] Moritz Lipp et al. "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors". English. In: *ASIA CCS 2020 - Proceedings of the 15th ACM Asia Conference on Computer and Communications Security.* ACM/IEEE, Oct. 2020.

[MR18] Giorgi Maisuradze and Christian Rossow. "Ret2spec: Speculative Execution Using Return Stack Buffers". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2109–2122. ISBN: 9781450356930. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761.

[Mur+20] Kit Murdock et al. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *41st IEEE Symposium on Security and Privacy (S&P'20).* 2020.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology – CT-RSA 2006.* Ed. by David Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN: 978-3-540-32648-9. DOI: 10.1007/11605805_1.

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching". In: *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture.* MICRO 29. Paris, France: IEEE Computer Society, 1996, pp. 24–35. ISBN: 0818676418.

[SBL04] Taeweon Suh, Douglas M. Blough, and Hsien-Hsin S. Lee. "Supporting Cache Coherence in Heterogeneous Multiprocessor Systems". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2.* DATE '04. USA: IEEE Computer Society, 2004, p. 21150. ISBN: 0769520855.

[Sez05] André Seznec. "Analysis of the O-GEometric History Length Branch Predictor". In: *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA.* IEEE Computer Society, 2005, pp. 394–405. DOI: 10.1109/ISCA.2005.13. URL: https://doi.org/10.1109/ISCA.2005.13.

[Sez11] André Seznec. "A New Case for the TAGE Branch Predictor". In: *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO.* Ed. by Carlo Galuzzi et al. Porto Alegre, Brazil: ACM, Dec. 2011, pp. 117–127. DOI: 10.1145/2155620.2155635. URL: https://hal.inria.fr/file/index/docid/639193/filename/MICRO44_Andre_Seznec.pdf.

[Sez14] André Seznec. "TAGE-SC-L Branch Predictors". In: *JILP - Championship Branch Prediction.* Minneapolis, United States, June 2014. URL: https://hal.inria.fr/hal-01086920.

[Sez16] André Seznec. "TAGE-SC-L Branch Predictors Again". In: *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5).* Seoul, South Korea, June 2016. URL: https://hal.inria.fr/hal-01354253.

[Smi98] James E. Smith. "A Study of Branch Prediction Strategies". In: *25 Years of the International Symposia on Computer Architecture (Selected Papers).* ISCA '98. Barcelona, Spain: Association for Computing Machinery, 1998, pp. 202–215. ISBN: 1581130589. DOI: 10.1145/285930.285980. URL: https://doi.org/10.1145/285930.285980.

[SS95] James E. Smith and Gurindar S. Sohi. "The Microarchitecture of Superscalar Processors". In: *Proceedings of the IEEE, Vol. 83, No. 12, December 1995.* IEE, 1995, p. 15.

[YF14] Yuval Yarom and Katrina Falkner. “Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf.

[YP91] Tse-Yu Yeh and Yale N. Patt. “Two-Level Adaptive Training Branch Prediction”. In: *Proceedings of the 24th Annual International Symposium on Microarchitecture*. MICRO 24. Albuquerque, New Mexico, Puerto Rico: Association for Computing Machinery, 1991, pp. 51–61. ISBN: 0897914600. DOI: 10.1145/123465.123475. URL: https://doi.org/10.1145/123465.123475.

[ZPL01] Yuanyuan Zhou, James Philbin, and Kai Li. “The Multi-Queue Replacement Algorithm for Second Level Buffer Caches”. In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 91–104. ISBN: 1-880446-09-X.

Articles

[Ald+19] Alejandro Cabrera Aldaya et al. “Port Contention for Fun and Profit”. In: *IEEE Symposium on Security and Privacy (S&P)* (Sept. 16, 2019). DOI: 10.1109/SP.2019.00066. URL: https://eprint.iacr.org/2018/1060.pdf.

[BCR10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. “Translation Caching: Skip, Don’t Walk (the Page Table)”. In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 48–59. ISSN: 0163-5964. DOI: 10.1145/1816038.1815970. URL: https://doi.org/10.1145/1816038.1815970.

[Bin+11] Nathan Binkert et al. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://dl.acm.org/doi/pdf/10.1145/2024716.2024718.

[BL93] Thomas Ball and James R. Larus. “Branch Prediction for Free”. In: *SIGPLAN Not.* 28.6 (June 1993), pp. 300–313. ISSN: 0362-1340. DOI: 10.1145/173262.155119. URL: https://doi.org/10.1145/173262.155119.

[Ge+16] Qian Ge et al. “A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware”. In: *J. Cryptogr. Eng.* 8 (2016). https://eprint.iacr.org/2016/613, pp. 1–27.

[GST17] Daniel Genkin, Adi Shamir, and Eran Tromer. “Acoustic Cryptanalysis”. In: *J. Cryptology* 30 (2017), pp. 392–443. DOI: 10.1007/s00145-015-9224-2.

[Ioa05] John P. A. Ioannidis. “Why Most Published Research Findings Are False”. In: *PLoS Medicine* (Aug. 2005), pp. 696–701. DOI: 10.1371/journal.pmed.0020124. URL: https://journals.plos.org/plosmedicine/article/file?id=10.1371/journal.pmed.0020124&type=printable.

[Kes99] R. E. Kessler. “The Alpha 21264 Microprocessor”. In: *IEEE Micro* 19.2 (Mar. 1999), pp. 24–36. ISSN: 0272-1732. DOI: 10.1109/40.755465. URL: https://doi.org/10.1109/40.755465.

[Koc+18] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *CoRR* abs/1801.01203 (2018). arXiv: 1801.01203. URL: http://arxiv.org/abs/1801.01203.

[Koc+20] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Commun. ACM* 63.7 (June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742. URL: https://doi.org/10.1145/3399742.

[Lip+20a] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *Commun. ACM* 63.6 (May 2020), pp. 46–56. ISSN: 0001-0782. DOI: 10.1145/3357033. URL: https://doi.org/10.1145/3357033.

[LS83] Johnny K. F. LEE and A. J. SMITH. “Analysis of branch prediction strategies and branch target buffer design”. In: *Perform. Evaluation* 3.4 (1983), p. 313. DOI: 10.1016/0166-5316(83)90042-1. URL: https://doi.org/10.1016/0166-5316(83)90042-1.

[Mit16a] Sparsh MITTAL. “A Survey of Recent Prefetching Techniques for Processor Caches”. In: *ACM Comput. Surv.* 49.2 (Aug. 2016). ISSN: 0360-0300. DOI: 10.1145/2907071. URL: https://doi.org/10.1145/2907071.

[Mit16b] Sparsh MITTAL. “A Survey of Techniques for Dynamic Branch Prediction”. In: *Concurrency and Computation Practice and Experience* (Apr. 2016), pp. 1–48. DOI: 10.1002/cpe.4666. URL: https://www.researchgate.net/profile/Sparsh_Mittal/publication/324166804_A_Survey_of_Techniques_for_Dynamic_Branch_Prediction/links/5adfe5b8458515c60f63cf62/A-Survey-of-Techniques-for-Dynamic-Branch-Prediction.pdf.

[MSU97] Pierre MICHAUD, André SEZNEC, and Richard UHLIG. “Trading Conflict and Capacity Aliasing in Conditional Branch Predictors”. In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pp. 292–303. ISSN: 0163-5964. DOI: 10.1145/384286.264211. URL: https://doi.org/10.1145/384286.264211.

[SM06] André SEZNEC and Pierre MICHAUD. “A Case for (Partially) TAgged GEometric History Length Branch Prediction”. In: *J. Instr. Level Parallelism* 8 (2006). URL: http://www.jilp.org/vol8/v8paper1.pdf.

[Smi82] Alan Jay SMITH. “Cache Memories”. In: *ACM Comput. Surv.* 14.3 (Sept. 1982), pp. 473–530. ISSN: 0360-0300. DOI: 10.1145/356887.356892. URL: https://doi.org/10.1145/356887.356892.

[Spr+97] Eric SPRANGLE et al. “The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference”. In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pp. 284–291. ISSN: 0163-5964. DOI: 10.1145/384286.264210. URL: https://doi.org/10.1145/384286.264210.

[Van85] Wim VAN ECK. “Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?” In: *North-Holland Computers & Security* (1985), pp. 269–286. DOI: 10.1016/0167-4048(85)90046-X. URL: https://www.win.tue.nl/~aeb/linux/hh/emr.pdf.

[Wan+18] Guanhua WANG et al. “oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis”. In: *CoRR* abs/1807.05843 (2018). arXiv: 1807.05843. URL: http://arxiv.org/abs/1807.05843.

[YP92] Tse-Yu YEH and Yale N. PATT. “Alternative Implementations of Two-Level Adaptive Branch Prediction”. In: *SIGARCH Comput. Archit. News* 20.2 (Apr. 1992), pp. 124–134. ISSN: 0163-5964. DOI: 10.1145/146628.139709. URL: https://doi.org/10.1145/146628.139709.

[Zad+13] Jonas ZADDACH et al. “Implementation and Implications of a Stealth Hard-Drive Backdoor”. In: *ACSAC* (Dec. 2013), pp. 279–288. DOI: 10.1145/2523649.2523661. URL: http://s3.eurecom.fr/docs/acsac13_zaddach.pdf.

Technical Reports

[Ash17] Chuan Zhu ASHKAN TOUSI. *The Arm Research Starter Kit, System Modeling using Gem5*. Tech. rep. ARM Ltd., 2017. URL: https://github.com/arm-university/arm-gem5-rsk/blob/master/gem5_rsk.pdf.

[McF93] Scott MCFARLING. *Combining Branch Predictors*. Tech. rep. Digital Equipment Corporation, Western Research Laboratory (DEC WRL), 1993. URL: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf.

[NSA82] NSA. *TEMPEST Fundamentals*. Tech. rep. National Security Agency, 1982. URL: https://cryptome.org/jya/nacsim-5000/nacsim-5000.htm.

[Yar] Yuval Yarom. *Mastik: A Micro-Architectural Side-Channel Toolkit*. Tech. rep. The University of Adelaide. URL: https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf.

Books

[Pop34] Karl R. Popper. *The Logic of Scientific Discovery*. Routledge, 1934. ISBN: 1-1344-7002-9. URL: http://www.strangebeautiful.com/other-texts/popper-logic-scientific-discovery.pdf.

[SBP09] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. CreateSpace Independent Publishing Platform, 2009. ISBN: 978-1441418869. URL: https://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf.

[TA13] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. 6th Edition. Pearson, 2013. ISBN: 978-0-13-291652-3.

Manuals and Specifications

[16] *ARM Cortex-A72 MPCore Processor. Technical Reference Manual*. Revision r0p3. ARM Ltd. 2016. URL: https://developer.arm.com/documentation/100095/0003/.

[20] *Arm Architecture Reference Manual. Armv8, for Armv8-A architecture profile*. Version DDI 0487F.b (ID040120). ARM Ltd. 2020. URL: https://developer.arm.com/documentation/ddi0487/latest.

[Sta20] Richard M. Stallman. *Using the GNU Compiler Collection*. Version 11.0. Free Software Foundation (FSF). GNU Press, 2020. URL: https://gcc.gnu.org/onlinedocs/gcc.pdf.

Websites

[Apv18] Axelle Apvrille. *Does malware based on Spectre exist*. Consulted on 2020-09-19. Fortinet. 2018. URL: https://www.virusbulletin.com/virusbulletin/2018/07/does-malware-based-spectre-exist/.

[Can+] Claudio Canella et al. *TransientFail Repository*. Consulted on 2020-08-30. Institute of Applied Information Processing and Communications (IAIK). URL: https://github.com/IAIK/transientfail.

[Hor18] Jann Horn. *Speculative Execution, Variant 4, Speculative Store Bypass*. Consulted on 2020-09-18. Google Project Zero. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[Lee] Ben Lee. *Dynamic Branch Prediction*. Consulted on 2020-09-02. Oregon State University. URL: https://web.engr.oregonstate.edu/~benl/Projects/branch_pred/.

[Lowa] Jason Lowe-Power. *Building gem5*. Consulted on 2020-09-08. gem5 developers. URL: https://www.gem5.org/documentation/general_docs/building.

[Lowb] Jason Lowe-Power. *gem5 Documentation*. Consulted on 2020-09-13. gem5 developers. URL: https://www.gem5.org/documentation/.

[Lowc] Jason Lowe-Power. *gem5 Documentation*. Consulted on 2020-09-13. gem5 developers. URL: http://www.m5sim.org/Documentation.

[Lowd] Jason Lowe-Power. *gem5: Mailing Lists*. Consulted on 2020-09-18. gem5 developers. URL: https://www.gem5.org/mailing_lists/.

[Lowe] Jason Lowe-Power. *gem5: Publications*. Consulted on 2020-09-18. gem5 developers. URL: https://www.gem5.org/publications/.

[Lowf] Jason Lowe-Power. *Learning gem5*. Consulted on 2020-09-02. University of California. URL: http://learning.gem5.org/.

[Low+20] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+. A new era for the open-source computer architecture simulator.* July 2020. URL: https://arxiv.org/pdf/2007.03152.pdf.

[Mut18] Onur Mutlu. *Computer Architecture Lectures. Branch Prediction.* Consulted on 2020-08-31. ETH Zürich. 2018. URL: https://safari.ethz.ch/architecture/fall2018/doku.php?id=schedule.

[PLH] Christophe Pouzat, Arnaud Legrand, and Konrad Hinsen. *Reproducible Research, Methodological Principles for a Transparent Science.* Consulted on 2020-08-27. INRIA, CNRS. URL: https://www.fun-mooc.fr/courses/course-v1:inria+41016+self-paced/about.

[Sch+17] Stephan van Schaik et al. *Reverse Engineering Hardware Page Table Caches Using Side-Channel Attacks on the MMU.* Consulted on 2020-09-17. Vrije Universiteit Amsterdam. 2017. URL: https://download.vusec.net/papers/revanc_ir-cs-77.pdf.

[Sez] André Seznec. *An Optimized 2bcgskew Branch Predictor.* Consulted on 2020-09-16. IRISA, INRIA. URL: https://www.irisa.fr/caps/people/seznec/Optim2bcgskew.pdf.

[Shi] Ryota Shioya. *Visualizing the out-of-order CPU model.* Consulted on 2020-09-13. Nagoya University. URL: http://learning.gem5.org/tutorial/presentations/vis-o3-gem5.pdf.

[Suna] Zhiyi Sun. *Enable ARM PMU.* Consulted on 2020-09-07. URL: https://github.com/zhiyisun/enable_arm_pmu/tree/dev.

[Sunb] Zhiyi Sun. *How to Use Performance Monitor Unit (PMU) of 64-bit ARMv8-A in Linux.* Consulted on 2020-09-07. URL: https://zhiyisun.github.io/2016/03/02/How-to-Use-Performance-Monitor-Unit-(PMU)-of-64-bit-ARMv8-A-in-Linux.html.

[Tor] Linus Torvalds. *Spectre Side Channels.* Consulted on 2020-09-06. The Linux Foundation. URL: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

[V-E] V-E-O. *PoC of CVE/Exploit.* Consulted on 2020-09-09. URL: https://github.com/V-E-O/PoC.

Acronyms

API Application Programming Interface. 36, 39, 44, 46, 55, 62

BHR Branch History Register. 14

BTB Branch Target Buffer. 13, 19, 23, 24, 34, 52

BTFN Backward Taken, Forward Not-Taken. 13

CA Cross-Address Space. 35,

CERT Computer Emergency Response Team. 1

CISC Complex Instruction Set Computer. 20

CNRS National Centre for Scientific Research. 3–5

CPU Central Processing Unit. 2, 6, 9, 40,

DVFS Dynamic Voltage Frequency Scaling. 32, 40

FS Full-system Simulation. 29, 51–53, 67, 78, 80

GHR Global History Register. 14, 15, 17, 23

HT Hyper-Threading.

IDE Integrated Development Environment. 5,

INRIA National Institute for Research in Computer Science and Automation. 3, 5

INSA National Institute of Applied Sciences. 3, 4

IO Input-Output. 53

IP In-Place. 34, 35, 50, 66,

IRISA Research Institute of Computer Science and Random Systems. 3, 4

ISA Instruction Set Architecture. , *Glossary:* ISA

LLC Last-Level Cache. 10, 11, 31

LRU Least Recently Used. 12, 25

MESI Modified Exclusive Shared Invalid. 26

MIT Massachusetts Institute of Technology. 1

MMU Memory Management Unit. 11, 26

MOESI Modified Owned Exclusive Shared Invalid. 26

MOOC Massive Open Online Course. 5,

MSR Model Specific Register. , *Glossary:* MSR

OoO Out-of-Order. 9, 51, 52

OOP Object Oriented Programming. 28

OOP Out-of-Place. 35,

PA Physical Address. 11

PC Program Counter. 13–16, 18–20

PHT Pattern History Table. 14–17, 19, 24, 34, 35, 50, 52, 66

PIPT Physically Indexed and Physically Tagged. 25,

PMC Performance Monitoring Counter. 39, , *Glossary:* PMC

PMU Performance Monitoring Unit. , *Glossary:* PMU

PoC Proof of Concept. 44, 45,

RAS Return Address Stack. 20, 24, 52

RISC Reduced Instruction Set Computer. 20

RSB Return Stack Buffer. 34, 46,

SA Same-Address Space. 34, 35, 50, 66,

SCU Snoop Control Unit. 26

SE System-call Emulation. 29, 51–53, 67

SMT Simultaneous Multithreading. , *Glossary:* SMT

TLB Translation Lookaside Buffer. 26, 32, 53

VA Virtal Address. 11

Index