Spring 2022

Processor Microarchitecture Security

Shuwen Deng
Yale University Graduate School of Arts and Sciences, shuwen.deng@yale.edu

Abstract

Processor Microarchitecture Security

Shuwen Deng

2022

As computer systems grow more and more complicated, various optimizations can unintentionally introduce security vulnerabilities in these systems. The vulnerabilities can lead to user information and data being compromised or stolen. In particular, the ending of both Moore's law and Dennard scaling motivate the design of more exotic microarchitectural optimizations to extract more performance – further exacerbating the security vulnerabilities. The performance optimizations often focus on sharing or re-using of hardware components within a processor, between different users or programs. Because of the sharing of the hardware, unintentional information leakage channels, through the shared components, can be created. Microarchitectural attacks, such as the high-profile Spectre and Meltdown attacks or the cache covert channels that they leverage, have demonstrated major vulnerabilities of modern computer architectures due to the microarchitectural optimizations.

Key components of processor microarchitectures are processor caches used for achieving high memory bandwidth and low latency for frequently accessed data. With frequently accessed data being brought and stored in caches, memory latency can be significantly reduced when data is fetched from the cache, as opposed to being fetched from the main memory. With limited processor chip area, however, the cache size cannot be very large. Thus, modern processors adopt a cache hierarchy with multiple levels of caches, where the cache close to processor is faster but smaller, and the cache far from processor is slower but larger. This leads to a fundamental property of modern processors: *the latency of accessing data in different cache levels and in main memory is different.* As a result, the timing of

memory operations when fetching data from different cache levels, e.g., the timing of fetching data from closest-to-processor L1 cache vs. from main memory, can reveal secret-dependent information if attacker is able to observe the timing of these accesses and correlate them to the operation of the victim's code. Further, due to limited size of the caches, memory accesses by a victim may displace attacker's data from the cache, and with knowledge, or reverse-engineering, of the cache architecture, the attacker can learn some information about victim's data based on the modifications to the state of the cache – which can be observed by the timing measurements.

Caches are not only structures in the processor that can suffer from security vulnerabilities. As an essential mechanism to achieving high performance, cache-like structures are used pervasively in various processor components, such as the translation lookaside buffer (TLB) and processor frontend. Consequently, the vulnerabilities due to timing differences of accessing data in caches or cache-like structures affect many components of the processor.

The main goal of this dissertation is the *design of high performance and secure computer architectures*. Since the sophisticated hardware components such as caches, TLBs, value predictors, and processor frontend are critical to ensure high performance, realizing this goal requires developing fundamental techniques to guarantee security in the presence of timing differences of different processor operations. Furthermore, effective defence mechanisms can be only developed after developing a formal and systematic understanding of all the possible attacks that timing side-channels can lead to.

To realize the research goals, the main main contributions of this dissertation are:

- Design and evaluation of a novel three-step cache timing model to understand theoretical vulnerabilities in caches
- Development of a benchmark suite that can test if processor caches or secure cache

designs are vulnerable to certain theoretical vulnerabilities.

- Development of a timing vulnerability model to test TLBs and design of hardware defenses for the TLBs to address newly found vulnerabilities.
- Analysis of value predictor attacks and design of defenses for value predictors.
- Evaluation of vulnerabilities in processor frontends based on timing differences in the operation of the frontends.
- Development of a design-time security verification framework for secure processor architectures, using information flow tracking methods.

This dissertation combines the theoretical modeling and practical benchmarking analysis to help evaluate susceptibility of different architectures and microarchitectures to timing attacks on caches, TLBs, value predictors and processor frontend. Although cache timing side-channel attacks have been studied for more than a decade, there is no evidence that the previously-known attacks exhaustively cover all possible attacks. One of the initial research directions covered by this dissertation was to develop a model for cache timing attacks, which can help lead towards discovering all possible cache timing attacks. The proposed three-step cache timing vulnerability model provides a means to enumerate all possible interactions between the victim and attacker who are sharing a cache-like structure, producing the complete set of theoretical timing vulnerabilities. This dissertation also covers new theoretical cache timing attacks that are unknown prior to being found by the model. To make the advances in security not only theoretical, this dissertation also covers design of a benchmarking suite that runs on commodity processors and helps evaluate their cache's susceptibility to attacks, as well as can run on simulators to test potential or future cache designs. As the dissertation later demonstrates, the three-step timing vulnerability model can be naturally applied to any cache-like structures such as TLBs,

and the dissertation encompasses a three-step model for TLBs, uncovering of theoretical new TLB attacks, and proposals for defenses. Building on success of analyzing caches and TLBs for new timing attacks, this dissertation then discusses follow-on research on evaluation and uncovering of new timing vulnerabilities in processor frontends. Since security analysis should be applied not just to existing processor microarchitectural features, the dissertation further analyzes possible future features such as value predictors. Although not currently in use, value predictors are actively being researched and proposed for addition into future microarchitectures. This dissertation shows, however, that they are vulnerable to attacks. Lastly, based on findings of the security issues with existing and proposed processor features, this dissertation explores how to better design secure processors from ground up, and presents a design-time security verification framework for secure processor architectures, using information flow tracking methods.

Processor Microarchitecture Security

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Shuwen Deng

Dissertation Director: Jakub Szefer

May, 2022

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my family, my friends, and my collaborators for their kind and strong support to make this thesis and dissertation possible.

My deep gratitude goes first to my Ph.D. advisor, Professor Jakub Szefer, who guided me and supported me throughout my Ph.D. study. I still remember the time when I chose Yale six years ago, I made up my mind to work together with this young and full-of-aspiration scholar. Life always works not as expected. I went through discouragement and failure but I can always get strong backing from Jakub, who helped me digest the problem and start over again. He is always full of passion to stick to the research and find potentials from the seemingly impossible scenarios as well as explore totally new areas. I learn a lot from that and he will always be my academic role model.

I would like to give the great appreciation to my committee members. Professor Rajit Manohar is one of the smartest person I met. He is so devoted to the research and has provided a lot of good suggestions to the research and academic lives. Professor Ruzica Piskac is very kind to students and very passionate to the research. Every time I talk with her, I feel like I am talking to my elder sister and she is very responsive and gave lots of responses to my puzzles.

I want to express my gratitude to my CASLAB labmates Shanquan Tian, Bowen Huang, Ilias Giechaskiel, Ferhat Erata, Chuanqi Xu, Sanjay Deshpande, Theodoros Trochatos, Anthony Etim for their support. I enjoyed the research time being with you so much. I want to thank my pre-labmates, Dr. Wenjie Xiong and Dr. Wen Wang. They are the best seniors who support me a lot since I joined the lab. I missed the time of three girls in our lab a lot and missed the Wen-ish line. As the women researchers, they show incredible power to their

research and stick to their mind, where I also learn a lot. I would like to also thank the staff at Yale SEAS: Cara Gibilisco, Kevin Ryan, Annette Myers, Pamela DeFilippo, Vanessa Epps, and Rebekka Blaha, for all their timely support.

I would like to also thank my awesome collaborators. Prof. Onur Demir and Doğuhan Gümüşoğlu are sincerely nice people to work together. I still remember hard time in my first research project when they helped me overcome the first obstacle I met in my Ph.D. to start building up my strength. Nikolay Matyunin together with Prof. Stefan Katzenbeisser are also powerful teammates. They are rigorous to the research and very reliable people. Allen Mi is a very smart student and I learn a lot of quantum computing knowledge from him. Prof. Xuehai Qian provides a lot of valueable suggestions to my thesis and applications and lead me to the transition from a student to a new role in academia.

Now I want to express my special thanks to my intern managers and mentors Dr. Jin Yang, Dr. Zhenkun Yang, Prof. Steve Keckler, and Dr. Michael Sullivan. Zhenkun and Michael are very good mentors to provide considerate care for me when I joined the company. During the intern time, I met them everyday to share thoughts and receive feedback from the update. They guide me how to initialize the work in industry and propose valuable suggestions for both the project and how to work in the company. My managers Jin and Steve are very nice and inspiring. They help me build the confidence and encourage me to achieve high with their solid research suggestions.

I want to thank my friends Wenxuan Deng, Chang Liu, Xiayuan Wen, Fengjiao Liu who support me throught my Ph.D. life. I am also very lucky to know many other friends at Yale: Rui Li, Yihang Yang, Zijun Tang, Yifan Chen, Chen Gu, Meredith Reba, Ann Cowlin, Naijia Liu, Juanjuan Lu, Sihao Wang, and Mohan Shen. They make me feel I joined a new family at New Haven and I really enjoyed and will miss the time spending with them. Out of Yale, I also want to thank my lifelong friends Minghao Yin, Yiying Li, Jixin Guo, Tianqi Li, Yijing Gao, Yuemeng Wang, Junhui Wu, Ziqi Liu, Miao Yu, Wenqi Yin, Boyi Zheng, Xinyi Liu, Mengyun Liu, Yuting Hou, Qianru Li, Xueyin Yu. With their support, I finally grasped the chance to join Yale and finish the study here. The last part of this module belongs to my boyfriend Yuntao Xu. He is a determined person with his own will from which I learn a lot. He cooks so well that he heals me every time when I miss home. We

share the happiness and sorrow and grow up together.

Last but not least, I want to express my whole gratitude to my family. My mother, who gave me life and always support me to be a stronger and happier person. I know every footprint of mine brands in your heart so I hope next time your nightmare that was full of child version of me and other bad memories which troubled you so much can be replaced with sweet dream about our current happy life and fruitful future. My father is always like a teacher of my life who gives me very valuable and instructive suggestions that drag me out of the dilemmas. We two will keep exercising every day and stimulate each other to live long from now on. My grandfathers and my grandmothers, who raised me starting from an infant, gave their selfless and deepest love to me. Grandma, I miss you so much and I hope you are happy long after and I wish you could help comb my hair again on our leisure Friday. My grandfather, who is the most earnest person I have ever seen, built the foundation of me since my childhood and fostered every single good habit of me. My family are my soft blood and my hard bones. I could not achieve a single step without their support.

To my family, especially my grandmother.

Chapter 1

Introduction

The recent Spectre, Meltdown, and Foreshadow attacks [3, 4, 5, 6], which allow the attackers to obtain secrets that cannot be revealed through non-speculative execution, exploit fundamental design flaws existing in almost all modern processors at the microarchitecture level. As another example, network structure and parameters of neural network models (i.e., weights) can also be leaked throughout the microarchitectural attacks targeting caches [7]. These and other numerous security attacks abuse microarchitectural features which were initially designed to improve performance, but as an side-effect they open up timing channels that can be used to leak information. Secrets that can be leaked can include passwords stored in a password manager or browser, personal photos, emails, instant messages and even business-critical documents [3, 4]. To solve these important security weaknesses, this dissertation presents research that addresses and mitigates number of different types of microarchitectural timing security vulnerabilities in computer processors.

1.1 Dissertation Contributions

This dissertation covers a number of research themes and contributions listed below.

1.1.1 Microarchitectural Vulnerability Modeling

In order to steal secret information from a computer system, the attacker only needs to find one possible attack. On the other hand, to protect the system's security, a defender needs

to prevent all possible types of attacks. However, there has so far been little research on systematic analysis of how existing processors and secure hardware proposals can, or cannot, protect against different types of the timing attacks and whether they cover all possible types of attacks. To provide a means of the analysis, this dissertation presents research on a systematic approach to finding all possible cache [8, 9] and translation look-aside buffer (TLB) [10] timing vulnerabilities. To achieve this, a model is established based on two observations that: 1) all existing cache timing attacks focusing on the data use three memory operations and 2) timing attacks can be analyzed by checking the behavior of one cache block, since all blocks are updated in the same manner by the cache logic. Following these observations, a three-step model focusing on one cache block for evaluating all possible timing attacks is presented in this dissertation. The model is validated with a soundness analysis where we demonstrated that the three-step model can cover all possible cache timing side-channel vulnerabilities: either a vulnerability can be expressed using three or fewer steps, or if it is expressed using more than three steps, it can be reduced to a existing vulnerability type that requires only three steps. Given all possible states and three-step combinations of memory accesses by potential victim and attacker programs, the model then demonstrated 88 types of theoretical timing vulnerabilities in processor L1 caches [9] and 24 types of theoretical timing vulnerabilities in the translation look-aside buffers [10].

1.1.2 Cache and TLB Timing Vulnerabilities and Defenses

Based on the success of the three-step model, a benchmark suite was developed for both x86 and Arm. The goal of the benchmark suite is to evaluate all the vulnerabilities of real processor caches to timing attacks by running the benchmarks on real hardware, and analyzing their results. Any cache timing vulnerabilities found to be possible in a real processor can be used, for example, by Spectre variants to extract sensitive information. The benchmarks can also be run in simulation. With the benchmarks, it is possible to help test processor caches or future secure cache designs, and understand which timing attacks they are vulnerable to. In order to model the attacks in the real processors, the commercial processor features need to be considered, including hyper-threading and time-slicing of execution of programs, accessing memory using either read or write operations, invalidation

using flush instructions or via cache coherence protocol operations, etc. The x86 and the Arm benchmark suite were tested on 9 commodity Intel and AMD processors and 34 mobile devices with Arm processors to show how the differences in processor implementations and microarchitectures can result in different types of potential vulnerabilities. Further, the benchmarks can be ran in simulation to help designers of new secure processors and caches evaluate their designs' susceptibility to cache timing attacks and provide new insights about processor features affecting security. As an example, the presented research has analyzed secure cache designs to understand if they can enhance the security of devices. The work shows the security of PL [1] and RF [2] caches but also uncovers new weaknesses.

Developing systematic approach to checking for attacks is necessary, not just for caches, but TLBs and other cache-like structures since any cache-like structure with varying timing in the microarchitecture can be vulnerable to timing attacks. This dissertation in particular explores TLBs as another critical cache-like structure in modern processors. Comparing to caches, TLB accesses are triggered by memory translation requests and TLBs may have more complicated logic, e.g., since they the support various memory page sizes. In this case, to systematically analyze TLB vulnerabilities, a modified three-step model had to be developed. Based on the three-step model for TLBs, means to automatically generate micro security benchmarks that can test for the TLB vulnerabilities were developed and are presented in this dissertation. After showing the insecurity of standard TLBs using the TLB micro security benchmarks, two new secure TLB designs were proposed and evaluated on the RISC-V Rocket Core platform. Based on the analysis, the proposed secure TLBs can defend not only against the previously publicized attacks but also against other new timing attacks in TLBs found using the TLB three-step model. The performance overhead was evaluated on an FPGA-based setup, and, for example, shows that a Random Fill (RF) TLB which was introduced as one of the defenses to the new vulnerabilities, has less than 10% overhead while defending all the attacks from the TLB three-step model developed and presented in this dissertation.

1.1.3 Vulnerabilities and Defenses of Value Predictors

Looking beyond caches and TLBs, this dissertation also explores security of value predictors. Although not realized in silicon yet, value predictors are actively researched [11, 12], and attacks and defenses should be analyzed at design time before new features are added to real machines. Many existing attacks on real processors can be attributed to features that were introduced without proper design-time security analysis. By exploring value predictor attacks and defenses, the research presented in this dissertation fills in the missing understanding of the security of value predictors. To address security of value predictors, the first, systematic model for analyzing value predictor attacks to demonstrate different variants of attacks that can leverage value predictors to leak information was developed. The work from this dissertation also demonstrated the first, new attacks on value predictors, which can be used to attack real applications and bypass existing protection schemes and further proposed and evaluated security techniques for securing value predictors by randomizing, delaying the prediction or always doing the prediction even if prediction confidence level is low.

1.1.4 Processor Frontend Vulnerabilities

The design of the processor frontend ensures that ideally it will not become a bottleneck, so that the backend is always fed with sufficient instructions to process. Consequently, majority of timing or power variations that can be observed in processors are due to components in the backend, such as the caches or the execution units, as that is the bottleneck where execution differences can be more easily observed. However, as this dissertation shows, new timing and power covert channels were uncovered due to the processor frontend in modern Intel processors. The root causes of new channels are the multiple paths in the processor frontend that the micro-ops can take: through the Micro-Instruction Translation Engine (MITE), through the Decode Stream Buffer (DSB), also called the Micro-op Cache, or through the Loop Stream Detector (LSD). Each path has its own unique timing and power signatures, which lead to the security threats such as fingerprinting running applications. In addition, the switching between the different paths can also lead to observable timing or power differences which could be exploited by attackers. Furthermore, the dissertation

research demonstrated new ways for leaking execution information about SGX enclaves or a new in-domain Spectre variant. In addition, a new method for fingerprinting the microcode patches of the processor by analyzing the behavior of different paths in the frontend is possible and demonstrated. The frontend-based side channel can also be utilized to fingerprinting what type of workloads that is co-located with it on a same SMT-core, which demonstrates the user preference for mobile devices usage, for example, and can be further utilized as the data input of advertisement recommendation.

Based on these findings, defending the frontend vulnerabilities will require new approaches for design of the frontend. One the one hand, partitioning can be used to prevent interference between SMT threads. The LSD and DSB should be always partitioned, as well as the MITE should be partitioned. On the other hand, some attacks can be defended by randomizing MITE mapping or by adding random operations to MITE. Only by exploring new vulnerabilities, as presented in this dissertation, can architects start to think about proper defenses.

1.1.5 Preliminary Study of Vulnerabilities in Accelerators Beyond CPUs

Classical processors are not only ones vulnerable to side-channel attacks. Graphical Processing Units (GPUs) are among most utilized accelerators that are used today. GPUs can be used to accelerate machine learning, cryptographic, or other tasks. Similar to processors, there is recent trend to share the GPUs among different programs or users. In such case, it is desirable to isolate the different users and programs on these systems, such that one program cannot extract information or data from another without explicit program-to-program communication through supported channels. However, with GPU partitioning features (Multi-instance GPU, or MiG) which might be exploited to transparently enable co-operative multi-process GPU applications, covert-channel attacks become threats to the security of the GPU architecture. This dissertation covers preliminary research on security of MiG in GPUs.

Apart from accelerators such as GPUs, small quantum computers now are being deployed as accelerators available for cloud-based access. Quantum computers based on the Noisy Intermediate-Scale Quantum (NISQ) principles are being actively researched and developed

and made available to access over the internet. As bigger and bigger NISQ quantum computers are introduced, researchers have begun to explore architectures for multi-programming and sharing such computers among different users and cloud-based access for remote users to rent quantum computers will be a dominant use-case. The sharing of quantum computers in a cloud-based setting, however, opens up new security and privacy threats that need to be explored, as well as necessitates the development of defensive techniques. This dissertation covers preliminary research on security of quantum computer systems used as computation accelerators.

1.1.6 Hardware Security Verification Framework

A number of secure processor architectures have been designed over the last decade. They all implemented some new security protection mechanisms in hardware, typically leveraging encryption and hashing to protect user's code or data. The ideas presented by these architectures have been recently incorporated into commercial designs, such as AMD's SEV [13] or Intel's SGX [14] processor architecture security extensions. Today, however, most of these architectures have not been thoroughly and formally verified from the security perspective. Any security flaws with the hardware will undermine the security of the whole platform. There is then a need for security verification frameworks, such as presented in this paper. To help performing security verification of such architectures, research presented in this dissertation also presents a design-time security verification framework for secure processor architectures. The SecChisel [15] framework was built upon the Chisel hardware construction language and tools by adding security tags, and used information flow analysis to verify the security properties of architecture at design-time. The framework performs automatic security tag propagation analysis in a new SecChisel parser and information flow checking using the Z3 SMT solver.

1.2 Dissertation Organization

This dissertation is organized as follows.

Chapter 1 - Introduction This chapter briefly summarized the dissertation work.

Motivation for the research was given and each research theme was summarized separately before listing the dissertation organization.

Chapter 2 - Background This chapter introduces background knowledge required to understand this dissertation work. First, the basic computer microarchitecture is introduced, where caches and TLBs, value predictors, and frontend are separately explained. Second, the concepts of side-channel and covert-channel attacks are introduced. Last, the definition of formal verification and security verification is briefly discussed.

Chapter 3 - Vulnerability Modeling of Timing Attacks on Caches This chapter is mainly composed of two parts: vulnerability modeling approach and secure cache evaluation using the modeling approach. In order to systematically find all the effective timing vulnerabilities, all the possible states for each cache block are first enumerated. Processed by a cache three-step simulator and reduction rules, all the effective vulnerabilities are derived. The model proves to be able to cover the data-based timing vulnerabilities in three steps and focuses on one targeted cache block. Given all the effective vulnerabilities which are found, a number of existing secure caches is tested and evaluated to check the effectiveness of the secure caches against timing attacks derived from the three-step model. The secure cache techniques are summarized accordingly. Finally the estimated performance and secure tradeoffs are shown. Some ideal secure cache proposals are also given.

Chapter 4 - Evaluation of Timing Vulnerabilities of Caches and TLBs This chapter describes how the theoretical three-step modeling approach was used to put it into practice the evaluation on real hardware. To create benchmarks for real processors, the research had to considered different real hardware settings, including memory access timing, running victim and attacker threads in hyper-threading or time-slicing way, different memory access operations and different invalidation-related operations. And further meta C program was written to automatically generate binaries for each of the benchmark tests with certain side-channel vulnerability-required techniques applied. The benchmark can help evaluate different processors to quantify the security of the commodity machines and help build customized defenses for the systems. The vulnerabilities found were validated by running all the three-step combinations on the real hardware. This chapter also covers how the vulnerability modeling and evaluation approach was further extended from x86 to

Arm; and set up a framework to evaluate mobile devices in a cloud testbed. This chapter also presents work on implementation of two secure caches on gem5 to practically evaluate benchmarks and the secure caches. Finally, the chapter covers how the cache three-step model is extended to evaluate side-channel attacks on TLBs.

Chapter 5 - Vulnerability Evaluation of Value Predictors This chapter shows the microarchitecture attacks can be found beyond caches or TLBs. The chapter focuses on value predictors. Threat model and approach for evaluating the value predictor security is presented. Then the new attacks on value predictors are demonstrated, which can be used to attack applications and bypass existing protection schemes and develop the systematic model for analyzing value predictor attacks. Finally three security techniques are proposed to evaluate three security techniques for securing value predictors.

Chapter 6 - Processor Frontend Attacks This chapter covers frontend attacks which were developed and which can covertly send bits between hyper-threads or on the same thread in time-sliced setting using internal-interference. Both timing and power-based variants are developed as well as attacks leveraging special instruction prefixes to force frontend path switches. Frontend attacks' ability to leak information from Intel SGX enclaves and use of the frontend covert-channels as part of a new Spectre attack variant are also shown. Finally, frontend fingerprinting to detect which microcode patch has been applied and practical frontend-based side-channel used to leak information about victim application type are demonstrated.

Chapter 7 - Preliminary Study of Vulnerabilities in Accelerators Beyond CPUs This chapter shows microarchitectural attacks beyond CPUs. The chapter discusses preliminary work on attacks on multi-tenant GPUs as well as quantum computers used as accelerators in cloud-based setting.

Chapter 8 - Hardware Security Verification This chapter shows first security verification framework based on the Chisel hardware construction language, which leverages information flow tracking and an SMT solver. The framework supports verification of nested modules, without having to check individual module separately, static and dynamic tags, declassification mechanisms, and interference tables for third party modules. The framework is evaluated using AES and SHA security RoCCs within a RISC-V Rocket Chip, showing

fast runtime and the ability to detect information leaks due to hardware bugs or Trojans.

Chapter 9 - Conclusion and Future Directions This chapter concludes the whole dissertation. Each work is reviewed and potential future research directions are also given.

Chapter 2

Background

This chapter introduces the basic background knowledge that is needed to understand the work of this thesis.

2.1 Computer Microarchitecture

As can be seen from Figure 2.1, within the computer system, overall design consists of different layers. The typical software layers in a computer system include Application or Machine Code which cover the software running on a commodity computing system today. The typical hardware layers include ISA (Instruction Set Architecture), Microarchitecture, Gate and Registers. Among them, operations of processor below the ISA layer is often hidden from the users. For example, microarchitecture design below the ISA level is proprietary to each vendor and not fully disclosed. Hidden or unknown functionality of the microarchitecture, however, can lead to potential security vulnerabilities.

Typically microarchitecture contains registers, caches, TLBs, execution units, and peripherals such as memory controllers, and others, as has been shown in Figure 2.2. Execution units contain arithmetic logic units (ALUs), floating point units (FPUs), load/store units, branch prediction, and single instruction multiple data (SIMD) units, for example. In particular load/store units work with the memory hierarchy which is used to store information for use in the computer. Caches and TLBs help improve the memory access latency and assist with memory address translation, respectively.

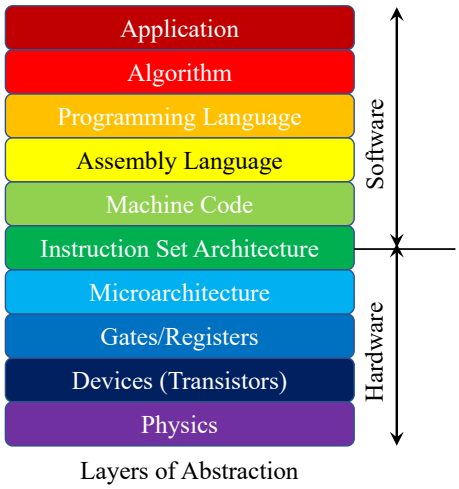


Figure 2.1: Example software and hardware layers of today's computer systems.

2.1.1 Caches and TLBs

Modern memory stores data within memory cells built from transistors and other components on an integrated circuit [16]. There are two main categories of memories, volatile and non-volatile. Non-volatile memories include flash memory, ROM, PROM, EPROM, EEPROM, etc. Examples of volatile memories are Dynamic Random-Access Memory (DRAM), which is used for primary storage, and Static Random-Access Memory (SRAM), which is used for processor caches and TLBs.

Caches in particular are smaller, faster memories located closer to a processor core. They store data and instructions from frequently used main memory locations. As can be seen in Figure 2.3, in general, processors have a hierarchy of multiple cache levels (L1, typically with separate instruction-specific and data-specific storage, L2, and L3, also called last-level cache). Higher-level caches, where L1 is the highest level and last-level cache is the lowest level, have smaller but faster memory, and the lower-level caches have larger but slower memories. "Locality" principle of cache indicates that a processor tends to access the same set of memory locations repetitively over a short period of time, which are stored by the cache and have high likelihood that the data can be reused so that higher performance is gained by maintaining the data in the cache.

However, the caches are of a finite size, and not all data can fit in them. In this case, data found in the cache (cache hit) can be accessed more quickly. On the other hand, if the

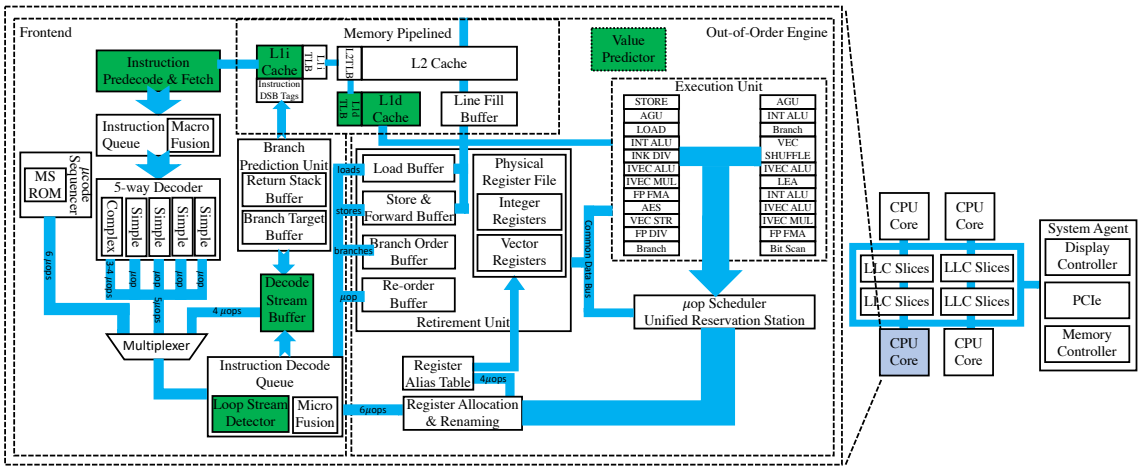


Figure 2.2: Critical microarchitectural states that possibly contain timing and power variations which can be utilized to perform side-channel or covert-channel attacks. Within the frontend, buffer-related structures such as return stack buffer, branch target buffer, decode stream buffer, and loop stream detector are all possible targets. For the out-of-order engine, there are load/store buffers, branch order buffer, and re-order buffer, as well as port contention through the execution units. Value predictor is not implemented on commercial processors but are actively developed. The whole memory pipeline should be counted as potential targets of triggering side-channel or covert-channel attacks. Microarchitectures filled with green are the targets studied in this dissertation.

data is not found (cache miss), it requires fetching data from the main memory and may also need to evict data from the cache in order to store the incoming data. These timing differences related to the memory operations involving caches, such as accesses resulting in cache hits vs. misses, can reveal information about the addresses or even data in the cache (for instruction caches it may be possible to reveal information about instructions as well). The cache coherence protocol can also change the cache states and affect the timing of the memory operations. The cache coherence may invalidate a cache block from a remote core, resulting in a cache miss in the local core, for example. Also, the timing of a cache flush operations varies depending on whether the data to be flushed is in the cache or not. Flushing an address containing dirty (modified) data using *clflush* is slow as it has to be moved back to the main memory, while flushing an address containing clean (unmodified data, same as in the main memory) is fast, as the data can be simply discarded, for example. From these timing differences of memory-related operations, the attacker can infer a data's specific memory address or corresponding cache index value, and thus learn some information about the victim's secrets.

Translation Look-aside Buffers (TLBs) are also cache-like structures. They store virtual

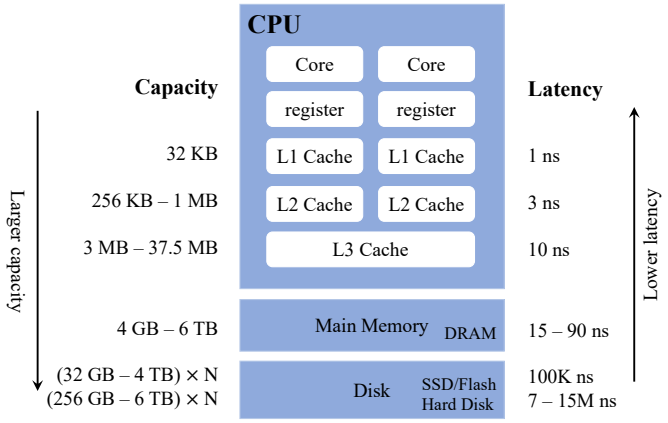


Figure 2.3: Memory hierarchy (including caches) of a typical modern computer processor.

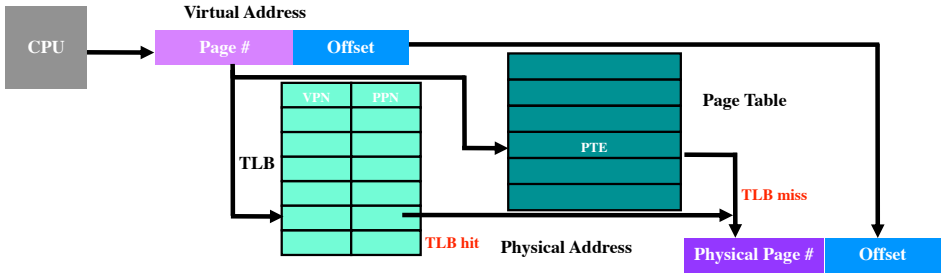


Figure 2.4: Simplified schematic of virtual to physical memory address translation in modern computer processors. VPN represents Virtual Page Number; PPN represents Physical Page Number; PTE represents Page Table Entries.

address to physical address translation, as is shown in Figure 2.4. The virtual memory is the memory space as seen from the perspective of a process; this space is often split into pages of mixed page sizes (4KB and 2MB). The page table stored in main memory keeps track of where the virtual pages are stored in the physical memory. This method of having virtual to physical memory uses two or more memory accesses to access each memory location (first one or more accesses to get the page table entry, and then one access for the actual data). First, the page table is looked up based on the virtual page number to get the physical page number. Second, the physical page number can be combined with the page offset to give the actual physical address where the target data is located. To reduce the overheads of the memory accesses, the TLB is often implemented to reduce the time taken to access the memory locations by caching the virtual to physical translations so that the memory accesses to obtain the page table entry can be avoided if the entry is already in the TLB. Upon each memory reference, the hardware checks the TLB to see whether the virtual to

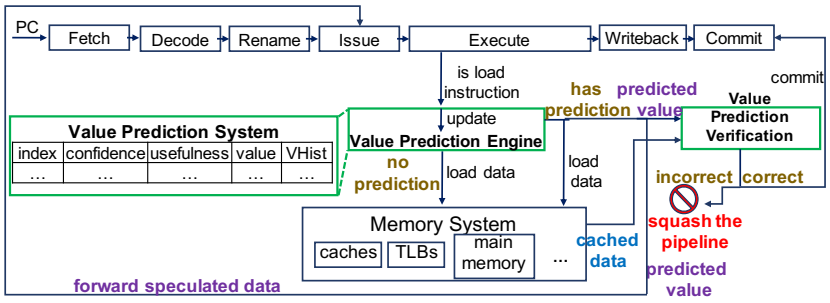


Figure 2.5: Processor pipeline with a Value Prediction System (VPS).

physical translation is there. If yes, it is a TLB hit, and the translation is read from the TLB. The physical page number is returned and is used to access the memory. If the translation is not in the TLB, it is a TLB miss and the page table must be checked by accessing main memory, which results in a different access timing compared with a TLB hit. Similar to caches, TLBs may have multiple levels.

2.1.2 Value Predictors

A processor pipeline with a value predictor is shown in Figure 2.5. A typical value predictor [17] uses the instruction address (program counter) to keep track of the loaded values. For each instruction, once a value is predicted correctly for more than a $confidence$ number of times, the predictor starts to use the previous (last) value when a cache miss occurs. This allows instructions to proceed while the actual data value is still being fetched, providing data with high confidence and preventing the performance penalty of a cache miss. Different variants of value predictors have been demonstrated, which can improve the processors' performance from 4.8% [11] to 11.2% [18].

For a typical predictor shown in Figure 2.5, for each load instruction, the Value Prediction System (VPS) keeps track of the index, the data value to be predicted, and the past value history (VHist). The index can be the program counter (PC), or the data address, depending on the type of value predictor. The VPS typically further uses the full address as the index, e.g., [12]. Using a subset of the address bits is possible, but will introduce conflicts between different addresses and reduce the prediction rate. On each load, the value history and predicted value are updated. If the predicted value is verified to be correct (after the actual

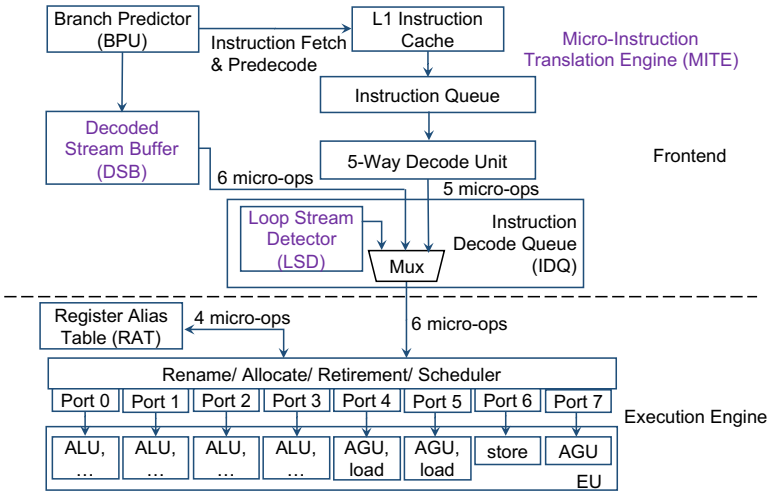


Figure 2.6: Microarchitecture details of the frontend and the execution engine, based on [19].

load data is available), the $confidence$ and $usefulness$ values are increased and there are no changes to the original load and dependent instructions. Meanwhile, a misprediction will cause not only the predicted load but also dependent instructions to be squashed and reissued. Within the VPS, if there are not enough entries, the entry with the smallest $usefulness$ value will be evicted.

2.1.3 Processor Frontend

Within the processor frontend, instruction decoding and delivery to the backend has multiple paths: through the Micro-Instruction Translation Engine (MITE), the Decoded Stream Buffer (DSB), also called the micro-op cache, and the Loop Stream Detector (LSD), as is seen from Figure 2.6.

Given that MITE path has low throughput and high power consumption, the DSB has been added and the micro-ops decoded by MITE are inserted into the DSB [19] in modern Intel processors. If the micro-ops are available in the DSB, the micro-op stream is sent directly from DSB to the Instruction Decode Queue (IDQ), bypassing the MITE, therefore saving power and improving throughput. The instruction delivery path from DSB is also shorter than MITE (shorter by 2 – 4 cycles), so the pipeline latency is reduced as well [19].

Further, there is also the LSD located within the IDQ. If the micro-op stream belongs to a qualified loop (discussed in Section 6.2), all the micro-ops of the loop code can be issued

directly from LSD to the backend, bypassing DSB as well. The purpose of the LSD is to help save power, but it also can help performance by providing higher instruction delivery throughput. When branch mis-prediction occurs, e.g., at the end of the loop, or the number of micro-ops within the loop exceeds the limit that the LSD can handle, LSD is not used and micro-ops are delivered from the DSB. Furthermore, if the micro-ops exceed the DSB limit or belong to a newly accessed micro-ops, they are processed by the MITE. We also note that the DSB is inclusive of LSD, and MITE is inclusive of DSB as well [19], e.g., eviction of micro-ops from DSB will cause their eviction from LSD. Although DSB and LSD are partitioned in Intel processor when two hyper-threads are actively running, our analysis indicates that DSB in Intel processors is fully assigned to one thread if the other is idle or not executing. When the second thread becomes active, DSB becomes partitioned, which forces DSB evictions of micro-ops of the first thread to occur. Further, MITE is a shared resource, and activity of two threads mutually affects the micro-op decoding.

LSD Behavior. The LSD can continuously stream the same sequence of up to 64 micro-ops, directly from the IDQ to the backend [19]. While the LSD is active, the rest of the frontend is effectively disabled. In order to generate detectable timing and power difference between LSD vs. DSB and DSB vs. MITE, one can control micro-op number within a loop to either make it fit in the LSD where instruction delivery starts with LSD only, or exceed the LSD limit so the processor falls back to use DSB or MITE, creating detectable timing and power changes.

DSB Behavior. The DSB is constructed as a cache-like structure with 32 sets and 8 ways per set [19] in recent Intel processors. Each line can store up to 6 micro-ops or 32 bytes (so DSB can hold at most 1536 micro-ops in total). Based on our reverse engineering as well as Intel manuals [19], we find that when there is only one thread running on the hardware core, instructions' virtual address bits `addr[4:0]` are used as the byte offset within the 32-byte window, and `addr[9:5]` are the set index bits into the 32 DSB sets. However, when two threads are running in parallel on the hardware core, the DSB is set partitioned, and half the sets are assigned to each thread based on our experimental results. This means that although the DSB is partitioned by sets when two threads are running, if there is only one thread being active, the thread is assigned to all the DSB sets. Whether the DSB is currently

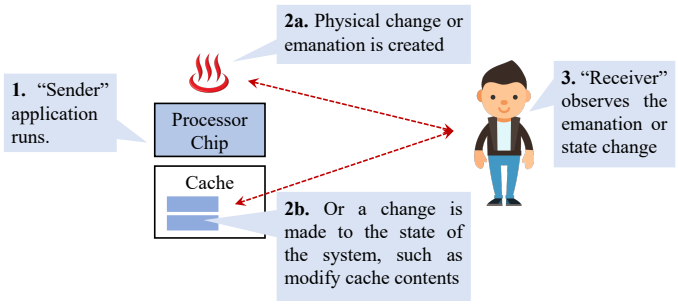


Figure 2.7: Typical covert channel setup.

partitioned or not can be detected by an application by checking the increased MITE usage (when DSB is partitioned, more instructions will conflict with each other causing DSB evictions and increased MITE usage). The behavior was tested on Intel Xeon E-2174G with LSD disabled to show the conflicts are not influenced by LSD. We also tested on Intel Gold 6226 with LSD enabled, and observe similar results. Further, we tested Intel Gold 6226 with LSD enabled, but each test thread was set to access larger blocks of instructions which do not fit in LSD (forcing processor to use DSB even if LSD is enabled), and similar results are observed on this machine.

MITE Behavior. Regarding the MITE structure, the instruction cache, instruction queue, and the decode unit are shared among the two threads. Typically the instruction cache is 32 KB and 8-way associative and instruction queue contains 50 entries.

2.2 Side-Channel and Covert-Channel Attacks

A covert channel is a communication channel that was not originally proposed to transfer information. As is shown in Figure 2.7, it can become an intentional communication between a sender and a receiver by leveraging unusual methods for information communications. Compared with a side channel, it is easier to establish a covert channel communication and covert channel can be used as a precursor for the side-channel attacks.

A side channel is similar to a covert channel, but in a side channel the sender does not intend to communicate to the receiver and transfer information. Instead, the sending (i.e., leaking) of data is a side effect of the sender's behavior and implementation as well as the hardware and the software used. In a side channel, we call the sender a "victim" and the

receiver an "attacker".

The sender's behavior that can be utilized by the side channels and the covert channels to transfer information can be timing, power, thermal emanations, electro-magnetic (EM) emanations, acoustic emanations, for example.

2.2.1 Examples of Previously Discovered Timing Vulnerabilities

Researchers have previously proposed to use the timing differences in memory-related operations to attack software, e.g., [20, 21, 22, 23, 24]. Especially, the timing side-channel attacks often focus on cryptographic applications, e.g., attacks on software using AES encryption or decryption with table lookups [25]. Further, there are many timing covert-channel attacks, where the sender and receiver cooperate to leak data. Previously, security vulnerabilities have been uncovered in all the different levels of caches [22, 26, 27, 28], e.g., the last-level cache mentioned in study [28] and cross-core cache covert channels [29], as well as due to port contention in the execution engine [30], branch predictors [31, 32], or memory controllers [33], for example. Security community has especially focused on the speculative execution attacks, following disclosure of Spectre [3] and Meltdown [4]. Other recently explored vulnerabilities include attacks that abuse branch prediction, but not for Spectre-like attacks. This includes BranchScope vulnerability [31] or Jump over ASLR (address space layout randomization) type vulnerabilities [32]. There are also attacks that leverage prefetchers [34] and value predictors [35]. Most recently, researchers have also demonstrated microarchitectural replay attacks [36] and attacks abusing network-on-chip (NoC) [37].

In modern processor caches, two types of memory-related operations exhibit timing variations that can be abused for timing side or covert channel attacks in processor data caches. First, memory access operations, such as loads and stores can be fast (e.g., a cache hit) or slow (e.g., a cache miss). Second, invalidation-related operations, such as cache flush, can be fast (e.g., there is no dirty data in cache so flush finishes quickly) or slow (e.g., there is dirty data in the cache so it has to be written back, resulting in longer timing). These variations in timing have been exploited to leak sensitive information. Especially, a large number of different cache timing side-channel and covert-channel attacks have been

presented in literature, e.g., [20, 21, 22, 23, 24]. And, there are many secure hardware cache designs which aim to prevent these different attacks, e.g., [1, 38, 39, 40, 2, 41, 42, 43, 44, 45]. However, even if the cache-based attacks are mitigated, TLB-based attacks are the next attack vector that malicious attackers might use.

Compared with caches for which there is large number of known attacks, there are only two published TLB-based timing attacks[1]. TLBleed attack [48] uses timing channels combined with machine learning to create attack which is able to leak bits of secret keys from the RSA algorithm (they also show attack for the EdDSA algorithm). They leverage the Prime + Probe [49] attack strategy previously applied in processor caches. Prior to TLBleed, the Double Page Fault attack [50] leveraged the Cache Collision [23] attack strategy previously applied in processor caches. It requires the victim to access some kernel memory pages twice, and uses the fact that an access to a previously allocated kernel virtual pages will bring in TLB entries, even if page fault is generated and accesses permission checks failed. The timing of the second access thus reveals information on whether a inherent TLB hit happened.

Besides memory systems and execution units, processor frontend can also be a potential target for side- and covert-channel attacks. The existing security attacks [51, 52] focus on studying eviction of DSB and how it can cause timing differences that attackers can exploit. They used reverse-engineering to extract hidden features of the frontend and presented attack scenarios by using frontend as a timing side-channel to transmit secret data as well as proposed hardware mitigation with both security and performance analyses.

1. The Leaky Cauldron [46] attack is also related to TLB and targets Intel SGX. However, it does not depend on hits and misses in the TLB, but instead it relies on the assumption that the attacker can evict the enclave entries in the TLB, so an enclave's memory access will trigger a page table walk, and the malicious OS can get the page access pattern trace. The Malicious Management Unit [47] attack makes use of the Memory Management Unit (MMU) to build eviction set of virtual addresses to allow the page table entries to map to certain cache sets in the CPU caches (especially in the Last-Level Cache). In this case, eviction sets which can bypass the software-based defenses are formed and can trigger cache timing attacks in LLC. However, similar to the Leaky Cauldron [46] attack, this attack also does not depend on hits and misses in the TLB.

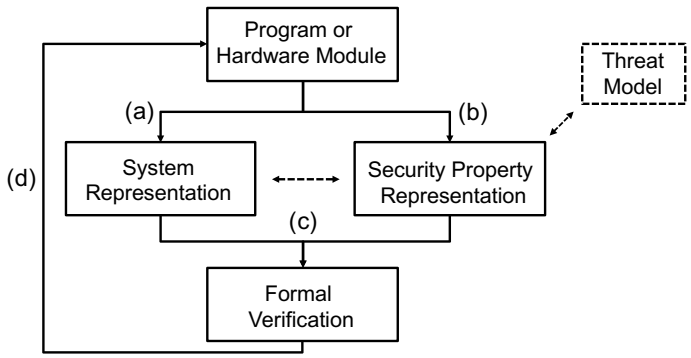


Figure 2.8: General procedure for security verification.

2.3 Formal Verification and Security Verification

For the software and hardware systems, formal verification is the process of proving or disproving the correctness of the target program or hardware module using formal methods of mathematics with certain formal specification or property.

A typical flow of the security verification process is shown at a high level in Figure 2.8. The starting point is the target program or hardware module, either an already existing system or a design of some new system whose security properties need to be verified. From the target system, or design, a representation of the system needs to be obtained in the verification tools, (a) in Figure 2.8. In parallel, the security properties of the target system need to be specified, (b) in Figure 2.8. The security properties are closely tied to the system's assumed threat model. The security properties can be specified separately or together within the representation of the system, in which case (a) and (b) would be done together. The final step is the actual verification process which takes the system representation and security properties as input, and returns whether the verification passed or failed, (c) in Figure 2.8. If the verification fails, the design needs to be updated and re-evaluated, (d) in Figure 2.8.

Chapter 3

Vulnerability Modeling of Timing Attacks on Caches

Research presented in this chapter focuses on ways to systematically analyze and obtain the complete set of cache timing side-channel vulnerabilities, including known and new unknown vulnerabilities. The work later combines the theory with the practice to create security benchmarks based on the theoretical vulnerabilities. The chapter further shows evaluation of secure caches' effectiveness against the vulnerabilities. Prior research has mostly focused on uncovering individual attacks, while to help to systematically protect whole computer systems, this research aims to uncover all the theoretical vulnerabilities, so they can be defended against.

3.1 Three-Step Model

This section explains how we developed the three-step modeling approach and used it to model the behavior of the cache logic and to enumerate all the possible cache timing vulnerabilities in caches.

3.1.1 Introduction to the Three-Step Model

We have observed that all of the existing cache timing attacks can be modeled with three steps of memory-related operations. Here, "memory-related operation" refers to loads, stores,

or different flushes that can be done by the victim or the attacker on the same core or different cores. When the victim and the attacker are on different cores, cache coherence will also be triggered when one of the memory-related operations is performed.

The three-step model has three steps, as the name implies. In *Step* 1, a memory operation is performed, placing the cache in an initial state known to the attacker (e.g., a new piece of data at some address is put into the cache or the cache block is invalidated). Then, in *Step* 2, a second memory operation alters the state of the cache from the initial state. Finally, in *Step* 3, a final memory operation is performed, and the timing of the final operation *Step* 3 enables a receiver to learn how *Step* 2 has perturbed the state established by *Step* 1.

For example, in Flush + Reload [27] attack, in *Step* 1, a cache block is flushed by the attacker. In *Step* 2, security critical data is accessed by, for example, victim's AES encryption operation. In *Step* 3, the same cache block as the one flushed in *Step* 1 will be accessed and the time of the access will be measured by the attacker. If the victim's secret-dependent operation in *Step* 2 accesses the cache block, in *Step* 3 there will be a cache hit and fast timing of the memory operation will be observed, and the attacker learns the victim's secret address.

We write the three steps as: *Step* 1 $\rightsquigarrow$ *Step* 2 $\rightsquigarrow$ *Step* 3, which represents a sequence of steps taken by the attacker or the victim. To simplify the model, we focus on memory-related operations affecting one single cache block (also called cache slot, cache entry, or cache line). Cache block is the smallest unit of the cache. Since all the cache blocks are updated following the same cache state machine logic, it is sufficient to consider only one cache block.

3.1.2 States of the Three-Step Model

When modeling the attacks, we propose that there are 17 possible states for a cache block. Table 4.8 lists all the 17 possible states of the cache block for each step in the three-step model and their definitions. Figure 3.1 graphically shows for each possible state how the memory location maps to the cache block.

In each sub-figure of Figure 3.1, left-most part shows the state or states being described in the sub-figure. Middle part shows the cache entries and their states. For all sub-figures, the middle cache block (shown in bold) is the targeted cache block. Right-most part shows

Table 3.1: The 17 possible states for a single cache block in the three-step model.

State	Description
V_u	A memory location u belonging to the victim is accessed and is placed in the cache block by the victim (V). Attacker does not know u, but u is from a set x of memory locations, a set which is known to the attacker. It may have the same index as a or a^{alias}, and thus conflict with them in the cache block. The goal of the attacker is to learn the index of the address u. The attacker does not know the address u, hence there is no A_u in the model.
A_a or V_a	The cache block contains a specific memory location a. The memory location is placed in the cache block due to a memory access by the attacker, A_a, or the victim, V_a. The attacker knows the address a, independent of whether the access was by the victim or the attacker themselves. The address a is within the range of sensitive locations x. The address a is known to the attacker.
$A_{a^{alias}}$ or $V_{a^{alias}}$	The cache block contains a memory address a^{alias}. The memory location is placed in the cache block due to a memory access by the attacker, $A_{a^{alias}}$, or the victim, $V_{a^{alias}}$. The address a^{alias} is within the range x and not the same as a, but it has the same address index and maps to the same cache block, i.e. it "aliases" to the same block. The address a^{alias} is known to the attacker. To note that a^{alias} definition effectively focuses on describing the index bits and this can represent leaking different subset of address bits through the cache channel.
A_d or V_d	The cache block contains a memory address d. The memory address is placed in the cache block due to a memory access by the attacker, A_d, or the victim, V_d. The address d is not within the range x. The address d is known to the attacker.
A^{inv} or V^{inv}	The cache block is now invalid. The data and its address are "removed" from the cache block by the attacker, A^{inv}, or the victim, V^{inv}, as a result of cache block being invalidated, e.g., this represents a cache flush of the whole cache.
A_a^{inv} or V_a^{inv}	The cache block state can be anything except a in this cache block now. The data and its address are "removed" from the cache block by the attacker, A_a^{inv}, or the victim, V_a^{inv}. E.g., by using a flush instruction such as $clflush$ that can flush specific address, or by causing certain cache coherence protocol events that force a to be removed from the cache block. The address a is known to the attacker.
$A_{a^{alias}}^{inv}$ or $V_{a^{alias}}^{inv}$	The cache block state can be anything except a^{alias} in this cache block now. The data and its address are "removed" from the cache block by the attacker, $A_{a^{alias}}^{inv}$, or the victim, $V_{a^{alias}}^{inv}$. E.g., by using a flush instruction such as $clflush$ that can flush specific address, or by causing certain cache coherence protocol events that force a^{alias} to be removed from the cache block. The address a^{alias} is known to the attacker.
A_d^{inv} or V_d^{inv}	The cache block state can be anything except d in this cache block now. The data and its address are "removed" from the cache block by the attacker A_d^{inv} or the victim V_d^{inv}. E.g., by using a flush instruction such as $clflush$ that can flush specific address, or by causing certain cache coherence protocol events that force d to be removed from the cache block. The address d is known to the attacker.
V_u^{inv}	The cache block state can be anything except u in the cache block. The data and its address are "removed" from the cache block by the victim V_u^{inv} as a result of cache block being invalidated, e.g., by using a flush instruction such as $clflush$, or by certain cache coherence protocol events that force u to be removed from the cache block. The attacker does not know u. Therefore, the attacker is not able to trigger this invalidation and A_u^{inv} does not exist in the model.
$\star$	Any data, or no data, can be in the cache block. The attacker has no knowledge of the memory address in this cache block.

the memory region in relation to the cache block. Note that the addresses a and a^{alias} are within the sensitive set of addresses x, while d is outside the set of sensitive addresses (for simplicity the set is shown as a contiguous region, but it can be any set). Also, A represents the operations performed by the attacker and V represents the victim's operations.

Figure 3.1a shows the description of the possible state V_u, where address u is within sensitive set and unknown to the attacker. Therefore, it can possibly map to any cache block including the target cache block shown in the middle of the sub-figures. Since its position in the cache and specific address is unknown, we show V_u in dashed lines. Meanwhile, Figure 3.1e shows the description of the possible state V_u^{inv}, which is result of the victim invalidating data at the sensitive address u. Further, Figure 3.1f shows the description of

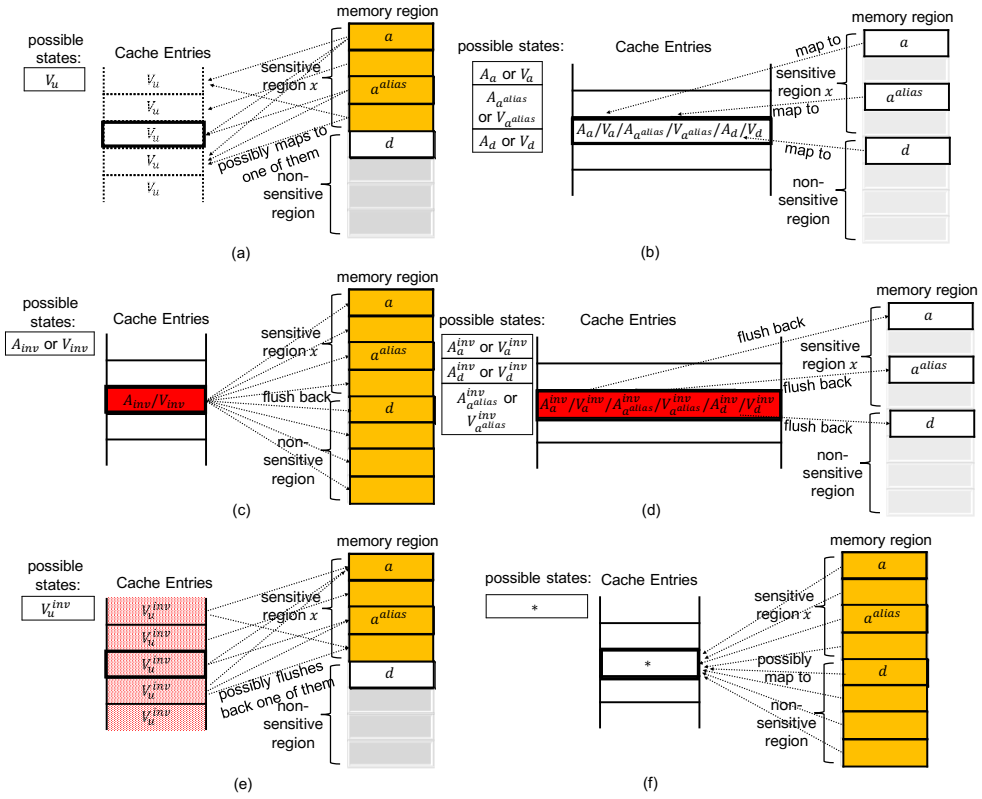


Figure 3.1: The 17 possible states for a cache block in the three-step model: (a) V_u state; (b) $A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d$ states; (c) A^{inv}/V^{inv} states; (d) $A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv}/A_d^{inv}/V_d^{inv}$ states; (e) V_u^{inv} state; (f) $\star$ state.

the possible state $*$, which represents the lack of knowledge of the address for the attacker to this corresponding cache block. It can refer to any or no address in the memory.

Figure 3.1b shows the description of the possible state $A_a/V_a/A_{a^{alias}}/V_{a^{alias}}\ /A_d/V_d$. Their addresses are all known to the attacker and map to the same targeted cache block. Both a and a^{alias} are within the sensitive set of addresses x and a^{alias}, as its name indicates, is a different address than a but still within set x and maps to the same cache block as a. Meanwhile, address d is outside of the set x. Next, Figure 3.1d shows the description of the possible state $A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv}/A_d^{inv}/V_d^{inv}$, which correspond to invalidation of the address shown in the subscript of the state. Some additional possible invalidation states, A^{inv}/V^{inv}, are shown in Figure 3.1c. These states indicate no valid address is in the cache block. Therefore, all the possible addresses that mapped to this cache block before the invalidation, e.g., a, a^{alias}, d and u (if it mapped to this block), will be flushed back to the memory by the step A^{inv}/V^{inv}.

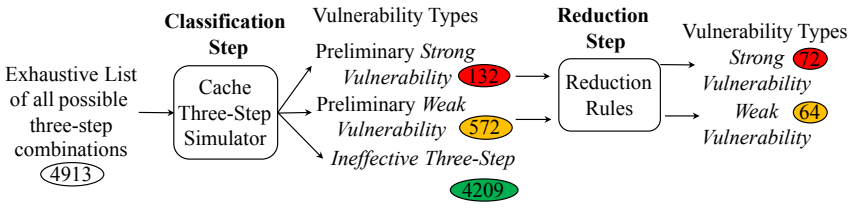


Figure 3.2: Procedure to derive the effective types of three-step timing vulnerabilities. Ovals refer to the number of vulnerabilities in each category.

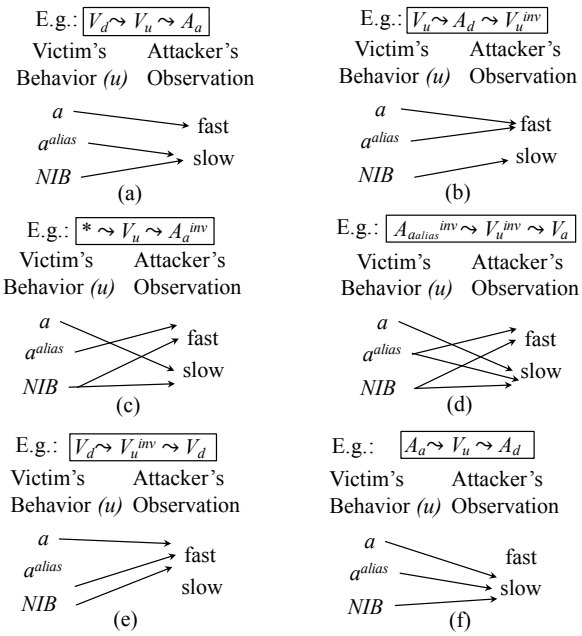


Figure 3.3: Examples of relations between victim's behavior (u) and attacker's observation for each vulnerability type: (a),(b) *Strong Vulnerability*; (c),(d) *Weak Vulnerability*; (e),(f) *Ineffective Three-Step*.

3.1.3 Derivation of All Cache Timing Vulnerabilities

With the 17 candidate states for each step, there are in total $17 * 17 * 17 = 4913$ combinations of three steps. We developed a cache three-step simulator and a set of reduction rules to process all the three-step combinations and decide which ones can indicate a real attack. As is shown in Figure 3.2, the exhaustive list of the 4913 combinations will first be input to the cache three-step simulator, where the preliminary classification of vulnerabilities is derived. The effective vulnerabilities will then be sent as the input to the reduction rules to remove the redundant three steps and obtain final list of vulnerabilities.

Algorithm 1 Pseudo code for the cache three-step simulator.

Input: $state[]$: a list containing 17 possible states for each of the step
Output: $strong[]$: a list containing all the vulnerabilities that belong to the *Strong* type
$weak[]$: a list containing all the vulnerabilities that belong to the *Weak* type
$ineffective[]$: a list containing all the ineffective type

for $step1 \in len(state[])$ **do**
  for $step2 \in len(state[])$ **do**
    for $step3 \in len(state[])$ **do**
      $steps = [state[step1], state[step2], state[step3]]$
      $candidates = []$ // array to store all possible candidate combinations of this three-step pattern
      $res = []$ // array to store all possible timing observation regading different candidate combinations for this three-step pattern
      if ($u\_related(steps[0])$ **or** $u\_related(steps[1])$ **or** $u\_related(steps[2])$) **then**
        for $possi\_candidate \in 3$ **do**
          $candidates.append((change\_u(steps[0], possi\_candidate),$
          $change\_u(steps[1], possi\_candidate), change\_u(steps[2], possi\_candidate)))$
          // V_u's candidates are V_a, $V_{a^{alias}}$ and V_{NIB}; V_u^{inv}'s candidates are V_a^{inv}, $V_{a^{alias}}^{inv}$ and V_{NIB}^{inv}. Both candidate's number is 3.
        end for
        for $i \in 3$ **do**
          $res.append(output\_timing(candidates[i]))$
        end for
        if $judge\_type(res) == Strong$ **then**
          strong.append(steps)
        else
          if $judge\_type(res) == Weak$ **then**
            $weak.append(steps)$
          else
            $ineffective.append(steps)$
          end if
        end if
      else
        $ineffective.append(steps)$
        continue
      end if
    end for
  end for
end for

Cache Three-Step Simulator

We developed a cache three-step simulator that simulates the state of one cache block and derives the attacker's observations in the last step of the three-step patterns that it analyzes, for different possible u. Since u is in secure range x, the possible candidates of u for a cache block are a, a^{alias} and NIB (Not-In-Block). Here, NIB indicates the case that u does not have same index as a or a^{alias} and thus does not map to this cache block.

The cache three-step simulator is implemented as a Python script and it's pseudo-implementation is shown in Algorithm 1. Simulator's inputs are 17 possible states for each of the step. Outputs are all the vulnerabilities that belong to the *Strong* or the *Weak* type or the *Ineffective* type. The simulator uses a nested `for` loop to check all possible combinations (4913) of the three-step patterns. For each step of each pattern, if it is V_u, this step will be

extended to be one of three candidates: V_a, $V_{a^{alias}}$ and V_{NIB}. If it is V_u^{inv}, this step will be extended to be one of three candidates: V_a^{inv}, $V_{a^{alias}}^{inv}$ and V_{NIB}^{inv}. We wrote a function *output_timing* that takes three known memory access steps as input and output whether fast or slow timing will be observed for the last step. In this case, for each of the u-related step's candidate, we can derive a timing observation. Using these timing observation, function *judge_type* decides whether a three-step pattern is a potential vulnerability by analyzing whether the attacker is able to observe different and unambiguous timing for different values of u or not.

The simulator categorizes all the three-step patterns into three categories, as listed below. Figure 3.3 shows two examples for the *Strong Vulnerability* (a, b), *Weak Vulnerability* (c, d) and *Ineffective Three-Step* (e, f), categories respectively.

1. *Strong Vulnerability:* When a fast or slow timing is observed by the attacker, he or she is able to uniquely distinguish the value of u (either it maps to some known address or has the same index with some known address). In this case, the vulnerability has strong information leakage (i.e. attacker can directly obtain the value of u based on the observed timing). We categorize these vulnerabilities to be strong. E.g., for $V_d \rightsquigarrow V_u \rightsquigarrow A_a$ vulnerability shown in Figure 3.3a, if u maps to a, the attacker will always derive fast timing. If u is a^{alias} or NIB, slow timing will be observed. This indicates that the attacker is able to unambiguously infer the victim's behavior (u) from the timing observation.
2. *Weak Vulnerability:* When fast or slow timing is observed by the attacker, he or she knows it corresponds to more than one possible value of u (e.g., a or a^{alias}). For these vulnerabilities, timing variation can still be observed due to different victim's behavior. However, the attacker cannot learn the value of the index of the address u unambiguously. E.g., for type $\star \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ shown in Figure 3.3c, when fast timing is observed, u possibly maps to a^{alias} or NIB (the reason for the possibility of u mapping to NIB to derive fast timing is that due to the $\star$ in *Step* 1, the cache have chances to not own a before the final step and then A_a^{inv} will receive an fast invalidation). On the other hand, when slow timing is observed, u possibly maps to a

or NIB. This pattern leads to uncertain u guess about value of u based on timing observation.

3. *Ineffective Three-Step:* The remaining types are treated to be ineffective. E.g., for type $A_a \rightsquigarrow V_u \rightsquigarrow A_d$ shown in Figure 3.3f, no matter what the value of u is, attacker's observation is always a slow timing.

After computing the type of all the three-step patterns, the cache three-step simulator will output effective (*Strong Vulnerability* or *Weak Vulnerability*) three-step patterns. We only list and analyze the *Strong* vulnerabilities in this work. *Weak* vulnerabilities are left for future work when channels with smaller channel capacities are desired to be analyzed.

Reduction Rules

We also have developed rules that can further reduce the output list of all the effective three steps from the cache three-step simulator. Figure 3.2 shows how the output of the simulator is filtered through the reduction rules to get the final list of vulnerabilities. Reduction's goal is to remove vulnerabilities of repeating or redundant types from the lists to form effective *Strong Vulnerability* or *Weak Vulnerability* output. A script was developed that automatically applies below reduction rules to the output of the simulator to get the final list of vulnerabilities. A three-step combination will be eliminated if it satisfies one of the below rules:

1. Three-step patterns with two adjacent steps which are repeating, or which are both known to the attacker, can be eliminated, e.g., $A_d \rightsquigarrow A_a \rightsquigarrow V_u$ can be reduced to $A_a \rightsquigarrow V_u$, which is equivalent to $\star \rightsquigarrow A_a \rightsquigarrow V_u$. Therefore, $A_d \rightsquigarrow A_a \rightsquigarrow V_u$ is a repeat type of $\star \rightsquigarrow A_a \rightsquigarrow V_u$ and can be eliminated.
2. Three-step patterns with a step involving a known address a and an alias to that address a^{alias} gives the same information. Thus three step combinations which only differ in use of a or a^{alias} cannot represent different attacks, and only one combination needs to be considered. For example, $V_u \rightsquigarrow A_{a^{alias}} \rightsquigarrow V_u$ is a repeat type of $V_u \rightsquigarrow A_a \rightsquigarrow V_u$, and we will eliminate the first pattern.

Table 3.2: The table shows all the cache timing vulnerabilities where the last step is a memory access-related operation. For *Step* 3, *fast* indicates a cache hit must be observed to derive sensitive address information, while *slow* indicates a cache miss must be observed.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
Cache Internal Collision	A^{inv}	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	V^{inv}	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	A_d	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	V_d	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	$A_{a^{alias}}$	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	$V_{a^{alias}}$	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	A_a^{inv}	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
	V_a^{inv}	V_u	V_a (fast)	IH	Cache Internal Collision attack [23]
Flush + Reload	A_a^{inv}	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	V_a^{inv}	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	A^{inv}	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	V^{inv}	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	A_d	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	V_d	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	$A_{a^{alias}}$	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
	$V_{a^{alias}}$	V_u	A_a (fast)	EH	Flush + Reload attack [27, 53], Evict + Reload attack [54]
Reload + Time	V_u^{inv}	A_a	V_u (fast)	EH	**new**
	V_u^{inv}	V_a	V_u (fast)	IH	**new**
Flush + Probe	A_a	V_u^{inv}	A_a (slow)	EM	SpectrePrime, MeltdownPrime attack [55]
	A_a	V_u^{inv}	V_a (slow)	IM	**new**
	V_a	V_u^{inv}	A_a (slow)	EM	**new**
	V_a	V_u^{inv}	V_a (slow)	IM	**new**
Evict + Time	V_u	A_d	V_u (slow)	EM	Evict + Time attack [49]
	V_u	A_a	V_u (slow)	EM	Evict + Time attack [49]
Prime + Probe	A_d	V_u	A_d (slow)	EM	Prime + Probe attack [49, 21], Alias-driven attack [56]
	A_a	V_u	A_a (slow)	EM	Prime + Probe attack [49, 21], Alias-driven attack [56]
Bernstein's Attack	V_u	V_a	V_u (slow)	IM	Bernstein's attack [22]
	V_u	V_d	V_u (slow)	IM	Bernstein's attack [22]
	V_d	V_u	V_d (slow)	IM	Bernstein's attack [22]
	V_a	V_u	V_a (slow)	IM	Bernstein's attack [22]
Evict + Probe	V_d	V_u	A_d (slow)	EM	**new**
	V_a	V_u	A_a (slow)	EM	**new**
Prime + Time	A_d	V_u	V_d (slow)	IM	**new**
	A_a	V_u	V_a (slow)	IM	**new**
Flush + Time	V_u	A_a^{inv}	V_u (slow)	EM	**new**
	V_u	V_a^{inv}	V_u (slow)	IM	**new**

3. Three-step patterns with steps V_u and $V_{u^{inv}}$ in adjacent consecutive steps with each other will only keep the latter step and eliminate the first step. For example, $A_a \rightsquigarrow V_u \rightsquigarrow V_{u^{inv}}$ can be reduced to $A_a \rightsquigarrow V_{u^{inv}}$ and further equivalent to $\star \rightsquigarrow A_a \rightsquigarrow V_{u^{inv}}$. So $A_a \rightsquigarrow V_u \rightsquigarrow V_{u^{inv}}$ can be eliminated.

Categorization of *Strong*Vulnerabilities

As is shown in Figure 3.2, after applying the reduction rules, there are remaining 72 types of *Strong* vulnerabilities. In Section 3.1.5, we analyze the soundness of the three-step model to demonstrate that the three-step model can cover all possible cache timing side-channel vulnerabilities. And if there is a vulnerability, it can always be reduced to a model that requires only three steps. Table 3.2 lists all the vulnerability types of which the last step is a memory access and Table 3.3 shows all the vulnerability types of which the last step is an

Table 3.3: The table shows the second part of the timing cache side-channel vulnerabilities where the last step is an invalidation-related operation. For *Step* 3, *fast* indicates no corresponding address of the data is invalidated, while *slow* indicates invalidation operation makes some data invalid, causing longer processing time.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
Cache Internal Collision Invalidation	A^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	V^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	A_d	V_u	V_a^{inv} (slow)	IH	**new**
	V_d	V_u	V_a^{inv} (slow)	IH	**new**
	$A_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
	$V_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
Flush + Flush	A_a^{inv}	V_u	V_a^{inv} (slow)	IH	Flush + Flush attack [26]
	V_a^{inv}	V_u	V_a^{inv} (slow)	IH	Flush + Flush attack [26]
	A_a^{inv}	V_u	A_a^{inv} (slow)	EH	Flush + Flush attack [26]
	V_a^{inv}	V_u	A_a^{inv} (slow)	EH	Flush + Flush attack [26]
Flush + Reload Invalidation	A^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	V^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	A_d	V_u	A_a^{inv} (slow)	EH	**new**
	V_d	V_u	A_a^{inv} (slow)	EH	**new**
	$A_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
	$V_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
Reload + Time Invalidation	V_u^{inv}	A_a	V_u^{inv} (slow)	EH	**new**
	V_u^{inv}	V_a	V_u^{inv} (slow)	IH	**new**
Flush + Probe Invalidation	A_a	V_u^{inv}	A_a^{inv} (fast)	EM	**new**
	A_a	V_u^{inv}	V_a^{inv} (fast)	IM	**new**
	V_a	V_u^{inv}	A_a^{inv} (fast)	EM	**new**
	V_a	V_u^{inv}	V_a^{inv} (fast)	IM	**new**
Evict + Time Invalidation	V_u	A_d	V_u^{inv} (fast)	EM	**new**
	V_u	A_a	V_u^{inv} (fast)	EM	**new**
Prime + Probe Invalidation	A_d	V_u	A_d^{inv} (fast)	EM	**new**
	A_a	V_u	A_a^{inv} (fast)	EM	**new**
Bernstein's Invalidation Attack	V_u	V_a	V_u^{inv} (fast)	IM	**new**
	V_u	V_d	V_u^{inv} (fast)	IM	**new**
	V_d	V_u	V_d^{inv} (fast)	IM	**new**
	V_a	V_u	V_a^{inv} (fast)	IM	**new**
Evict + Probe Invalidation	V_d	V_u	A_d^{inv} (fast)	EM	**new**
	V_a	V_u	A_a^{inv} (fast)	EM	**new**
Prime + Time Invalidation	A_d	V_u	V_d^{inv} (fast)	IM	**new**
	A_a	V_u	V_a^{inv} (fast)	IM	**new**
Flush + Time Invalidation	V_u	A_a^{inv}	V_u^{inv} (fast)	EM	**new**
	V_u	V_a^{inv}	V_u^{inv} (fast)	IM	**new**

invalidation-related operation. The *Attack Strategy* column gives a common name for each set of one or more specific vulnerabilities that would be exploited in an attack in a similar manner. The *Vulnerability Type* column gives the three steps that define each vulnerability. The *Macro Type* column proposes the categorization the vulnerability belongs to. "E" is for external interference vulnerabilities. "I" is for internal interference vulnerabilities. "M" is for miss-based vulnerabilities. "H" is for hit-based vulnerabilities.

To ease the understanding of all the vulnerability types, we group the vulnerabilities based on attack strategies (left most column in Table 3.2 and Table 3.3), these strategies correspond to well-known names for the attacks, if such exist, otherwise we provide a new name. In Section 3.1.4 we provide description for each attack strategy to show the main ideas behind them.

The list of vulnerability types can be further collected into four simple macro types which cover one or more vulnerability types: internal interference miss-based (IM), internal interference hit-based (IH), external interference miss-based (EM), external interference hit-based (EH), as labeled in the *Macro Type* column of Table 3.2 and Table 3.3. All the types of vulnerabilities that only involve the victim's behavior, V, in the states in *Step* 2 and *Step* 3 are called internal interference vulnerabilities (I). The remaining ones are called external interference (E). Some vulnerabilities allow the attacker to learn that the address of the victim accesses map to the set the attacker is attacking by observing *slow* timing due to a cache miss or $fast$ timing due to invalidation of data not in the cache[1]. We call these miss-based vulnerabilities (M). The remaining ones leverage observation of $fast$ timing due to a cache hit or *slow* timing due to an invalidation of an address that is currently valid in the cache, and are called hit-based vulnerabilities (H).

Many vulnerability types have been explored before. E.g., the Cache Collision attack [23] is effectively based on the Internal Collision. But we found many new ones as well. The types labeled **new** correspond to new attack not previously discussed in literature. We believe these 43 are new attacks not previously analyzed nor known.

3.1.4 Description of Attack Strategies

This subsection gives overview of the attack strategies from Section 3.1. For each attack strategy, an overview of the three steps of the strategy is given. One advantage of the three-step model is that it gives precise definition of each attack. The attack strategy names used before (and added by us for strategies which did not have such names) may be useful to recall the attacks' high-level operation.

Cache Internal Collision: In *Step* 1, cache block's data is invalidated by flushing or eviction done by either the attacker or the victim. Then, the victim accesses secret data in *Step* 2. Finally, the victim accesses data at a known address in *Step* 3, if there is a cache hit, then it reveals that there is an internal collision and leaks value of u.

Flush + Reload: In *Step* 1, either the attacker or the victim invalidates the cache

1. Invalidation is fast when the corresponding address, which is to be invalidated, does not exist in the cache since no operation is needed for the invalidation in this case.

block's data by flushing or eviction. Then, the victim access secret data in *Step* 2. Finally, the attacker tries to access some data in *Step* 2 using a known address. If a cache hit is observed, then addresses from last two steps are the same, and the attacker learns the secret address. This strategy has similar *Step* 1 and *Step* 2 as **Cache Internal Collision** vulnerability, but for *Step* 3, it is the attacker who does the reload access.

Reload + Time (new name assigned in this work): In *Step* 1, secret data is invalidated by the victim. Then, the attacker does some known data access in *Step* 2 that could possibly bring back the invalidated the victim's secret data in *Step* 1. In *Step* 3, if the victim reloads the secret data, a cache hit is observed and the attacker can derive the secret data's address.

Flush + Probe (new name assigned in this work): In *Step* 1 the victim or the attacker access some known address. In *Step* 2, the victim invalidates secret data. In *Step* 3, reloading of *Step* 1's data and observation of a cache miss will help the attacker learn that the secret data maps to the known address from *Step* 1.

Evict + Time: In *Step* 1, some victim's secret data is put into the cache by the victim itself. In *Step* 2, the attacker evicts a specific cache set by performing a memory related operation that is not a flush. In *Step* 3, the victim reloads secret data, and if a cache miss is observed, the will learn the secret data's cache set information. This attack has similar *Step* 1 and *Step* 3 as **Flush + Time** vulnerability, but for *Step* 2, in **Evict + Time**, the attacker invalidates some known address allowing it to find the full address of the secret data, instead of evicting a cache set to only find the secret data's cache index as in the **Flush + Time** attack.

Prime + Probe: In *Step* 1, the attacker primes the cache set using data at address known to the attacker. In *Step* 2, the victim accesses the secret data, which possibly evicts data from *Step* 1. In *Step* 3, the attacker probes each cache set and if a cache miss is observed, the attacker knowns the secret data maps to the cache set he or she primed.

Bernstein's Attack: This attack strategy leverages the victim's internal interference to trigger the miss-based attack. For one case, the victim does the same secret data access in *Step* 1 and *Step* 3 while in *Step* 2, the victim tries to evict one whole cache set's data by known data accesses. If cache miss is observed in *Step* 3, that will tell the attacker the

cache set is the one secret data maps to. For another case, the victim primes and probe a cache set in *Step* 1 and *Step* 3 driven by the attacker while in *Step* 2, the victim tries to access the secret data. Similar to the first case, observing cache miss in *Step* 3 tells the attacker the cache set is the one secret data maps to.

Evict + Probe (new name assigned in this work): In *Step* 1, Victim evict the cache set using the access to a data at an address known to the attacker. In *Step* 2, the victim accesses secret data, which possibly evicts data from *Step* 1. In *Step* 3, the attacker probes each cache set using the same data as in *Step* 1, if a cache miss is observed the attacker knowns the secret data maps to the cache set he or she primed. This attack strategy has similar *Step* 2 and *Step* 3 as **Prime + Probe** attack, but for *Step* 1, it is the victim that does the eviction accesses.

Prime + Time (new name assigned in this work): In *Step* 1, the attacker primes the cache set using access to data at an address known to the attacker. In *Step* 2, the victim accesses secret data, which possibly evicts data from *Step* 1. In *Step* 3, the victim probes each cache set using the same data *Step* 1, if a cache miss is observed the attacker knowns the secret data maps to the cache set he or she primed in *Step* 1. This attack strategy has similar *Step* 1 and *Step* 2 as **Prime + Probe** attack, but for *Step* 3, it is the victim that does the probing accesses.

Flush + Time (new name assigned in this work): The victim accesses the same secret data in *Step* 1 and *Step* 3; while in *Step* 2, the attacker tries to invalidate data at a known address. If cache miss is observed in *Step* 3, that will tell the attacker the data address he or she invalidated in *Step* 2 maps to the secret data.

Invalidation related **(new names assigned in this work):** Vulnerabilities that have names ending with "invalidation" in Table 3.3 correspond to the vulnerabilities that have the same name (except for the "invalidation" part) in Table 3.2. The difference between each set of corresponding vulnerabilities is that the vulnerabilities ending with "invalidation" use invalidation related operation in the last step to derive the timing information, rather than the normal memory access related operations.

3.1.5 Soundness Analysis of the Three-Step Model

In this section we analyze the soundness of the three-step model to demonstrate that the three-step model can cover all possible timing cache vulnerabilities in normal caches. If there is a vulnerability that is represented using more than three steps, we show the steps can be reduced to only three steps, or a three-step sub-pattern can be found in the longer representation of the vulnerability.

In the below analysis, we use β to denote the number of memory related operations, i.e., steps, in a representation of a vulnerability. We show that $\beta = 1$ is not sufficient to represent a vulnerability, $\beta = 2$ covers some vulnerabilities but not all, $\beta = 3$ represents all the vulnerabilities, and $\beta > 3$ can be reduced to only three steps, or a three-step sub-pattern can be found in the longer representation. Known addresses refer to all the cache states that interference with the data a, a^{alias} and d Unknown address refers u. An access to a known memory address is denoted as *known_access_operation*, and an invalidation of a known memory address is denoted as *known_inv_operation*. The *known_access_operation* and *known_inv_operation* together make up *not_u_operation*s. An unknown memory related operation (containing u) is denoted as *u_operation*.

Patterns with $\beta = 1$

When $\beta = 1$, there is only one memory related operation, and it is not possible to create interference between memory related operations since two memory related operations are the minimum requirement for an interference. Furthermore, $\beta = 1$ corresponds to the three-step pattern with both *Step* 1 and *Step* 2 being $\star$, since the cache state $\star$ gives no information, and *Step* 3 being the one operation. These types of patterns are all examined by the cache three-step simulator and none of these types are found to be effective. Consequently, a vulnerability cannot exit when $\beta = 1$.

Patterns with $\beta = 2$

When $\beta = 2$, it satisfies the minimum requirement of an interference for memory related operations and corresponds to the three-step cases where *Step* 1 is $\star$, and *Step* 2 and *Step* 3

are the two operations. These types are all examined by the cache three-step simulator and some of them belong to *Weak Vulnerabilities*, like $\{ \star \rightsquigarrow A_a \rightsquigarrow V_u\}$. Therefore, three-step cases where *Step* 1 is $\star$ have corresponding effective vulnerabilities shown in Table 3.2. Consequently, $\beta = 2$ can represent some weak vulnerabilities, but not all vulnerabilities as there exist some that are represented with three steps, as discussed next.

Patterns with $\beta = 3$

When $\beta = 3$, we have tested all possible combinations of three-step memory related operations in Section 4.3.1 using the cache simulator for the three-step model. We found that there are in total 72 types of *Strong Vulnerabilities* and 64 types of *Weak Vulnerabilities* that are represented by patterns with $\beta = 3$ steps. Consequently, $\beta = 3$ can represent all the vulnerabilities (including some weak ones where *Step* 1 is $\star$). Using more steps to represent vulnerabilities is not necessary, as discussed next.

Patterns with $\beta > 3$

When $\beta > 3$, the pattern of memory related operations for a vulnerability can be reduced using the following rules. First a set of subdivision rules is used to divide the long pattern into shorter patterns, following the below rules. Each subdivision rule should be applied recursively before applying the next rule.

Subdivision Rule 1: If the longer pattern contains a sub-pattern such as $\{ ... \rightsquigarrow \star \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where $\star$ is assigned as *Step* 1 of the second pattern. This is because $\star$ gives no timing information, and the attacker loses track of the cache state after $\star$. This rule should be recursively applied until there are no sub-patterns left with a $\star$ in the middle or as last step ($\star$ in the last step will be deleted) in the longer pattern.

Subdivision Rule 2: Next, if a pattern (derived after recursive application of the *Rule 1* contains a sub-pattern such as $\{ ... \rightsquigarrow A_{inv}/V_{inv} \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where A_{inv}/V_{inv} is assigned as *Step* 1 of the second pattern. This is because A_{inv}/V_{inv} can be used as the flushing step for *Step* 1, e.g., vulnerability $\{ A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a(fast)\}$ shown in Table 3.2. A_{inv}/V_{inv} cannot be a candidate for middle

steps or the last step because it will remove all timing information, making the attacker unable to deduce the final timing. This rule should be recursively applied until there are no sub-patterns left with a A_{inv}/V_{inv} in the middle or the last step (A_{inv}/V_{inv} in the last step will be deleted).

Next, for each of the patterns resulting from the subdivision of the original pattern, we define *Commute Rule*s, *Union Rule*s and *Reduction Rule*s for a each set of two adjacent steps in these remaining patterns. In Table 3.4, we show the rules for combining adjacent steps, regardless of the attacker's access (A) or the victim's access (V). The table shows whether the corresponding two steps can be commuted, reduced or unioned (and the reduced or the unioned result if the rules can be applied).

Commute Rules: Suppose there are two adjacent steps M and N for a memory sequences $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$. If commuting M and N lead to the same observation result, i.e., $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$ and $\{... \rightsquigarrow N \rightsquigarrow M \rightsquigarrow ...\}$ will have the same timing observation information in the final step for the attacker, we can freely exchange the place of M and N in this pattern. In this case, we have more chance to *Reduce* and *Union* the steps within the memory sequence by the following rules. In the possible commuting process, we will try every possible combinations to commute different pairs of two steps that are able to apply the *Commute Rule*s and then further apply *Reduce Rule*s and *Union Rule*s to see whether the commute is effective, i.e., there can be steps reduced or unioned after the proper commuting process. The following two adjacent memory related operations can be commuted:

- *Commute Rule 1*: For two adjacent steps, if one step is a *known_access_operation* and another step is a *known_inv_operation*. and the addresses they refer to are different, these two steps can be commuted no matter which position of the two steps they are in within the whole memory sequence. It will show a "yes" for the corresponding two-step pattern for the *Commute Rule 1* column if these two can be commuted in Table 3.4.
- *Commute Rule 2*: A superset of two-step patterns that can apply *Commute Rule 1* can be commuted if the second step of these two adjacent steps is not the last step in the whole memory sequence. There are some two adjacent steps that can only be

Table 3.4: Rules for combining two adjacent steps.

First Step	Second Step	*Commute Rule 1*	*Commute Rule 2*	*Union Rule* or *Reduce Rule*	Combined Step	First Step	Second Step	*Commute Rule 1*	*Commute Rule 2*	*Union Rule* or *Reduce Rule*	Combined Step
a	a	yes	yes	yes	a	a^{inv}	a	no	no	yes	a
a	a^{alias}	no	no	yes	a^{alias}	a^{inv}	a^{alias}	yes	yes	yes	a^{alias}
a	d	no	no	yes	d	a^{inv}	d	yes	yes	yes	d
a	u	no	no	no	–	a^{inv}	u	no	no	no	–
a	a^{inv}	no	no	yes	a^{inv}	a^{inv}	a^{inv}	yes	yes	yes	a^{inv}
a	$a^{alias^{inv}}$	yes	yes	yes	a	a^{inv}	$a^{alias^{inv}}$	yes	yes	yes	$Union(a^{inv}, a^{alias^{inv}})$
a	d^{inv}	yes	yes	yes	a	a^{inv}	d^{inv}	yes	yes	yes	$Union(a^{inv}, d^{inv})$
a	u^{inv}	no	no	no	–	a^{inv}	u^{inv}	no	yes	no	–
a^{alias}	a	no	no	yes	a	$a^{alias^{inv}}$	a	yes	yes	yes	a
a^{alias}	a^{alias}	yes	yes	yes	a^{alias}	$a^{alias^{inv}}$	a^{alias}	no	no	yes	a^{alias}
a^{alias}	d	no	no	yes	d	$a^{alias^{inv}}$	d	yes	yes	yes	d
a^{alias}	u	no	no	no	–	$a^{alias^{inv}}$	u	no	no	no	–
a^{alias}	a^{inv}	yes	yes	yes	a^{alias}	$a^{alias^{inv}}$	a^{inv}	yes	yes	yes	$Union(a^{inv}, a^{alias^{inv}})$
a^{alias}	$a^{alias^{inv}}$	no	no	yes	$a^{alias^{inv}}$	$a^{alias^{inv}}$	$a^{alias^{inv}}$	yes	yes	yes	$a^{alias^{inv}}$
a^{alias}	d^{inv}	yes	yes	yes	a^{alias}	$a^{alias^{inv}}$	d^{inv}	yes	yes	yes	$Union(d^{inv}, a^{alias^{inv}})$
a^{alias}	u^{inv}	no	no	no	–	$a^{alias^{inv}}$	u^{inv}	no	yes	no	–
d	a	no	no	yes	a	d^{inv}	a	yes	yes	yes	a
d	a^{alias}	no	no	yes	a^{alias}	d^{inv}	a^{alias}	yes	yes	yes	a^{alias}
d	d	yes	yes	yes	d	d^{inv}	d	no	no	yes	d
d	u	no	no	no	–	d^{inv}	u	no	yes	no	–
d	a^{inv}	yes	yes	yes	d	d^{inv}	a^{inv}	yes	yes	yes	$Union(a^{inv}, d^{inv})$
d	$a^{alias^{inv}}$	yes	yes	yes	d	d^{inv}	$a^{alias^{inv}}$	yes	yes	yes	$Union(d^{inv}, a^{alias^{inv}})$
d	d^{inv}	no	no	yes	d^{inv}	d^{inv}	d^{inv}	yes	yes	yes	d^{inv}
d	u^{inv}	no	yes	no	–	d^{inv}	u^{inv}	no	yes	no	–
u	a	no	no	no	–	u^{inv}	a	no	no	no	–
u	a^{alias}	no	no	no	–	u^{inv}	a^{alias}	no	no	no	–
u	d	no	no	no	–	u^{inv}	d	no	yes	no	–
u	u	yes	yes	yes	u	u^{inv}	u	no	no	yes	u
u	a^{inv}	no	no	no	–	u^{inv}	a^{inv}	no	yes	no	–
u	$a^{alias^{inv}}$	no	no	no	–	u^{inv}	$a^{alias^{inv}}$	no	yes	no	–
u	d^{inv}	no	yes	no	–	u^{inv}	d^{inv}	no	yes	no	–
u	u^{inv}	no	no	yes	u^{inv}	u^{inv}	u^{inv}	yes	yes	yes	u^{inv}

commuted if the second step of these two adjacent steps is not the last step in the whole memory sequence. There will be a "yes" for the corresponding two-step pattern for the *Commute Rule 2* column and a "no" for the corresponding two-step pattern for the *Commute Rule 1* column in Table 3.4.

Reduction Rules: If the memory sequence after applying *Commute Rule*s have a sub-pattern that has two adjacent steps both related to known addresses or both related to unknown address (including repeating states), the two adjacent steps can be reduced to only one following the reduction rules (if the two-step pattern has "yes" for the Column "*Union Rule or Reduce Rule*" and has no *Union* result for the "Combined Step" column in Table 3.4).

- *Reduction Rule 1*: For two *u_operation*s, although u is unknown, both of the accesses target on the same u so can be reduced to only keep the second access in the sequence of steps.
- *Reduction Rule 2*: For two known adjacent memory access related operations (*known_access_operation*), they always result in a deterministic state of the second memory access related cache block, so these two steps can be reduced to only one step.
- *Reduction Rule 3*: For two adjacent steps, if one step is *known_access_operation* and another one is *known_inv_operation*, no matter what order they have, and the address they refer to is the same, these two can be reduced to one step, which is the second step.

Union Rules: Suppose there are two adjacent steps M and N for a memory sequences $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$. If combing M and N leads to the same timing observation result, i.e., $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$ and $\{... \rightsquigarrow Union(M, N) \rightsquigarrow ...\}$ will have the same timing observation information in the final step for the attacker, we can combine step M and N to be a joint one step for this memory sequence, defined as $Union(M, N)$. Two adjacent steps that can be combined are discussed in the following cases:

- *Union Rule 1*: Two invalidations to two known different memory addresses can be applied *Union Rule 1*. *known_inv_operation* are two operations both invalidating

some known address, therefore, they can be combined to only one step. The *Union Rule* can be continuously done to union all the adjacent invalidation step that invalidates known different memory addresses.

Finally, each long memory sequence will recursively apply these three categorizations of the rules in the order: *Commute Rule*s first to put *known_access_operation*s and *known_inv_operation* that targets the same address as near as possible, as well as *u_operation*s and *not_u_operation*s. The *Reduced Rule*s are then checked and applied to the processed sequence to reduce the steps. Then the *Union Rule* is applied to the processed sequence of steps.

The recursion at each application to these rules should be always applied and reduce at least one step until the resulting sequence matches one of the two possible cases:

- the long ($\beta > 3$) memory sequence with *u_operation* and *not_u_operation* is further reduced to a sequence where there are at most three steps in the following patterns:
  - $u\_operation \rightsquigarrow not\_u\_operation \rightsquigarrow u\_operation$
  - $not\_u\_operation \rightsquigarrow u\_operation \rightsquigarrow not\_u\_operation$

  There might be possible extra $\star$ or A^{inv}/V^{inv} before these three-step pattern, where:
  - An extra $\star$ in the first step will not influence the result and can be directly removed.
  - If an extra A^{inv}/V^{inv} in the first step:
    * If followed by *known_access_operation*, A^{inv}/V^{inv} can be removed due to the actual state further put into the cache block.
    * If followed by *known_inv_operation* or V_u^{inv}, A^{inv}/V^{inv} can also be removed since the memory location is repeatedly flushed by the two steps.
    * If followed by V_u, worst case will be $A^{inv}/V^{inv} \rightsquigarrow V_u \rightsquigarrow not\_u\_operation \rightsquigarrow u\_operation$, which is either an effective vulnerability or $A^{inv}/V^{inv} \rightsquigarrow V_u \rightsquigarrow A_d^{inv}/V_d^{inv} \rightsquigarrow u\_operation$, where $V_u \rightsquigarrow A_d^{inv}/V_d^{inv}$ can further apply *Commute Rule 2* to reduce and be within three steps.

In this case, the steps are finally within three steps and the checking is done.

- There exists two adjacent steps that cannot be affected by any rules anymore and require additional checks listed below.

The only remaining two adjacent steps that cannot be applied by any of the three categorizations of the rules are the following:

- $\{... \rightsquigarrow A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d/A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv} \rightsquigarrow V_u \rightsquigarrow ...\}$
- $\{... \rightsquigarrow A_a/V_a/A_{a^{alias}}/V_{a^{alias}} \rightsquigarrow V_u^{inv} \rightsquigarrow ...\}$
- $\{... \rightsquigarrow V_u \rightsquigarrow\rightsquigarrow ...A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d/A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv}\}$
- $\{... \rightsquigarrow V_u^{inv} \rightsquigarrow A_a/V_a/A_{a^{alias}}/V_{a^{alias}} \rightsquigarrow ...\}$

We manually checked all of the two adjacent step patterns above and found that adding extra step before or after these two steps can either generate two adjacent step patterns that be processed by the three rules, where further step can be reduced, or construct effective vulnerability, where the corresponding pattern can be treated as effective.

Algorithm for Reducing and Checking Memory Sequence

The Algorithm 2 is used to: i) reduce a β-step ($\beta > 3$) pattern to a three-step pattern, thus demonstrating that the corresponding $\beta > 3$ step pattern actually is equivalent to the output three-step pattern and represents a vulnerability that is captured by an existing three-step pattern, or ii) demonstrate that the β-step pattern can be mapped to one or more three-step vulnerabilities. It is not possible for a β-step vulnerability pattern to not be either i) or ii) after doing the *Rule* applications Key outcome of the analysis is that any β-step pattern is not a vulnerability, or if it is a vulnerability it maps to either outputs i) or ii) of the algorithm.

Inside the Algorithm 2, contain() represents a function to check if a list contains a corresponding state, is_ineffective() represents a function that checks the corresponding memory sequence does not contain any effective three-steps. has_interval_effective_ three_steps() represents a function that check if the corresponding memory sequence can be mapped to one or more three-step vulnerabilities.

Algorithm 2 β-Step ($\beta > 3$) Pattern Reduction

Input: β: number of steps of the pattern
step_list: a two-dimensional dynamic-size array. *step_list*[0] contains the states of each step of the original pattern in order. *step_list*[1], *step_list*[2], ... are empty initially.
Output: *reduce_list*: reduced effective vulnerability pattern(s) array. It will be an empty list if the original pattern does not correspond to an effective vulnerability.

reduce_list = []
while *step_list*.contain($\star$) **and** $\star$.index **not** 0 **do**
    step_list = *Subdivision_Rule_*1 (*step_list*)
end while
while (*step_list*.contain(A_{inv}) **and** A_{inv}.index **not** 0) **or** (*step_list*.contain(V_{inv}) **and** V_{inv}.index **not** 0) **do**
    step_list = *Subdivision_Rule_*2 (*step_list*)
end while
while !(*step_list.set_list*.is_ineffecitve **or** *step_list.set_list*.has_interval_effective_ three_steps) **do**
    step_list = *Commute_Rules* (*step_list*)
    step_list = *Reduction_Rules* (*step_list*)
    step_list = *Union_Rule* (*step_list*)
    if !(*step_list.set_list*.is_ineffecitve **or** *step_list.set_list*.has_interval_effective_ three_steps) **then**
        reduce_list += *Rest_Checking* (*step_list*)
    end if
end while
return *reduce_list*

3.1.6 Cache Three-Step Model Summary

In conclusion, the three-step model can model all possible timing cache vulnerability in normal caches. Vulnerabilities which are represented by more than three steps can be always reduced to one (or more) vulnerabilities from the three-step model; and thus, using more than three step is not necessary.

3.2 Secure Caches Evaluation

To address the threat of the cache timing–based attacks, different secure cache designs have been previously presented in academic literature. The secure processor caches are designed with different assumptions and often address only specific types of timing side-channel or covert-channel attacks. To help analyze the security of these designs, this work uses the three-step modeling approach to reason about all the possible timing vulnerabilities. Especially, since the work demonstrates a number of new timing attacks, the existing secure caches have never been analyzed with respect to these new attacks before.

3.2.1 Different Types of Secure Caches

Various secure caches which have been presented in literature to date [38, 39, 1, 40, 2, 41, 42, 43, 45, 44, 57, 58, 59, 60, 61, 62, 63, 64]. Later, in Section 3.2.2 we will apply the three-step model to check if the secure caches can defend some or all of the vulnerabilities in the model.

This section gives brief overview of the 18 secure cache designs that have been presented in academic literature in the last 15 years. To the best of our knowledge, these cover all the secure cache designs proposed to date. Most of the designs have been realized in functional simulation, e.g., [42, 58]. Some have been realized in FPGA, e.g., [62], and a few have been realized in real ASIC hardware, e.g., [65]. No specific secure caches have been implemented in commercial processors to the best of our knowledge, however, CATalyst [57] leverages Intel's CAT (Cache Allocation Technology) technology available today in Intel Xeon E5 2618L v3 processors, and could be deployed today.

When the secure cache description in the cited papers did not mention the issue of using flush or cache coherence, we assume the victim or the attacker cannot invalidate each other's cache blocks by using $clflush$ instructions or through cache coherence protocol operations; but they can flush or use cache coherence to invalidate their own cache lines. The victim and the attacker also cannot invalidate protected or locked data. Further, if the authors specified any specific assumptions (mainly about the software), we list the assumption as part of the description of the cache. What's more, when the level of cache hierarchy was unspecified, we assume the secure caches' features can be applied to all levels of caches, including L1 cache, L2 cache and Last Level Cache (LLC). If the inclusivity of the caches was not specified, we assume they target inclusive caches. Following the below descriptions of each secure cache design, the analysis of the secure caches is given in Section 3.2.2.

SP* cache [43, 66][2] uses partitioning techniques to statically partition the cache ways into *High* and *Low* partition for the victim and the attacker according to their different process IDs. The victim typically belongs to *High* security and attacker belongs to *Low* security. Victim's memory accesses cannot modify *Low* partition (assigned to processes

2. Two existing papers give slightly different definitions for an "SP" cache, thus we selected to define a new cache, the SP* cache, that combines secure cache features of the Secret-Protecting cache from [43] with secure cache features of the Static-Partitioned cache from [66].

such as the attacker), while the attacker's memory accesses cannot modify *High* partition (assigned to the victim). However, the memory accesses of both the victim and the attacker can result in a hit in either *Low* or *High* partition if the data is in the cache.

SecVerilog cache [39, 38] statically partitions cache blocks between security levels L (*Low*) and H (*High*). Each instruction in the source code for programs using SecVerilog cache needs to include a *timing label* which effectively represents whether the data being accessed by that instruction is *Low* or *High* based on the code and this *timing label* can be similar to a process ID that differentiates attacker's (*Low*) instructions from victim's (*High*) instructions. The cache is designed such that operations in the *High* partition cannot affect timing of operations in the *Low* partition. For a cache miss due to *Low* instructions, when the data is in the *High* partition, it will behave as a cache miss, and the data will be moved from the *High* to the *Low* partition to preserve consistency. However, *High* instructions are able to result in a cache hit in both *High* and *Low* partitions, if the data is in the cache.

SecDCP cache [42] builds on the SecVerilog cache and uses partitioning idea from the original SecVerilog cache, but the partitioning is dynamic. It can support at least two security classes H (*High*) and L (*Low*), and configurations with more security classes are possible. They use the percentage of cache misses for L instructions that was reduced (increased) when L's partition size was increased (reduced) by one cache way to adjust the number of ways of the cache assigned to the *Low* partition. When adjusting number of ways in the cache dedicated to each partition, if L's partition size decreases, the process ID is checked and L blocks are flushed before the way is reallocated to H. On the other hand, if L's partition size increases, H blocks in the adjusted cache way remain unmodified so as to not add more performance overhead, and they will eventually be evicted by L's memory accesses. However, the feature of not flushing *High* partition data during way adjustment may leak timing information to the attacker.

NoMo cache [44] dynamically partitions the cache ways among the currently "active" simultaneous multithreading (SMT) threads. Each thread is exclusively reserved Y blocks in each cache set, where Y is within the range of $[0, \lfloor \frac{N}{M} \rfloor]$, where N is the number of ways and M is the number of SMT threads. NoMo-0 equals to traditional set associative cache while NoMo-$\lfloor \frac{N}{M} \rfloor$ partitions cache evenly for the different threads and there are no non-reserved

ways. The number of Y assigned to each thread is adjusted based on its activeness. When adjusting number of blocks assigned to a thread, Y blocks are invalidated for cache sets to protect timing leakage. Eviction is not allowed within each thread's own reserved ways while it is possible for the shared ways. Therefore, to avoid eviction caused by the unreserved ways, we assume NoMo-$\lfloor \frac{N}{M} \rfloor$ is used to fully partition the cache. When the attacker and the victim share the same library, there will be a cache hit if accessing the shared data, and the normal cache hit policy holds to guarantee the cache coherence.

SHARP cache [45] uses both partitioning and randomization techniques to prevent victim's data from being evicted or flushed by other malicious processes and it targets on the inclusive caches. Each cache block is augmented with the core valid bits (CVB) to indicate which private cache (process) it belongs to (similar to the Process ID), where CVB stores a bitmap and i-th bit in the bitmap is set if the line is present in i-th core's private cache. Cache hit is allowed among different processes' data. When there is cache miss and data needs to be evicted, data not belonging to any current processes will be evicted first. If there is no such data, data belonging to the same process will be evicted. If there is no existing data in the cache that is in the same process, a random data in the cache set will be evicted. This random eviction will generate an interrupt to the OS to notify it of a suspicious activity. For pages that are read-only or executable, SHARP cache disallows flushing using $clflush$ in user mode. However, invalidating victim's blocks by using cache coherence protocol is still possible.

Sanctum cache [41] focuses on isolation of enclaves (equivalent to Trusted Software Module in other designs) from each other and the operating system (OS). In terms of caches, they implements security features for L1 cache, TLB and LLC. Cache isolation of LLC is achieved by assigning each enclave or OS to different DRAM address regions. It uses page-coloring-based cache partitioning scheme [67, 68] and a software security monitor that ensures per-core isolation between OS and enclaves. For L1 cache and TLB, when there is a transition between enclave and non-enclave mode, the security monitor will flush the core-private caches to achieve isolation. Normal flushes triggered by the enclave or the OS can only be done within enclave or not within enclave code. Also, timing side-channel attacks exploiting cache coherence are explicitly not prevented, thus behavior on cache coherence

operations is not defined. This cache listed extra software assumptions as follows:

Assumption 1. Software security monitor guarantees that victim and attacker process cannot share the same cache blocks. It uses page coloring [67, 68] to ensure that victim and attacker's memory is never mapped to the same cache blocks for the LLC.

Assumption 2. The software runs on a system with a single processor core where victim and attacker alternate execution, but can never run truly in parallel. Moreover, security critical data is always flushed by the security monitor when program execution switches away from the victim program for the L1 cache and the TLB.

MI6 cache [62] is part of the memory hierarchy of the MI6 processor, which combines Sanctum [41] cache's security feature with disabling speculation during the speculative execution of memory-related operations. During normal processor execution, for L1 caches and TLB, the corresponding states will be flushed across context switches between software threads. For the LLC, set partitioning is used to divide DRAM into contiguous regions. And cache sets are guaranteed to be strictly partitioned (two DRAM regions cannot map to the same cache set). Each enclave is only able to access its own partition. Speculation is simply disabled when enclave interacts with the outside world because of small performance influence based on the rare cases of speculation. This cache listed extra software assumptions as follows:

Assumption 1. Software security monitor guarantees that the victim and the attacker process cannot share the same cache blocks. It uses page coloring [67, 68] to ensure that victim's and attacker's memory are never mapped to the same cache blocks for the LLC.

Assumption 2. The software runs on a system with a single processor core where victim and attacker alternate execution, but can never run truly in parallel. Moreover, security critical data is always flushed by the security monitor when program execution switches away from the victim program for the L1 cache and the TLB.

Assumption 3. When an enclave is interacting with the outside environment, the corresponding speculation is disabled by the software.

InvisiSpec cache [61] is able to make speculation invisible in the data cache hierarchy. Before a *visibility point* shows up, when all of its prior control flow instructions resolve, unsafe speculative loads (USL) will be put into a speculative buffer (SB) without modifying

any cache states. When reaching the visibility point, there are two cases. In one case, the USL and successive instructions will be possibly squashed because of mismatch of data in the SB and the up-to-date values in the cache. In another case, the core receives possible invalidation from the OS before checking of memory consistency model and no comparison is needed. When speculative execution happens, the hardware puts the data into SB, as to identify visibility point for dealing with final state transition of the speculative execution. InvisiSpec cache targets on Spectre-like attacks and futuristic attacks. However, InvisiSpec cache is vulnerable to all non-speculative side channels.

CATalyst cache [57] uses partitioning, especially Cache Allocation Technology (CAT) [69] available in the LLC of some Intel processors. CAT allocates up to 4 different Classes of Services (CoS) for separate cache ways so that replacement of cache blocks is only allowed within a certain CoS. CATalyst first uses CAT mechanism to partition caches into secure and non-secure parts (non-secure parts may map to 3 CoS while secure parts map to 1 CoS). Secure pages are assigned to virtual machines (VMs) at a granularity of a page, and not shared by more than one VM. Here, attacker and victim reside in different VMs. Combined with CAT technology and pseudo-locking mechanism which pins certain page frames managed by software, CATalyst guarantees that malicious code cannot evict secure pages. CATalyst implicitly performs preloading by remapping security-critical code or data to secure pages. Flushes can only be done within each VM. And cache coherence is achieved by assigning secure pages to only one processor and not sharing pages among VMs. This cache listed extra software assumptions as follows:

Assumption 1. Security critical data is always preloaded into the cache at the beginning of the whole program execution.

Assumption 2. Security critical data is always able to fit within the secure partition of the cache. I.e. all data in the range x can fit in the secure partition.

Assumption 3. The victim and the attacker process cannot share the same memory space between each other.

Assumption 4. Use pseudo-locking mechanism by software to make sure that victim and attacker process cannot share the same cache blocks.

Assumption 5. Secure pages are reloaded immediately after the flush, which is done by

the virtual machine monitor (VMM) to make sure all the secure pages are still pinned in the secure partition.

DAWG cache [60] (Dynamically Allocated Way Guard) partitions the cache by cache ways and provides full isolation for hits, misses and metadata updates across different protection domains (between the attacker and the victim). DAWG cache is partitioned for the attacker and the victim and each of them keep their own different *domain_id* (which is similar to process ID used in general caches). Each *domain_id* has its own bit fields, one is called *policy_fillmap*, for masking fills and selecting the victim to replace, another is called *policy_hitmap*, for masking hit ways. Only both the tag and the *domain_id* are the same will a cache hit happen. Therefore, DAWG allows read-only cache lines to be replicated across ways for different protection domain. For a cache miss, the victim can only be chosen within the ways belonging to the same *domain_id*, recorded by the *policy_fillmap*. Consistently, the replacement policy is updated with the victim selection and the metadata derived from the *policy_fillmap* for different domains is updated as well. The work also proposes the idea to dynamically partitions the cache ways following the system's workload changes but does not actually implement it.

RIC cache [59] (Relaxed Inclusion Caches) proposes a low-complexity cache to defend against eviction-based timing side-channel attacks on the LLC. Normally for an inclusive cache, if the data R is in the LLC, it is also in the higher level cache, and eviction of the R in the LLC will cause the same data in the higher level cache, e.g., L1 cache to be invalidated, making eviction-based attacks in the higher level cache possible (e.g., attacker is able to evict victim's security critical cache line). For RIC, each cache line is extended with a single bit to set the relaxed inclusion. Once the relaxed inclusion is set for that cache line, the corresponding LLC line eviction will not cause the same line in the higher-level cache to be invalidated. Two kinds of data will be set relaxed inclusion bit: *read only data* and *thread private data* when they are loaded into the cache. These two kinds of data are claimed by the work to cover all the critical data for ciphers. Therefore, RIC will not prevent writable in-private critical data, which is currently not found in any ciphers. Apart from that, RIC requires flushing the corresponding cache lines in the cases that the RIC bits are modified or for thread migration events to avoid the timing leakage during migration.

PL cache [1] provides isolation by partitioning the cache based on cache blocks. It extends each cache block with a process ID and a lock status bit. The process ID and the lock status bits are controlled by the extended load and store instructions (*ld.lock*/*ld.unlock* and *st.lock*/*st.unlock*) which allow the programmer and compiler to set or reset the lock bit through use of the right load or store instruction. In terms of cache replacement policy, for a cache hit, PL cache will perform the normal cache hit handling procedure and the instructions with locking or unlocking capability can update the process ID and the lock status bits while the hit is processed. When there is a cache miss, locked data cannot be evicted by data that is not locked and locked data among different processes cannot be evicted by each other. In this case, the new data will be either loaded or stored without caching. In other cases, data eviction is possible. This cache listed extra software assumption as follows:

Assumption 1. Security critical data is always preloaded into the cache at the beginning of the whole program execution.

RP cache [1] uses randomization to de-correlate the memory address accessing and timing of the cache. For each block of RP cache, there is a process ID and one protection bit P set to indicate if this cache block needs to be protected or not. A permutation table (PT) stores each cache set's pre-computed permuted set number and the number of tables depends on number of protected processes. For memory access operations, cache hits need both process ID and address to be the same. When a cache miss happens to data D of a cache set S, if the to-be-evicted data and to-be-brought-in data belong to the same process but have different protection bit, arbitrary data of a random cache set S' will be evicted and D will be accessed without caching. If they belong to different processes, D will be stored in an evicted cache block of S' and mapping of S and S' will be swapped as well. Otherwise, the normal replacement policy is executed.

Newcache cache [40, 65] dynamically randomizes memory-to-cache mapping. It introduced a ReMapping Table (RMT), and the mapping between memory addresses and this RMT is as in a direct mapped cache, while the mapping between the RMT and actual cache is fully associative. The index bits of memory address are used to look up entries in the RMT to find the cache block that should be accessed. It stores the most useful cache lines

rather than hold a fixed set of cache lines. This index stored in RMT combined with the process ID is used to look up the actual cache where each cache line is associated with its real index and process ID. Each cache block is also associated with a protection bit (P) to indicate if it is security critical. For cache replacement policy, it is very similar to RP cache. Cache hit needs both process ID and address to be the same. When cache miss happens to data D, arbitrary data will be evicted and D will be accessed without caching if they belong to the same process but either one of their protection bit is set. If the evicted data and brought-in data have different process IDs, D will randomly replace a cache line since it is fully associative in the actual cache. Otherwise, the normal replacement policy for direct mapped cache is executed.

Random Fill cache [2] de-correlates cache fills with the memory access using random filling technique. New instructions used by applications in Random Fill cache can control if the requested data belongs to a normal request or a random fill request. Cache hits are processed as in normal cache. For the security critical data accesses of the victim, a $Nofill\ request$ is executed and the requested data access will be performed without caching. Meanwhile, on a $Random\ Fill\ request$, arbitrary data, from the range of addresses, will be brought into the cache. In the paper [2], the authors show that random fill of spatially near data does not hurt performance. For other processes' memory accesses and normal victim's memory accesses, $Normal\ request$ will be used to achieve normal replacement policy. Victim and attacker are able to remove victim's own security critical data including using $clflush$ instructions or cache coherence protocol since the flush will not influence timing side-channel attack prevention (the random filling technique is used for this).

CEASER cache [63] is able to mitigate conflict-based LLC timing side-channel attacks using address encryption and dynamic remapping. CEASER cache does not differentiate whom the address belongs to and whether the address is security critical. When memory access tries to modify the cache state, the address will first be encrypted using Low-Latency BlockCipher (LLBC) [70], which not only randomizes the cache set it maps, but also scatters the original, possibly ordered and location-intensive addresses to different cache sets, decreasing the probability of conflict misses. The encryption and decryption can be done within two cycles using LLBC. Furthermore, the encryption key will be periodically changed

to avoid key reconstruction. The periodic re-keying will cause the address remapping to dynamically change.

SCATTER cache [64] uses cache set randomization to prevent timing attacks. It builds upon two ideas. First, a mapping function is used to translate memory address and process information to cache set indices, the mapping is different for each program or security domain. Second, the mapping function also calculates a different index for each cache way, in a similar way to the skewed associative caches [71]. The mapping function can be keyed hash or keyed permutation derivation function – a different key is used for different application or security domain resulting in a different mapping from address to cache sets for each. Software (e.g., the operating system) is responsible for managing the security domains and process IDs which are used to differentiate the different software and assign it different keys for the mapping. For the hardware extension, a cryptographic primitive such as hashing and an index decoder for each scattered cache way is added. SCATTER cache also stores the index bits of the physical address to efficiently perform lookups and writebacks. There is also one bit per page-table entry added to allow the kernel to communicate with the user space for security domain identification.

Non Deterministic cache [58] uses cache access delay to randomize the relation between cache block access and cache access timing. There is no differentiation of data caching between different process ID or whether the data is secure or not. A per-cache-block counter records the interval of its data activeness, and is increased on each global counter clock tick when the data is untouched. When the counter reaches a predefined value, the corresponding cache line will be invalidated. Non Deterministic Cache randomly sets the local counters' initial value that is less than the maximum value of the global counter. In this case, the cache delay is changed to be randomized. Cache delay interval controlled by this non-deterministic execution can lead to different cache hit and miss statistics because the invalidation is determined by the randomized counter of each cache line, and therefore de-correlates any cache access time from the address being accessed. However, the performance degradation is tremendous.

3.2.2 Analysis of the Secure Caches

In this section, we manually evaluate the effectiveness of the 18 secure caches [38, 39, 1, 40, 2, 41, 42, 43, 45, 44, 57, 58, 59, 60, 61, 62, 63, 64]. We analyze how well the different caches can protect against the 72 types of vulnerabilities defined before, which cover all the possible *Strong* (according to the definition in Seciton 3.1) cache timing vulnerabilities. Following the analysis, discuss what types of secure caches and features are best suited for defending different types of timing attacks.

Effectiveness of the Secure Caches Against timing Attacks

Table 3.5 and Table 3.6 list the result of the analysis of which caches can prevent which types of attacks. Some caches are able to prevent certain vulnerabilities, denoted by a checkmark, ✓, and green color in the table. For example, SP* cache can defend against $V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow) (one type of Evict + Time [49]) vulnerability. For some other caches and vulnerabilities, the cache is not able to prevent the vulnerabilities and it is indicated by $\times$ and red color. For example, SecDCP cache cannot defend against $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [22]) vulnerability. A cache is judged to be able to prevent a cache timing vulnerability if:

1. A cache can prevent a timing attack if the timing of the last step in a vulnerability is always constant and the attacker can never observe fast and slow timing difference for the given set of three steps. For instance, in a regular set-associative cache, the $V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [27]) vulnerability will allow the attacker to know that address a maps to secret u when the attacker observes fast timing, compared with observing slow timing in the other cases. However, in case of the RP cache [1] will make the timing of the last step to be always slow because RP cache does not allow data of different processes to derive cache hit between each other.
2. A cache can prevent a timing attack if the timing of last step is randomized and cannot have original corresponding relation between victim's behavior and attacker's observation. For instance, $A_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast) (one type of Prime + Probe Invalidation) vulnerability when executed on a normal set-associative cache will allow

the attacker to know that the address d has the same index with secret u when observing fast timing, compared with slow timing in the other cases. However, when executing this attacks on the Random Fill cache [2], for example a slow timing will not determine that u and d have the same index as the secret, since in Random Fill cache u would be accessed without caching and another random data would be cached instead in the cache.

3. A cache can prevent a timing attack if it disallows certain steps from the three-step model to be executed, thus prevents the corresponding vulnerability. For instance, when PL cache [1] preloads and locks the security critical data in the cache, vulnerabilities such as $A_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (slow) (one type of Prime + Time Invalidation) will not be possible since a preloaded locked security critical data will not allow A_d in *Step* 1 to replace it. In this case, A_d cannot be in the cache, so this vulnerability cannot be triggered in PL cache.

From the security perspective, the entries of the secure cache in Table 3.5 and Table 3.6 should have as many green colored cells as possible. If a cache design has any red cells, then it cannot defend against that type of vulnerability – attacker using the timing vulnerability that corresponds to the red cell can attack the system.

The third column in Table 3.5 and Table 3.6 shows a normal set associative cache, which cannot defend against any type of timing vulnerabilities. Meanwhile, the last column of Table 3.5 and Table 3.6 shows the situation where the cache is fully disabled. As is expected, the timing vulnerabilities are eliminated and timing attacks will not succeed. Disabling caches, however, has tremendous performance penalty. Similarly, second-to-last column shows Nondeterministic Cache, which totally randomizes cache access time. It can defend all the attacks, but again will have a tremendous cost.

For each of the entry that shows the effectiveness of a secure cache against a vulnerability, there are two results listed. Left one is for normal execution, and the right one is for speculative execution. Some secure caches such as InvisiSpec cache target timing channels in speculative execution. For most of the caches that do not differentiate speculative execution and normal execution, the two sub-columns for each cache are the same.

Table 3.5: Existing secure caches' protection against all possible timing vulnerabilities where the last step is a memory access-related operation. A ✓ in a green cell means this cache can prevent the corresponding vulnerability. A ○ in a pink cell means this cache can prevent the corresponding vulnerability in some degree. A × in a red cell means this cache cannot prevent this vulnerability. Furthermore, for each cache, we analyze normal execution (left column under the cache name) and speculative execution (right column under the cache name).

Type	Vulnerability	Set Associative Cache		SP* Cache		Sec-Verilog Cache		SecDCP Cache		NoMo Cache		SHARP Cache[6]		Sanctum Cache[2]		MI6 Cache		InvisiSpec Cache[9]		CATalyst Cache[2,5]		DAWG Cache		RIC[8]		PL Cache		RP Cache		Newcache		Random Fill Cache		CEASER Cache		SCATTER Cache		Non Deterministic Cache[13]		Cache Disabled	
Cache Internal Collision	$A_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×	×	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×	×	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×	×	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	×	×	✓	✓	✓	✓	×	×	✓	✓	×	×	×	×	○	○	✓	✓
	$A_a^{alias} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	✓	✓	×	×	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$V_a^{alias} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	×	×	×	×	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
Flush + Reload	$A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10],	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10],	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓[12]	✓[12]	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[12]	✓[12]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10],	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[12]	✓[12]	○	○	✓	✓
	$A_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10],	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓[12]	✓[12]	○	○	✓	✓
Reload + Time	$V_u^{inv} \rightsquigarrow A_a \rightsquigarrow V_u$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_a \rightsquigarrow V_u$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Flush + Probe	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10],	✓[10],	✓[10],	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[11]	✓[11]	✓[11]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Evict + Time	$V_a \rightsquigarrow A_d \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓[4]	✓[4]	✓[10],	✓[10],	✓[10],	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓[4]	✓[4]	✓[10],	✓[10],	✓[10],	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Probe	$A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10],	✓[10],	✓[10],	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10],	✓[10],	✓[10],	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Bernstein's Attack	$V_a \rightsquigarrow V_a \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Evict + Probe	$V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[10]	✓[10],	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[10]	✓[10],	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Time	$A_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Flush + Time	$V_a \rightsquigarrow A_a^{inv} \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10],	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	×[3]	×[3]	○	○	✓	✓

[1] [1] Dynamic adjustment of ways for different threads is assumed to be properly used according to the running program's cache usage. [2] Some software assumptions listed in the entries in this column have been implemented by the cache's related software. [3] Flush is disabled, but cache coherence might be used to do the data removal. [4] For L1 cache and TLB, flushing is done during context switch. [5] The techniques are implemented in L1 cache, TLB and last-level cache which consist of the whole cache hierarchy, where L1 cache and TLB require software flush protection and the last-level cache can be achieved by simple hardware partitioning. To protect all levels of caches, the software assumptions need to be added. [6] The technique is now only implemented in last-level cache. [7] The technique now only targets shared cache. [8] The technique only targets inclusion last-level cache. [9] The technique targets data cache hierarchy. [10] For the last-level cache, cache is partitioned between the victim and the attacker. [11] The technique can control the probabilities of the vulnerability to be successful to be extremely small. [12] The technique can work in shared, read only memory while not working in shared, writable memory. [13] Random delay but not random mapping can only decrease the probabilities of attacker in some limited degree.

Table 3.6: Existing secure caches' protection against all possible timing vulnerabilities where the last step is an invalidation-related operation. A ✓ in a green cell means this cache can prevent the corresponding vulnerability. A ○ in a pink cell means this cache can prevent the corresponding vulnerability in some degree. A × in a red cell means this cache cannot prevent this vulnerability. Furthermore, for each cache, we analyze normal execution (left column under the cache name) and speculative execution (right column under the cache name).

Type	Vulnerability	Set Associative Cache		SP* Cache		Sec-Verilog Cache		SecDCP Cache		NoMo Cache		SHARP Cache[6]		Sanctum Cache[2]		MI6 Cache		InvisiSpec Cache[9]		CATalyst Cache[2,5]		DAWG Cache		RIC[8]		PL Cache		RP Cache		Newcache		Random Fill Cache		CEASER Cache		SCATTER Cache		Non Deterministic Cache[13]		Cache Disabled	
Cache Internal Collision Invalidation	$A_{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$A_a^{alias} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$V_a^{alias} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
Flush + Flush	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Flush + Reload Invalidation	$A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
Reload + Time Invalidation	$V_u^{inv} \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u^{inv} \rightsquigarrow V_a \rightsquigarrow V_u^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Flush + Probe Invalidation	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[11]	✓[11]	✓[11]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Evict + Time Invalidation	$V_a \rightsquigarrow A_d \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Probe Invalidation	$A_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Bernstein's Invalidation Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow V_d \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓[8]	✓[8]	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Evict + Probe Invalidation	$V_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Time Invalidation	$A_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Flush + Time Invalidation	$V_a \rightsquigarrow A_a^{inv} \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	×[3]	×[3]	○	○	✓	✓

[1] [1] Dynamic adjustment of ways for different threads is assumed to be properly used according to the running program's cache usage. [2] Some software assumptions listed in the entries in this column have been implemented by the cache's related software. [3] Flush is disabled, but cache coherence might be used to do the data removal. [4] For L1 cache and TLB, flushing is done during context switch. [5] The techniques are implemented in L1 cache, TLB and last-level cache which consist of the whole cache hierarchy, where L1 cache and TLB require software flush protection and the last-level cache can be achieved by simple hardware partitioning. To protect all levels of caches, the software assumptions need to be added. [6] The technique is now only implemented in last-level cache. [7] The technique now only targets shared cache. [8] The technique only targets inclusion last-level cache. [9] The technique targets data cache hierarchy. [10] For the last-level cache, cache is partitioned between the victim and the attacker. [11] The technique can control the probabilities of the vulnerability to be successful to be extremely small. [12] The technique can work in shared, read only memory while not working in shared, writable memory. [13] Random delay but not random mapping can only decrease the probabilities of attacker in some limited degree.

3.2.3 Summary of Secure Cache Techniques

Among the secure cache designs presented in the prior section, there are three main techniques that the caches utilize: differentiating sensitive data, partitioning, and randomization.

Differentiating sensitive data (columns for CATalyst cache to columns for Random Fill cache in Table 3.5 and Table 3.6) allows the victim or attacker software or management software to explicitly label a certain range of the data of victim which they think is sensitive. The victim process or management software is able to use cache-specific instructions to protect the data and limit internal interference between victim's own data. E.g., it is possible to disable victim's own flushing of victim's labeled data, and therefore prevent vulnerabilities that leverage flushing. This technique allows the designer to have stronger control over security critical data, rather than forcing the system to assume all of victim's data is sensitive. However, how to identify sensitive data and whether this identification process is reliable are open research questions for caches that support differentiation of sensitive data.

This technique is independent of whether a cache uses partitioning or randomization techniques to eliminate side channels between the attacker and the victim. Caches that are able to label and identify sensitive data have the advantage in preventing internal interference since they are able to differentiate sensitive data from the normal data and can make use of special instructions to give more privileges to sensitive data. However, it requires careful use when identifying the actual sensitive data and implementing corresponding security features on the cache.

Comparing PL cache with SP* cache, although both of them use partitioning, flush is able to be implemented to be disabled for victim's sensitive data in PL cache, where $V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow) (one type of Flush + Time) is prevented. Newcache is able to prevent $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [22]) while most of the caches without ability to differentiate sensitive data cannot because Newcache disallows replacing data as long as either data to be evicted or data to be cached is identified to be sensitive. However, permitting differentiation of sensitive data can potentially backfire on the cache itself. For example, Random Fill cache cannot prevent $V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow) (one type of Evict + Time [49]) which most of the other caches can prevent or avoid, because the

random fill technique loses its intended random behavior when the security critical data is initially loaded into the cache in *Step* 1.

Partitioning-based caches usually limit the victim and the attacker to be able to only access a limited set of cache block (columns for SP* cache to column for PL cache in Table 3.5 and Table 3.6). E.g., either there is static or dynamic partitioning of caches which allocates some blocks to *High* victim and *Low* attacker. The partitioning can be based not just on whether the memory access is victim's or attacker's, but also on where the access is to (e.g., *High* partition is determined by the data address) For speculative execution, attacker's code can be the part of speculation or out-of-order load or store, which is able to be partitioned (e.g., using speculative load buffer) from other normal operations. The partitioning granularity can be cache sets, cache lines or cache ways. Partitioning-based secure caches are usually able to prevent external interference by partitioning but are weak at preventing internal interference. When partitioning is used, interference between the attacker and the victim, or data belonging to different security levels, should not be possible and attacks based on external interference between the victim and the attacker will fail. However, the internal interference of victim's own data is hard to be prevented by the partitioning based caches. What's more, partitioning is recognized to be wasteful in terms of cache space and inherently degrades system performance [1]. Dynamic partitioning can help limit the negative performance and space impacts, but it could be at a cost of revealing some information when adjusting the partitioning size for each part. It also does not help with internal interference prevention.

In terms of the three-step model, the partitioning-based caches excel at making use of partitioning techniques to disallow the attacker to set initial states (*Step* 0) of victim partition by use of flushing or eviction, and therefore bring uncertainty to the final timing observation made by the attacker.

SP* cache can prevent external miss-based interference, but it still allows the victim and the attacker to get cache hits due to each other's data, which makes hit-based vulnerabilities happen, e.g., $V_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (one type of Cache Internal Collision [23]) vulnerability is one of the examples that SP* cache cannot prevent. SecVerilog cache is similar to SP* cache but prevents the attacker from directly getting cache hit due to victim's data for

confidentiality and therefore prevents vulnerabilities such as $A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [27]). SHARP cache mainly uses partitioning combined with random eviction to minimize the probability of evicting victim's data and prevent external miss-based vulnerabilities. It is vulnerable to hit-based or internal interference vulnerabilities such as $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [22]) vulnerability. DAWG cache will only allow the data to get a cache hit if both its address and the process ID are the same. Therefore, compared with normal partitioning cache such as SP* cache, it is able to prevent vulnerabilities such as $V_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast) (one type of Prime + Flush).

SecDCP and NoMo cache both leverage dynamic partitioning to improve performance. Compared to SecVerilog cache, SecDCP cache introduces certain side channels which manifest themselves when the number of ways assigned to the victim and attacker changes, e.g., $V_u \rightsquigarrow A_a^{inv} \rightsquigarrow V_u$ (slow) (one type of Flush + Time) vulnerability. NoMo cache behaves more carefully when changing the number of ways during dynamic partitioning, however, it requires victim's sensitive data to fit into the assigned partitions, otherwise it will be put into the unreserved way and allow eviction by the attacker. SecDCP does not have unreserved way. All the space in the cache will be either belongs to *High* or *Low* partition.

Sanctum cache and CATalyst cache are both controlled by a powerful software monitor and they disallow secure page sharing between victim and attacker to prevent vulnerabilities such as $A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [27]). Sanctum cache does not consider internal interference while CATalyst cache is more carefully designed to prevent different vulnerabilities with the implemented software system, so far supporting preventing all of the vulnerabilities, but only works for LLC and with high software implementation complexity and some assumptions that might be hard to achieve in other scenarios, e.g., assuming the secure partition is big enough to fit all the secure data. MI6 cache is the combination of Sanctum and disabling speculation when interacting with the outside world. Therefore, in normal execution, it behaves the same as Sanctum. For speculative execution, because it will simply disable all the speculation when involving the outside world, the external interference vulnerability such as $V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow) (one type of Evict + Probe) vulnerability will be prevented.

InvisiSpec cache does not modify the original cache state but places the data in a

speculative buffer partition during the speculation or out-of-order load or store. Since during speculation cache state is not actually updated, the speculative execution cannot trigger any of the steps in the three-step model. RIC cache focuses on eviction based attack and therefore are good at preventing even some internal miss-based vulnerability such as $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [22]) but are bad at all hit-based vulnerabilities. PL cache is line-partitioned and uses locking techniques for victim's security critical data. It can prevent many vulnerabilities because preloading and locking secure data disallow the attacker or non-secure victim data to set initial states ($Step\ 0$) for victim partition, and therefore brings uncertainty to the final observation by the attacker, e.g., $A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (one type of Cache Internal Collision [23]) vulnerability is prevented.

Randomization-based caches (columns for SHARP cache, and columns for RP cache to columns for Non Deterministic cache in Table 3.5 and Table 3.6) inherently de-correlate the relationship between information of victim's security critical data's address and observed timing from cache hit or miss, or between the address and observed timing of flush or cache coherence operations. For speculative execution, they also de-correlate the relationship between the address of the data being accessed during speculative execution or out-of-order load or store and the observed timing from a cache hit or miss. Randomization can be used when bringing data into the cache, evicting data, or both. Some designs randomize the address to cache set mapping. As a result of the randomization, the mutual information from the observed timing, due to having or not having data in the cache, could be reduced to 0, if randomization is done on every memory access. Some secure caches use randomization to avoid many of the miss-based internal interference vulnerabilities. However, they may still suffer from hit-based vulnerabilities, especially when the vulnerabilities are related to internal interference. However, randomization is also likewise recognized to increase performance overheads [58]. It also requires a fast and secure random number generator. Most of the randomization is cache-line-based and can be combined with differentiation of sensitive data to be more efficient.

RP cache allows eviction between different sensitive data, which leaves vulnerabilities such as $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [22]) still possible, while Newcache prevents this. Both of the RP cache and Newcache are not able to prevent hit-

based internal-interference vulnerabilities such as $A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (one type of Cache Internal Collision [23]). Random Fill cache is able to use total de-correlation of memory access and cache access of victim's security critical data to prevent most of the internal and external interference. However, when security critical data is initially directly loaded into the cache block for *Step* 1, Random Fill cache will not randomly load security critical data and allows vulnerabilities such as $V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow) (one type of Flush + Time) vulnerability to exist. CEASER cache uses encryption scheme plus dynamic remapping to randomize mapping from memory addresses to cache sets. However, this targets eviction based attacks and cannot preventing hit-based vulnerabilities such as $V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast) (one type of Flush + Probe Invalidation). SCATTER cache encrypts both the cache address and process ID when mapping into different cache index to further prevent more hit-based vulnerabilities for shared and read only memory. Non Deterministic cache totally randomizes timing of cache accesses by adding delays and can prevent all attacks (but at tremendous performance cost).

Estimated Performance and Security Tradeoffs

Table 3.7 shows the implementation and performance results of the secure caches, as listed by the designers in the different papers. At the extreme end, there is the Non Deterministic cache: with random delay, the secure cache can prevent all the cache timing vulnerabilities in some degree – while their paper reports only 7% degradation in performance, we expect it to be much more for more complex application than AES algorithm. Disabling caches eliminates the attacks, but at a huge performance cost. Normally, a secure cache needs to sacrifice some performance in order to de-correlate memory access with the timing. The secure caches that tend to be able to prevent more vulnerabilities usually have weaker performance compared with other secure caches.

Towards an Ideal Secure Cache

Based on the above analysis, a good secure cache should consider all the 72 types of *Strong* vulnerabilities, e.g., external and internal interference, hit-based and miss-based vulnerabilities. Considering all factors and based on Table 3.5 and Table 3.6, we have several

Table 3.7: Existing secure caches' implementation method, performance, power and area comparison.

Metric		Set Associative Cache	SP* Cache [27, 20]	SecVerilog Cache [57, 56]	SecDCP Cache [48]	NoMo Cache [12]	SHARP Cache [53]	Sanctum Cache [10]	MI6 Cache[7]	InvisiSpec Cache [52]	CATalyst Cache [29]	DAWG Cache [25]	RIC [22]	PL Cache [49]	RP Cache [49]	Newcache [50, 31]	Random Fill Cache [30]	CEASER Cache [38]	SCATTER Cache[51]	Non Deterministic Cache [23]	Cache Disabled
Cache Configuration	L1 Cache	–	–	4-way 32kB	private 2-way 32kB D/I	8-way 32KB D/I	private 4-way 32KB D/I	8-way 32KB D/I	8-way 32KB D/I	private 8-way 64KB D, 4-way 32KB I	–	private 2× 8-way 32 KB	4-way 32 KB D/I	direct-mapped, 2-way and 4-way 4KB to 32KB	2-way 4-way 16KB 32KB	2-way, 4-way or 8-way	4-way 32 KB	private 8-way 32KB	8-way 32KB	2-way 2 KB D/I	–
	L2 Cache	–	–	–	shared 8/16-way 1/2MB	unified 8-way 256KB	private 8-way 256KB	256KB 8-way L2	1MB, 16-way, max 16 requests	–	–	private 8-way 256 KB	8-way 256 KB	–	–	–	8-way 2 MB	private 8-way 256KB	–	shared 4-way 128 KB	–
	LLC	–	–	–	16-way 2MB	shared 16-way 2MB	shared 16-way 2MB	8MB 16- way LLC partitioned into core-local slices	coherent with I and D	shared 16-way 2MB	20-way 20 MB	shared 8× 16-way 2 MB	shared 16-way 2 MB/512 KB	–	–	–	–	shared 16-way 8MB	16-way 2MB	–	–
Benchmark		–	RSA, AES and MD5	MiBench, ciphers and hash functions of OpenSSL	SPEC 2006	SPEC 2006	SPEC INT2006, SPEC FP2006 and PARSEC	SPEC INT2006	SPEC INT2006	SPEC INT2006, SPEC FP2006 and PARSEC	SPEC 2006 and PARSEC	PARSEC and GAP Benchmark Suite (GAPBS)	SPEC 2006	AES, SPEC 2000	AES, SPEC 2000	SPEC 2000	SPEC 2006	SPEC 2006 and GAP	SPEC 2017	AES cryptographic algorithm	–
Implementation		–	–	MIPS processor	Gem5 simulator [4]	Pin [32] based trace-driven x86 simulator	MARSS [35] cycle-level full-system simulator	Rocket Chip Generator [28]	RiscyOO processor [58] + Xilinx FPGA	Gem5 simulator + CACTI 5 [45]	Intel Xeon E5 2618L v3 processors	zsim [39] execution driven x86-64 simulator and Haswell [37] hardware	Cacti [43] version 6.5	M-Sim v2.0 [42]	M-Sim v2.0	CACTI 5.0	Gem5 simulator	Pin-based x86 simulator	Gem5 simulator	HotLeakage simulator [60]	–
Performance Overhead		–	1%	–	12.5% better over static cache partitioning	1.2% average, 5% worst	3%-4%	–	–	reduce the execution slowdown of Spectre from 74% to only 21%	average slowdown of 0.7% for SPEC and 0.5% for PARSEC	L1 and L2 most 4%-7%	improves 10%	12%	0.3%, 1.2% worst	within the 10% range of the real miss rate	3.5%, 9% if setting the window size to be largest	1% for performance optimization	3.5% for performance optimization	7% with simple benchmarks	–
Power		–	–	–	–	–	–	–	–	L1 0.56 mW, LLC 0.61 mW	–	–	–	–	average 1.5 nj	< 5% power	–	–	–	–	–
Area Overhead		–	–	–	–	–	–	–	–	L1-SB LLC-SB Area (mm2) 0.0174 0.0176	–	–	0.176%	–	–	–	–	–	–	–	–

suggestions and observations for a secure cache design which can defend timing attacks:

- Internal interference is important for caches to prevent timing attacks and is the weak point of most of the secure caches. To prevent this, the following three subpoints should be considered:
  - Miss-based internal interference can be solved by randomly evicting data to decorrelate memory access with timing information when either data to be evicted or data to be cached is sensitive, e.g., Newcache prevents $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [22]) vulnerability.
  - Hit-based internal interference can be solved by randomly bringing data into the cache, e.g., Random Fill cache prevents $A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (Cache Internal Collision) vulnerability.
  - To limit internal interference at lower performance cost, rather than simply assume all of victim's data is sensitive, it is better to differentiate real sensitive data from other data in the victim code. However, identification of sensitive information needs to be carefully used, e.g., Random Fill cache is vulnerable to $V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (fast) (one type of Evict + Time [49]) vulnerability which most of the secure caches are able to prevent.
- Direct partitioning between the victim and the attacker, although may hurt cache space utilization or performance, is good at disallowing attacker to set known initial state to victim's partition and therefore prevents external interference. Alternatively, careful use of randomization can also prevent external interference.

It should be noted that some cache designs only focus on certain levels, e.g., CATalyst cache only works at the last level cache. In order to fully protect the whole cache system from timing attacks, all levels of caches in the hierarchy should be protected with related security features. E.g., Sanctum is able to prevent all levels of caches from L1 to last-level cache. Consequently, secure cache design needs to be realizable at all levels of the cache hierarchy and not just one.

Chapter 4

Evaluation of Timing Vulnerabilities of Caches and TLBs

Having presented the theoretical modeling approach in Chapter 3, this chapter presents evaluation of the attacks on x86 commercial processors and Arm mobile devices. The chapter further presents appliation of the three-step model to TLBs, which are cache-like structures, and shows evaluation on RISC-V processor.

4.1 Cache Timing Vulnerabilities and x86 Benchmark Suite

To address the need to understand and evaluate all the different possible types of attacks, this work presents both a theoretical model of all possible timing attacks in caches, and a benchmark suite that can test for the theoretical vulnerabilities on real processors, or simulations of new designs.

4.1.1 Modeling of Cache Timing Attacks

The goal of this work is to present the first set of benchmarks which can be used to evaluate all the vulnerabilities of processor caches to timing attacks. Such attacks can be used, for example, by Spectre variants, e.g., [3, 4, 72, 73], to extract sensitive information. For each benchmark, if there is observable timing difference on a particular processor, it means that the processor may be vulnerable to the corresponding attack.

Assumptions and Threat Model

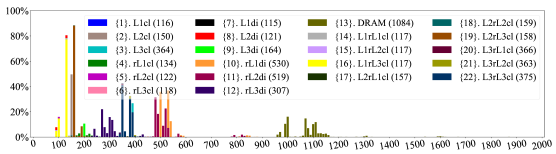
We assume that there is a victim process running on the CPU core and performing secret-dependent memory accesses. There is also a malicious attacker process on the same or different CPU core, whose aim is to determine a secret memory address or address index used by the victim. Both attacker's and victim's accesses affect a cache block in one of the L1 data caches, through which a possible timing channel exists.

The goal of the benchmarks is to evaluate for which types of accesses by the victim and the attacker there is indeed a timing vulnerability in caches. The presented benchmarks are not actual security exploits, rather they implement memory-related operations that correspond to all possible timing attacks. Each benchmark outputs whether there is a statistically significant timing difference that the attacker could observe to extract information from the timing channel about the secret and unknown address u of the victim.

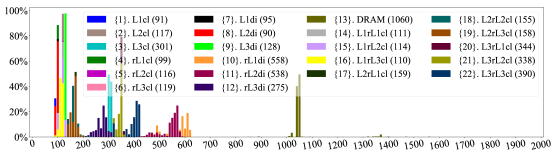
The current model focuses on all possible timing attacks in the L1 data cache. The model includes uses of any memory-related operations (load, store, flush) and cache coherence protocol. The model assumes a multi-core and possibly hyper-threading processor, with a cache hierarchy of local and remote L1 cache, L2 cache, and a shared L3 cache (which is possibly divided into different cache slices).

Current benchmarks do not consider timing attacks of other levels in cache hierarchy besides L1, but it should be straightforward to extend to the other levels. We do not consider directory-related attacks [74] or attacks based on replacement policy [75], but it should be possible to model these by adding more states to the model (and still keep an only total of three steps). This work does not cover TLB attacks [48, 50], but there is already a theoretical model for TLBs [76], and similar benchmarks can be developed for TLB attacks (possibly merge with our benchmarks).

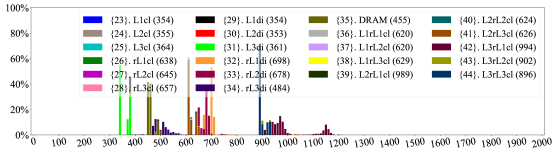
The work considers more than just "fast" and "slow" timings. This means that the influence of structures such as Miss Status Handling Registers (MSHRs), load and store buffers between processor and caches, and line-fill buffers between cache levels are accounted for. However, benchmarks for timing attacks that are just due to these structures could likely be developed. Our analysis is also general for all the cases of the three-step model and



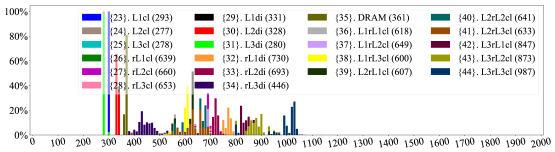
(a) Timing of read access on Intel Xeon E5-1620 processor



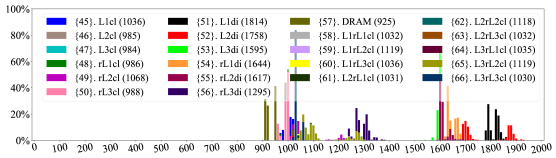
(b) Timing of read access on Intel Xeon E5-2690 processor



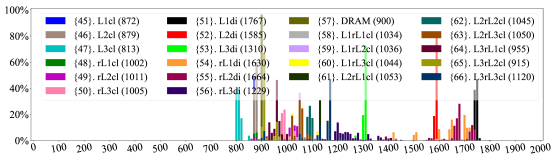
(c) Timing of write access on Intel Xeon E5-1620 processor



(d) Timing of write access on Intel Xeon E5-2690 processor



(e) Timing of flush operation on Intel Xeon E5-1620 processor



(f) Timing of flush operation on Intel Xeon E5-2690 processor

Figure 4.1: Histograms of read, write, and flush operations' timing (each contains 8 operations for timing measurement) under all possible data movements considered in this work. The timing is for the timing observation step, i.e. *Step* 3, in the tested three-step patterns. Note, different processors have different timing, and not all different types of data movements can be distinguished on different processors. The data is presented for Intel Xeon E5-1620 (a, c, e) and Intel Xeon E5-2690 (b, d, f) processors. Numbers in the "{}" in the legend denote the different data movement types. {1} - {22} correspond to read operation, {23} - {44} correspond to write operation, {45} - {66} correspond to flush operation, to access clean L1 data, clean L2 data, clean L3 data, remote clean L1 data, remote clean L2 data, remote clean L3 data, dirty L1 data, dirty L2 data, dirty L3 data, remote dirty L1 data, remote dirty L2 data, remote dirty L3 data, DRAM data, clean data in both L1 and remote L1, clean data in both L1 and remote L2, clean data in both L1 and remote L3, clean data in both L2 and remote L1, clean data in both L2 and remote L2, clean data in both L2 and remote L3, clean data in both L3 and remote L1, clean data in both L3 and remote L2, clean data in both L3 and remote L3, respectively. Numbers in the "()" in the legend show the average cycles needed for completing that type of memory operation. The x axis shows the access latency in cycles.

we do not differentiate if the access is from the instruction or a prefetcher.

The flush operation in this work refers to the *clflush* instruction in x86, which causes data to be flushed from all levels of caches (including data in other cores) back to the main memory. The timing are measured from when each of the memory-related operations is issued until the instruction commits in the processor pipeline.

Improved Modeling of Real Processors

We expand the original model [8] by considering more realistic cases for a processor's memory-related operation. The expanded modeling allows us to cover all possible attacks, and uncover new vulnerabilities. For example, some proposals [45] discuss disabling flush instruction to prevent Flush+Reload [27] based attacks. However, because we consider different flush operations, our benchmarks show that using remote access to invalidate (flush) the cache could also result in a vulnerability.

Timing Observation on Local vs. Remote Core. Our cache attack model assumes a multi-core system and possibly a hyper-threading system as well. We model such a system using two cores: a "local" and a "remote" core, each with L1, L2, and shared L3 caches. The target cache block is located in the local core. Remote core affects the target cache block on the local core by using cache coherence protocol. E.g., perform write operations on the remote core to invalidate the local core's data using cache coherence protocol. As future work, more detailed modeling of multi-core system can be done.

For each read, write or flush operation, it may target the data that is in the local L1 cache, L2 cache, or L3 cache slice, or that is in the remote L1 cache, L2 cache, or L3 cache slice. The cache block can be either in a clean or dirty state for the above 6 locations ($6 \times 2 = 12$ types). The clean data may also be in both local or remote core, which can be in any cache hierarchy (L1, L2, or L3 cache) for both cores ($3 \times 3 = 9$ types). Otherwise, the data is not in any level of the cache hierarchy, i.e., it is in the DRAM (1 type). We consider all these 66 timings (3 operations $\times (12 + 9 + 1) = 66$) to be different from each other and use these 66 types of timings in our three-step cache simulator, discussed in Section 4.1.2, to determine if a three-step combination can be used in an attack.

Figure 4.10 shows the histograms of these 66 types of timing observations for Intel Xeon E5-1620 and E5-2690 processors. Based on the histograms, we found that some operations are differentiable from each other, while some are not. In general, the timing is processor-specific, so we need to consider and examine all various cache timings and cannot just assume "fast" and "slow" timings as was done previously [8].

Hyper-Threading vs. Time-Slicing. We consider that the victim and the attacker

on one core can either run in time-slicing setting or run in parallel as two hyper-threads (if there is hyper-threading support in the processor). For the case of accesses on "local" vs. "remote" cores, the accesses on local and remote cores can be done in parallel.

Read (Load) Access vs. Write (Store) Access. For operations related to memory accesses, our model considers that they can be either read (load) access or write (store) access. The timing of writes is not well explored in attacks, except for one work [77]. For example, for Flush + Reload attack, the previous attack [27] uses the load operation in the final step to reload secret data and observe timing. In our model, we also test store operation in the final step to access secret data and reveal that attacks with write in the final step are also effective, for example.

Flush vs. Write Invalidation. In our model, we consider that a flush operation can be achieved by a $clflush$ type instruction, that flushes data from all caches back to main memory, or that by writing the corresponding line in the remote core it will trigger cache coherence and result in the local cache line being invalidated.

4.1.2 Derivation of All Vulnerabilities

In this work, we build a new cache three-step simulator based on the new model discussed in Section 4.1.1. It considers different memory-related operations and differentiates among the 66 timing variations discussed in Section 4.1.1 that are related to L1 cache timing attack for the final timing observation step. Further, we give categorizations of vulnerabilities to find common features that attacks exploit.

Judging the Effectiveness of Three-Step Combination

In order for a three-step combination to be effective for an attack, at least the unknown victim's address u should be involved in one of the three steps since u is the unknown secret the attacker tries to learn. In this case, the vulnerability will have V_u or V_u^{inv} as one or more of the three-steps to represent the operations on the secret u.

Based on the 17 states shown in Table 4.8, for the three-step model, the attacker tries to learn the value of u by guessing if u equals to: a, a^{alias} or NIB. a denotes the address that is within the set of sensitive locations x and maps to the target cache line. a^{alias} denotes

N.	Vulnerability Type			Type	Attack	Attack Strategy
	$S1$	$S2$	$S3$			
1	A^{inv}	V_u	V_a	*I-A*	[23]	Cache Collision
2	V^{inv}	V_u	V_a	*I-A*	[23]	
3	A_a^{inv}	V_u	V_a	*I-A*	[23]	
4	V_a^{inv}	V_u	V_a	*I-A*	[23]	
5	A_a^{inv}	V_u	A_a	*E-A*	[27, 54]	Flush + Reload
6	V_a^{inv}	V_u	A_a	*E-A*	[27, 54]	
7	A^{inv}	V_u	A_a	*E-A*	[27, 54]	
8	V^{inv}	V_u	A_a	*E-A*	[27, 54]	
9	V_u^{inv}	A_a	V_u	*E-A*	new [8]	Reload + Time
10	V_u^{inv}	V_a	V_u	*I-A*	new [8]	
11	A_a	V_u^{inv}	A_a	*E-A*	[55]	Flush + Probe
12	A_a	V_u^{inv}	V_a	*I-A*	new [8]	
13	V_a	V_u^{inv}	A_a	*E-A*	new [8]	
14	V_a	V_u^{inv}	V_a	*I-A*	new [8]	
15	V_u	A_a^{inv}	V_u	*E-A*	new [8]	Flush + Time
16	V_u	V_a^{inv}	V_u	*I-A*	new [8]	
17	A^{inv}	V_u^{inv}	A_a	*E-A*	**new**	Cache Coherence Flush + Reload
18	A^{inv}	V_u^{inv}	V_a	*I-A*	**new**	
19	V^{inv}	V_u^{inv}	A_a	*E-A*	**new**	
20	V^{inv}	V_u^{inv}	V_a	*I-A*	**new**	
21	A_a^{inv}	V_u^{inv}	A_a	*E-SA*	**new**	Cache Coherence Prime + Probe
22	A_a^{inv}	V_u^{inv}	V_a	*I-SA*	**new**	
23	V_a^{inv}	V_u^{inv}	A_a	*E-SA*	**new**	
24	V_a^{inv}	V_u^{inv}	V_a	*I-SA*	**new**	
25	A_d^{inv}	V_u^{inv}	A_d	*E-S*	**new**	
26	A_d^{inv}	V_u^{inv}	V_d	*I-S*	**new**	
27	V_d^{inv}	V_u^{inv}	A_d	*E-S*	**new**	
28	V_d^{inv}	V_u^{inv}	V_d	*I-S*	**new**	
29	V_u^{inv}	A_a^{inv}	V_u	*E-SA*	**new**	Cache Coherence Evict + Time
30	V_u^{inv}	V_a^{inv}	V_u	*I-SA*	**new**	
31	V_u^{inv}	A_d^{inv}	V_u	*E-S*	**new**	
32	V_u^{inv}	V_d^{inv}	V_u	*I-S*	**new**	
33	V_u	V_a	V_u	*I-SA*	[22]	Bern-stein's Attack
34	V_u	V_d	V_u	*I-S*	[22]	
35	V_d	V_u	V_d	*I-S*	[22]	
36	V_a	V_u	V_a	*I-SA*	[22]	
37	V_d	V_u	A_d	*E-S*	new [8]	Evict + Probe
38	V_a	V_u	A_a	*E-SA*	new [8]	
39	A_d	V_u	V_d	*I-S*	new [8]	Prime + Time
40	A_a	V_u	V_a	*I-SA*	new [8]	
41	V_u	A_d	V_u	*E-S*	[49]	Evict + Time
42	V_u	A_a	V_u	*E-SA*	[49]	
43	A_d	V_u	A_d	*E-S*	[49, 56]	Prime + Probe
44	A_a	V_u	A_a	*E-SA*	[49, 56]	

(a) Timing vulnerabilities with *Step*3 as memory access operation.

N.	Vulnerability Type			Type	Attack	Attack Strategy
	$S1$	$S2$	$S3$			
45	A^{inv}	V_u	V_a^{inv}	*I-A*	new [8]	Cache Colli. Inv.
46	V^{inv}	V_u	V_a^{inv}	*I-A*	new [8]	
47	A_a^{inv}	V_u	V_a^{inv}	*I-A*	[26]	Flush + Flush
48	V_a^{inv}	V_u	V_a^{inv}	*I-A*	[26]	
49	A_a^{inv}	V_u	A_a^{inv}	*E-A*	[26]	
50	V_a^{inv}	V_u	A_a^{inv}	*E-A*	[26]	
51	A^{inv}	V_u	A_a^{inv}	*E-A*	new [8]	Flush + Reload Inv.
52	V^{inv}	V_u	A_a^{inv}	*E-A*	new [8]	
53	V_u^{inv}	A_a	V_u^{inv}	*E-A*	new [8]	Reload + Time Inv.
54	V_u^{inv}	V_a	V_u^{inv}	*I-A*	new [8]	
55	A_a	V_u^{inv}	A_a^{inv}	*E-A*	new [8]	Flush + Probe Inv.
56	A_a	V_u^{inv}	V_a^{inv}	*I-A*	new [8]	
57	V_a	V_u^{inv}	A_a^{inv}	*E-A*	new [8]	
58	V_a	V_u^{inv}	V_a^{inv}	*I-A*	new [8]	
59	V_u	A_a^{inv}	V_u^{inv}	*E-A*	new [8]	Flush + Time Inv.
60	V_u	V_a^{inv}	V_u^{inv}	*I-A*	new [8]	
61	A^{inv}	V_u^{inv}	A_a^{inv}	*E-A*	**new**	Cache Coherence Flush + Reload Inv.
62	A^{inv}	V_u^{inv}	V_a^{inv}	*I-A*	**new**	
63	V^{inv}	V_u^{inv}	A_a^{inv}	*E-A*	**new**	
64	V^{inv}	V_u^{inv}	V_a^{inv}	*I-A*	**new**	
65	A_a^{inv}	V_u^{inv}	A_a^{inv}	*E-SA*	**new**	Cache Coherence Prime + Probe Inv.
66	A_a^{inv}	V_u^{inv}	V_a^{inv}	*I-SA*	**new**	
67	V_a^{inv}	V_u^{inv}	A_a^{inv}	*E-SA*	**new**	
68	V_a^{inv}	V_u^{inv}	V_a^{inv}	*I-SA*	**new**	
69	A_d^{inv}	V_u^{inv}	A_d^{inv}	*E-S*	**new**	
70	A_d^{inv}	V_u^{inv}	V_d^{inv}	*I-S*	**new**	
71	V_d^{inv}	V_u^{inv}	A_d^{inv}	*E-S*	**new**	
72	V_d^{inv}	V_u^{inv}	V_d^{inv}	*I-S*	**new**	
73	V_u^{inv}	A_a^{inv}	V_u^{inv}	*E-SA*	**new**	Cache Coherence Evict + Time Inv.
74	V_u^{inv}	V_a^{inv}	V_u^{inv}	*I-SA*	**new**	
75	V_u^{inv}	A_d^{inv}	V_u^{inv}	*E-S*	**new**	
76	V_u^{inv}	V_d^{inv}	V_u^{inv}	*I-S*	**new**	
77	V_u	V_a	V_u^{inv}	*I-SA*	new [8]	Bernstein's Inv. Attack
78	V_u	V_d	V_u^{inv}	*I-S*	new [8]	
79	V_d	V_u	V_d^{inv}	*I-S*	new [8]	
80	V_a	V_u	V_a^{inv}	*I-SA*	new [8]	
81	V_d	V_u	A_d^{inv}	*E-S*	new [8]	Evict + Probe Inv.
82	V_a	V_u	A_a^{inv}	*E-SA*	new [8]	
83	A_d	V_u	V_d^{inv}	*I-S*	new [8]	Prime + Time Inv.
84	A_a	V_u	V_a^{inv}	*I-SA*	new [8]	
85	V_u	A_d	V_u^{inv}	*E-S*	new [8]	Evict + Time Inv.
86	V_u	A_a	V_u^{inv}	*E-SA*	new [8]	
87	A_d	V_u	A_d^{inv}	*E-S*	new [8]	Prime + Probe Inv.
88	A_a	V_u	A_a^{inv}	*E-SA*	new [8]	

(b) Timing vulnerabilities with *Step*3 as invalidation operation.

Table 4.1: The table shows all the L1 cache timing vulnerabilities. The *N.* column assigns each type of vulnerability a number. The *Vulnerability Type* column shows the three steps that define each vulnerability. The *Type* column proposes the categorization the vulnerability belongs to. "E" and "I" are for internal and external interference types, respectively. "S", "A" and "SA" are set-based, address-based types and the types that are both set-based and address-based, respectively. The *Attack* column shows if a vulnerability has been previously presented in the literature. The *Attack Strategy* column gives a common name for each set of vulnerabilities that would be exploited in an attack in a similar manner. *Inv.* means invalidation. Light-blue colored rows are the vulnerabilities which are first presented in this work.

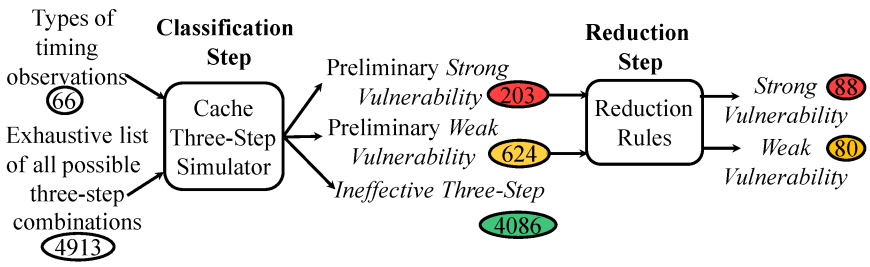


Figure 4.2: The derivation process of all the *Strong* and *Weak* types of L1 cache timing vulnerabilities.

any data address that belongs to sensitive locations x and also maps to the cache line but is not a. Apart from all possible sensitive address mapping to the target cache line, u may not map to the target cache line the attacker is measuring. We denote these addresses as NIB (not-in-block). Therefore, u can be either a, a^{alias}, or NIB. If the attacker is able to find access time of one value significantly different from the other two values, he or she is able to learn the value of u and the corresponding three-steps is a *Strong* type vulnerability. Meanwhile, if the attacker is not able to clearly distinguish whether u is a, a^{alias}, or NIB based on the timing, but there are still timing differences observed, then the corresponding attacks belong to *Weak* type of vulnerabilities. Otherwise, if the timing is always the same regardless of different values of u, it will be an *Ineffective* three-step combination.

New Cache Three-Step Simulator

Figure 4.2 shows the derivation process of vulnerabilities. We wrote Python scripts to develop the cache three-step simulator. The simulator takes all 4913 three-step combinations and 66 types of timing observations as input, checks and outputs the three-steps that belong to *Strong*, *Weak* vulnerabilities, or *Ineffective* types, respectively. For the step that is u-related, since u is in secure range x, the possible candidates of u for a cache block are a, a^{alias}, and NIB, so the simulator checks the timing when u is a, a^{alias}, and NIB, respectively. The timing variance exists if different possible values of u correspond to different timings of the 66 types. We enumerate all possible operations (read/write for access, remote write/flush for invalidation) for a step and consider different timings for each operation. Therefore, each three-step pattern may have different types of timing observations. The rules from our prior three-step model work [8], on which the prior chapter was based, are used to remove repeat

and redundant three-step patterns.

As shown in Figure 4.2, based on the much finer-grained categorization of timing differences, we derived in total 88 *Strong* effective vulnerabilities and 80 *Weak* effective vulnerabilities after removing repeat three-step patterns. They are shown in Table 4.1, where light-blue colored rows (in total 32 types) are the new vulnerabilities (compared to study [8], also presented in the prior chapter) which we found through running of new cache three-step simulator (16 types of the original *Strong* effective vulnerabilities [8] become *Weak* vulnerabilities when considering multi-core systems). We provide new names for the new attacks in *Attack Strategy* in Table 4.1 while re-use existing names if the attacks were presented before. As validated in Section 4.1.4 through tests on real processors, there are no other effective vulnerabilities except the types we derive in Table 4.1.

Categorizations of the Vulnerabilities

We first categorize different vulnerabilities as based on internal (I) or external (E) interference. The types that only involve the victim's behavior, V, in the states of *Step* 2 and *Step* 3 are internal interference vulnerabilities (I). The remaining ones are external interference (E) timing vulnerabilities.

In prior work [8, 78], cache vulnerabilities are categorized as hit-based and miss-based vulnerabilities, based on the cache behaviors the attackers want to observe (cache misses or hits). This definition does not fit our model since there are different types of timings for L1 data hits and misses in the real machines. For example, attacks can derive information using timing difference from two types of cache misses.

Therefore, we further categorize the vulnerabilities as address-based (A) if they are able to derive the cache line address of u by observing cache hit of u and obtaining different timing compared with other candidate data. Set-based (S) vulnerabilities are the ones that can know the mapped set of u by conflicting and generating eviction between u and candidate data addresses. The third type are the ones that potentially derive information from set or address (SA) depending on timing differences derived for all the candidates of u. For example, SA type #33 vulnerability $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ can be set-based if a and u are not the same but map to the same cache set, which differs in timing between {2}{8}{24}{30}, a

local L2 hit, and {1}{7}{23}{29}, a local L1 hit. Or it can be address-based if a maps to u and *Step* 1 (V_u) and *Step* 2 (V_a) are accessed by different operations (read or write), which have different timing between reads of L1 clean data and dirty data, {1} and {7}, or writes of L1 clean data and dirty data, {23} and {29}.

4.1.3 Benchmark Implementation

For each vulnerability, there are three steps, where each can be: read or write access for a memory access operation, or flush or write in the remote core for an invalidation-related operation. Thus, there are in total of $2^3 = 8$ cases considering different types of operations. Further, if the vulnerabilities have both the victim and the attacker running in one core, these two parties can run either time-slicing or multi-threading. Based on that, one case may be doubled for running in two settings. So for one vulnerability type, there are corresponding 8 - 16 cases depending on the specific vulnerability. In total, there are 1094 benchmarks for all 88 *Strong* type vulnerabilities. We wrote C programs to automatically generate the binaries for each of the 1094 benchmarks.

Evaluating Three-Step Combinations

For a specific benchmark that implements one case of the three-step combinations, following the idea of the cache three-step simulator in Section 4.1.2, if the step is V_u, the benchmarks separately test the timing when V_u is V_a, $V_{a^{alias}}$, or V_{NIB}. If the step is V_u^{inv}, the benchmarks separately test the timing when V_u^{inv} is V_a^{inv}, $V_{a^{alias}}^{inv}$, or V_{NIB}^{inv}. The timing of the last step in the three-step pattern is measured. For each of the cases, there is RUN_NUM number of trials, and Welch's t-test [79] is used to distinguish the distributions of the measured timings. We consider two distributions to be significantly different from each other if the probability of observing the data given that they come from the same distribution is less than 0.05%.

For an effective vulnerability, one of the three candidates of V_u (or V_u^{inv}) should generate timing distribution that is statistically different from the other two candidates, which we use to extract information from the runs. This is for the *Strong* vulnerability types which are 88 types in total. The 80 *Weak* vulnerability types are not currently considered in the benchmarks but can be straightforward to add if needed. At end of each benchmark run,

the benchmark outputs if there was significant timing difference – "vulnerability is found", or not – "vulnerability not found".

Timing Measurement and Noise Minimization

We use *rdtsc* instruction in our benchmarks to do timing measurements, which is the most effective method compared with hardware performance counters, which may be limited [26] or lacking-determinism [80], or using a "counting" thread. AMD's *rdtsc* instruction is not as accurate as Intel machine's, but there are many works [81, 50] showing that it is also able to be used for cache timing attacks.

Noise and variation in the timing measurements could further result in false negatives (if the time measurement was not accurate enough to distinguish different timings of accesses) or false positives (if timing changes resulted in timing measurement differences even though there is no timing difference). We isolate cores to reduce the software noise to minimize the false positives. To reduce the false negatives from the noise, instead of measuring just one cache block, we arbitrarily chose 8 cache blocks from different cache sets to do operation on. Further, the measurements are all repeated RUN_NUM times and collect statistical data. The *fence* instructions are added between each memory-related instruction to enforce an ordering constraint for the attacks.

To reduce the variation of the timing among different cache sets and further minimize the false negatives, the timing measurement of the last step is repeated for each test if the last step is u-related step. Specifically, right after the third step's timing measurement, we trigger and measure the timing of this step again, which is guaranteed to result in an L1 cache hit timing or timing to invalidate the data that is not in the caches, depending on the concrete memory operations. We then compare the timing of the third step with the repeated third step. This eliminates any variations in timing among different cache sets.

Benchmark Code Example

Figure 4.3 shows an example pseudo code of #42 vulnerability $V_u \rightsquigarrow A_a \rightsquigarrow V_u$'s benchmark for read ($V_u$), write ($A_a$), and write ($V_u$) access of the three steps and running in hyper-threading setting.

```
1.  #define LIM 0.0005
2.  //mutex to sequence three-step operations
3.  mutex = mmap(mutex_size, PROT_READ|PROT_WRITE, …)
4.  init_array(arr); //initialize data array and load it into L1, L2, and L3
5.  mutex[0] = NOONE_RUN;
6.  if ((pid_l1=fork()) < 0) { exit(1); //fail to fork process}       //local attacker
7.  else if (pid_l1==0){
8.      CPU_SET(att_num, &mycpuset);
9.      sched_setaffinity(getpid(), sizeof(cpu_set_t), &mycpuset);
10.     for (int m=0; m<RUN_NUM; m++){
11.         for (int j=0; j<4; j++){
12.             // before the attack, initialize mutex
13.             if(mutex[0]==NOONE_RUN){ mutex[0]=STEP1_RUN;}
14.             // step 2 Aa
15.             while(mutex[0]!=STEP2_RUN) sched_yield;
16.             attacker_write_8_access(a);
17.             mutex[0]=STEP3_RUN;
18.     }} exit(0);}
19. if ((pid_l2=fork()) < 0) { exit(1); //fail to fork process}       //local victim
20. else if (pid_l2==0){
21.     CPU_SET(vic_num, &mycpuset);
22.     sched_setaffinity(getpid(), sizeof(cpu_set_t), &mycpuset);
23.     for (int m=0; m<RUN_NUM; m++){
24.         for (int j=0; j<4; j++){
25.             // step 1 Vu
26.             while(mutex[0]!=STEP1_RUN) sched_yield;
27.             if(j==0) victim_read_8_access(a);
28.             else if (j==1) victim_read_8_access(a_alias);
29.             else if (j==2) victim_read_8_access(NIB);
30.             else if (j==3) dummy_operation;
31.             mutex[0]=STEP2_RUN;
32.             // step 3 Vu and measure time
33.             while(mutex[0]!=STEP3_RUN) sched_yield;
34.             if(j==0) {victim_write_8_access_time(a, t);
35.                       victim_write_8_access_time(a, t_r);}
36.             else if (j==1) {victim_write_8_access_time(a_alias, t);
37.                             victim_write_8_access_time(a_alias, t_r);}
38.             else if (j==2) {victim_write_8_access_time(NIB, t);
39.                             victim_write_8_access_time(NIB, t_r);}
40.             else if (j==3)  dummy_operation;
41.             // timing store
42.             store_third_step_timing(j,t);
43.             store_repeat_access_timing(j,t_r);
44.             mutex[0]=STEP1_RUN;
45.     }}
46. //timing analysis
47. if((p_value(a, a_alias)<LIM && p_value(a, NIB)<LIM)||(
48.   p_value(a, NIB)<LIM&& p_value(a_alias, NIB)<LIM)||(
49.   p_value(a, a_alias)<LIM&& p_value(a_alias, NIB)<LIM)
50. &&((!u_last_step) ||
51. ((pvalue(a_dif, a_alias_dif)<LIM&& pvalue(a_dif, NIB_dif)<LIM)||(
52.   pvalue(a_dif, NIB_dif)<LIM && pvalue(a_alias_dif, NIB_dif)<LIM)||(
53.   pvalue(a_dif, a_alias_dif)<LIM&& pvalue(a_alias_dif, NIB_dif)<LIM))))
54.       printf("Vulnerability is found");
55. else printf("Vulnerability not found");
56. exit(0);}
```

Figure 4.3: Example pseudo code of #42 vulnerability $V_u \rightsquigarrow A_a \rightsquigarrow V_u$ for read (V_u), write (A_a), and write (V_u) case running in hyper-threading setting.

First, we define probability bound of Welch's t-test (line 1) and initialize a shared array (line 2-3) used by mutexes to control the sequence of the three-step accesses. Then, the data (stored in the array) that will be accessed by the victim and the attacker is loaded into the L1 cache (line 4), and consequently possibly brought into L2 and L3 caches. We use *fork()* (line 6 and line 19) to create sub-process, one for the victim and one for the attacker in this example. Each remote and local victim and attacker will have one sub-process throughout the whole test. Each sub-process is assigned to a hardware thread (line 8-9, line 21-22). When running hyper-threading, two local or two remote sub-processes are run in different hardware threads of one CPU, if applicable. If running time-slicing, sub-processes are assigned to one hardware thread. Within each sub-process, the test will be run for a certain predefined RUN_NUM (line 10 and line 23) times so the timing statistics can be done based on a large number of runs. We set RUN_NUM at 600 to minimize noise and maintain a suitable test set number for Welch's t-test to measure distributions.

As discussed in Section 4.1.2, for all the effective vulnerabilities, there will be at least one V_u step (or V_u^{inv}). Within each test, the three candidate values (i.e., V_a, $V_{a^{alias}}$, or V_{NIB})

will be tested for the V_u or V_u^{inv} (line 11 and line 24). The "dummy operation" branch is used to avoid making the third branch to be the last branch, which we found experimentally has an abnormal stable longer timing measurement result.

Figure 4.3 shows a test performing *Step* 1 (V_u, line 25-31), *Step* 2 (A_a, line 14-17) and *Step* 3 (V_u, line 32-44). Last step *Step* 3 is performed twice and results are stored (line 41-43). The first access of *Step* 3 is done to measure if the attacker can observe timing differences when running different values of u. The second access of the third step will always be a hit (fast timing, and is used to obtain baseline "fast" timing for that cache set, as we observed different cache sets can have different timing). In this case, we can collect results of difference between the first access timing and the second access timing for each candidate of u to limit the possibilities that timing difference is due to different cache sets but not different values of u.

In the end, Welch's t-test is first applied to each statistical distribution of candidate values for V_u (or V_u^{inv}) to see whether the attacker can observe different timing when V_u refers to different addresses (line 47-49). If the three-step patterns have u-related step as the last step (implemented by *u_last_step* in line 50), to remove the noise in the timing among different cache sets, the second access timing is considered. Welch's t-test is applied to test the difference of the first and the second access of the last step *Step* 3. Only if one candidate's distribution has significant timing difference compared with the other two, the cache sets' noise is shown to be not the reason of timing difference and the corresponding vulnerability is judged to be effective (line 51-53).

4.1.4 Validation of the Three-Step Model

To validate if there are any other vulnerabilities that are left out apart from all the effective vulnerabilities we derived from our cache three-step simulator, we empirically ran benchmarks for all the $17^3 = 4913$ three-step combinations for 9 processor configurations.

We discovered a number of three-steps, besides the *Strong*, *Weak* and repeat types, returned by the benchmarks to have timing variations but consider all of them as false positives. The false positives that show up in every processor we tested all have the second or the third step to be A^{inv}, V^{inv} or $\star$. The corresponding types cannot be any effective

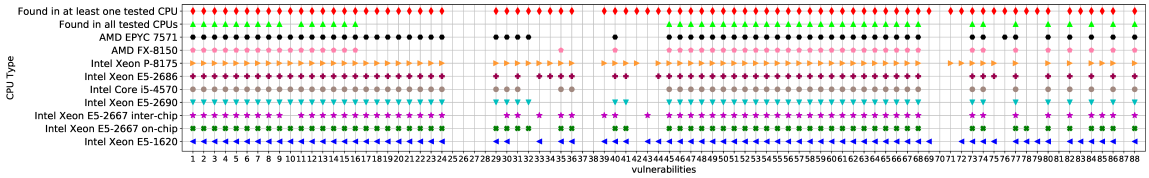


Figure 4.4: Evaluation of 88 *Strong* types of vulnerabilities on different machines. A dot means the corresponding processor is vulnerable to the vulnerability type. Intel Xeon E5-2667 in our lab has two sockets. Therefore, the local and remote core can be both in one socket, i.e., run on-chip; or local and remote core can be in different sockets, i.e., run inter-chip.

Table 4.2: Configurations of the experimental machines, which all have 64B L1 cache line size. (1) denotes the number of hardware threads sharing one L1 cache; (2) denotes the number of hardware threads per socket; (3) denotes the number of sockets.

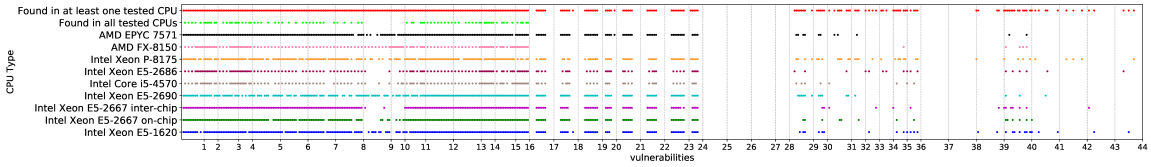
Model Name	L1-D Cache	L1-I Cache	L2 Cache	L3 Cache	(1)	(2)	(3)
Intel Xeon E5-1620	32KB, 8-way	32KB, 8-way	256KB, 8-way	10MB, 20-way	2	8	1
Intel Xeon E5-2667	32KB, 8-way	32KB, 8-way	256KB, 8-way	15MB, 20-way	2	12	2
Intel Xeon E5-2690	32KB, 8-way	32KB, 8-way	256KB, 8-way	20MB, 20-way	2	16	1
Intel Core i5-4570	32KB, 8-way	32KB, 8-way	256KB, 8-way	6MB, 12-way	1	4	1
Intel Xeon E5-2686	32KB, 8-way	32KB, 8-way	1MB, 16-way	33MB, 11-way	1	4	1
Intel Xeon P-8175	32KB, 8-way	32KB, 8-way	256KB, 8-way	45MB, 20-way	2	8	1
AMD FX-8150	16KB, 4-way	64KB, 2-way	2MB, 16-way	8MB, 64-way	1	8	1
AMD EPYC 7571	32KB, 8-way	64KB, 4-way	512KB	8MB	2	4	1

vulnerabilities because these three types of states will make the attacker lose track of useful information due to whole cache flush (A^{inv}, V^{inv}) or zero-knowledge state inference ($\star$) if they are in *Step* 2 or *Step* 3. Reason of three-steps with the second or the third step as A^{inv}, V^{inv} to seem to be effective in the result of running the benchmarks is that whole cache flush currently cannot be implemented under user-level privilege. We use approximate method to implement these states in the benchmark by invalidating every address that is related to the attacks. An approximate method is also used for $\star$ to simulate the zero-knowledge state. Therefore, the timings of A^{inv}, V^{inv} and $\star$ have extra noise leading to the false positives.

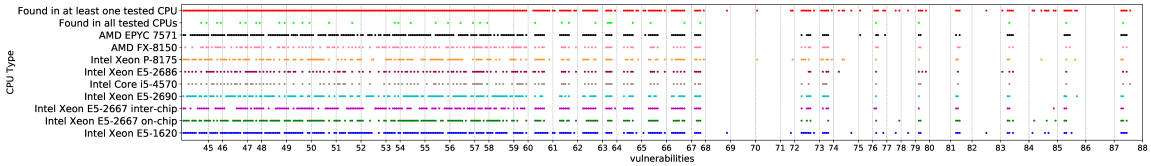
Overall, we found that there are no effective vulnerabilities that are not covered by the vulnerabilities we derived.

4.1.5 Evaluation and Security Discussion

The experimental results reported for Intel processors were performed on Intel Core i5-4570, Xeon E5-2690, E5-2667, E5-1620, P-8175 and E5-2686 CPUs. The AMD tests were on AMD EPYC 7571 and AMD FX-8150. P-8175, E5-2686 and AMD EPYC 7571 instance are from



(a) #1 - #44 vulnerability testing results on different machines.



(b) #45 - #88 vulnerability testing results on different machines.

Figure 4.5: Evaluation of 88 *Strong* types of vulnerabilities for all the benchmark tests. A dot means the corresponding processor is vulnerable to the vulnerability case. For each vulnerability, a fixed number of cases (see Section 4.1.3) are tested according to the vulnerability type. And there are in total 1094 cases for 88 *Strong* types of vulnerabilities.

Amazon EC2. Table 4.2 shows the processor configurations.

Vulnerability Evaluation on Commodity CPUs

We evaluated 88 *Strong* effective vulnerabilities shown in Table 4.1. Figure 4.4 lists the experimental results when testing the 9 types of processor configurations upon 88 effective vulnerabilities. For each type of processors, a dot showing up in the figure means that the machine is vulnerable to this vulnerability. Apart from the 9 types of tested processors, Figure 4.4 has a row showing if the vulnerability is found in at least one tested processor, i.e., *or* result, and another row showing if the vulnerability is found in all tested processors, i.e., *and* result.

Figure 4.4 shows that 88 effective vulnerabilities are mostly found in all the tested CPUs. Since our new cache three-step simulator considers the ideal case where 66 types of timing observations all have unique results, it outputs all the possible vulnerability types. For commodity processors, a subset of them is shown to be effective. This is due to the actual cache implementation and timing measurement methods, making some of the timing of the 66 types not differentiable, as is shown in histograms of Figure 4.10. Figure 4.4 also demonstrates that different machines are vulnerable to different types of attacks. The *and* result of 9 types of processor configuration experiments have relatively small percentage of

Table 4.3: Percentage of vulnerability cases that are effective for different types of timing observation steps for different machine configurations. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of cases to which the corresponding processor is vulnerable. Machines labeled * do not support hyper-threading in hardware.

Model Name	Local Read	Local Write	Remote Write to Inv.	Flush to Inv.
Intel Xeon E5-1620	137/277	118/277	127/277	129/263
Intel Xeon E5-2667 on-chip	121/277	117/277	80/277	119/263
Intel Xeon E5-2667 inter-chip	127/277	111/277	124/277	72/263
Intel Xeon E5-2690	128/277	101/277	77/277	107/263
Intel Core i5-4570*	82/277	66/277	57/277	63/263
Intel Xeon E5-2686*	87/277	74/277	69/277	80/263
Intel Xeon P-8175	124/277	120/277	75/277	105/263
AMD FX-8150*	68/277	65/277	89/277	65/263
AMD EPYC 7571	125/277	125/277	124/277	114/263
in all CPUs	49/277	34/277	10/277	30/263
at least one CPU	175/277	150/277	162/277	162/263

vulnerabilities to which machines are all vulnerable. We further list the statistical results as CTVS for each machine in Section 4.1.5.

Analysis of Vulnerabilities Found

Figure 4.5 shows the results of benchmarks for all the cases of the 88 vulnerability types. Machines not supporting hyper-threading have much fewer effective cases. Similar to Figure 4.4, the dot means the related processor is vulnerable to the specific case. The gray vertical lines are used to group all the cases per vulnerability (there are thus 88 vertical bars and groupings). We further collect the data in Figure 4.5 and group them with different *Step* 3 types as the timing observation steps in Table 4.3 to compare effects of different operations on processor cache timing attacks.

Local read and local write of timing observation step. Previous attacks normally used *read* access to implement the side-channel attacks, as analyzed in Section 4.1.1. However, write access is shown in Figure 4.5 and Table 4.3 to be an effective method to implement attacks as well. It has generally smaller rate compared with read access to trigger effective vulnerabilities of different cases, especially for tested machine Intel Xeon E5-1620 and E5-2690. For the 44 types of vulnerabilities (#1 - #44) that have access operation as timing observation step, Figure 4.5 demonstrates that there are 38 out of 44 vulnerabilities to which at least one machine is vulnerable when using read as the timing observation step. While using write access as the timing observation step, 34 out of 44 vulnerabilities are vulnerable

Table 4.4: Percentage of vulnerability cases that are effective for the victim (Vic.) and the attacker (Att.) running the same core (time-slicing or hyper-threading), running different cores or within the victim for different machine configurations. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of cases to which the corresponding processor is vulnerable. Machines labeled * do not support hyper-threading.

Model Name	Vic., Att. Same Core		Vic., Att. on Different Cores	Within Victim
	Time-Slicing	Hyper-Threading		
Intel Xeon E5-1620	181/390	174/390	51/90	105/224
Intel Xeon E5-2667 on-chip	156/390	146/390	52/90	83/224
Intel Xeon E5-2667 inter-chip	151/390	146/390	49/90	88/224
Intel Xeon E5-2690	144/390	138/390	46/90	85/224
Intel Core i5-4570*	143/390	0/390	46/90	79/224
Intel Xeon E5-2686*	166/390	0/390	50/90	94/224
Intel Xeon P-8175	148/390	143/390	38/90	95/224
AMD FX-8150*	155/390	0/390	43/90	89/224
AMD EPYC 7571	159/390	171/390	55/90	103/224
in all CPUs	67/390	0/390	18/90	38/224
at least one CPU	223/390	217/390	61/90	148/224

to at least one machine.

Invalidation using cache coherence or flush for timing observation step. According to Table 4.3, the percentage of vulnerabilities to which the machine is vulnerable mainly depends on processor types when comparing different invalidation-related operation as the timing observation step. Among the tested processors, Intel Xeon E5-2667 running inter-chip, AMD FX-8150 and AMD EPYC are more vulnerable to remote write as the timing observation step. Intel Xeon E5-1620, E5-2667 running on-chip, E5-2690, E5-2686, P-8175, and Core i5-4570 are more vulnerable to flush observation step. Overall, for the 44 types of vulnerabilities (#45 - #88) that have invalidation for timing observation, remote write and flush operations both have 38 out of 44 vulnerabilities to which at least one machine is vulnerable.

Running time-slicing or hyper-threading. Besides different kinds of operations, we also collect results in Table 4.4 for running time-slicing and hyper-threading when the victim and the attacker run on the same core (either local or remote core). There are also vulnerabilities for which the victim and the attacker run on different cores, or vulnerabilities only having victim steps. Based on the results, running time-slicing is more vulnerable compared with running hyper-threading for Intel processors. While AMD processor EPYC 7571 shows that running hyper-threading is more vulnerable. Furthermore, hyper-threading provides more choices for the attacker to exploit the corresponding vulnerability.

Table 4.5: Cache Timing Vulnerability Score (CTVS) for each of the tested processors. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of types to which the corresponding processor is vulnerable. Smaller is better. "I" and "E" are internal and external interference vulnerabilities, respectively. "S" and "A" are set-based and address-based vulnerabilities, respectively. "SA" are the ones that are both set-based and address-based.

Model Name	CTVS	*I-A* Vul.	*I-S* Vul.	*I-SA* Vul.	*E-A* Vul.	*E-S* Vul.	*E-SA* Vul.
Intel Xeon E5-1620	73/88	20/20	6/12	12/12	20/20	5/12	10/12
Intel Xeon E5-2667 on-chip	66/88	20/20	3/12	11/12	20/20	3/12	9/12
Intel Xeon E5-2667 inter-chip	64/88	19/20	2/12	12/12	20/20	3/12	8/12
Intel Xeon E5-2690	62/88	20/20	1/12	10/12	20/20	2/12	9/12
Intel Core i5-4570	61/88	20/20	1/12	10/12	20/20	1/12	9/12
Intel Xeon E5-2686	66/88	20/20	2/12	11/12	20/20	3/12	10/12
Intel Xeon P-8175	73/88	20/20	5/12	12/12	20/20	5/12	11/12
AMD FX-8150	50/88	18/20	1/12	7/12	18/20	0/12	6/12
AMD EPYC 7571	62/88	20/20	2/12	10/12	20/20	1/12	9/12
in all CPUs	47/88	17/20	0/12	6/12	18/20	0/12	6/12
at least one CPU	79/88	20/20	9/12	12/12	20/20	7/12	11/12

Take-Aways and Need for Cache Timing Vulnerability Benchmarks

Table 4.5 shows the Cache Timing Vulnerability Score (CTVS) which represents the percentage of the vulnerabilities that are effective for the machine. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of types to which the corresponding processor is vulnerable. For CTVS number, smaller is better. For all the 88 *Strong* type vulnerabilities, AMD FX-8150 has relatively better CTVS compared with Intel machines. Xeon E5-1620 and P-8175 are the most vulnerable ones among Intel processors. Otherwise, the Xeon family and AMD EPYC 7571 are generally similar.

CTVS numbers vary by different machines and type of vulnerabilities. In Table 4.5, CTVS numbers for *in all CPUs* are small, demonstrating that only a few attacks can be effective in all the processors. These numbers are expected to be even smaller if more processors are tested. CTVS numbers for *at lest one CPU* are large, confirming that nearly all of the vulnerabilities derived by the new three-step model are found in real processors.

A type vulnerabilities generally have higher effective rates than S type vulnerabilities. This is because that S type vulnerabilities normally differentiate timing between L1 cache and L2 cache accesses, i.e., accessing or invalidating L1 or L2 data, which are shown in the histograms in Figure 4.10 to be much smaller compared with the difference between L1 cache hit and DRAM hit, for example. Especially, the timing difference between remote

write to invalidate dirty L1 data and L2 data is almost non-differentiable, resulting in that related vulnerabilities (especially #25 - #28) are found to be not effective in all tested processors (shown in Figure 4.4). A type vulnerabilities generally rely on timing differences between L1 cache hit and DRAM hit, or L1 cache hit and remote L1 cache hit; histograms in Figure 4.10 demonstrate that these access types have large timing differences, making these vulnerabilities much more effective. SA type vulnerabilities generally leverage the timing differences between clean L1 data invalidation and dirty L1 data invalidation or between local access of remote clean L1 data and remote dirty L1 data; histograms in Figure 4.10 again show large timing differences for these, and related vulnerabilities are found to be very effective by CTVS. Meanwhile, for I and E type vulnerabilities, they do not have an explicit distinction of CTVS numbers for the tested processors.

Use CTVS to build custom defenses. CTVS has shown that different processors are vulnerable to different attacks. Consequently, customized software or hardware defenses can be deployed for each processor based on the CTVS score, rather than defending vulnerabilities not present in the specific processor's caches. For software defenses, the access patterns from the benchmarks could be used as a reference for scanning software to find if it has similar patterns, e.g., to find malicious software that has such attack patterns.

Understand limits of existing defenses using three-step model. Further, CTVS and our three-step model have shown new attack types which are unknown before, and thus, not considered by defenses based on monitoring performance counters, e.g., study [82]. This points to the requirement of using new or different performance counter types in works that use active monitoring, for example.

Understand micro-architecture using CTVS. The vulnerability score can also be used to help understand the implementation of different processors especially the micro-architectures. For example, according to Figure 4.4, vulnerability #78 $V_u \rightsquigarrow V_d \rightsquigarrow V_u^{inv}$ and #79 $V_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ fully show up on Intel E5-1620 while do not show up on Intel E5-2690. As shown in Figure 4.10, flushing clean L1 data (L1cl) to DRAM and flushing clean L2 data (L2cl) to DRAM have large timing differences for Intel E5-1620 (1036 vs. 985 average cycles shown in Figure 4.10(e)), but are non-differentiable for Intel E5-2690 (872 vs. 879 average cycles shown in Figure 4.10(f)). With the smaller difference, it is not possible

to distinguish the timing with high confidence and corresponding vulnerabilities are highly likely unexploitable on this processor.

Diving deeper, the reason for the timing variation may due to the different clock speed of Intel E5-1620 and Intel E5-2690 (3.6GHz vs. 2.9GHz), where faster clock speed will make long memory-related operations more differentiable, even if the absolute timing differences are the same. Besides that, Intel E5-1620 does not support Flex Memory Access, which improves memory access efficiency. Intel E5-2690 supports it, making two operations less differentiable on timing.

4.2 Cache Timing Vulnerabilities and Arm Evaluation

This work shows for the first time, a systematic, large-scale analysis of Arm devices and the detailed results of attacks the processors are vulnerable to. Compared to x86, Arm uses different architectures, microarchitectural implementations, cache replacement policies, etc., which affects how attacks can be launched, and how security testing for the vulnerabilities should be done.

4.2.1 Threat Model and Assumptions

We assume that there is a victim that has secret data which the attacker tries to extract through timing of memory-related operations. The victim performs some secret-dependent memory accesses (V_u) and the goal for the attacker is to determine a particular memory address (or cache index) accessed by the victim. The attacker is assumed to have some additional information, e.g., he or she knows the algorithm used by the victim, to correlate the memory address or index to values of secret data.

In addition to regular reads, writes, and flush operations, we assume that the attacker can make use of cache coherence protocol to invalidate other core's data, by triggering read or write operations on the *remote* core as one of the steps of the that attack.

A negative result of a benchmark means there is likely no such timing channel in the cache or the channel is too noisy to be observable. Meanwhile, a positive result may be due to structures other than cache, such as prefetchers, Miss Status Handling Registers

(MSHRs), load and store buffers between processor and caches, or line fill buffers between cache levels. Our benchmarks focuses on L1 data caches, but we consider that timing results could be due to all the different structures. Detailed benchmarks for these structures or other levels of caches are left for future work.

4.2.2 Arm Security Benchmarks

In this section, we present the first set of benchmarks which is used to evaluate L1 cache timing vulnerabilities of Arm processors. To implement the security benchmarks on Arm, as listed below, we developed solutions to key challenges accordingly.

Heterogeneous CPU Architectures

Arm processors implement the *big.LITTLE* architecture with *big* and *little* processor cores having different cache sizes. This presents a new challenge, as the architecture is fundamentally different from multi-core systems where all cores have identical cache sizes and configurations. This was not considered in our previous work [9] which only dealt with x86, nor in previous studies [83, 84, 85, 86] which only tested attacks on one core type.

The *local* core is the one wherein is located the target cache line that the attacker wants to learn. Meanwhile, the *remote* core is a different core where the target cache line is not located, but which could affect the *local* core and its caches, e.g., via cache coherence protocol. Thus, both cross-core and cross-CPU vulnerabilities are considered in our work by testing the victim and attacker operations on different combinations of *local* and *remote* cores. Especially, with different *big* and *little* processor cores, a *local* or *remote* core can be either of *big* or *little* core type, resulting in four combinations.

Because we consider different core types, unlike prior work, and caches are not even between the *big* and *little* cores, we define how to correctly specify the cache configurations for the benchmarks when running the tests:

- If the first two steps of the three-step model describing a particular vulnerability both occur in the remote core, use the remote core's cache configuration.
- In all other cases, use the local core's cache configuration.

In the three-step model, when testing for vulnerabilities, main interference (leading to potential timing differences) occurs within the first two steps, while the final, third step is used for the timing observation used to determine if there is possible attack or not. Therefore, the above method of choosing the cache configuration focuses on where the main interference is occurring in the three steps.

Random Replacement Policy in Arm

Modern Arm cores use the random replacement policy in the L1 cache [83]. This policy is significantly different from the Least Recently Used (LRU) replacement policy, and poses fundamental challenges for eviction and probing steps in 48 out of 88 vulnerability types.

In particular, this makes the set-only-based vulnerabilities (*SO*-Type) harder to implement. The reason is that occupying a cache set in caches using a random replacement policy is not as easy as in caches using LRU or similar policies, where accessing a certain number of ways (denoted as `cache_associativity_num`) of cache lines in a cache set is able to evict all data in the set. In caches using the random replacement policy, the *cache set thrashing problem* [87], referring to self-evictions within the eviction set, which affects accessing all the ways of the cache set in eviction-based vulnerabilities. To avoid this problem, we use a smaller set size to avoid set thrashing in our benchmarks. We set the eviction set size to `cache_associativity_num-1` and then repeat each step's memory operations 10 times. Using this technique, we are able to reduce set thrashing significantly given the random replacement policy. However, in this case, exactly one way will not be occupied after the repeated memory operations. This will cause victim's access in one out of `cache_associativity_num` ways to be not detectable, but this is acceptable as vulnerabilities can still be detected as we show in our evaluation.

Measuring and Differentiating Timing

For benchmarking Arm cache timing vulnerabilities, this work is the first to utilize statistical tests – Welch's t-test [79] – to differentiate distributions of timings to check if vulnerabilities can result in attacks. The *pvalue* is the threshold used to judge the effectiveness of the vulnerabilities. Based on our evaluation, we select 0.00049 for the *pvalue* in our tests,

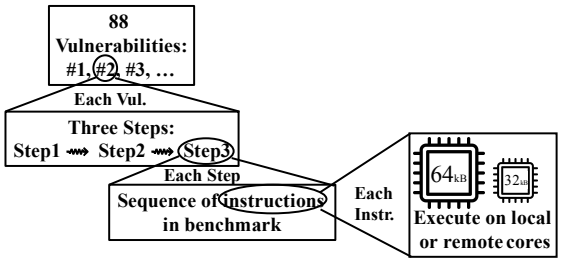


Figure 4.6: Relationship of the 88 vulnerabilities, each of which is described using three steps from the three-step model. The steps are further translated into sets of assembly instructions for the benchmarks, and the code can be run on either *big* or *small* cores in the tested systems.

improved from our previous work on x86 [9], and use this to determine if different timing distributions are distinguishable. We chose Welch's t-test since it is generally used in attack evaluations [88, 89, 90]. There is also Kolmogorov-Smirnov's two-sample test [91] that can be used to differentiate distributions. However, in the case of cache timing side channel, there is only two possible timing observations (i.e., hit or miss), t-test is sensitive to the mean of distributions, and thus fit in this case.

The statistical tests are used to differentiate timings of memory related operations. However, cycle-accurate timings are not accessible without root access on Arm, while x86 provides accurate assembly instructions to record timing (e.g., `rdtsc`). Consequently, we developed code that can get reliable timing measurements in user-level applications using the `clock_gettime()` system call. We experimented with other different performance counters and thread timers, but these proved not to be applicable or accurate enough for our benchmarks.

When performing timing measurements, in our experience, Arm devices further exhibit a lot of system noise when running the tests on real devices in the cloud-based device farms, possibly due to OS activity, or other background services. Therefore, we set the benchmarks to run more than $30,000$ repetitions for each benchmark for each device to average out the noise. Further, when running each step operated by either the victim or the attacker, we isolate the core to avoid influence of other application processes from user-level applications.

Algorithm 3 Assembly code sequence for read or write accesses.

```
1: asm ___volatile___ (
2: “DSB SY n”
3: “ISB n”
4: “LDR/STR %0, [%1] n”
5: “DSB SY n”
6: “ISB n”
7: : “=r” (destination)
8: : “r” (array[i]));
```

Algorithm 4 Assembly code sequence for flush.

```
1: asm ___volatile___ (
2: “DSB ISH n”
3: “ISB n”
4: “DC CIVAC, %0 n”
5: “DSB ISH n”
6: “ISB n”
7: : : “r” (array[i]));
```

Summary of the Benchmark Structure

Following the above features, we developed benchmarks for all 88 vulnerabilities. As shown in Figure 4.6, there are three steps for each vulnerability, and each step is realized by a sequence of instructions. The instruction sequences from each step can execute on local or remote cores. When performing the steps, there are two possible cases for the victim's or attacker's memory related operation: read or write access for a memory access operation; and flush or write in the remote core for an invalidation-related operation. Thus, for each vulnerability, there are in total of $2^3 = 8$ types considering different cases of each step's operation. Further, if a vulnerability being tested has both the victim and the attacker running on one core, these two parties can run either time-slicing or multi-threading. Consequently, the 8 cases are doubled to account for both time-slicing and multi-threading execution. Thus, for each vulnerability being tested, there are correspondingly 8-16 cases depending on the specific vulnerability. Each vulnerability is realized as a single benchmark program. In total there are 1094 benchmarks for all 88 types of vulnerabilities.

The 1094 benchmarks are automatically generated. The basic code sequences, e.g., Alg. 3 and 4, are composed into programs, with one program for each benchmark. Additional instructions are used in the benchmarks to pin execution of the code to different processor cores when testing different configurations. The resulting 1094 programs are compiled and executed on the devices under test as detailed in the next section.

4.2.3 Cloud-Based Framework

In this section, we report on the first cloud-based platform for testing cache channels on Arm devices. Our prior work only considered x86 [9] with several processors manually set to test, and work by others only manually tested only few Arm devices [83, 84, 85, 86].

Android Device Testbeds

We build our evaluation framework using testing platforms for mobile devices, namely the Visual Studio App Center [92], the Amazon AWS Device Farm [93], and the Firebase Test Lab [94]. We developed a framework which allows us to run custom binary benchmarks and retrieve the results in an automated manner.

In these cloud deployments, it is not possible to execute benchmark files through a remote shell and download the results. Instead, the entire functionality must be implemented as a user-level native Android application. Consequently, the benchmark executables are inserted into the application package (APK) of a custom Android application we developed. Figure 4.7 illustrates the resulting test setup, which will be open-sourced.

Extracting Cache Configurations

To build the benchmark, cache and CPU configuration information are needed. The configuration can be automatically identified by reading the corresponding system information located at */sys/devices/system/cpu/cpux/* (where x stands for the CPU core number) on each tested device. However, depending on the SELinux policies applied by the vendor and Android version, access to these files is restricted on some devices [95]. For these device models, we manually identify and verify their cache configurations from public resources. Finally, we store both automatically- and manually-extracted cache configuration parameters in a single database, and include this database into the APK, so that it can be used when running the benchmarks.

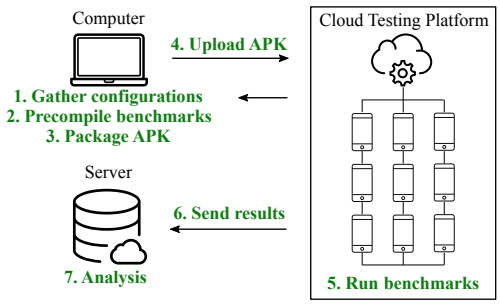


Figure 4.7: Overview of the evaluation framework using the cloud-based testing platforms for Android mobile devices.

Packaging Security Benchmarks

Starting from Android 9, the operating system does not allow to execute files from an arbitrary writeable location on the filesystem [96]. Instead, only native library dependencies within an Android application can be executed. Consequently, we pre-compile and place the benchmark files in the resource subfolder of the APK package which contains native libraries (*src/main/resources/lib/arm64-v8a*), as the OS grants read-and-execute permissions for all binary files in this subfolder.

Running Benchmarks

We give an overview of our evaluation framework in Figure 4.7. Once the cache configuration is extracted (step 1), the corresponding benchmarks are precompiled (step 2) and packaged (step 3), we upload the application package to the cloud testing platforms (step 4). The implemented application does not require any user interaction. Instead, it contains an instrumented unit test which automates the execution of benchmarks. The tests can be run simultaneously on multiple devices (step 5). The process of uploading and running the application is automated using the APIs provided by the cloud platform provider.

On each device, the application first identifies the device model by accessing the *Build.MODEL* property. This information is used to look up the corresponding cache configuration parameters in the database. Afterwards, the application executes the precompiled benchmarks one by one, using the corresponding parameters. In order to automatically retrieve the results of benchmarks from multiple devices, we implement an HTTP server which can receive POST requests from Android applications. Each request contains the

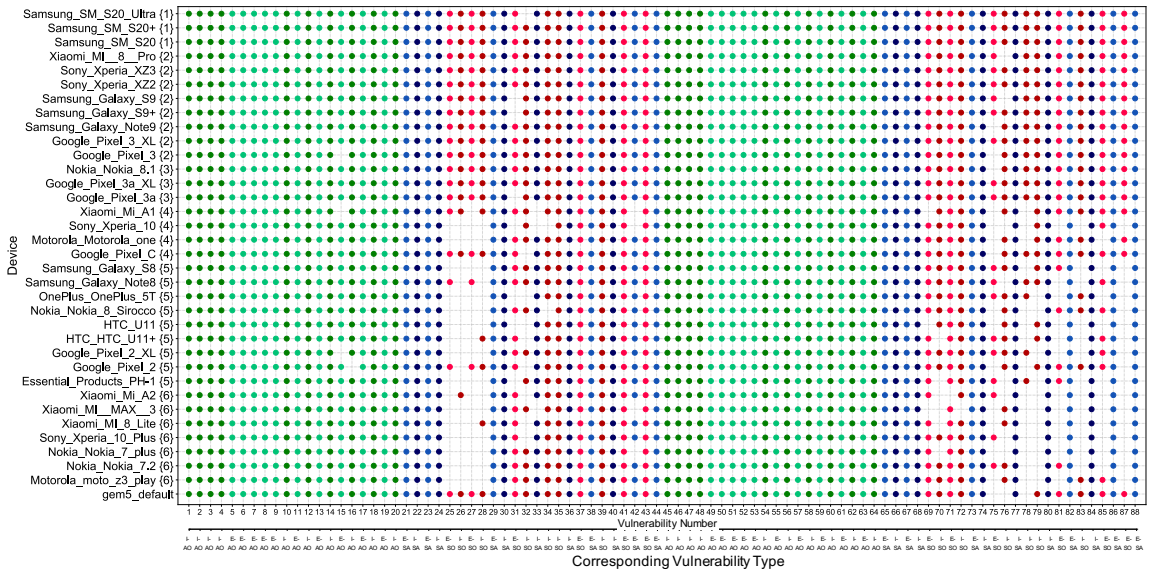


Figure 4.8: Evaluation of the 88 types of vulnerabilities on different Arm devices. We further tested other Arm cores, including an X-Gene 2 core and a Neoverse core to test Arm processors on servers. The results generally have similar patterns as the mobile devices so we show only results for the mobile devices from the cloud-based testbeds. A solid dot means the corresponding processor is found to be vulnerable to the vulnerability type. The "*I-SO*" (colored by dark red) and "*E-SO*" (colored by light red) are internal-interference set-only-based and external-interference set-only-based vulnerabilities, respectively. The "*I-AO*" (colored by dark red) and "*E-AO*" (colored by light red) are internal-interference address-only-based and external-interference address-only-based vulnerabilities, respectively. The "*I-SA*" (colored by dark red) and "*E-SA*" (colored by light red) are internal-interference set-or-address-based and external-interference set-or-address-based vulnerabilities, respectively. The devices are grouped according to their core types. Each device's core is labeled by a number shown after the device name, with corresponding cores shown in Table 4.6. The order is from the most vulnerable core to least vulnerable among the cores. The last line shows `gem5` testing results of default `gem5`, to show that `gem5` simulation gives similar results to real devices.

results in textual or binary format. As the execution time of the whole set of benchmarks on a device can take several hours, the application periodically sends the intermediate results to the server. In this way, we can precisely monitor the current state of the execution on each device. Finally, the results are collected from the server (step 6) for further analysis (step 7).

4.2.4 Arm Benchmark Evaluation

We tested a number of different devices. The corresponding processor core types are shown in Table 4.6 – note that some devices use the same processor or SoC configuration, as shown in in Table 4.6. The results of the tests are shown in Figure 4.8, which shows the vulnerabilities that can possibly be exploited on the device, based on sufficient timing differences in the memory operations corresponding to each three-step attack. Figure 4.8

Table 4.6: CPUs and SoC types found in the evaluated devices. The *Core Name* (with corresponding number used in Figure 4.8), *Core Freq.*, and *L1 Cache Config.* columns show the processor core names, their frequency ranges, and typical cache configurations. The *Vul. Num.* column shows the average number (out of 88) of vulnerabilities that show up during tests; smaller value is better.

Core Name	Core Freq.	L1 Cache Config.	SoC Name	Vul. Num.
Kryo 585[1] Gold/ Silver	2.42-2.84/ 1.8	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 865	88
Kryo 385[2] Gold/ Silver	2.5-2.8/ 1.6-1.7	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 845	87
Kryo 360[3] Gold/ Silver	2.0-2.2/ 1.7	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 670/ 710	87
Cortex A53[4]	1.9-2.2	32 KB 4-way	Nvidia Tegra X1/ Qualcomm Snapdragon 625/ 630	81
Kryo 280[5] Gold/ Silver	2.35-2.5/ 1.8-1.9	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 835	79
Kryo 260[6] Gold/ Silver	1.8-2.2/ 1.6-1.8	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 636/ 660	76

consists of 88 columns, each corresponding to one of the three-step vulnerabilities. The vulnerabilities are colored based on the different types.

In addition to smartphones, we further tested other Arm cores, leveraging Amazon EC2 [97] with an X-Gene 2 core and Chameleon cloud [98] with a Neoverse core to test Arm processors on servers. Arm server chip results generally have similar patterns as the mobile devices. Therefore, in this work, we show only results for the mobile devices from the cloud-based testbeds.

Microarchitectures' Impacts on the Vulnerabilities

Below we list some of the observations gained from our evaluation. Only through the extensive benchmarking of caches on a large set of devices, can such insights be discovered.

Store Buffer. The STB (STore Buffer) is used during write accesses to hold store operations. This structure makes clean and dirty L1 data access timing easier to be distinguished. For example, $I - SA$-Type vulnerability #33 differentiates timing between reads of dirty L1 data and reads of clean L1 data, or between writes of dirty L1 data and writes of clean L1 data, which is a typical vulnerability that allows STB to make it more effective. From the evaluation results, Kryo 360 Gold/Silver cores are more susceptible to vulnerabilities such as #33, compared to Cortex A53 core, which confirms the fact that the STB is presented in Kryo 360 Gold/Silver cores but not in Cortex A53 core, based on reference manuals.

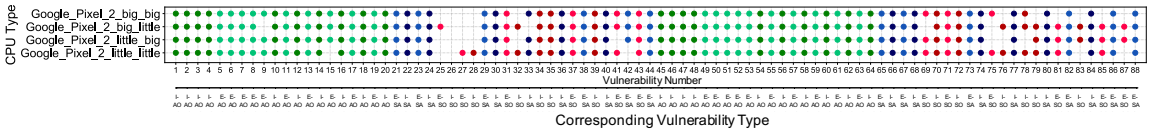


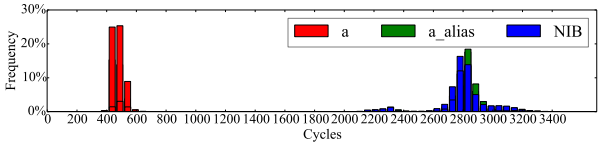
Figure 4.9: Evaluation of the 88 types of vulnerabilities on different cores of Google Pixel 2. "big_big" means running both local and remote core on big cores, "big_little" means running local core on the big core, remote core on the little core. Same naming is applied to "little_big" and "little_little". Dot coloring is the same as in Figure 4.8.

Snoop Control Unit. The Snoop Control Unit (SCU) contains buffers that can handle direct cache-to-cache transfers between cores without having to read or write any data to the lower cache by maintaining a set of duplicate tags that permit each coherent data request to be checked against the contents of the other caches in the cluster. With the SCU, when comparing the timing between remote writes to invalidate local L1 data and remote writes to invalidate local L2 data, the SCU will accelerate the coherence operations. This makes the different cache coherence influence non-differentiable in timing on the cores that have the SCU.

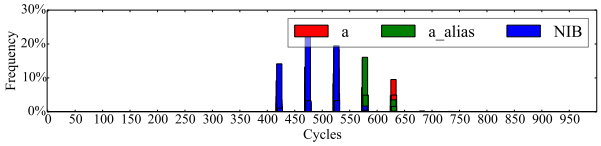
For example, $I-SO$-Type vulnerabilities #78-#79 mainly use timing differences between flushing of L1 data and flushing of L2 data, or between remote writes to invalidate local L1 data and remote writes to invalidate local L2 data. From the evaluation results, vulnerabilities #78-#79 occur much less frequently on Kryo 280 Gold/Silver cores and Cortex A53 cores compared to Kryo 360 and Kryo 385 Gold/Silver cores. This supports the observation that the Kryo 280 Gold/Silver cores and Cortex A53 cores have a Snoop Control Unit (SCU), which helps prevent these types of vulnerabilities, while Kryo 360 and Kryo 385 Gold/Silver cores do not have it.

Transient Memory Region. Transient Memory Region allows for setting a memory region as transient. Data from this region, when brought into L1 cache, will be marked as transient. As result, during eviction, if this cache line is clean, it will be marked as invalid instead of being allocated in the L2 cache.

Although this may help avoid polluting the cache with unnecessary data, internal and external SO-Type vulnerabilities #33-#44 that we are able to differentiate between L1 and L2 cache hits can now differentiate between an L1 cache hit and a data access from DRAM. This makes this type of vulnerability more effective on cores that support this feature, which



(a) Sample histogram of *AO*-Type vulnerability



(b) Sample histogram of *SO*-Type vulnerability

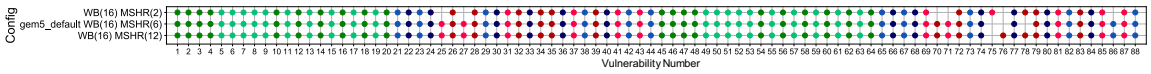
Figure 4.10: Samples of different types of vulnerabilities' timing histograms for different candidate values for V_u.

are Kryo 360/385 Gold/Silver cores, compared to other cores, such as Cortex A53.

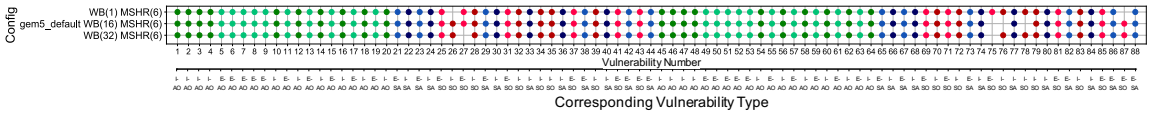
Heterogeneous Caches' Impact on Vulnerabilities

We also evaluated how Arm's *big.LITTLE* architecture impacts the attacks, where we set *local* and *remote* core to be either *big* or *little* processor core. In Figure 4.9, we present evaluation results for one example device, Google Pixel 2. A similar pattern was observed for all other tested devices.

SO-Type and *SA*-Type vulnerabilities which differentiate L1 and L2 cache hit timings (#41-#44) are mostly vulnerable to the case when the *local* core uses the *big* core. This is mainly because the bigger cache (e.g., 64K 16-way vs. 32K 4-way) of the *big* core results in larger timing differences for the vulnerabilities that require priming each cache set, reducing the proportion of system noise at the same time. *SO*-Type and *SA*-Type vulnerabilities which differentiate writing to remote dirty L1 and L2 cache data (#73-#76) are successful when *local* and *remote* core both use the *little* core. Dirty data are usually not stored in the cache line but stored in other locations such as write buffer. Write buffer is possibly processed in an out-of-order way. Therefore, fewer number of writes due to fewer number of ways in *little* core are more likely to have relatively differentiable timing. *SO*-Type and *SA*-Type vulnerabilities which differentiate writing remote L1 and remote L2 cache data (#77-#88) are mostly successful when *local* and *remote* cores use different core types (*big* or *little*). This is due to the fact that *big* and *little* cores are often in different quad-core clusters in the SoC, where coherence time across quad-core cluster results in higher timing



(a) Benchmark `gem5` simulation results for different MSHR sizes.



(b) Benchmark `gem5` simulation results for different write buffer sizes.

Figure 4.11: Evaluation of 88 types of vulnerabilities on different number of write buffer (WB) and Miss Status Handling Register (MSHR) sizes. A solid dot means the corresponding processor is found to be vulnerable to the vulnerability type. The "*SO*" (colored red) and "*AO*" (colored green) are set-only-based and address-only-based vulnerabilities, respectively. "*SA*" (colored blue) are the ones that are set-or-address-based. The "*E*" (colored in lighter color) and "*I*" (colored in darker color) are internal- and external-interference vulnerabilities, respectively.

differences when accessing data located in the remote cluster.

Core Frequency's Impact on Vulnerabilities

High clock frequency tends to make long memory operations more differentiable, and will make timing attacks easier to exploit the difference. From the evaluation results, we found that devices with higher clock frequency will likely have more effective timing-channel vulnerabilities presented.

This is especially visible in *SO*-Type vulnerabilities, most of which differentiate between L1 and L2 cache hits, which have a relatively small cycle difference, e.g., less than 10 cycles. However, if the core's frequency increases, the timing difference is also increased, which makes cycle distributions more differentiable and an attack possibly easier to execute.

Influence of Write Buffer and MSHR Sizes

We design our benchmarks so they can also be used in simulation. We use the Arm *big.LITTLE* configuration to run the benchmarks in Full System (FS) mode or Syscall Emulation (SE) mode on `gem5`. The simulator is configured to use the Exynos [99] configuration to model real Android devices and uses the O3CPU model with a 5-stage pipeline. The last line of Figure 4.8 shows the benchmark results when using the default configuration on the `gem5` simulator. Overall, we find that baseline `gem5` results have good correspondence with real CPUs in terms of the cache timing vulnerabilities.

Next, we evaluate different configurations of the Miss Status Holding Register (MSHR) and the write buffer (WB), both tested on `gem5`. Results are shown in Figure 4.11: A larger MSHR size leads to more vulnerabilities to be observed. MSHR is a hardware structure for tracking outstanding misses. Larger MSHR sizes lead to more outstanding misses that can be handled, which may stabilize the memory access timings and give more consistent results.

Changing the size of WB does not have an explicit influence on the vulnerability results. WB stores the write request, which frees the cache to service read requests while the write is taking place. It is especially useful for very slow main memory, where subsequent reads are able to proceed without waiting. We use the "SimpleMemory" option of `gem5`, which is relatively simple compared with the implementation of real devices and may not have the same slow memory timing in this case. As the result shows, bigger WB may improve performance and can be added without degrading security, while bigger MSHR may improve performance but at some cost to security.

Patterns in Vulnerability Types

It can clearly be observed from the colored dots in Figure 4.8 that *AO*-Type vulnerabilities are observable in almost all devices and in the simulation, because these types of vulnerabilities, e.g., differentiate L1 cache hits and DRAM hits, which have large timing differences. Such timing distribution results can be observed in Figure 4.10a. *SA*-Type vulnerabilities also occur relatively often, but are much more unstable compared with *AO*-Type vulnerabilities, which shows that different devices have large but quite variable timing differences among different memory operations, e.g., between clean abd dirty L1 data invalidation or between local access of remote clean and dirty L1 data. *SO*-Type vulnerabilities are least effective. This is because the timing differences between the observations such as L1 and L2 cache hits are so small that they are sometimes indistinguishable due to system noise. For example, timing distribution evaluation result shown in Figure 4.10b have small timing difference.

I-Type and *E*-Type vulnerabilities do not show explicit evaluation differences. In this case, another take-away message is that protecting only the external-interference vulnerabilities is not enough at all. Internal-interference vulnerabilities can be as effective as the external-interference vulnerabilities for attacks.

Estimating the Real Attack Difficulty

To estimate the real attack difficulty, we can leverage the distance and likelihood (using p-value) of different timing measurement distributions. As is shown in Figure 4.10 in Section 4.2.4, *AO*-Type or *SA*-Type vulnerabilities are easier to exploit since they depend on timing differences of L1 cache hits vs. DRAM accesses; meanwhile *SO*-Type vulnerabilities are more difficult to exploit, since they depend on the timing differences between L1 and L2 cache hits, which are much smaller compared to the former.

Further, our benchmarks show the overall attack surface. If a motivated attacker only needs to use one attack to derive sensitive information, he or she will likely start with *AO*-Type or *SA*-Type vulnerabilities. However, the bigger the attack surface is, the more options he or she has, and if there are defenses for *AO*-Type or *SA*-Type types of vulnerabilities, attackers could still leverage *SO*-Type vulnerabilities. The goal of this work is to show the whole attack surface on Arm devices, including vulnerabilities and attack types that are not previously presented in the literature. Which attack could be used in practice depends on the attacker's motivation and resources, but thanks to this work, the overall attack surface is better understood.

Results Compared with Other Work

For our benchmark results shown in Figure 4.8, strategies exploited by existing Arm attacks – Evict+ Time (#41-#42 in the Figure), Prime+Probe (#43-#44 in the Figure), Flush+Reload[1] (#5-#8 in the Figure), and Flush+Flush (#47-#50 in the Figure) – all indeed show up as effective vulnerabilities for mobile devices that were tested. This confirms that our benchmarks can cover existing work. Note that the 5 types of vulnerabilities explored by prior work, e.g., the Evict+ Time, etc., can be realized using more than one vulnerability from the 88 types, thus prior work covers 12 types, leaving 76 types not considered, for the total of 88 vulnerabilities that are possible.

1. Our Flush+Reload benchmarks test for a stronger variant of the Evict+Reload vulnerability shown in [84, 83].

Summary of Vulnerability Trends

To summarize, the patterns of the vulnerabilities uncovered thanks to the systematic benchmarking on the devices are:

- Microarchitectural features: performance increasing features such as the store buffer can degrade security, while features such as the snoop control unit can be helpful, indicating that security and performance are not always at odds with each other, and some features can help both.
- Heterogeneous cache size: larger coherence timing for accesses involving cores in different clusters, compared to within same cluster, may lead to more vulnerabilities being effective.
- Core frequency: larger core frequency generally correlates with more vulnerabilities.
- WB and MSHR sizes: WB size does not impact security, while larger MSHR may allow more vulnerabilities to be effective.
- Vulnerability type effectiveness: relations of number of effective vulnerabilities showed are: AO-Type $>$ SA-Type $>$ SO-Type; meanwhile, I-Type and E-Type vulnerabilities are similarly effective on the tested devices.
- Tested device results: relations of number of effective vulnerabilities showed are: Kryo 585 $>$ Kryo 385 $\approx$ Kryo 360 $>$ Core A53 $>$ Kryo 280 $>$ Kryo 260.

4.2.5 Sensitivity Testing of Benchmarks

To understand how the benchmarks are affected by possible misconfigurations, we performed a number of sensitivity tests. In addition to evaluating how the benchmarks behave, the sensitivity study allows us to understand how knowledge (or lack of knowledge) of the correct cache configuration affects the attacker's ability to attack the system.

Analysis of Sensitivity Testing

The most important cache parameters for sensitivity tests are: *associativity*, *line size*, and *total cache size*. We use $asso_d$, $line_d$, and tot_d to respectively denote the value of the parameters of the actual target device. Meanwhile, $asso_b$, $line_b$, and tot_b denote the cache

parameters used by the benchmarks. The parameters used in the tests are varied and are different from the actual, correct parameters to test the sensitivity of the results to misconfiguration. As we show, setting the configuration incorrectly in the benchmarks changes the mapping of the addresses used by the benchmarks, and influences the number of vulnerabilities judged to be effective on a device.

We implement the sensitivity tests in the following way. A large array is maintained to locate three different candidates of the secret value (a, a_{alias}, or NIB). We consider two addresses that only differ in the low $log_2(line_b)$ bits to belong to the same cache line, and two addresses that are a distance of $C \times tot_b/asso_b$ (C is a integer) apart to map to the same cache set. For each step, we access $asso_b$ number of addresses for each cache set to occupy or cause collision in the whole cache set. To increase the signal to noise ratio in our measurements, rep cache sets are accessed in each of the steps of a benchmark (in our setting this number is 8).

When $asso_b$, $line_b$, or tot_b deviates from $asso_d$, $line_d$, or tot_d, the following situations could happen: 1 the number of addresses being accessed in one cache set is less than $asso_d$, so interferences that should happen are not observed; 2 the addresses that should map to a target cache set actually map to several cache sets, and contention in the target cache set might not happen or will become contention in several sets; and 3 the addresses that should map to different cache sets actually map to the same cache set, introducing noise to the channel. We show later that the total number of attacks judged to be effective is less when an incorrect configuration is used – however, there are still attacks that are effectively independent of the configuration setting.

In the following, we denote one L1 cache hit timing as t_{L1} and one L2 cache hit timing as t_{L2}. When the configuration of the benchmark is correct, if the secret maps to the same cache set as some known address that was accessed, t_{L2} will be observed, while if they are not mapped, t_{L1} will be observed. In this case, timing observations for mapped and unmapped cases are $asso_d \times t_{L2}$ and $asso_d \times t_{L1}$.

Cache Associativity. Associativity usually influences the number of accesses that map to a target cache set. We distinguish two cases:

- $asso_b < asso_d$: In this case, due to smaller number of ways accessed in each step, fewer evictions will occur (situation 1). If a data address maps to the same set as the secret data, timing observation will be $n \times t_{L2} + (asso_d - n) \times t_{L1}$ instead of $asso_d \times t_{L2}$. Here, $0 < n < asso_b$. Due to the random replacement policy, only n (not all $asso_b$) cache lines will be evicted. This will make the timing less distinguishable compared with the unmapped case, in which timing should be equal to $asso_d \times t_{L1}$.
- $asso_b > asso_d$: When $tot_b = tot_d$, this setting will lead to accesses that should map to one cache set actually mapping to several cache sets (situation 2). This will result in measuring more than rep of cache sets for one step, which possibly introduces more noise in the observation.

Cache Line Size. Line size generally influences which cache set is chosen within an attack (benchmark) step. Again, we distinguish two cases:

- $line_b < line_d$: In this setting, the accesses that should map to different cache sets in the benchmark actually map to the same cache set (situation 3). This will lead to the result that the benchmark measures less than rep cache sets effectively, causing a reduced signal to noise ratio. For example, when choosing $line_b = line_d/2$, then two addresses that differ in $line_b$ will map to the same cache line instead of different lines in difference sets. This results in having more L1 cache hits, from $asso_d \times t_{L2}$ to $asso_d/2 \times t_{L2} + asso_d/2 \times t_{L1}$, which makes it less distinguishable compared with unmapped case where timing is $asso_d \times t_{L1}$.
- $line_b > line_d$: In this setting, since we always access the first 64 bits in a cache line, the addresses that should map to the same sets in the benchmark (with the incorrect configuration) still map to the same set (if the correct configuration was used). However, when $line_b$ is larger or equal to $cache\_set/rep$[2] times of $line_d$, the address for NIB in the benchmark will wrap back and map to the same cache set as a and a_{alias} (situation 3), causing a false negative result.

Total Cache Size. Cache size mainly influences the data addresses accessed in each

2. In the example of Section 4.2.5, this number is equal to $128/8 = 16$.

step of an attack (benchmark).

- $tot_b < tot_d$: In this setting, the accesses that should map to one cache set in the benchmark actually map to several cache sets (situation 2), because $tot_b/asso_d < tot_d/asso_d$. This further causes the number of data accesses in each set to be less than the number of ways being accessed in the target cache set, i.e., $asso_d$ (situation 1). Thus, for the mapped case, it is equivalent to observing $n \times t_{L2}$ timing instead of $asso_d \times t_{L2}$ timing for this cache set, where $0 < n < tot_b/tot_d \times asso_d$ due to the random replacement policy. This could decrease the signal to noise ratio.
- $tot_b > tot_d$: Let $C' = tot_b/tot_d$. In most cases, C' will be an integer, assuming a cache size (both tot_b and tot_d) of 2^N bytes. In this setting, the cache addresses that are different by $tot_b/asso_d = C' \times tot_d/asso_d$ in the benchmark, will still map to a different cache set in target device[3]. Further, if C' is too large, this will cause unexpected system noise if prefetching, copy-on-write, etc., functions are enabled in the device.

Analysis by Vulnerabilities Types. For *AO*-Type and *SA*-Type Vulnerabilities, the timing observation for $V_u = a$ is different from $V_u = a_{alias}$ or $V_u = NIB$. In these types of vulnerabilities, the attack does not rely on the interference between different cache lines in a cache set. How the addresses map to the cache set does not affect the result, and the cache configurations will not influence the effectiveness of the vulnerabilities. Also, these types usually rely on relatively larger timing differences, so the signal to noise ratio is large.

SO-Type vulnerabilities usually derive the V_u information by observing evictions of the originally accessed data in a prior attack step. For *SO*-Type vulnerabilities, we need to access all the $asso_d$ ways to prime the whole cache set in order to observe the timing difference, therefore, *SO*-Type vulnerabilities will actually be influenced by the setting of parameters including *associativity*, *line size*, and *total cache size*.

Summary. Based on the above, we make three observations about the configurations' impact on the benchmarks and the corresponding attacks and how easy they are to perform:

3. When C' is not an integer, e.g., $C' = 1.5$, then the address to set mapping will be different than the case when $tot_b = tot_d$, which is equivalent to having addresses mapped to other cache set, resulting in fewer number of addresses mapped to the target cache set (situation 1).

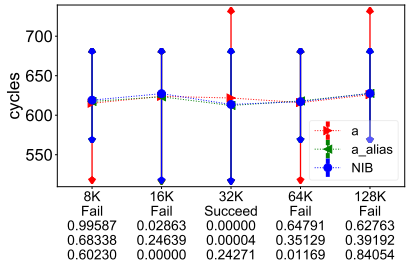


Figure 4.12: Timing histogram of a vulnerability case when changing the cache size. The error bar shows the range of timing distribution and the dot shows the average timing cycles. "Succeed" under the configuration means the vulnerability is effective while "Fail" means not. Three values under "Succeed" or "Fail" are the *pvalue* for each two timing distributions out of three. If it is smaller than 0.00049, we judge the two timing distributions to be differentiable, otherwise not.

1. Attackers can still attack the system even when they are uncertain about the cache configuration. This is especially true for *AO*-Type or *SA*-Type attacks since they are not impacted much by the (mis) configuration.
2. Most of the differences are due to *SO*-Type attacks, which do not work well when incorrect setting is selected.
3. Setting correct configurations causes more vulnerabilities to be judged effective for a device. Incorrect settings can cause an underestimation of the total number of vulnerabilities presented.

Evaluation of Sensitivity Testing

We tested a wide range of devices and found similar trends among the results. Here we give results for one example phone, Google Pixel 2, to show how the sensitivity test is implemented and evaluated.

The L1 cache configuration of small core of Google Pixel 2 is 32KB, 4-way set-associative with line size to be 64B. We test this configuration by changing one of the three parameters (associativity, line size or cache size), while keeping the other two the same to avoid interference between different parameters. The different configuration values we choose in our evaluation are listed in Table 4.7.

In the example test case shown in Figure 4.12, timing distribution differences between three candidates are larger for the correct configuration, compared to the wrong configurations. The vulnerability is effective under the correct configuration while it fails for the incorrect

Table 4.7: Configuration test results for cache associativity, line size and cache size of Google Pixel 2. Black bold numbers show the largest effective number of vulnerabilities for each category. Middle column shows the correct configuration values for this device, other columns show smaller (left side) and bigger (left side) values that were tested for each parameter of the cache.

Config.		Effective Vul. Number for Different Configuration				
Associativity	$asso_b$ Value	1	2	4	8	16
	Total Vul. Num.	75	78	**82**	75	75
	SO-Type Num.	17	17	**20**	13	12
Line Size	$line_b$ Value	16	32	64	128	256
	Total Vul. Num.	77	75	**82**	80	79
	SO-Type Num.	14	12	**18**	17	17
Cache Size	tot_b Value	8192	16384	32768	65536	98304
	Total Vul. Num.	79	77	**82**	79	77
	SO-Type Num.	16	15	**20**	16	14

configuration of the benchmark.

As shown in Table 4.7, we found that differences between the number of correct configuration and incorrect configuration for all effective vulnerabilities and *SO*-Type only effective vulnerabilities are roughly the same. For example, when changing the associativity, difference of all effective vulnerability numbers between 4 (82) and 8 (75) is 7, which is the same as difference of *SO*-Type numbers (between 4 (20) and 8 (13)). This also shows that wrong configurations will still lead to *AO*-Type and *SA*-Type vulnerabilities to be effective even if the configuration is wrong.

As shown in Table 4.7 as well, attacks are most effective under the correct configuration. When setting the wrong value for either one of the three cache configurations, the number of vulnerabilities that are effective decreases. On the other hand, this shows that hiding the cache architecture information or giving wrong configurations on-purpose is not a reliable defense against the attacks.

4.2.6 Evaluation of Arm Secure Caches

As shown in the previous sections, current commercial Arm architectures are indeed vulnerable to most of the attack types. A potential defense are secure caches. To help understand if existing secure cache designs could help defend the attacks in Arm processors, we implemented and evaluated the Partition-Locked (PL) [1] and Random Fill (RF) [2] caches together with our benchmarks in the `gem5` simulator. We show that they can defend many of the attacks, but we also uncover new vulnerabilities in the secure cache designs. In

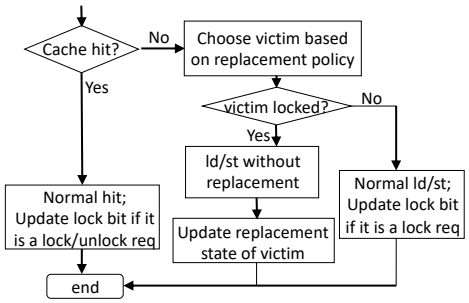


Figure 4.13: PL cache replacement logic flow-chart, as proposed in [1].

this section, we focus on the security analysis of the secure cache designs. Performance evaluations of PL cache and RF cache can be found in [1] and [2], where reasonable overhead is shown.

PL Cache Design and Implementation

Cache replacement is considered as the root cause of many cache side-channel attacks, and partitioned caches were proposed to prevent the victim's cache line from being evicted by the attacker. PL cache [1] is a flexible cache partitioning design, where the victim can choose cache lines to be partitioned. In the PL cache, each cache line is extended with a lock bit to indicate if the line is locked in the cache. When a cache line is locked, the line will not be evicted by any cache replacement until it is unlocked. Figure 4.13 shows the replacement logic of the PL cache. If a locked cache line is selected to be evicted, the eviction will not happen, and the incoming cache line will be handled uncached. If the victim locks the secret-related address properly and the cache is big enough to hold all the locked cache lines, the PL cache is secure against all types of timing vulnerabilities, because the secret-related address will always be in the cache.

To evaluate the PL cache against different vulnerabilities, we implement it in the L1 data cache and add new instructions to lock (and unlock) cache lines in the `gem5` simulator. The evaluation in `gem5` is run in SE mode using a single O3CPU core, where each benchmark has an additional `lock` step for locking the victim's cache line.

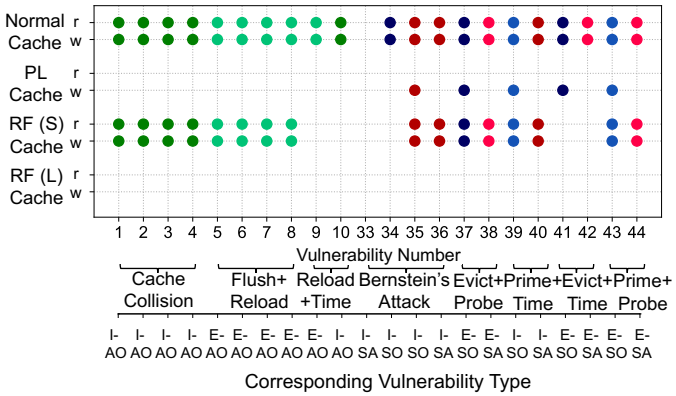


Figure 4.14: Evaluation results of security benchmarks on PL cache, RF cache, and a normal set-associative cache, for comparison. Solid dots, half solid dots or empty dot mean all of the, part of the, or no vulnerability cases are vulnerable to the cache, respectively.

Security Evaluation of the PL Cache

Figure 4.14 shows the results of evaluation of the PL cache (and the RF cache, as well as the baseline set-associative cache). For the PL cache, *AO*-Type vulnerabilities such as Flush+ Reload fail, because the sensitive data is locked in the cache, and cannot be evicted by the benchmark steps that simulate the attacker. Without locking, a normal cache is vulnerable to these attacks, as shown in Figure 4.14.

For *SO*-Type or *SA*-Type vulnerabilities such as Bernstein's attack, theoretically the PL cache should prevent all of them as well. However, from the experimental results we find that when the steps are implemented using write (store), some of the attacks will still be successful in the PL cache. This is mainly due to the write buffer structure, which is not considered in original design of the PL cache [1]. These attack strategies all require conflicts of known and unknown secret cache lines. Although being locked before the attack runs, the secret cache lines may be further brought into the write buffer due to a write access and then leave the cache structure to "bypass" the locking features, making the attack successful. On the other hand, without the influence of the write buffer, we find that the attack cases that have all three steps to be non-write accesses to be always prevented on PL cache, as expected. The vulnerabilities leveraging the cache coherence states and multiple cores were not considered in original PL cache design, but can be tested in future.

The PL cache evaluation highlights the need for systematic security evaluation using

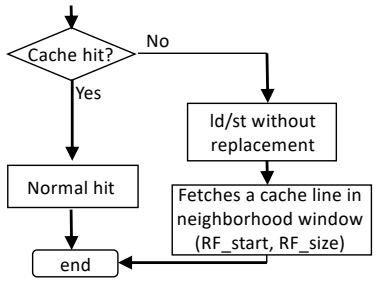


Figure 4.15: RF cache replacement logic flow-chart, as proposed in [2].

benchmarks. Thanks to the approach, the original PL cache design is found to have a new write-based attack. More importantly, our benchmarks can be useful for designing future secure caches and testing them in *gem*5.

RF Cache Design and Implementation

To prevent interference caused by cache replacement, Random Fill (RF) cache [2] has been proposed to de-correlate the cache fill that causes the cache replacement and the victim's cache access. On a cache miss, the missing cache line will be handled without being fetched in the cache, instead a cache line in the neighborhood window $[addr - RF\_start, addr - RF\_start + RF\_size]$ will be fetched, as shown in Figure 4.15. In this way, the memory access pattern is de-correlated from the cache lines fetched in the cache. Since fetching cache lines in the neighborhood window may still carry information about the original $addr$, the security of RF cache depends on the parameters RF_start and RF_size.

We implement the RF cache in the L1 data cache, as suggested by the work [2]. Note that here the cache line will still be fetched into L2 cache, but vulnerabilities targeting the L1 cache should be defended. Parameters RF_start and RF_size can be configured in `gem5`. The benchmark suite for evaluation is identical to the normal three-step benchmarks, no additional step is required for the RF cache, e.g., no special locking step is needed.

Security Evaluation of the RF Cache

RF cache can potentially defend all attacks because the victim's access to a secret address will not cause the corresponding cache line to be fetched into cache, but a random cache line in a neighborhood window will be fetched instead. However, fetching a cache line in the

neighborhood window still can transfer information about the victim's cache access. We tested two different RF cache configurations, one with small neighborhood window (5 cache lines) and one with large neighborhood window (128 cache lines[4]).

To reduce noise in the tests, the benchmarks test 8 contiguous cache lines and measure the total timing. When the neighborhood window of the RF cache is small, the cache line fetched into the cache will be not far from the address being accessed, and can still be observed by the third step of the benchmark with a high probability. As shown in Figure 4.14, for a small neighborhood window (S), a number of vulnerabilities are still effective, such as Flush+Reload and Prime+Probe.

For a large neighborhood window (L), no effective vulnerabilities are detected by the benchmark. For *SO*-Type vulnerabilities, the large neighborhood window de-correlates the memory access and the cache set to be accessed, so that the vulnerabilities can be prevented. For *AO*-Type vulnerabilities, the channel capacity of the cache side channel decreases with the window size due to the reduced probability of the desired cache line being fetched into cache, as analyzed in [2]. The neighborhood window of 128 cache lines is enough to mitigate the channel in our setting where there are 128 cache sets.

The evaluation of the RF cache shows how the benchmark suite can be used to help choose the design parameter, and the benchmark can quickly evaluate the design prototypes.

Security Evaluation of Other Secure Caches

CEASER [63] is able to mitigate conflict-based LLC timing–based side-channel attacks using address encryption and dynamic remapping. The CEASER cache does not differentiate whom the address belongs to and whether the address is security critical. When a memory access tries to modify the cache state, the address will first be encrypted using a Low-Latency BlockCipher (LLBC) [100], which not only randomizes the cache set it maps to, but also scatters the original, possibly ordered, and location-intensive addresses to different cache sets, decreasing the probability of conflict misses. The encryption key will be periodically changed to avoid key reconstruction. CEASER-S [101] allows CEASER to divide the cache ways into

4. There are 128 cache sets in the evaluated L1 cache.

multiple partitions of all the cache ways and allows the line to be mapped to a different set in each partition via principles of skewing. The modified "skew" idea of CEASER-S cache assigns each partition a different multiple instance of CEASER to determine the set mappings to strengthen the random mapping. These two caches, focusing on randomizing cache set mapping, targets *SO*-type or *SA*-type attacks and cannot prevent *AO*-type vulnerabilities.

ScatterCache [102] uses cache set randomization to prevent timing attacks. It builds upon two ideas. First, a mapping function is used to translate memory addresses and process information to cache set indices. The mapping is different for each program or security domain. Second, the mapping function also calculates a different index for each cache way. The mapping function can be keyed hash or keyed permutation derivation function – a different key is used for each application or security domain resulting in a different mapping from addresses to cache sets. Software (e.g., the operating system) is responsible for managing the security domains and process IDs, which are used to differentiate the software processes and assign them with different keys for the mapping. As hardware extension, a cryptographic primitive such as hashing and an index decoder for each scattered cache way is added. ScatterCache is able to prevent *SO*-type or *SA*-type vulnerabilities by assigning a different index for each cache way and security domain. It encrypts both the cache address and process ID when mapping into the cache index, therefore, ScatterCache is able to prevent *E-AO*-type vulnerabilities such as Flush+Reload, but not *I-AO*-type vulnerabilities such as Cache Collision vulnerabilities.

Time-Predictable Secure Cache (TSCache) [103] relies on random placement to exhibit randomized execution times. To achieve side-channel attack robustness, random placement must also decouple cache interference of the attacker from the victim. Memory addresses from victim and attacker's processes must not contend systematically in the same cache set. Instead, each memory address from each process must be randomly and independently placed in a set, thus randomizing interference. This is achieved by operating the address (tag and index bits) together with a random number called random seed. Each task is forced to have a different seed so that conflicts between attacker's and victim's cache lines are random and independent across runs, thus defeating any contention-based attacks. The same seed is given to allow the communication between runnables of a given software components of

an application via shared memory. TSCache exploits random placement to de-correlate set mapping with the corresponding address index bits. Therefore, it can be used to prevent SO-type or SA-type vulnerabilities but may not be able to prevent AO-type vulnerabilities.

4.3 TLB Timing Vulnerabilities and Secure TLBs

We now extend the three-step model to TLBs. We show how to automatically generate security microbenchmarks that test for the TLB vulnerabilities. After showing the insecurity of standard TLBs, two new secure TLB designs are presented to mitigate all the TLB vulnerabilities found.

4.3.1 Modeling TLB Timing Vulnerabilities

To analyze all the possible timing TLB attacks, this section presents a three-step modeling approach which can be used to reason about the behavior of the TLB logic and to derive all the possible timing vulnerabilities.

Threat Model and Assumptions

A TLB timing attack involves an attacker and a victim. In many cases they are executing on the same processor core, a set of cores, or a set of hyper-threads which share same physical TLB, but this is not required for all types of the attacks. In this work, we use A and V to denote the attacker and the victim with different process IDs. For the attacks where the attacker and the victim are in the same address space, attacker is able to trigger some known address memory operation as if it were the victim, e.g., states V_a and $V_{a^{alias}}$ in Table 4.8 can be actually attackers.

We assume, in hardware, all memory operations are identified by the virtual memory address, *vaddr* (including null address in case of certain TLB flush related operations) and the process ID (including null process ID in case of certain TLB flush related operations), e.g., ASID in RISC-V.

The victim is assumed to have some security critical memory range, x, within which the access pattern depends on the secret the attacker wants to learn. An example of a security

critical region is the set of page entries accessed during execution of the RSA functions of *libgcrypt*, where the value of the key bit (either 0 or 1) determines which specific memory pages are accessed. The timing of the accesses to the security critical memory range is affected by the timing of TLB related operations, and it can reveal information such as cryptographic keys.

The attacker is assumed to know the victim software, e.g., what implementation of a cryptographic algorithm it uses, but not the secret cryptographic keys. He or she is assumed to know the size, $ssize$, and the location, $sbase$ (in virtual memory) of the security critical memory range x. And, the attacker is assumed to know the TLB state machine logic; although during run-time of the processor the attacker cannot access the internal state of the hardware TLB – he or she can only observe the timing of the memory operations and try to deduce the state of the TLB from the timing.

The attacker can measure the timing of its own memory operations or operations of the victim; but cannot access the actual sensitive data being processed by the victim. In most cases, the attacker can also force the victim to execute specific operations, e.g., force the victim to perform decryption while the attacker measures timing. Thus, even if some operation is done by the victim, it is under control of the attacker so attacker can measure the timing. The timing can be identified by the attacker as $fast$ or $slow$.

The timing attacks can be both side-channel attacks and covert-channel attacks. The difference between the two is that the victim in the side-channel scenario is the sender in the covert-channel scenario. Regardless of the channel type, we use V for victim (or sender) and A for attacker (or receiver).

Our threat model assumes that high-level OS page table related channels are already mitigated. E.g., TLB miss can take variable amount of time depending on whether there already exists a page table translation, or whether the OS has to create a new translation entry during a page fault. We focus on address translation data of the TLB structure. We do not consider possible TLB timing channels that are due to port contention, LRU replacement, or any directory structures. We also do not consider Page Walk Cache [104, 105] effect on storing intermediate translations of memory pages[5].

5. So far Page Walk Cache does not exist in RISC-V architecture.

Introduction of the Three-Step Model

One observation we make is that all existing TLB timing attacks take three steps. In *Step* 1, a memory operation is performed, placing the TLB block (also called TLB slot or TLB entry) in a known initial state (e.g., a new translation is put into the block or block is invalidated). Then, in *Step* 2, a second memory operation alters the state of the TLB block from the initial state. Finally, in *Step* 3, a final memory operation is performed, and the timing of the final operation reveals some information about the relationship among the addresses from *Step* 1, *Step* 2 and *Step* 3. Attacks with more than three steps can be reduced to a three-step attack, as shown in Appendix A.

We write the three steps as: *Step* 1 $\rightsquigarrow$ *Step* 2 $\rightsquigarrow$ *Step* 3 which indicates a sequence of steps taken by the attacker or the victim. Table 4.8 lists all the 10 possible states of the TLB block for each step of our three-step model.

Each step in the model represents a state of a TLB block, since all the TLB blocks are updated following the same TLB state machine logic, it is sufficient to consider only a TLB block as it is the smallest unit of the TLB. Different implementations of TLBs involve different mapping functions for the TLB blocks. However, this does not affect the model, as the steps target on only one single TLB block. Different TLBs may make it more difficult in practice for attacker and victim to access the same block, but once they can achieve that – qualifying the practical difficulty of achieving certain steps is not part of the model, the model shows if there is a possibility of an attack or not.

Derivation of All TLB Vulnerabilities

Based on the states possible in each step there are in total $10 * 10 * 10 = 1000$ combinations of possible three-steps. We developed an algorithm that can process the list of all the three-steps, and eliminates ones which cannot lead to an attack. A three-step combination cannot become a vulnerability if it satisfies one of the below rules:

1. A $\star$ is not possible in *Step* 2 or *Step* 3, having $\star$ in the step means the TLB is in an unknown state and this removes useful information for the attacker.
2. A V_u must be in one of the steps. If there is no unknown u in the steps, there is

Table 4.8: The 10 possible states for a single TLB block in the three-step model.

States	Description
V_u	The TLB block contains translation for a memory address u, translation which is placed in the TLB block due to a memory access by the victim. Attacker does not know u, but u is from a range x of memory locations, range which is known to the attacker. The address u may have same page index as A_a or V_a and thus conflict with them in the TLB block. The goal of the attacker is to learn the page address or index of V_u.
A_a or V_a	The TLB block contains translation for a memory address a. The translation is placed in the TLB block due to a memory access by the attacker, A_a, or the victim, V_a. The attacker knows the address a, independent of whether the access was by the victim or the attacker themselves. The address a is from within the range of sensitive locations x. The address a may or may not be the same as the address u.
$A_{a^{alias}}$ or $V_{a^{alias}}$	The TLB block contains translation for a memory address a^{alias}. The translation is placed in the TLB block due to a memory access by the attacker, $A_{a^{alias}}$, or the victim, $V_{a^{alias}}$. The address a^{alias} is within the range x. It is not the same as a, but it has same page index and can map to the same TLB block, i.e. it "aliases" to the same block.
A_{inv} or V_{inv}	The TLB block previously containing translation for a memory address is now invalid. The translation is "removed" from the TLB block by the attacker A_{inv} or the victim V_{inv} as the result of TLB block being invalidated, e.g., due to synchronization updates to in-memory memory-management data structures or due to context switch between processes which causes OS to flush per-core TLB entries.
A_d or V_d	The TLB block contains translation for a memory address d. The translation is placed in the TLB block due to a memory access by the attacker, A_d, or the victim, V_d. The address d is not within the range x.
$\star$	Any data, or no data, can be in the TLB block. The attacker has no knowledge of page address in this TLB block.

nothing for the attacker to learn.

3. A $\star$ in one step, followed by V_u in next step cannot lead to an attack, since the TLB block needs to be in some known state before V_u is placed into it.
4. Three-step patterns with two adjacent steps repeating, or both known to the attacker, can be eliminated[6].
5. Steps involving a known address a and an alias to that address a^{alias}, give same information, thus three step combinations which only differ in use of a or a^{alias} cannot represent different attacks, and only one combination needs to be considered, e.g., $V_u \rightsquigarrow A_{a^{alias}} \rightsquigarrow V_u$ is a repeat type of $V_u \rightsquigarrow A_a \rightsquigarrow V_u$, and one of the two can be eliminated from the model.
6. An *inv* related state cannot be in *Step* 2 or *Step* 3 because it is so far not possible for most ISAs to allow flushing of the TLB from user space. (See more discussion in Appendix B).
7. If measured timing corresponds to more than one possible sensitive address translation of the victim, the corresponding vulnerability is removed. E.g., $\star \rightsquigarrow A_a \rightsquigarrow V_u$ is

6. Some of the possible attacks involve only two steps, but these attacks are represented by three-step model where first step is an explicit $\star$, i.e., they are represented by patterns $\star \rightsquigarrow \cdots$.

Table 4.9: The table shows all the timing TLB vulnerabilities. *Attack Strategy* column gives our common name for each set of one or more specific vulnerabilities that would be exploited in an attack in a similar manner (many of the names are borrowed from cache timing attacks in literature). *Vulnerability Type* column gives the three steps that define each vulnerability. For *Step* 3, *fast* indicates a TLB hit must be observed, while *slow* indicates a TLB miss must be observed. *Macro Type* column proposes the categorization the vulnerability belongs to. "E" is for external interference vulnerabilities. "I" is for internal interference vulnerabilities. "M" is for miss-based vulnerabilities. "H" is for hit-based vulnerabilities. *Attack* column shows if a type of vulnerability has been previously presented in literature.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
TLB Internal Collision	A_{inv}	V_u	V_a (fast)	IH	(1)
	V_{inv}	V_u	V_a (fast)	IH	(1)
	A_d	V_u	V_a (fast)	IH	(1)
	V_d	V_u	V_a (fast)	IH	(1)
	$A_{a^{alias}}$	V_u	V_a (fast)	IH	(1)
	$V_{a^{alias}}$	V_u	V_a (fast)	IH	(1)
TLB Flush + Reload	A_{inv}	V_u	A_a (fast)	EH	**new**
	V_{inv}	V_u	A_a (fast)	EH	**new**
	A_d	V_u	A_a (fast)	EH	**new**
	V_d	V_u	A_a (fast)	EH	**new**
	$A_{a^{alias}}$	V_u	A_a (fast)	EH	**new**
	$V_{a^{alias}}$	V_u	A_a (fast)	EH	**new**
TLB Evict + Time	V_u	A_d	V_u (slow)	EM	**new**
	V_u	A_a	V_u (slow)	EM	**new**
TLB Prime + Probe	A_d	V_u	A_d (slow)	EM	(2)
	A_a	V_u	A_a (slow)	EM	(2)
TLB version of Bernstein's Attack	V_u	V_a	V_u (slow)	IM	**new**
	V_u	V_d	V_u (slow)	IM	**new**
	V_d	V_u	V_d (slow)	IM	**new**
	V_a	V_u	V_a (slow)	IM	**new**
TLB Evict + Probe	V_d	V_u	A_d (slow)	EM	**new**
	V_a	V_u	A_a (slow)	EM	**new**
TLB Prime + Time	A_d	V_u	V_d (slow)	IM	**new**
	A_a	V_u	V_a (slow)	IM	**new**

(1) Double Page Fault attack [50]. (2) TLBleed attack [48].

removed because when observing a fast timing, u can possibly map to a, or first step's potential u that is included in $\star$.

After applying the script which implements our simplification algorithm, 34 three-step access patterns remain as candidates for possible timing TLB attacks. By checking the timing variation manually, these 34 access patterns are further reduced to a list of 24 types of timing TLB vulnerabilities, listed in Table 4.9.

To summarize all the vulnerability types, Table 4.9 shows the list of all the 24 vulnerability types, along with a more coarse-grained attack strategies, which cover one or more vulnerability types. The list of vulnerability types can be further collected into four simple macro types: internal interference miss-based (IM), internal interference hit-based (IH), external interference miss-based (EM), external interference hit-based (EH).

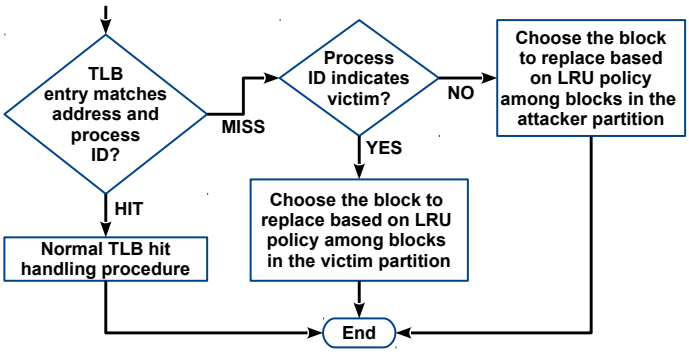


Figure 4.16: SP TLB access handling procedure flow chart.

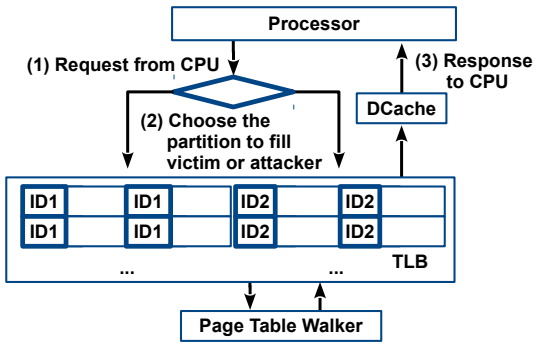


Figure 4.17: Sample block diagram of SP TLB with victim (ID1) and attacker (ID2) partition being allocated 50% of the TLB space.

All types of vulnerabilities only involving the victim, V, in the states in *Step* 2 and *Step* 3 are called internal interference vulnerabilities (I). The remaining ones are called external interference (E). Some vulnerabilities allow attacker to learn that the page address of the victim maps to the TLB set of the attacker by observing *slow* timing due to a TLB miss. we call these miss-based vulnerabilities (M). The remaining ones leverage observation of $fast$ timing due to a TLB hit, and are called hit-based vulnerabilities (H).

Most of the vulnerability types have not been explored before, except for two groups. The Double Page Fault attack [50] is effectively based on the Internal Collision, and it maps to types labeled (1) in the Attack column in Table 4.9. The TLBleed attack [48] is effectively based on the Prime+Probe strategy, and it maps to types labeled (2) in the Attack column in Table 4.9. All other vulnerability types correspond to new attacks not previously discussed.

4.3.2 Secure TLB Designs

In order to prevent timing vulnerabilities, we designed two secure TLBs, the SP TLB and the RF TLB. Designs of the secure TLBs follow the threat model discussed in Section 4.3.1. We focus on L1 D-TLB in this work, but it can be applied to instruction TLBs as well as other levels of TLB.

Static-Partition (SP) TLB

SP TLB is a SA TLB where certain ways are assigned to a victim process and other ways are assigned to all remaining processes, which by default are assumed to be potential attacker processes. The process ID, e.g., ASID in RISC-V, is used to differentiate the victim and the attacker. The number of ways assigned to each is set at design time, but could be further extended to be dynamic at run time.

SP TLB Access Handling Procedure

SP TLB isolates the accesses between the victim and the attacker. TLB hits are identical to SA TLB, where both address and process ID must match. For TLB misses, the victim's address translations cannot cause replacement in the attacker's partition, and the attacker's address translations cannot cause replacement in the victim's partition. Each partition maintains its own LRU policy, which can prevent some LRU attacks, but defense of LRU related attacks is not focus of this work as discussed in the threat model. The SP TLB access handling procedure is shown in Figure 4.16.

SP TLB Logic

The SP TLB (Figure 4.17) partitions the victim and the attacker by cache ways. The allocation of different partitions is configurable during the design time. Assuming there are M ways in total. The victim partition will take N ($0 < N < M$) ways while the attacker partition will take the remaining $M - N$ ways. In our implementation, the victim and the attacker part are allocated 50% of TLB ways by default. Process ID field of each entry in the TLB is reused by the SP TLB to determine whether a partition is victim's or attacker's. SP

TLB requires minimal changes to the TLB logic, and protects 14 out of the 24 vulnerabilities shown in Section 4.3.3.

Random-Fill (RF) TLB

To protect all the vulnerabilities, we propose Random-Fill TLB, which is able to de-correlate the requested memory access from actual TLB entries that are brought into the TLB, making the attacker's observations non-deterministic. For TLB hits, the behavior is the same as the SA TLB. For TLB misses, depending on the memory address region, a random address translation will be fetched into the TLB ("random fill"), while the originally requested address is directly sent back to the CPU without filling the TLB ("no fill"). The RF TLB also introduces the *Sec* bit which is used to identify certain memory translation entries are belonging to secure data.

RF TLB Access Handling Procedure

RF TLB access handling procedure is shown in Figure 4.18. D denotes the requested address translation. D' is a random address translation to be filled in the TLB (D and D' are possibly the same because of the randomization). R is the TLB entry that would be evicted by D, in TLB set S, according to the LRU replacement policy. R' is the TLB entry evicted by D'. Sec_R and Sec_D is the *Sec* bit of R and D, respectively, indicating whether the page address is in the secure region.

If D maps to an existing entry in the TLB (page address and process ID matches), a normal TLB hit handling procedure will occur. Otherwise:

- If Sec_R is 0 and Sec_D is 0, normal TLB miss occurs.
- If Sec_R is 1 and Sec_D is 0, then D' is chosen as a random virtual non-secure page address, within the same sets of TLB entries as the secure region, and filled in TLB, evicting R'. Meanwhile, R will not be evicted and results of D request will be sent to the processor directly. Thus an attacker cannot deterministically evict the secure address chosen by the replacement policy.

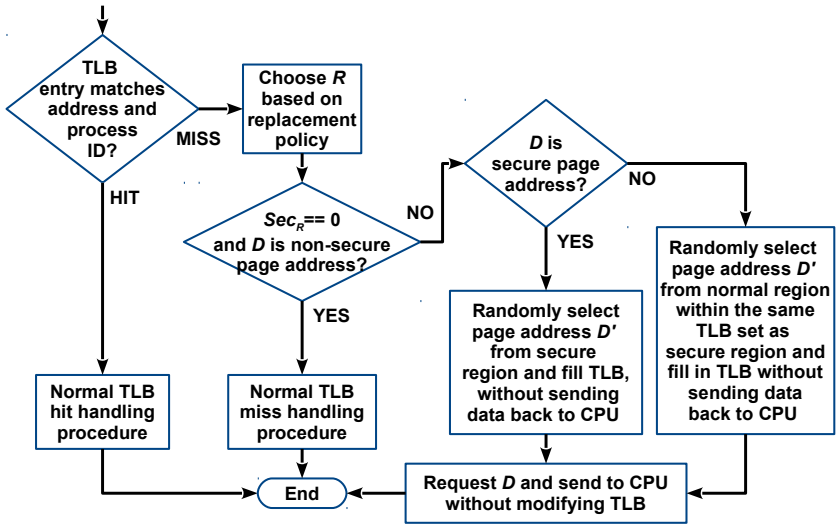


Figure 4.18: RF TLB access handling procedure flow chart.

- If Sec_D is 1, then D' is chosen as a random virtual address within the secure region, and filled in TLB, evicting R'. Results of D request will be sent to the processor directly. Thus, an attacker cannot observe TLB state changes due to secure page address D, but he or she instead observers TLB state changes due to random page address D'.

RF TLB uses the randomization approach to randomly bring in data from specified memory ranges to confuse the attacker. It does not randomize all of the TLB accesses so as to limit the performance impact. The RF TLB is able to prevent all types of timing vulnerabilities shown in Table 4.9, which are discussed in Section 4.3.3.

RF TLB Logic

RF TLB block diagram is shown in Figure 4.19b. All the bold lines and blocks are the added hardware and logic extension. In the TLB array, an extra field (a secure bit *Sec*, either 0 or 1) is added to each of the TLB entries to indicate whether it contains an address translation within the secure region. In addition, the existing process ID field (e.g., ASID in RISC-V) in each TLB entry is used to differentiate the victim and the attacker process. By default, we set specific process ID 1 for the victim program and all other ASIDs to be attackers.

An extra set of registers is added to store the process ID of the victim process and the start address, *sbase*, and the size, *ssize* of the secure region (the base and size are defined

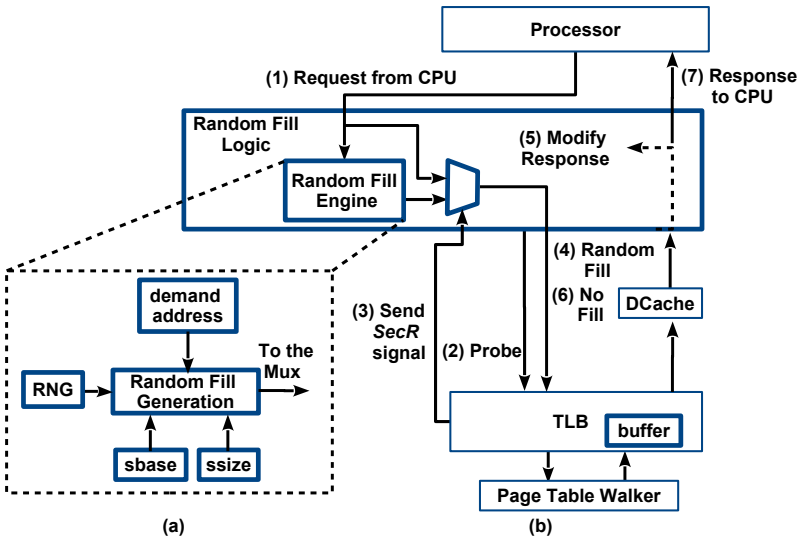


Figure 4.19: RF TLB: (a) Random Fill Engine, (b) RF TLB block diagram.

in terms of pages, usually 4KiB). The registers can be managed by a trusted OS to change the victim process ID and secure regions when different victim programs need protection.

An extra *buffer* is added which stores equivalent of one TLB entry. It is used as temporary storage for translation data that is returned to the CPU, but which should not be placed in the TLB. It will be cleaned up after the address is returned.

The Random Fill Engine (RFE), shown in Figure 4.19a is used to generate addresses which should be used for TLB updates[7]. In Figure 4.19b, (1-2), the "no fill" *fill_type* will first be sent to TLB. On a TLB miss, the TLB will *probe* the page address without filling TLB entries to see if the chosen entry has a valid secure page address translation. Then, (3) the Sec_R bit is set and sent back. Next, (4) if it is a request to the secure region or the Sec_R bit is one, a random fill request will be triggered. If the original request is in the secure region, a random virtual page address is derived from RFE within the secure region $[sbase, sbase + ssize]$, and a translation will be put into the TLB entry. If the original request comes from the non-secure region, most of the higher bits of the requested address are remain the same while the bits that correspond to the TLB set index[8] will be randomized to make

7. We assume the OS has pre-generated page table entries that may correspond to the random virtual address generated by the RFE, which may not be actually used by the original program, to prevent OS or software-based timing attacks due to page faults when a page entry for a random address is looked up by the TLB logic.

8. The TLB set index to be randomized has bit size $S_n = log_2[min(ssize, nsets)]$, where $nsets$ is the number of sets in TLB. A random set index will be generated within the region $[sbase[S_n - 1, 0], sbase[S_n -$

the eviction indeterministic. Next, (5) the *Random Fill Logic* will modify the response and prevent the random fill result from being sent to the processor. Then, (6) the original page address is finally requested, and "no fill" *fill_type* will be sent to the TLB to obtain the translation. Finally, (7) this address will be stored in the *buffer*, without modifying TLB entries, and be sent back to the processor.

RF TLB vs. RF Cache

As a possible alternative, the "random fill" part can be done asynchronously during the idle cycles, as has been proposed for secure caches, e.g., [2]. However, using this way, programs which are TLB intensive for accesses to the secure region will starve "random fill" and result in no random entries being put into the TLB, which will negatively impact the security offered by the TLB.

The proposed RF TLB differentiates victim and attacker, secure and non-secure region and has a different random fill scheme for pages within or outside of the secure region for both attacker and victim. That helps RF TLB prevent all types of TLB timing vulnerabilities. Meanwhile, RF Cache [2] only differentiates victim and attacker and cannot prevent all types of cache timing vulnerabilities [66].

4.3.3 TLB Security Evaluation

Most of the types of the attacks derived with the three-step model do not correspond to already known attacks. Some exceptions include, for example, the TLBleed attack [48], which was demonstrated using the *libgcrypt*'s RSA cryptographic implementation. In the RSA implementation, whether the pointer *tp* is accessed depends on the secret *e_bit* in *_gcry_mpi_powm* function (line 17, Figure 4.20). In TLBleed, the attacker can use the TLB Prime + Probe attack strategy to deploy an attack which allows them to learn whether *tp* is accessed, to know the secret bit. However, such examples for most of the other attacks do not exist.

Thus, we developed micro security benchmarks which can be used to test TLBs to check

$1, 0] + min(ssize, nsets)]$ for random fill.

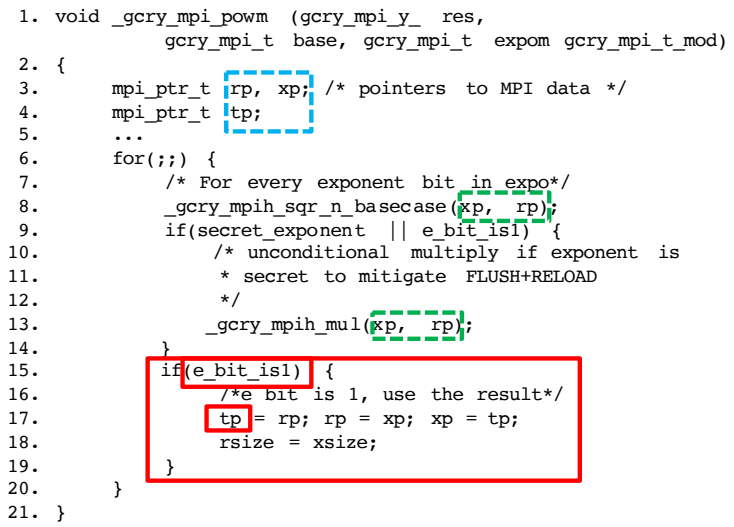


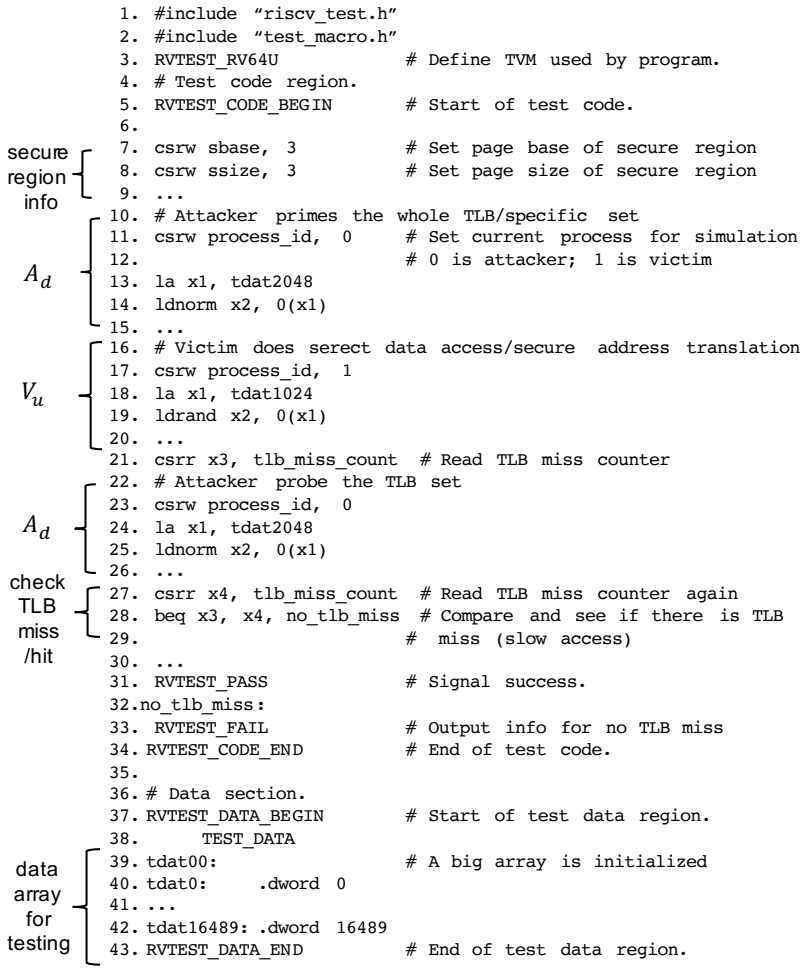
Figure 4.20: Code sample for one of the variants of modular exponentiation from *libgcrypt* version 1.8.2 used in experiments. Pointers *rp*, *xp* and *tp* are defined (blue dashed square). *rp* and *xp* are used for both *e_bit* is 1 or 0 (green dashed square). *tp* will only be accessed when *e_bit* is 1 (red square).

if they are vulnerable to each of the attack types. To generate the micro security benchmarks, we leverage a Python script that follows a three-step template to generate assembly code of all the types of vulnerabilities showed in Table 4.9.

Figure 4.21 is a micro security benchmark example of the $A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow) variant of TLB Prime + Probe vulnerability. Inside the benchmark, first there is standard prologue with include statements (line 1-5), then the secure region (*sbase*, *ssize*) is set (line 7-8). For the specific vulnerability, the three steps are executed in the order of A_d (line 10-15), V_u (line 16-20) and A_d (line 22-26). Out-of-secure-address-region d will be accessed using the *norm* type of memory access while inside-secure-address-region u will use *rand* type of memory access, corresponding to the non-secure and secure page address accesses illustrated in Section 4.3.2, respectively. Attacker measures the final step's timing (line 21, 27-29). The same script can be used to generate assembly tests for all SA TLB, SP TLB, and RF TLB.

Channel Capacity

An attacker gains knowledge about the secret address translation through TLB timing channel by observing the timing of address translation in a TLB block. The observed timing may depend on the victim's prior behavior.




```
1. #include "riscv_test.h"
2. #include "test_macro.h"
3. RVTEST_RV64U            # Define TVM used by program.
4. # Test code region.
5. RVTEST_CODE_BEGIN       # Start of test code.
6.
7. csrw sbase, 3           # Set page base of secure region
8. csrw ssize, 3           # Set page size of secure region
9. ...
10. # Attacker primes the whole TLB/specific set
11. csrw process_id, 0     # Set current process for simulation
12.                        # 0 is attacker; 1 is victim
13. la x1, tdat2048
14. ldnorm x2, 0(x1)
15. ...
16. # Victim does serect data access/secure address translation
17. csrw process_id, 1
18. la x1, tdat1024
19. ldrand x2, 0(x1)
20. ...
21. csrr x3, tlb_miss_count  # Read TLB miss counter
22. # Attacker probe the TLB set
23. csrw process_id, 0
24. la x1, tdat2048
25. ldnorm x2, 0(x1)
26. ...
27. csrr x4, tlb_miss_count  # Read TLB miss counter again
28. beq x3, x4, no_tlb_miss  # Compare and see if there is TLB
29.                        #  miss (slow access)
30. ...
31. RVTEST_PASS            # Signal success.
32.no_tlb_miss:
33. RVTEST_FAIL            # Output info for no TLB miss
34. RVTEST_CODE_END        # End of test code.
35.
36. # Data section.
37. RVTEST_DATA_BEGIN      # Start of test data region.
38.      TEST_DATA
39. tdat00:                # A big array is initialized
40. tdat0:     .dword 0
41. ...
42. tdat16489: .dword 16489
43. RVTEST_DATA_END        # End of test data region.
```

Figure 4.21: Code sample for TLB Prime + Probe micro security benchmark $A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow) variant, used in simulation testing of Rocket Core-based RISC-V.

There are two possible victim's behaviors B: whether the victim's secret-dependent memory access results in address translation, V_u, which maps to the TLB block tested by the attacker or not. There are also two possible attacker's observations O: whether attacker observes slow access due to a TLB miss or fast access due to a TLB hit.

To evaluate the relation between the victim's behaviors and the attacker's observations, we define p_1 and p_2 as listed next, and shown in Table 4.10: When the victim behavior triggers a translation of an address that maps to the TLB block the attacker tests, we use p_1 to denote the probability the attacker observes a TLB miss, and $1 - p_1$ as the probability the attacker observes a TLB hit. When the victim's behavior triggers a translation of an address that does not map to the TLB block the attacker tests, we use p_2 to denote the probability the attacker observing a TLB miss, and $1 - p_2$ for observing a hit.

Table 4.10: Probabilities of different victim behaviors B and attacker observations O.

		Attacker's observation O	
		Miss	Hit
Victim's behavior B	Memory access (or invalidation) maps to the same address/index attacker tests	p_1	$1-p_1$
	Memory access (or invalidation) does not map to the same address/index attacker tests	p_2	$1-p_2$

To provide the optimal scenario for attacker, we assume the probability of victim's access V_u mapping to the TLB block tested by the attacker to be the same as the probability of V_u not mapping to the block, i.e. both are $\frac{1}{2}$.

We use channel capacity [106] to quantify the amount of information about the secret address translation that the attack gains from a specific timing attack as follows:

$$\begin{aligned} C \equiv I(B;O) &\equiv \sum_{b,o} p(b,o) log \frac{p(b,o)}{p(b)p(o)} \\ &\equiv \frac{p_1}{2} log \frac{2p_1}{p_1+p_2} + \frac{p_2}{2} log \frac{2p_2}{p_1+p_2} \\ &\quad + \frac{1-p_1}{2} log \frac{2(1-p_1)}{2-p_1-p_2} + \frac{1-p_2}{2} log \frac{2(1-p_2)}{2-p_1-p_2} \end{aligned} \tag{4.1}$$

where $I(B;O)$ denotes the mutual information between victim's behavior B and attacker's observation O. The $p(b)$ and $p(o)$ are the marginal probability distribution functions of victim's behavior B and attacker's observation O. The $p(b,o)$ is the joint probability function of victim's behavior B and attacker's observation O.

The p_1 and p_2 will have different values for each type of vulnerability, and also depend on the type of the TLB. Especially, if a TLB is able to defend against a specific type of an attack, the mutual information C should be zero for that attack type. Otherwise, the attacker can gain some knowledge about the victim's behavior. Below we analyze the C for different TLB types, and compare with theoretical calculations.

Theoretical Result and Security Evaluation of the TLBs

We implemented SP TLB and RF TLB, as illustrated in Section 4.3.2, as well as a SA TLB, in Chisel code and integrated them into the Rocket Core-based RISC-V processor[9]. In

9. Chisel commit ID: 980778b, Rocket Chip commit ID: aca2f0c

		SA TLB								SP TLB								RF TLB							
Attack Category	Vulnerability Type	$n_{M,M}$	p_1*	p_1	$n_{N,M}$	p_2*	p_2	C^*	C	$n_{M,M}$	p_1*	p_1	$n_{N,M}$	p_2*	p_2	C^*	C	$n_{M,M}$	p_1*	p_1	$n_{N,M}$	p_2*	p_2	C^*	C
TLB Internal Collision	$A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	0	0	0	500	1	1	1	1	2	0.01	0	500	1	1	0.98	1	343	0.69	0.67	333	0.67	0.67	**0.01**	**0**
	$V_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	0	0	0	500	1	1	1	1	0	0	0	500	1	1	1	1	328	0.66	0.67	338	0.68	0.67	**0.01**	**0**
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	10	0.02	0	500	1	1	0.93	1	3	0.01	0	500	1	1	0.97	1	485	0.97	0.97	483	0.96	0.97	**0.01**	**0**
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	9	0.02	0	500	1	1	0.94	1	2	0.01	0	500	1	1	0.98	1	489	0.98	0.97	486	0.97	0.97	**0.01**	**0**
	$A_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	0	0	0	500	1	1	1	1	0	0	0	500	1	1	1	1	322	0.65	0.67	353	0.71	0.67	**0.01**	**0**
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	1	0.01	0	500	1	1	0.99	1	0	0	0	500	1	1	1	1	328	0.66	0.67	349	0.70	0.67	**0.01**	**0**
TLB Flush + Reload	$A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
TLB Evict +Time	$V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow)	500	1	1	0	0	0	1	1	0	0	0	26	0.05	0	**0.03**	**0**	0	0	0.01	0	0	0.01	**0**	**0**
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u$ (slow)	500	1	1	0	0	0	1	1	0	0	0	0	0	0	**0**	**0**	2	0.01	0.01	7	0.01	0.01	**0.01**	**0**
TLB Prime +Probe	$A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	500	1	1	1	0.01	0	0.99	1	0	0	0	20	0.04	0	**0.02**	**0**	167	0.33	0.33	158	0.32	0.33	**0.01**	**0**
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	500	1	1	1	0.01	0	0.99	1	0	0	0	2	0.01	0	**0.02**	**0**	135	0.27	0.26	148	0.30	0.26	**0.01**	**0**
TLB Bernstein's Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow)	500	1	1	1	0.01	0	0.99	1	500	1	1	1	0.01	0	0.99	1	0	0	0.01	10	0.02	0.01	**0.01**	**0**
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u$ (slow)	500	1	1	0	0	0	1	1	500	1	1	31	0.06	0	0.83	1	0	0	0.01	0	0	0.01	**0**	**0**
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	500	1	1	0	0	0	1	1	500	1	1	0	0	0	1	1	160	0.32	0.33	163	0.33	0.33	**0**	**0**
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	500	1	1	0	0	0	1	1	500	1	1	0	0	0	1	1	35	0.07	0.09	22	0.04	0.09	**0.01**	**0**
TLB Evict +Probe	$V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
TLB Prime +Time	$A_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**

Figure 4.22: Comparison of SA TLB, SP TLB and RF TLB simulation and theoretical results.

addition to implementing the TLBs, new TLB miss performance counters were implemented and are used by the simulation to determine *slow* or $fast$ TLB accesses based on whether miss occurs or not, respectively. The Chisel code for the whole processor with the new TLBs was used to generate cycle-accurate simulations.

The simulated hardware was used to execute the micro security benchmarks previously discussed in Section 4.3.3. For each benchmark, it was run 500 times each for "mapped" or "not mapped" (shown in Table 4.10) victim address for the tested TLB block, therefore in total 1000 times. Multiple runs are needed as the RF TLB leverages randomization and we need to average results over many runs. For each TLB type, each of the benchmarks was run, thus 24 vulnerability types × 1,000 simulations = 24,000 runs.

The security evaluation focused on 8-way 32-entry SA TLB as the example. With this

setup, the system software will take 4 out of 32 entries and distribute the 4 entries in different sets, so 28 different user pages are sufficient to prime the TLB. We assume two cases for the victim: one has 6 contiguous pages (3 pages out of the 6 are secure), another has 31 contiguous secure pages (to simulate contention between secure address translations).

Security of SA TLB, SP TLB and RF TLB

The theoretical and the simulation results of all the TLBs are listed and compared in Figure 4.22. p_1* and p_2* represent probabilities based on simulation. p_1 and p_2 are theoretical calculations. C* and C represent mutual information based on simulation and theoretical calculation, respectively. $n_{M,M}$ and $n_{N,M}$ denote number of misses when the victim's secret address and test address map and do not map to each other, respectively. Bold C* and C are the ones with value 0 or about 0, indicating that this TLB is able to prevent the corresponding vulnerability. Small numbers are rounded up.

SA TLB. SA TLB has simulated and theoretical $C = 0$ for TLB Flush + Reload, TLB Evict + Probe, and TLB Prime + Time attacks, thus it defends these attacks. SA TLB is not able to prevent internal TLB Collision ($p_1 = 0$, $p_2 = 1$, $C = 1$) and TLB Evict+Time, TLB Prime+Probe and TLB Bernstein's Attack ($p_1 = 1$, $p_2 = 0$, $C = 1$).

SP TLB. For SP TLB, all the vulnerabilities that SA TLB can prevent are also prevented by SP TLB. Further, TLB Evict + Time and TLB Prime + Probe vulnerability can be prevented by SP TLB. For these two types of vulnerabilities, $p_1 = p_2 = 0$, $C = 0$.

On the other hand, SP TLB is still vulnerable to TLB version of Bernstein's Attack ($p_1 = 1$, $p_2 = 0$, $C = 1$) and TLB Internal Collision vulnerability ($p_1 = 0$, $p_2 = 1$, $C = 1$) since victim's own address contention and TLB hit due to its own accesses cannot be defended by partitioning.

RF TLB. The RF TLB can defend all the vulnerabilities that SA TLB can defend. There are then 14 vulnerabilities left to consider, which can be further reduced to simplify the analysis: V_a and A_a belong to a, similarly, $V_{a^{alias}}$ and $A_{a^{alias}}$ are a^{alias}, V_d and A_d are d. Following this way, we can simplify the 14 patterns into 6 patterns for RF TLB, which are listed below. The *sec_range* stands for secure region, its value is 3 in the first 3 cases and 31, to simulate contention between secure address translations, in the last 3 cases. The *nset*

and *nway* stands for the number of sets and ways, whose value is 4 and 8 in the simulation tests, respectively. *prime_num* stands for the virtual page address number that can prime the whole 4-way 32-entry TLB in RISC-V. The theoretical probabilities p_1 and p_2 for the 6 combined patterns are:

- $V_u \rightsquigarrow d \rightsquigarrow V_u$ (slow): p_1=p_2=$\frac{1}{sec\_range}\times$ $\frac{1}{min(nset,sec\_range)\times nway}$=$\frac{1}{3}\times\frac{1}{3\times 8}$=0.01.
- $d/inv \rightsquigarrow V_u \rightsquigarrow a$ (fast): p_1=p_2=1-$\frac{1}{sec\_range}$=1 − $\frac{1}{3}$=0.67.
- $d \rightsquigarrow V_u \rightsquigarrow d$ (slow): p_1=p_2=$\frac{1}{sec\_range}$=$\frac{1}{3}$=0.33.
- $V_u \rightsquigarrow a \rightsquigarrow V_u$ (slow): p_1=p_2=$(\frac{nway}{sec\_range})^{nway}$=$(\frac{8}{31})^8$=0.01.
- $a^{alias} \rightsquigarrow V_u \rightsquigarrow a$ (fast): p_1=p_2=1-$\frac{1}{sec\_range}$ = 1 − $\frac{1}{31}$=0.97.
- $a \rightsquigarrow V_u \rightsquigarrow a$ (slow): Because V_u cannot get hit due to A_a, there are two cases:
  - $A_a \rightsquigarrow V_u \rightsquigarrow A_a$: p_1=p_2=$\frac{nway}{sec\_range}$=$\frac{8}{31}$=0.26.
  - $V_a \rightsquigarrow V_u \rightsquigarrow V_a$: p_1=p_2=$\frac{sec\_range-prime\_num}{sec\_range}$=$\frac{31-28}{31}$=0.09.

All the mutual information derived derived based on the above probabilities for the RF TLB is 0 for the theoretical calculations and about 0 for the simulation results as shown in Figure 4.22, indicating RF TLB is secure against these attacks.

Comparison of the Different TLBs. As can be seen from Figure 4.22, the simulation results match the theoretical values closely, indicating the actual hardware Chisel implementation of TLBs matches the theoretical calculations we presented.

For the TLBs, normal SA TLB can prevent 10 types of timing vulnerabilities due to requirement of deriving TLB hit for both address and process ID. For the SP TLB, it is able to prevent external interference by partitioning but is weak at preventing internal interference. Therefore, SP TLB is able to prevent 4 more types of vulnerabilities. However, internal hit-based vulnerabilities, such as $V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast), can still happen in SP TLB. For RF TLB, random fill technique is able to de-correlate the TLB fill with the memory access. This is able to prevent all classes of timing vulnerabilities listed in Table 4.9.

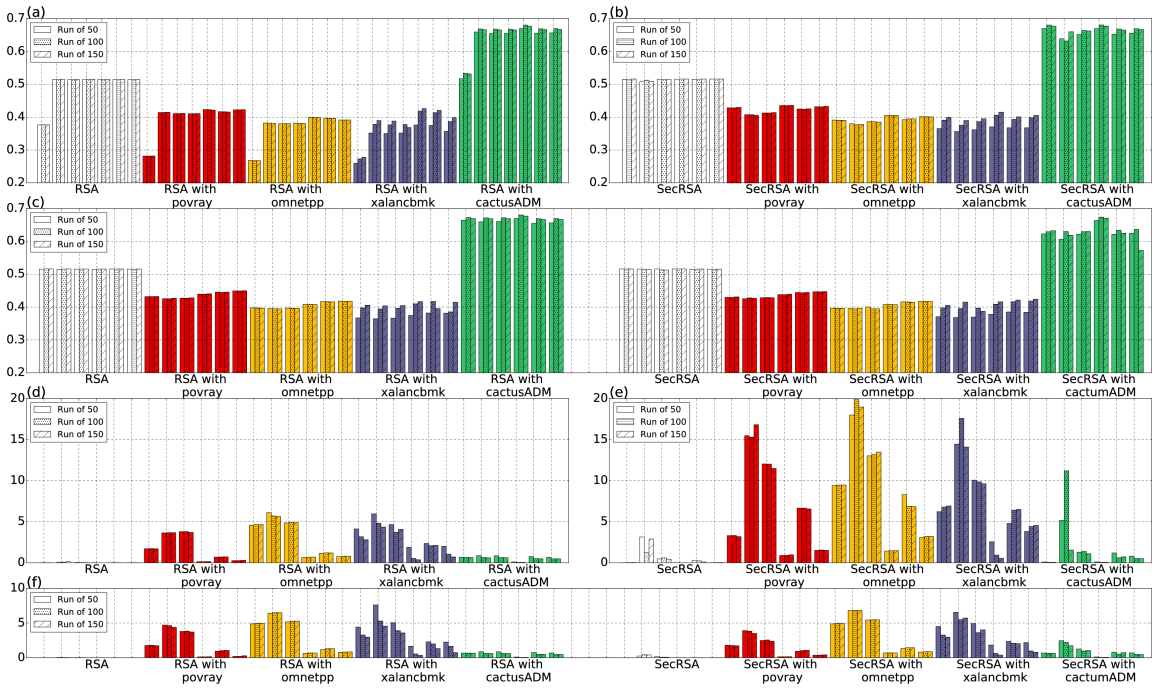


Figure 4.23: Evaluation of different configuration of TLBs. (a)-(c) IPC of SA TLB, SP TLB and RF TLB, respectively. (d)-(f) MPKI of SA TLB, SP TLB and RF TLB, respectively. Every set of bars in the graph follows the order: (1E only for IPC of SA TLB), FA 32, 2W 32, 4W 32, FA 128, 2W 128 and 4W 128.

4.3.4 Performance Evaluation

The SP TLB and RF TLB were implemented in Chisel hardware construction language and realized in the Rocket Core-based RISC-V processor. SP TLB related logic is about 300 lines of Chisel code, while RF TLB related logic is about 500 lines of Chisel code. The same hardware code was used for simulation (Section 4.3.3) and the performance evaluation (this Section), with minor changes for the FPGA version. Further, to allow for performance evaluation under realistic settings and with use of Linux, the RISC-V with the secure TLBs was synthesized on the Xilinx ZC706 and ZedBoard Evaluation FPGA boards.

To enable performance measurements, a TLB miss counter was added, and cycle counter and instruction counters were enabled in user mode. The counters are used to collect data during execution of the cryptographic program and benchmarks. The collected data were: instructions per cycle (IPC) and TLB misses per kilo instructions (MPKI).

Two TLB sizes were selected for evaluation. 32-entry, 4-way SA TLB corresponds to TLBs used in Intel's Haswell processors [107], while the 128 entry corresponds to TLBs used

in Intel's Nehalem microarchitecture [108].

The following L1 D-TLB configurations were tested. FA TLB with 32 entries (labeled *FA 32* in figures), SA TLB with 32 entries, 2-way (labeled *2W 32* in figures), SA TLB with 32 entries, 4-way (labeled *4W 32* in figures), and the same configurations, but for 128 entries (labeled *FA 128*, *2W 128*, and *2W 128*, respectively in figures). All of these were used for baseline Standard TLB, SP TLB and RF TLB. In addition, a naive solution to prevent all TLB attacks is to disable the TLB. While in our RISC-V setup it is not possible to fully disable the TLB, we include TLB with 1 entry (labeled *1E* in figures) as closest possible configuration to show its impact on performance. In total, 19 TLB configurations were tested on our FPGA setup.

The SP TLB was tested with half the ways to be set victim partition. The RF TLB was tested where the secure region was set by the software, see *SecRSA* discussion below.

For performance evaluation, we use the RSA implementation from TLBleed attack [48][10]. Further, we selected TLB-intensive SPEC 2006 integer and floating point benchmarks to evaluate the overheads introduced by the secure TLB designs. The four selected benchmarks are: 453.povray, 471.omnetpp, 483.xalancbmk, and 436.cactusADM[11]. The different configurations are listed below.

RSA. The *libgcrypt*'s RSA decryption routine was run 50, 100 and 150 times in series to simulate multiple uses of secret cryptographic key that the attacker may want to learn via the timing channels. Each time the same hard-coded key was used. No security is enabled for this configuration.

SecRSA. This is same as RSA configuration, except for SP and RF TLBs the security features are enabled to protect the RSA. For SP TLB, the SecRSA's process ID is set as the "victim", and all the address translations will be put in the victim partition in the SP TLB, while other processes' address translation will be in the attacker partition. For the RF TLB, SecRSA's *.data* section pages including the ones referenced by the *tp*, *rp* and *xp* pointers (the number of these pages is 3, and the pointers are previously discussed in Section 4.3.3)

10. RSA from Libgcrypt 1.8.2: https://gnupg.org/ftp/gcrypt/libgcrypt/

11. The "ref" or "train" inputs to SPEC benchmarks were used, the "train" inputs were used if the benchmark was not able to run with "ref" inputs on the FPGA setup due to memory size limitation.

Table 4.11: Area overhead of the new secure additions. Δ Slice LUT and Δ Slice Registers columns show the the difference from the 32-entry, 4-way SA TLB baseline.

	Configurations	Slice LUTs	Δ Slice LUTs	Slice Registers	Δ Slice Registers
	1E	35266	-777	18359	-4406
SA TLB	FA 32	36395	352	22199	-566
	2W 32	36298	255	23513	748
	4W 32	**36043**	–	**22765**	–
	FA 128	40177	4134	33815	11050
	2W 128	39684	3641	38630	15865
	4W 128	38107	2064	35694	12929
SP TLB	FA 32	36499	456	22251	-514
	2W 32	36387	344	23523	758
	4W 32	**36183**	**140**	**22798**	**33**
	FA 128	40568	4525	33824	11059
	2W 128	38609	2566	38521	15756
	4W 128	38049	2006	35659	12894
RF TLB	FA 32	38281	2238	22697	-68
	2W 32	38510	2467	25643	2878
	4W 32	**38266**	**2223**	**24018**	**1253**
	FA 128	42740	6697	34252	11487
	2W 128	42509	6466	45823	23058
	4W 128	41259	5216	39538	16773

are protected and accesses are randomized (see Section 4.3.2).

RSA with povray, **omnetpp**, **xalancbmk** and **cactusADM**. In order to better see the performance impact on the whole system when a secure program is running, the RSA as discussed above, was run in parallel with each of the selected TLB-intensive SPEC benchmarks. The RSA continuously performs the decryption (50, 100 and 150 times), while the SPEC benchmark runs in background.

SecRSA with povray, **omnetpp**, **xalancbmk** and **cactusADM**. Same as above, but security is enabled for RSA, as discussed in SecRSA case.

Standard TLB Performance

Standard TLB's IPCs and MPKIs are shown in Figure 4.23a and Figure 4.23d. Larger TLB has smaller MPKI and better IPC. RSA routine is relatively small, so it experiences very few MPKIs. When SPEC benchmarks are included, the MPKIs increase and IPC drops. Interestingly, although cactusADM was specified as TLB-intensive in [109], it is not affected much by TLB size. Additionally, in most of the cases, IPC and MPKI give similar result for 50, 100 and 150 runs. This is the same for SP TLB and RF TLB.

Note, the *1E* configuration approximates no TLB scenario. This has on average 38.3%

worst performance, based on IPC. Thus, achieving security by disabling the TLB will impact system performance significantly.

Further, FA TLB (i.e. *FA 32* and *FA 128*) have as expected better performance than SA configurations, and can prevent 8 more types of attacks compared with 10 types that SA TLB can prevent. However, FA TLBs have area impact of about 0.6% more Slice LUTs for 32 entries, and 3.3% more Slice LUT for 128 entries, see Section 4.3.4. The FPGA runs slow enough at 50MHz for ZC706 and at 25MHz for ZedBoard that the impact of FA configuration on the critical path is not observed.

SP TLB Performance

Performance evaluation results for the SP TLB are shown in Figure 4.23b and Figure 4.23e. For the SP TLB, half the ways are set to the victim partition. When victim program RSA (SecRSA) is run alone or run with a SPEC benchmark, the secure data of RSA is allocated to half of the ways in the victim partition, and all other data and all SPEC data is in the attacker partition. Overall, IPC is about 0.5% better compared to standard TLB. This may be due to the system code getting invoked more often for the SP TLB, than for the standard TLB, and the system code may have better overall IPC. From the evaluation result, SP additions for the TLB do not influence IPC too much.

The MPKI is significantly higher than standard TLB (207.5% more or 3.07x), as again the effective TLB size is one half. Assignment of different number of ways for victim and attacker partitions, and its impact on performance could be further explored. Further, ideas of coalescing in TLBs [109] could be explored to improve the effective TLB size for victim and attacker partitions.

RF TLB Performance

Performance evaluation results for the RF TLB are shown in Figure 4.23c and Figure 4.23f. The IPC is about 1.4% higher compared to SA TLB. Again, RF TLB may involve even more system code, in which case a better IPC is derived. Meanwhile, comparing the corresponding configurations, the MPKI of RF TLB is about 64.5% better than SP TLB, while 9.0% worse than SA TLB. Thus, RF TLB provides both better performance and better security than

SP TLB, while being as good as standard TLB in performance. It has about 39.4% better IPC than disabling TLB (approximated by the *1E* configuration) while providing the same security level.

Area Overhead

We further evaluate the area overhead of the new secure additions. We use the number of Slice Look-Up Table (LUT), Slice Registers, Block RAMs and DSPs from the FPGA synthesis reports for the Xilinx ZC706 FPGA as a proxy for the area. For all the configurations, the Block RAM usage is 24 and the DSP usage is 15. Slice LUT and Registers numbers are shown in Table 4.11. The baseline is again 32-entry, 4-way SA L1 D-TLB.

Comparing to 4-way 32-set SA TLB, 4-way 32-set SP TLB has 0.4% more Slice LUTs and 0.1% more Slice Registers. 4-way 32-set RF TLB has 6.2% more Slice LUTs and 5.5% more Slice Registers. On average for all the configurations of TLBs, SP TLB has about 0.2% less Slice LUTs and 1.3% less Slice Registers compared with SA TLB, while RF TLB has about 6.5% more Slice LUTs and 7.9% more Slice Registers compared with SA TLB.

4.3.5 Soundness Analysis of TLB Vulnerabilities

In this section we analyze the soundness of the three-step model to demonstrate that the three-step model can cover all possible SA TLB timing vulnerabilities. If there is a vulnerability, it can always be reduced to a model that requires only three steps.

Let β denote the number of memory page related operations in a vulnerability.

When $\beta = 1$, there is only one memory page related operation, and it is not possible to create interference between memory page related operations since two memory page related operations are the minimum requirement for an interference. Furthermore, $\beta = 1$ corresponds to the three-step pattern with both *Step* 1 and *Step* 2 to be $\star$ since the TLB state $\star$ gives no information. These types are not listed in Table 4.9, which shows all the effective vulnerabilities. Therefore, attack cannot happen when $\beta = 1$.

When $\beta = 2$, it satisfies the minimum requirement of an interference for memory page related operations and corresponds to the three-step cases where *Step* 1 is $\star$. Three-step cases

where *Step* 1 is $\star$ does not have corresponding effective vulnerabilities shown in Table 4.9. So $\beta \neq 2$.

When $\beta = 3$, we exhaustively list all possible three-step memory page related operations in Section 4.3.1 and we conclude that there are in total 24 types of effective vulnerabilities, of which 16 are new compared to what is known in literature. So $\beta = 3$.

When $\beta > 3$, the pattern of memory related operations for a vulnerability can be deducted using the following rules:

- *Rule 1*: If there is a sub-pattern such as $\{ ... \rightsquigarrow \star \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where $\star$ is assigned as *Step* 1 of the second pattern. This is because $\star$ gives no timing information, and the attacker loses track of the cache state after $\star$. This rule should be recursively checked until there are no sub-patterns with a $\star$ in the middle or last step ($\star$ in the last step will be deleted).
- *Rule 2*: If there is a sub-pattern such as $\{ ... \rightsquigarrow A_{inv}/V_{inv} \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where A_{inv}/V_{inv} is assigned as *Step* 1 of the second pattern. This is because A_{inv}/V_{inv} will flush all the timing information of the current block and it can be used as the flushing step for *Step* 1, e.g., vulnerability $\{ A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a(fast)\}$ shown in Table 4.9. It cannot be candidate for middle steps or the last step because it will flush all timing information, making the attacker unable to correspond the final timing with victim's sensitive address translation information. This rule should be recursively checked until there are no sub-patterns with a A_{inv}/V_{inv} in the middle or last step (A_{inv}/V_{inv} in the last step will be deleted).
- *Rule 3*: If the remaining memory page related operations have a sub-pattern that has two adjacent steps both related to known addresses or both related to unknown address (including repeating states), the two adjacent steps can be reduced to only one.
  - For two unknown adjacent memory page related operations (containing u, denoted as $u\_operation$), although u is unknown, both of the accesses target on the same u so can be reduced. E.g., $\{... \rightsquigarrow V_u \rightsquigarrow V_u \rightsquigarrow ...\}$ can be reduced to $\{... \rightsquigarrow V_u \rightsquigarrow ...\}$.

Table 4.12: The 7 specific-address-invalidation-related states for a single TLB block.

States	Description
V_u^{inv}	The TLB block previously containing translation for a memory address u is now invalid. Attacker does not know u, but u is from a sensitive memory range x of memory locations, range which is known to the attacker. The address u may have same page index as a and thus conflict with them in the TLB block.
A_a^{inv} or V_a^{inv}	The TLB block previously containing translation for a memory address a is now invalid. The attacker knows the address a, independent of whether the access was by the victim or the attacker themselves. The address a is from within the range of sensitive locations x. The address a may or may not be the same as the address u.
$A_{a^{alias}}^{inv}$ or $V_{a^{alias}}^{inv}$	The TLB block previously containing translation for a memory address a^{alias} is now invalid. The address a^{alias} is within the sensitive memory range x. It is not the same as a, but it has same page index and maps to the same TLB block, i.e. it "aliases" to the same block.
A_d^{inv} or V_d^{inv}	The TLB block previously containing translation for a memory address d is now invalid. The address d is not within the sensitive memory range x.

  - For two known adjacent memory related operations (denoted as *not_u_operation*), these two operations result in a deterministic state of the cache block, so these two steps can be reduced to only one step. E.g., $\{... \rightsquigarrow A_d \rightsquigarrow V_a \rightsquigarrow ...\}$ can be reduced to $\{... \rightsquigarrow V_a \rightsquigarrow ...\}$.

  The *Rule 3* should be recursively checked until there are no two adjacent steps both related to known addresses or both related to unknown address, i.e., the resulting pattern must be of a format of *u_operation* and *not_u_operation* alternating.

- *Rule 4*: After recursive reductions of *Rule 1*, *Rule 2* and *Rule 3*, either $\beta \leq 3$ holds, or the following sub-pattern still exists:

  - $... \rightsquigarrow u\_operation \rightsquigarrow not\_u\_operation \rightsquigarrow u\_operation \rightsquigarrow ...$

  If the pattern contains this sub-pattern, the three-step sub-pattern must be an effective vulnerability according to Table 4.9 and reduction rules shown in Section 4.3.1. The corresponding pattern can be treated effective and the checking is done.

We make use of the four *Rule*s in a way either i) to reduce a β-step ($\beta > 3$) pattern to be within three steps or ii) demonstrate that the β-step pattern can be mapped to one or more three-step vulnerabilities if it is effective.

Table 4.13: The table shows additional possible timing TLB vulnerabilities when TLB invalidations are possible. The column headings are the same as in Table 4.9.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
TLB Internal Collision	A_a^{inv}	V_u	V_a (fast)	IH	(1)
	V_a^{inv}	V_u	V_a (fast)	IH	(1)
TLB Flush + Reload	A_a^{inv}	V_u	A_a (fast)	EH	**new**
	V_a^{inv}	V_u	A_a (fast)	EH	**new**
TLB Reload + Time	V_u^{inv}	A_a	V_u (fast)	EH	**new**
	V_u^{inv}	V_a	V_u (fast)	IH	**new**
TLB Flush + Probe	A_a	V_u^{inv}	A_a (slow)	EH	**new**
	A_a	V_u^{inv}	V_a (slow)	IH	**new**
	V_a	V_u^{inv}	A_a (slow)	EH	**new**
	V_a	V_u^{inv}	V_a (slow)	IH	**new**
TLB Flush + Time	V_u	$A_{a^{inv}}$	V_u (slow)	EH	**new**
	V_u	$V_{a^{inv}}$	V_u (slow)	IH	**new**
TLB Internal Collision Invalidation	A^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	V^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	A_d	V_u	V_a^{inv} (slow)	IH	**new**
	V_d	V_u	V_a^{inv} (slow)	IH	**new**
	$A_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
	$V_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
TLB Flush + Flush	A_a^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	V_a^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	A_a^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	V_a^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
TLB Flush + Reload Invalidation	A^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	V^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	A_d	V_u	A_a^{inv} (slow)	EH	**new**
	V_d	V_u	A_a^{inv} (slow)	EH	**new**
	$A_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
	$V_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
TLB Reload + Time Invalidation	V_u^{inv}	A_a	V_u^{inv} (slow)	EH	**new**
	V_u^{inv}	V_a	V_u^{inv} (slow)	IH	**new**
TLB Flush + Probe Invalidation	A_a	V_u^{inv}	A_a^{inv} (fast)	EH	**new**
	A_a	V_u^{inv}	V_a^{inv} (fast)	IH	**new**
	V_a	V_u^{inv}	A_a^{inv} (fast)	EH	**new**
	V_a	V_u^{inv}	V_a^{inv} (fast)	IH	**new**
TLB Evict + Time Invalidation	V_u	A_d	V_u^{inv} (fast)	EM	**new**
	V_u	A_a	V_u^{inv} (fast)	EM	**new**
TLB Prime + Probe Invalidation	A_d	V_u	A_d^{inv} (fast)	EM	**new**
	A_a	V_u	A_a^{inv} (fast)	EM	**new**
TLB Bernstein's Invalidation Attack	V_u	V_a	V_u^{inv} (fast)	IM	**new**
	V_u	V_d	V_u^{inv} (fast)	IM	**new**
	V_d	V_u	V_d^{inv} (fast)	IM	**new**
	V_a	V_u	V_a^{inv} (fast)	IM	**new**
TLB Evict + Probe Invalidation	V_d	V_u	A_d^{inv} (fast)	EM	**new**
	V_a	V_u	A_a^{inv} (fast)	EM	**new**
TLB Prime + Time Invalidation	A_d	V_u	V_d^{inv}(fast)	IM	**new**
	A_a	V_u	V_a^{inv} (fast)	IM	**new**
TLB Flush + Time Invalidation	V_u	A_a^{inv}	V_u^{inv} (fast)	EH	**new**
	V_u	V_a^{inv}	V_u^{inv} (fast)	IH	**new**

(1) Double Page Fault attack [50].

Algorithm 5 β-Step ($\beta > 3$) Pattern Reduction

Input: β: number of steps of the pattern
step_list: a two-dimensional dynamic-size array. *step_list*[0] contains the states of each step of the original pattern in order. *step_list*[1], *step_list*[2], ... is empty initially.
Output: *reduce_list*: reduced effective vulnerability pattern(s) array. It will be an empty list if the original pattern does not correspond to an effective vulnerability.

```
while step_list.contain(⋆) and ⋆.index not 0 do
    Rule_1 (step_list)
end while
while (step_list.contain(A_inv) and A_inv.index not 0) or (step_list.contain(V_inv) and V_inv.index not 0) do
    Rule_2 (step_list)
end while
while step_list.contain(adjacent not_u_operation or u_operation) do
    Rule_3 (step_list)
end while
reduce_list = Rule_4 (step_list)
return reduce_list
```

In conclusion, the three-step model can model all possible timing SA TLB vulnerability with any β steps. Attacks which are represented by more than three steps can be always reduced to one (or more) vulnerabilities from our three-step model; and thus, using more than three step is not necessary.

4.3.6 Additional Attacks

Table 4.13 shows additional possible timing TLB vulnerabilities when different types of TLB invalidations are possible, which are listed in Table 4.12. The translation can be "removed" from the TLB block by the victim or the attacker as the result of TLB block being invalidated, e.g., through a dedicated TLB flush instruction. We are unaware of existing RISC-V ISAs or systems which have such features, but future extensions may add such features and they could cause security bugs. For example, invalidating a specific TLB entry for some processors on Linux is possible by using *mprotect()* system call, which changes the access protection bits for the calling process's memory pages.

If it is possible for the attacker or the victim to trigger invalidation of a specific address or entry in the TLB, then attacks such as TLB Flush + Time become possible. Invalidation of a specific address or entry is needed in *Step* 2 to derive information about V_u in the last step of the pattern.

If invalidation of TLB can be for a specific address or entry and has variable timing, then attacks such as TLB Flush + Flush become possible. One performance improvement to TLB could be that for each invalidation, check the TLB first. If TLB entry is already invalid, the invalidation is done. If it is valid, then during the second cycle, update the TLB entry to mark it as invalid. This may shorten each cycle, but would introduce $fast$ or $slow$ timing differences that lead to the further attacks.

Chapter 5

Vulnerability Evaluation of Value Predictors

This chapter focuses on on side- and covert-channel attacks beyond caches and TLBs. Especially, this is the first work to focus on understanding a special type of predictor, the value predictor, and demonstrating new security attacks on these predictors. Although not implemented in the real hardware today, numerous value predictor architectures have been proposed and are being considered for inclusion in future processors. Since the original last value predictor [17], a number of improvements have been developed, e.g., [11], including recent work in the last two years, e.g., [12]. These value predictors have demonstrated to improve processor performance, however, as we show, they can contribute to new security vulnerabilities in systems that may realize them.

By exploring value predictor attacks (and defenses), this work fills in the missing understanding of the security of value predictors. Attacks and defenses should be analyzed at design time before new features, such as the value predictors, are added to real machines. It has been shown many times before [27, 31] that attacks are found due to features introduced without proper design-time security analysis.

5.1 Threat Model and Assumptions

We present the first threat model for analyzing value predictors. The model assumes there is a sender (victim) process that has access to a secret and a receiver (attacker) process which aims to learn the secret. Two processes can execute on the same core or different cores. Especially, in internal-interference attacks (which involve only the sender's accesses), two processes do not need to share the value predictor, as long as the receiver can observe timing differences in the execution of the sender (as affected by the value predictor's state or use of the value predictor). The attacker is assumed to know the source code of the victim process which is not secret by itself. The attacker further can trigger the value prediction by meeting the right condition, e.g., making $confidence$ number of accesses, or other condition used by the Value Prediction System (VPS).

We assume predictors can be broadly *PC-based predictor* (use the program counter, i.e., instruction address, for indexing the predictor's state) or *data-address-based predictor* (use the address of the accessed data to index the predictor's state) and we assume the address is a virtual address[1]. The address can also incorporate other information, such as a process identifier, *pid*, if the value predictor uses that for indexing the predictor's state. We use the term *index* to describe the information used to index the predictor's state. I.e., in PC-based predictors, the index is the PC plus any potential identifiers.

This work focuses on load-based VPS[2], where 1) training[3] or 2) modifying the value predictor state, or 3) triggering the value predictor to make a prediction, requires a cache miss. The miss is assumed to occur naturally as part of the code's execution or can be forced by a malicious attacker that invalidates or flushes the cache. Having value prediction at the frontend or in the execution stage of the pipeline will not influence the attacks we propose in this work, since the attack mechanism is independent of stages in the pipeline where the VPS is used, as long as the prediction can happen before the actual value is obtained.

1. Physical-address-based attacks are also possible, but we focus on attacks using virtual address as most value predictors we studied use virtual addresses.

2. Non load-based VPS is possible, where the attacks can be triggered without causing cache misses.

3. To train the predictor to make a prediction, we assume a *confidence* number of accesses are required. Thus, the predictor will output a first prediction on the $confidence + 1$ access.

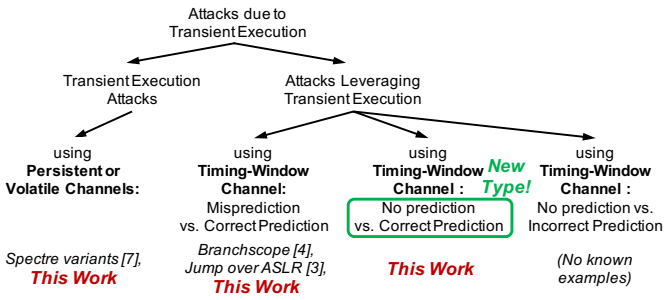


Figure 5.1: Taxonomy of timing-window microarchitectural channels. We are the first to present a no prediction vs. correct prediction timing attack.

5.2 Attack Taxonomy

Figure 5.1 shows the taxonomy of transient execution attacks and attacks leveraging transient execution. Attacks leveraging transient execution modify transient execution behavior based on the secret value. For this type, there are timing-window attacks that rely on misprediction vs. correct prediction timing, e.g., Branchscope [31], Jump over ASLR [32], or one of our attack variants. We also show a different attack variant that uses no prediction vs. correct prediction timing-window attack, which is a different, new type of attack. No prediction vs. incorrect prediction attacks theoretically exist but such types are not known. In addition, we also show value predictor attack variants that can be used with regular transient execution attacks as well.

5.3 New Value Predictor Attacks

In this section, we demonstrate two new proof-of-concept attacks on value predictors.

5.3.1 Train + Test Attack

The first new attack we present is the Train + Test attack, as is shown in Figure 5.2. In this attack, the attacker (receiver) is able to derive the value predictor index accessed by the victim (sender) during a load operation, and with the knowledge of the source code, the receiver can correlate the index accessed to the secret value 1 or 0 they are trying to learn. In this attack, first, the predictor is set to a known state in the train step by having a $confidence$ number of accesses to a known index. In the modify step a further $confidence$

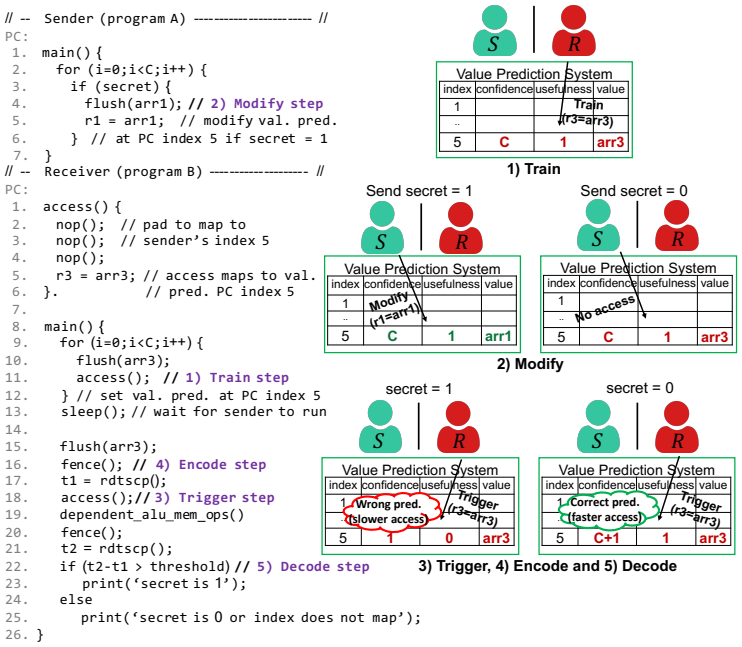


Figure 5.2: Proof-of-concept code and diagram of the value predictor's state for a new Train + Test attack presented in this work.

number of secret-related access can be made to set a new valid predictor state or 1 access can be made to cause the previously trained value to be invalidated. In the final trigger step, there is 1 memory access to a known index, as in the first step.

If modify step involves *confidence* number of accesses, as is shown in Figure 5.2, there will be a correct prediction in the last step if secret and known indices are different or *secret* is 0, since the predictor state was not modified by the middle modify step.[4] There will be a misprediction if indices are the same and the *secret* is 1 (predictor state was modified by the middle step that maps to the same index as the known index steps). If the modify step has 1 access and secret-dependent access by the sender maps to the same index as the known index access, it resets the *confidence* value to 0 and leads to no prediction in the last step. Otherwise, there will be a correct prediction as no modification of states in the middle step.

4. There can be a correct prediction also if the indices are the same and the secret data and known data happen to be the same. However, for index-focused attacks there is no assumption about data knowledge and the probability of this is approximately 1^{-64} for 64-bit data. Further, if this attack involves known data, it becomes equivalent to the Test + Hit attack.

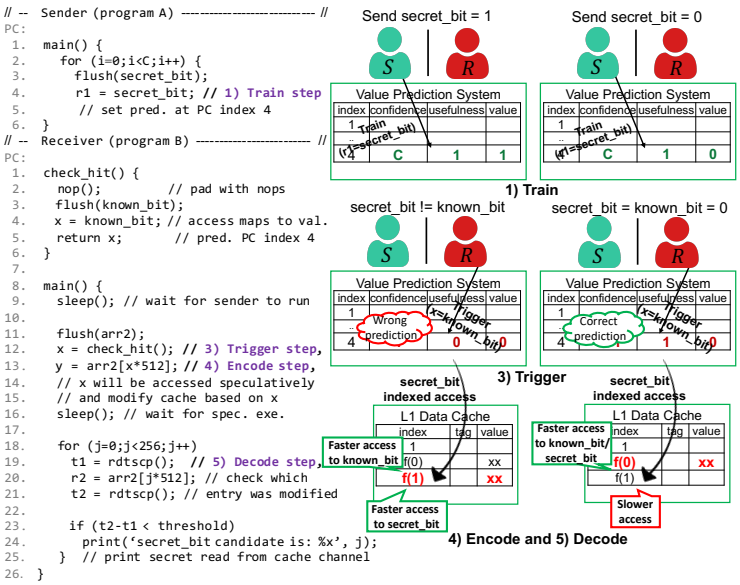


Figure 5.3: Proof-of-concept code and diagram of the value predictor's state for a new Test + Hit attack presented in this work.

5.3.2 Test + Hit Attack

The second attack we present is a Test + Hit attack, shown in Figure 5.3. In this attack, the receiver is able to derive the *secret_bit* value that was trained into the value predictor state by the sender's accesses. First, the sender accesses the secret at least a $confidence$ number of times to train the predictor. The receiver can for example force the sender to repeatedly execute the code that uses the secret value and causes it to be trained into the value predictor state. Next, the receiver makes access in the trigger step (no modify step is used in this attack) to a known data at the same index as the sender did. The access triggers the value predictor to make a prediction related to the secret value. When prediction occurs, during transient execution the output of the value predictor can be encoded into the persistent cache channel. Similar to Spectre attacks, array access is performed in Figure 5.3, where the index is the value from the value predictor. To recover the secret value from the cache channel, the receiver checks the timing of accessing the array elements, to learn which one was previously placed into the cache and thus recover the secret.

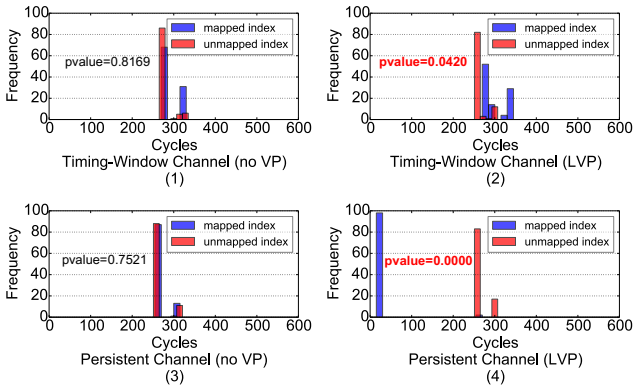


Figure 5.4: Timing distribution results of Train + Test attacks using timing-window channel (1-2) and persistent channel (3-4). Red *pvalue* means the related attack is effective, while black means it is not effective.

5.3.3 Experimental Setup for Evaluation

To evaluate new value predictor attacks, we implemented value predictors in a modified `gem5` simulator [110], and run the code on the simulator. The `gem5` simulator was used in *syscall emulation* mode (SE) with the O3CPU model and Ruby cache for testing. We implemented a baseline (non-secure) LVP [17] predictor. and an oracle VTAGE [111]. The oracle value predictor makes predictions only for the target load instruction to maximize the attacker's advantage. To judge whether an attack is successful, we report averages over 100 runs for each attack, with a 95%-confidence interval [88] calculated using the Student's t-test [112] to distinguish measured timing distributions.

5.3.4 Attack Evaluation and Results

For our evaluation, we focus on analyzing if the receiver can distinguish two types of timings – "mapped" vs. "unmapped" cases – as explained below. We use *pvalue* calculation result to determine if two types of timings can be distinguished. If the *pvalue* is smaller than 0.05, timing distributions are differentiable and the attack succeeds.

Train + Test Attack Results. For the timing-window channel, when the secret and known indices map to each other and *secret* is 1, misprediction leads to longer timing in the trigger step. Meanwhile, there will be a correct prediction in the trigger step when not mapped, since the predictor state set in the train step was not modified.

We also evaluate a persistent channel variant of this attack, where mapped case means

```
1. void _gcry_mpi_powm (gcry_mpi_y_ res,
2.           gcry_mpi_t base, gcry_mpi_t expom gcry_mpi_t_mod)
3. {
4.      mpi_ptr_t rp, xp; /* pointers to MPI data */
5.      mpi_ptr_t tp;
6.      ...
7.      for(;;) {
8.           /* For every exponent bit in expo*/
9.           _gcry_mpih_sqr_n_basecase(xp, rp);
10.          if(secret_exponent || e_bit_is1) {
11.             /* unconditional multiply if exponent is
12.              * secret to mitigate FLUSH+RELOAD
13.              */
14.             _gcry_mpih_mul(xp, rp);
15.          }
16.          if(e_bit_is1) {
17.              /*e bit is 1, use the result*/
18.              tp = rp; rp = xp; xp = tp;
19.              rsize = xsize;
20.          }
21.     }
22. }
```

Figure 5.5: Code of modular exponentiation from *libgcrypt*, adapted from [48]. The highlighted red-colored part shows conditional access to the `tp` index which can be leaked through value predictor attacks we present.

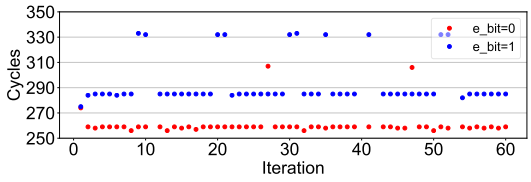


Figure 5.6: Sequences of the receiver's observation for each iteration when the `e_bit` is 0 or 1 (line 16 shown in Figure 5.5).

two indices are the same and *secret* is 1, resulting in misprediction in the trigger step, which encodes the data into the cache, and a cache hit is observed in the reload part of the covert channel. Otherwise, the unmapped case results in a cache miss.

In Figure 5.4 (1) and (3) it can be seen that without value predictor (no VP), different timing distributions cannot be distinguished, and the attacks are not possible. Meanwhile, in Figure 5.4 (2) and (4) it can be seen that when (non-secure) LVP value predictor is enabled, timing distributions for mapped and unmapped cases are different, and the secret value can be leaked.

Our proof-of-concept code can be extended to real applications. For example, Figure 5.5 shows the RSA related portion of *libgcrypt* code with conditional memory access. The code is already protected against Flush + Reload cache timing attacks [27]. However, when the value predictor is trained through repeated accesses (due to repeated invocations of the code with the same cryptographic key), the index of the `tp` access can be leaked through the value predictor attack, leaking the value of `e_bit_is1` as is shown in Figure 5.6. The success rate

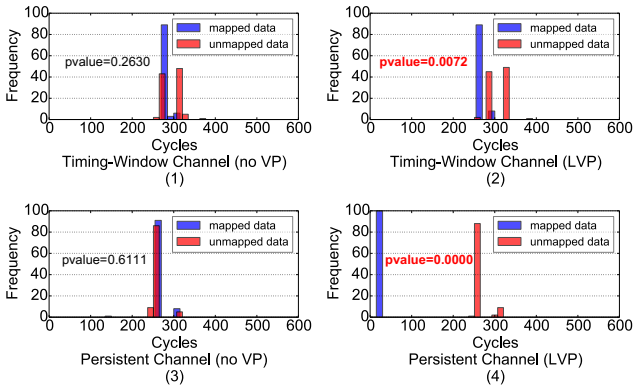


Figure 5.7: Timing distribution results of Test + Hit attacks using timing-window channel (1-2) and persistent channel (3-4). Red *pvalue* means the corresponding attack is effective, while black means it is not effective.

of correctly transmitting `e_bit` is 95.7% for 60 runs, which is enough to reconstruct the full key based on prior work [48]. Without further optimization, the transmission rate is 9.65Kbps.

Our attack is demonstrated on RSA modular exponentiation which does not use blinding techniques. We do this because most related work in architecture does not consider blinding and we target similar code for easier comparison. There are blinding techniques for both RSA and ECC, and we leave this for future work and do not discuss blinding here. However, we expect that a variant of our value prediction attack actually works if the blinding scheme is used. If the secret is accessed by a load or similar instruction during the blinding operation, we can use value prediction to extract the secret (it is not possible to extract the blinding factor, as it is random each time, while the secret is constant and gets trained into the value predictor). This type of attack is not possible with branch predictor or cache side channels but is possible to value predictor attacks.

Test + Hit Attack Results. For the timing-window channel, mapped data means that the data accessed in the train step and the trigger step are the same and a correct prediction can be derived in the trigger step (faster timing), while in the unmapped case the two data are different (slower timing). For the persistent channel, the mapped case means the secret value encoded by the train step is brought to the cache, causing reloading a fast timing. Therefore, the secret value can be observed through cache hits in the cache channel. Otherwise, only cache misses can be observed for the unmapped case.

As is shown in Figure 5.7, when no value predictor is used, there is no attack due to no significant difference in the timing distributions. Meanwhile, with (non-secure) LVP enabled, the mapped and unmapped timing distributions are significantly different.

Value Predictor Type Influence. We further evaluate whether the type of value predictor, e.g., LVP vs. VTAGE, has impacts on the attacks. For both predictor types, timing distributions between mapped and unmapped cases are significantly different to leak data. The VTAGE data and details are left for future work.

5.4 Derivation of All Expected Value Predictor Attacks

Based on the two proof-of-concept attacks, we further present the first model for analyzing value predictor security. The model further points to additional attack types. The model is based on exploring all possible steps that the victim or the attacker can perform to affect or observe the value predictor state, and how that can leak information.

1) Train Step: In this step, the value predictor is trained using load access at a certain index, to set up a deterministic predictor state for the PC's or data address's prediction entry. This step can be secret-related if performed by the sender who is the only one with logical access to the secret. In this case, this step is used to put the secret-related data into the predictor state so it can be revealed by other steps. Otherwise, this step is used to provide a known reference state that can be later used to derive secret information by observing state changes, which can be performed by the sender or the receiver.

To train the value predictor, the loads usually need to be accessed at least $confidence$ number of times to set the predicted state. However, for certain attacks, the access is made $confidence - 1$ number of times, so that the access in the next modify step can be detected if it pushes the total accesses to the $confidence$ number of times, and the prediction is triggered to output a value during a cache miss.

2) Modify Step: In some attacks, the modify step is needed to alter the value predictor's state set in the first step before an observation is made. This step is useful if the first step was to known data or index, and a state modification (due to secret-related access) is needed to observe potential interferences. This step is also useful if the first step is secret-related,

Table 5.1: Possible actions for each step of value predictor attacks.

Action	Description
S^{KD}, S^{KI}	Sender makes access to data, or respectively index, that it knows.
R^{KD}, R^{KI}	Receiver makes access to data, or respectively index, that it knows.
$S^{SD'}$, $S^{SD''}$	Sender makes access to secret data the receiver tries to learn. For attacks leveraging interference between sender's accesses, secret data D' and D'' may or may not be the same, which is what the receiver is trying to learn. For attacks that involve known data, the goal is to learn if the known D is or is not the same as the secret.
$S^{SI'}$, $S^{SI''}$	Sender makes access to secret-dependent index the receiver tries to learn. For attacks that involve the known index, the goal is to learn if the known I is or is not the same as the secret index.
—	This step is not used, this is only for the modify step for attacks that do not have any actions in the modify step.

and the state modification is due to another (possibly the same) secret-related access, or due to a known data or index access. For this step, most attacks will repeatedly execute load more than $confidence$ number of times to encode the value into the predictor's state. However, for some attacks, only 1 extra access in this step is needed if the train step uses $confidence - 1$ accesses.

3) Trigger Step: A single access is required in this step to probe the value predictor to observe the interference that can reveal the secret, or even directly observe the secret through timing variations.

4) Encode and 5) Decode Step: The sensitive information obtained from predictor states needs to be encoded into a channel to exfiltrate the information. This can be a persistent channel (e.g., cache channel), a volatile channel (e.g., port contention channel), or a timing-window channel (e.g., directly measure the timing of the load access and subsequent instructions). Depending on the types of three possible channels used, related ways are used to decode the secret.

5.4.1 Modeling Results

To understand possible attacks, we consider the first three steps, as the last two steps are about exfiltrating the information, and are not specific to value predictors. The first three steps can be performed by the sender S or the receiver R. The possible actions in each step are shown in Table 5.1. With these actions, there are in total 576 possible three-step combinations for the train, modify, and trigger steps: 8 step types for train step (S^{KD}, S^{KI}, R^{KD}, R^{KI}, $S^{SD'}$, $S^{SD''}$, $S^{SI'}$, and $S^{SI''}$) $\times$ 9 step types for modify step (the same as

train step plus "—" step) × 8 step types for trigger step (the same as train step) = 576 combinations. However, the majority of these 576 combinations do not represent attacks or can be reduced to simpler patterns. We define rules to determine if a pattern corresponds to a possible attack, and eventually show that there are exactly effective attacks, as discussed in Section 5.4.2. Rule description and soundness analysis of the model are not detailedly discuessed here.

5.4.2 Value Predictor Attack Variants

Following our analysis, there are value predictor attack variants. The attacks are summarized below and shown in Table 5.2. If the predictor indexing function uses *pid* or another identifier, and two known data or index steps are done by different processes (not both S nor both R), then the known data or index has to come from the shared library so both can access the same index. However, if the index is just based on the address and no *pid* (as is the case of many known value predictors [111, 18, 11, 12]), then no shared library is needed.[5]

Train + Test. Details of this attack were given in Section 5.3.

Test + Hit. Details of this attack were given in Section 5.3.

Train + Hit. This is a two-step attack where in the train step the predictor state is first set by the $confidence$ number of accesses to a known data. Next, 1 secret-related access is made. Correct prediction makes execution faster which shows that the known data is the same as secret-related data, otherwise execution is slower and two data are different. A timing-window channel can be used to observe the timing difference of correct prediction vs. misprediction.

Spill Over. This type of attack aims to determine if two secret-related states are the same or different. First, $confidence - 1$ number of accesses are made to secret data. Next, 1 access is made to possibly the same or different secret data. Finally, in the trigger step 1 access is made to the same secret data as in the first step. The last step will be predicted correctly if all the secrets accessed are the same. Otherwise, the $confidence$ value is not reached (since the middle step accesses a different value) and there is no prediction. A

5. Using *pid* only increases difficulties for attacks but does not eliminate it.

Table 5.2: List of value predictor attacks and attack categories that each attack belongs to. Each step is explained in Section 5.4.1. Each attack category is explained in Section 5.4.2.

Step 1 (Train)	Step 2 (Modify)	Step 3 (Trigger)	Attack Cateogry
S^{KD}	—	$S^{SD'}$	Train + Hit
S^{KI}	$S^{SI'}$	S^{KI}	Train + Test
S^{KI}	$S^{SI'}$	R^{KI}	Train + Test
R^{KD}	—	$S^{SD'}$	Train + Hit
R^{KI}	$S^{SI'}$	S^{KI}	Train + Test
R^{KI}	$S^{SI'}$	R^{KI}	Train + Test
$S^{SD'}$	$S^{SD''}$	$S^{SD'}$	Spill Over
$S^{SD'}$	—	S^{KD}	Test + Hit
$S^{SD'}$	—	R^{KD}	Test + Hit
$S^{SD'}$	—	$S^{SD''}$	Fill Up
$S^{SI'}$	S^{KI}	$S^{SI'}$	Modify + Test
$S^{SI'}$	R^{KI}	$S^{SI'}$	Modify + Test

Table 5.3: Value predictor attack evaluation for all the attack categories. Red *pvalue* means the corresponding attack is effective with transmission rate shown, while black means it is not effective. *Tran. Rate* is the transmission rate, or bandwidth, of the attack.

Attack Category	Timing-Window Channel		Persistent Channel	
	No VP	VP (Tran. Rate)	No VP	VP (Tran. Rate)
Train + Hit	0.1620	**0.0086** (7.72Kbps)	—	—
Train + Test	0.8169	**0.0420** (7.38Kbps)	0.7521	**0.0000** (6.88Kbps)
Spill Over	0.2989	**0.0000** (8.12Kbps)	—	—
Test + Hit	0.2630	**0.0072** (7.81Kbps)	0.6111	**0.0000** (7.43Kbps)
Fill Up	0.3734	**0.0083** (8.22Kbps)	0.4677	**0.0000** (6.85Kbps)
Modify + Test	0.2966	**0.0000** (8.00Kbps)	—	—

timing-window channel can show the timing difference of correct prediction vs. no prediction, to learn if the values are the same or not.[6]

Fill Up. This two-step attack has a $confidence$ number of accesses to secret data in the train step and 1 access to secret data in the trigger step, which is possibly the same or different secret. The last step has correct prediction if two secrets match, otherwise a misprediction is derived. A timing-window channel shows timing differences of correct and incorrect prediction to learn the secret. The secret can also be extracted from transient execution using a persistent or volatile channel since the predictor is trained on the secret.

Modify + Test. This is a flipped image of the Train + Test attack. First, a $confidence$

6. If all the data are the same, the secret value can be itself extracted from transient execution in the last step, but this reduces to the Fill Up attack since $confidence - 1$ plus 1 access add up to $confidence$ accesses captured by the train step in the Fill Up attack. Further, this is a weaker version of Fill Up attack, since it only leaks data if all the accesses are the same secret, while Fill Up leaks data in all the cases using a persistent or volatile channel.

number of secret-related accesses is performed in the train step. In the modify step there are $confidence$ accesses or 1 access to a known index to change or invalidate the predictor state, respectively. In the trigger step, there is 1 secret-related index access. A timing-window channel shows the timing difference when correct vs. incorrect prediction is made or when correct vs. no prediction is made.

All the attack variants discussed above can use a timing-window channel to observe the timing difference due to prediction states. Further, 2) Train + Test, 4) Test + Hit, and 5) Fill Up can use a persistent or volatile channel to extract secret data from transient execution since the predictor is trained on the secret before the trigger step. Evaluation results of all the attack categories are shown in Table 5.3, which proves the effectiveness of all the attack variants.

5.5 Secure Value Predictors

Security defenses such as InvisiSpec [61] can prevent existing transient execution attacks, but have not considered value prediction in particular, and are not effective against our new attacks. Consequently, we present value-predictor-specific defenses which shows an estimation of possible defense value predictors can consider.

5.5.1 Defense Techniques

Always predict a value (A-type) defense makes the predictor always predict the value based on a fixed value or on a history value regardless of whether confidence level is reached or not. In this case, the attacks based on differentiating from prediction vs. no prediction timing are protected. Delay side-effects (D-type) defense targets delaying the microarchitectural state changes and can only be used for preventing value predictor attacks based on persistent channels. Randomly predict a value (R-type) defense randomly predicts a value out of a window around the actual accessed value. Assuming the window size is S, the rate of randomly predicting the correct value is 1/S.

5.5.2 Defense Strategies Evaluation

When all the A-type, D-type, and R-type defenses are combined, all attacks we have considered can be defended. Note that R-type defense has a (predictable) probability of attacker learning the correct value based on the window size. This probability can be made arbitrarily small at some cost to performance.

The *Train+Test attack* can be prevented as long as the R-type defense is applied. D-type defense is effective only against the persistent-channel variant of Train+Test attack but not others. The *Modify+Test attack* can be prevented when the R-type defense is applied as well. The *Test+Hit attack* can be prevented by combining both A-type and R-type defense. D-type defense is effective against only the persistent-channel type of Test+Hit attack. The *Train+Hit attack* can be prevented by combining both A-type and R-type defense. The *Spill Over attack* can be prevented by the A-type defense directly. The *Fill-Up attack* can be prevented by R-type defense.

For the Train+Test attack, it will be prevented as long as the R-type defense is applied, and optionally D-type can be applied for preventing persistent channel variants. We evaluated the influence of the window size and found that a window size of 3 is the minimal size for this type of attack to guarantee security (p-value larger than 0.05 in our experiments), while at the same time maintaining the performance. Since Train+Test differentiates correct prediction vs. misprediction, A-type defense will not work, so this defense is not helpful in this case. For attack variants that use a persistent channel, D-type defense can be used.

For the Test+Hit attack, combining A-type and R-type defense can prevent the attack. For the R-type defense, in our experiments, on Test+Hit attack, the evaluation shows that window size of 9 is the minimal size for this type of attack to guarantee security (p-value larger than 0.05). This will cause large degradation to the performance. Therefore, a smaller window size is selected to maintain performance and partial security, e.g., size of 5. In addition, adding A-type defense is required to assist in fully preventing the attacks. D-type defense can be used for the persistent channel type of Test+Hit attack, but by itself will not defend non-persistent-channel attack types, so both A-type and R-type should be used.

Chapter 6

Processor Frontend Attacks

This chapter expands the security analysis to the processor frontend. Compared to previous work [51, 52], we are able to 1) present both eviction-based and misalignment-based attacks that leverage the Decode Stream Buffer (DSB), Loop Stream Detector (LSD), and Micro-Instruction Translation Engine (MITE), 2) show new power attacks, 3) evaluate SGX attacks, 4) analyze LSD influence, 5) use frontend behavior for microcode patch fingerprinting, 6) analyze instruction prefixes causing switching in the frontend paths for new attacks, and 7) present a new side-channel attack that identifies victim application type. The work presented in this chapter complements existing work by providing new attacks and security insights, including, to the best of our knowledge, fastest frontend attack reaching 1.4Mbps.

6.1 Threat Model and Assumptions

We assume there is one sender (victim) that holds security-critical information and one receiver (attacker) that tries to extract the secret information by measuring timing or power changes. For covert-channels, the sender and receiver cooperate and modulate usage of the DSB, LSB, and MITE to achieve the covert transmission. For side-channels, the attacker performs operations to interfere with the victim or monitor power or timing, while the victim is unaware of the attacker and operates on sensitive data. Our attack on SGX assumes that the attacker can trigger execution of the enclave and measure its timing or power. Our Spectre attack assumes an in-domain attack scenario: the attacker is within same thread,

Table 6.1: Specifications of the tested Intel CPU models.

Model	Gold 6226	Xeon E-2174G	Xeon E-2286G	Xeon E-2288G
Microarchitecture	Cascade Lake	Coffee Lake		
Core Number	12	4	6	8
Thread Number	24	8	12	8[a]
L1D Configuration	32KB, 8-way, 64 byte line size, 64 sets			
DSB Configuration	8-way, 32 byte window, 32 sets			
LSD Entries	64	— [b]	— [b]	64
Frequency	2.7GHz	3.8GHz	4.0GHz	3.7GHz
OS	18.04 Ubuntu			
SGX Support	No	Yes		

[a] We use Xeon E-2288G on Microsoft Azure cloud, this processor model is specific for Microsoft Azure and has hyper-threading disabled, although hyper-threading is supported by other E-2288G processors. [b] LSD is disabled in these machines.

e.g., as a sandboxed code where the disclosure gadget is executed. Our fingerprinting attack assumes attacker has prior access to the same CPU as the target one, so they can measure frontend performance under different microcode patches. All of the timing attacks can be performed fully from the user-level privilege using the `rdtscp` instruction for measuring timing. The power channels require access to Intel's RAPL [19] to get energy information. Even if the RAPL access is disabled for user-level code, privileged code can still use the power channels against SGX enclaves, for example.

6.2 Analysis of the Operation of the Frontend

Ensuring Observability of Frontend Timing

To achieve high backend throughput so that the frontend is the bottleneck, we do not want to touch data-related operations such as load and store because memory system may cause unpredictable timing differences, which are not due to frontend path changes. Load and store operations would also likely leave traces in the caches which may make any attacks more detectable. Based on our analysis, instruction mix sequence which maximizes the timing signature of the frontend for our attacks should satisfy the following three requirements:

- Total bytes of one access block should not exceed a 32 byte window (e.g., 4 *mov* and 1 *jmp* use in total 25 bytes).
- Total micro-op number should not exceed 6 micro-op limit that DSB can process by one DSB line (e.g., 4 *mov* and 1 *jmp* are decoded to total 5 micro-ops).

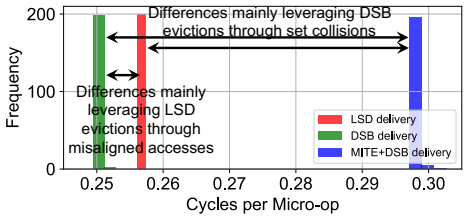


Figure 6.1: Example time histogram of Intel Xeon Gold 6226 processor of using LSD, DSB, or MITE+DSB paths. Timing difference between LSD/DSB and MITE+DSB are used for collision-based attacks (see Section 6.3.1) and differences between LSD and DSB paths are used for misalignment-based attacks (see Section 6.3.2).

- Avoid port contention. The 4 *mov* instructions exploit the ports as much as possible, plus 1 *jmp* instruction to end the cache line block, while avoiding load, store, or more complex instructions involved, which will cause influence or noise from other microarchitectural units.

As the result, 4 *mov* plus 1 *jmp* sequence is the *instruction mix block* which fits the requirement. Other instruction mix blocks are possible, although finding sufficient type and number of instruction mix blocks in real code may be a limitation of the proposed attacks.

Exploiting Frontend Path Timing Differences

As can be seen from histogram of Intel Xeon Gold 6226 processor shown in Figure 6.1, the timing difference of processing instruction mix blocks using DSB vs. MITE+DSB or LSD vs. DSB are clearly visible. In our attacks discussed later, we will use DSB vs. MITE+DSB timing differences to perform attacks related to DSB evictions through set collisions. On the other hand, the timing difference of processing using LSD vs. DSB will be used to perform attacks related to LSD evictions through misaligned accesses. Both of these also have power differences that separately can be used for power-based attacks.

Generating DSB Evictions Through Set Collisions

To force frontend path changes, we set up a series of instruction mix blocks and align the start of the instruction address of each block to map to the same DSB set, as shown in Figure 6.2. We make the *jmp* instructions at the end of each instruction mix block jump to the first instruction of next instruction mix block. In this case, executing the first *mov*

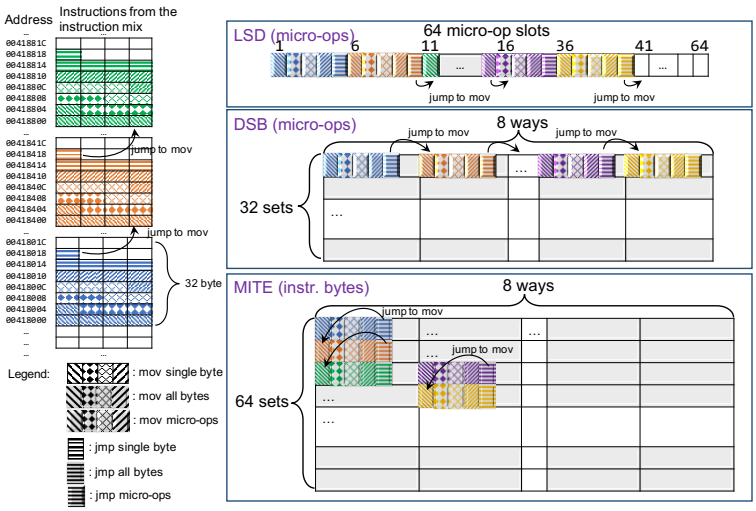


Figure 6.2: Example of mapping instruction mix blocks (Section 6.2) to MITE, DSB, and LSD. Each instruction mix block is 5 micro-ops (4 *mov* plus 1 *jmp*). If the number of chained 5 micro-op blocks is 8 then all will fit in LSD (since $8 \times 5 = 40 < 64$ micro-op limit of LSD) and they can all map to the same DSB set (since DSB is 8-way associative).

instruction of the first instruction mix block will trigger a series of instruction mix block execution. If the chain of instruction mix blocks is less than 12, all the blocks should fit in LSD. However, at the same time, each DSB set has 8 ways, so 8 blocks can map to same set. Consequently, if the chain of blocks is set to 8 (rather than 12), they can both fit in LSD and same DSB set. But, as soon as the chain is extended to 9 (or more) blocks that map to same set, eviction occurs in DSB, and in turn force LSD eviction due to inclusive nature of MITE, DSB, and LSD.

Inclusive feature of MITE, DSB, and LSD makes eviction of lines from DSB to cause flush of the LSD unit. Furthermore, eviction from DSB redirects micro-ops to be processed by MITE. Combing these, eviction from DSB will cause transition of micro-op delivery from LSD to both DSB and MITE.

Note that changing the chain of instruction mix blocks from 8 to 9 will not cause eviction or misses of L1 instruction cache. L1 instruction caches for the machines we tested are 8-way associative and contain 64 sets of 64 bytes. Consequently, the size of the L1 instruction is 4 times of DSB and instruction mix blocks mapping to the same DSB set will be mapped to different L1 instruction cache sets, as is shown in Figure 6.2. Changing chain length from 8 to 9 causes DSB and LSD eviction, but causes no misses in the L1 instruction cache.

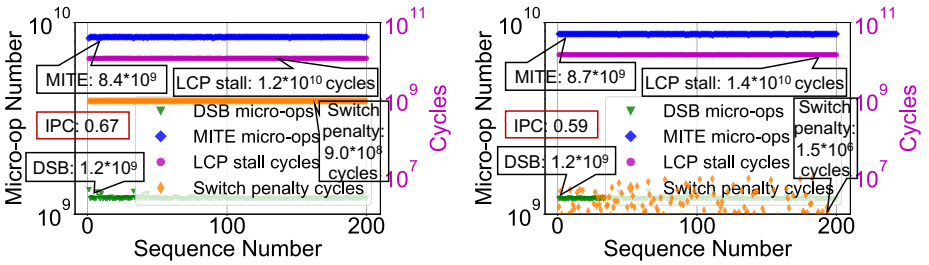


Figure 6.3: Intel Xeon Gold 6226 CPU performance counter readings for the different experiments with ordered-issued or mixed-issued types of *add* instructions. The numbers in the call-out boxes are the average micro-ops numbers for all the 200 rounds of experiments.

Generating LSD Evictions Through Misaligned Accesses

We further found that misaligned instructions will generate collisions in the LSD, even when the number of total accessed instruction mix blocks does not exceed the DSB way number. This can be achieved by setting up the initial addresses of instruction mix blocks to be misaligned, e.g., by aligning them on 16 byte boundaries that are not multiple of 32 bytes.

The alignment or misalignment of the blocks will cause different frontend path changes when processing micro-ops. When all the instruction mix blocks are misaligned, executing 4 chained instruction mix blocks that map to the same DSB set will trigger collisions in LSD which causes the micro-op delivery change from LSD to DSB. At the same time, as we discussed in Section 6.2, executing 4 chained aligned instruction mix blocks that map to the same DSB set will use LSD unit since the size of the 4 blocks (of 5 micro-ops each) is less than 64 micro-op limit of the LSD.

When considering accessing pattern, if accessing a chain of 7 instruction mix blocks which are all aligned, the 8^{th} access will determine the path used. If the 8^{th} access is aligned, all of the micro-ops will still be processed by the LSD. While if the 8^{th} instruction mix block is misaligned, LSD will be flushed and micro-ops will be redirected to use DSB in the frontend. We found that {aligned + misaligned} instruction mix block access pairs that will cause micro-ops to be changed from the LSD to the DSB paths are: {5 aligned + 2 misaligned}, {6 aligned + 2 misaligned}, {3 aligned + 3 misaligned}, {4 aligned + 3 misaligned}, and {5 aligned + 3 misaligned}. Similar to DSB evictions, misalignment will not cause L1 instruction cache misses.

Generating Different DSB Switch Penalties

In x86, Length Changing Prefixes (LCPs) are designed and incorporated into the x86 ISA to identify the instructions with non-default length, which may be used, e.g., with unicode processing and image processing [19]. For example, an instruction starting with *0x66h* prefix means there would be an operand size override. Such prefix can force CPU to use slower decoding MITE path and incur up to 3 cycles more penalty in addition to extra DSB-to-MITE switch penalty.

To demonstrate that generating different switch penalties is feasible, we set up two instruction mix blocks. The first instruction mix block is filled by 16 sets of two *add* instructions, with one normal *add* instruction followed by one *add* instruction with length changing prefixes (mixed issue), and repeating this alternating pattern to the end. The second one is filled with 16 normal *add* instructions followed by 16 *add* instructions with length changing prefixes (ordered issue), In both cases there are `32` instructions within the loop and we iterate the loop for `800` million times. Figure 6.3 shows the results of the measurement. The two instruction blocks generate similar number of micro-ops from MITE and DSB, but with detectable difference in the final performance (measured in instructions per cycle, or IPC), which is caused by different numbers of LCP stall cycles and DSB-to-MITE switch penalty cycles. This shows that the same type of frequently-used instructions can come with different front-end path switching penalties.

We also found other possibly useful, for an attacker, LCP behaviors including: a) use of LCP will force the front-end to switch from issuing instructions from DSB to issuing instructions from MITE, b) LCP instructions are only decoded sequentially and would incur measurable performance difference. Therefore, it is feasible to establish a covert channel based on instructions with LCPs.

6.3 Processor Frontend Vulnerabilities

In this section, we focus on implementation of the timing covert channels and attacks. Evaluation of the timing channels is in Section 6.4. Power-based channels and attacks are discussed in Section 6.4.6. Meanwhile, application of the attacks to SGX enclaves is

presented in Section 6.5, and for use with Spectre attacks in Section 6.6. Detection of the microcode patches, which can use both timing or power, is presented in Section 6.7. Finally, a new side-channel attack used to fingerprint applications is in Section 6.8.

Our timing covert-channel attacks can be differentiated based on the techniques used to covertly send different bits by switching between different frontend paths: using eviction (following ideas in Section 6.2), using misalignment (following ideas in Section 6.2), or using LCP stalls and DSB-to-MITE switch penalties (following ideas in Section 6.2).

For our attacks, there are generally three steps that the attacks follow:

- **Init Step**: A series of instruction accesses are performed in this step to set the micro-ops into certain frontend paths, for some attacks no initial step is needed, only start timing (or power) measurements.
- **Encode Step**: The sender accesses certain instructions to change frontend paths of micro-ops previously set in the initialization step according to the secret bit to be sent.
- **Decode Step**: The receiver accesses certain instructions and, depending on attack type, timing or power is measured to observe what changes occurred in the frontend, or for all three steps.

In addition, some of the attacks may require timing or power measurements in not just the last step, but the attacks still follow the three-step pattern.

In the attack descriptions we use the following variables to describe parameters of the system: N is the number of ways in the DSB. m is a 1-bit message to be transferred on the channel. d is the different number of instruction mix blocks used for an attack step, $d < N + 1$. M, only used for misalignment-based attacks, is the parameters of the receiver, $M < N + 1$. p is the number of iterations the receiver runs for initialization step and also for decoding step. q is the number of iterations the sender runs for encoding step. We note that use of multiple iteration increases time, but helps to reliably observe timing result with a low error rate. r, only used for attacks leveraging LCP, is the number of LCP instructions.

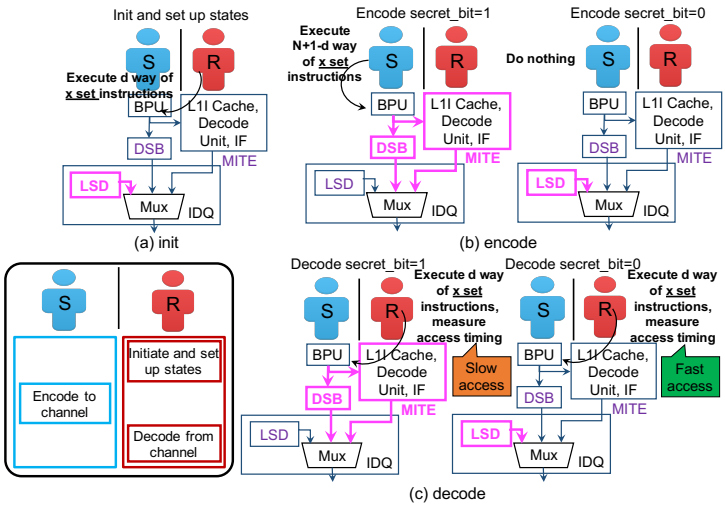


Figure 6.4: Overview of the MT Eviction-Based Attack.

6.3.1 Eviction-Based Timing Attack with Multi-Threading

For the eviction-based attack, in a multi-thread (MT) setting, we deploy a sender thread and a receiver thread on the same physical processor core, but different hardware threads, which causes them to share the frontend. When the instruction stream from the sender executes, the DSB will be partitioned and some of the receiver's instructions will be evicted from DSB, further triggering eviction from LSD so that the delivery of instructions falls back from the LSD to DSB+MITE, therefore generating detectable timing signature that the receiver can measure. When the instruction stream from the sender is not executing, the receiver thread will use whole DSB and the evictions will not happen. This process leaves no interference in traditional instruction and data caches.

In the MT Eviction-Based Attack, the sender and the receiver use in total $N + 1$ instruction mix blocks, denoted as lines 0 - N. For the MT eviction-based attack shown in Figure 6.4, in the Init Step, d ($d \leq N$) instruction mix blocks that map to a DSB set x are accessed for p times by the receiver. In the Encode Step, the sender will execute different instruction series according to the secret bit m. When sending $m = 1$, the sender will execute $N + 1 - d$ instruction mix blocks q times, these blocks map to DSB set x. In this case, the total number of ways accessed is larger than N, which causes eviction of DSB within the receiver and directs the micro-op delivery from LSD to DSB and MITE. When sending $m = 0$, the sender does nothing. In the Decode Step, the receiver will access the

same d instruction mix blocks accessed in the Init Step and time the Decode Step's access for p iterations. If eviction occurs, receiver's micro-ops in the Decode Step will be delivered from DSB and MITE, where longer timing will be measured, indicating message $m = 1$ was sent from the sender. On the other hand, if no evictions happen, receiver's micro-ops in the Decode Step will still be delivered from LSD, where much shorter timing is observed compared to the MITE+DSB path, indicating message $m = 0$ was sent from the sender.

For example, take $d = 6$ and $N = 8$, the instruction access sequences when sending $m = 1$ and $m = 0$ are as follows:

- **Init**: access blocks $1 - 6$ mapping to set x
- **Encode**: access blocks $7 - 9$ mapping to set x (if $m = 1$); no access (if $m = 0$)
- **Decode**: access blocks $1 - 6$ mapping to set x (if $m = 1$, DSB and MITE are used; if $m = 0$, LSD access is used)

6.3.2 Misalignment-Based Timing Attack with Multi-Threading

To achieve misaligned instruction access, sender and receiver first find virtual addresses of instructions that map to the same target set as what eviction-based attacks do, and then offset the initial address of every instruction mix block by 16 bytes (half of the DSB line size), to misalign the address.

For this type of attack, the total number of instruction mix blocks of the sender and the receiver are equal to or less than the N ways of the DSB, which has an advantage as it reduces the number of accesses and increases the transmission rate compared with eviction-based attacks.

The MT Misalignment-Based Attack is shown in Figure 6.5. Here, the sender and the receiver use in total M ($M \leq N$) instruction mix blocks. In the Init Step and the Decode Step, the receiver will access in total d (where $d < N$) sets of instructions mix blocks that map to one DSB set, this is repeated for p times. In this case, the receiver's instructions accessed in the Init Step will be processed by the LSD. For the sender, in the Encode Step, when sending $m = 1$, the sender will execute $(M - d)$ (where $M < N + 1$) sets of misaligned instructions that map to the same DSB set as the receiver for q iterations. In this case,

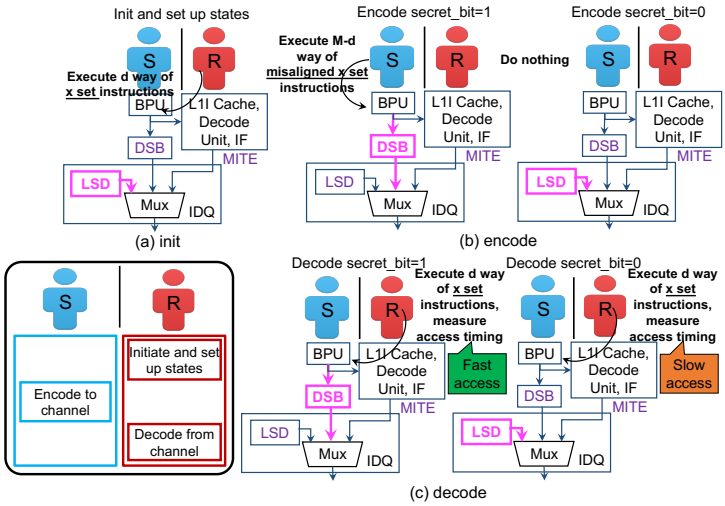


Figure 6.5: Overview of the MT Misalignment-Based Attack.

misalignment of the DSB causes the micro-op delivery to be redirected to DSB from LSD, which leads to faster access of receiver's instruction in the Decode Step. When sending $m = 0$, the sender does nothing. In this case, all the micro-ops will still be delivered by the LSD and the receiver's instruction access in the Decode Step will observe slower access time. We note that LSD is indeed slower in delivery which is demonstrated by the evaluation shown in Figure 6.1.

For example, take $d = 5, N = 8, M = 8$, the access sequences when sending $m = 1$ and $m = 0$ are as follows:

- **Init**: access instruction mix blocks $1 - 5$ mapping to set x
- **Encode**: access *misaligned* instruction mix blocks $6 - 8$ mapping to set x (if $m = 1$); no access (if $m = 0$)
- **Decode**: access blocks $1 - 5$ mapping to set x (if $m = 1$, DSB access is used; if $m = 0$, LSD access is used)

6.3.3 Non-MT Eviction-Based Attack without Multi-Threading

Our attack using internal-interference of the sender is shown in Figure 6.6. The number of iterations (q) of sender's encoding step and number of iterations (p) of receiver's initialization and decoding steps will be the same in this attack (i.e. $p = q$) in order to reliably observe one timing result with a low error rate. For one iteration, in the Init Step, the receiver

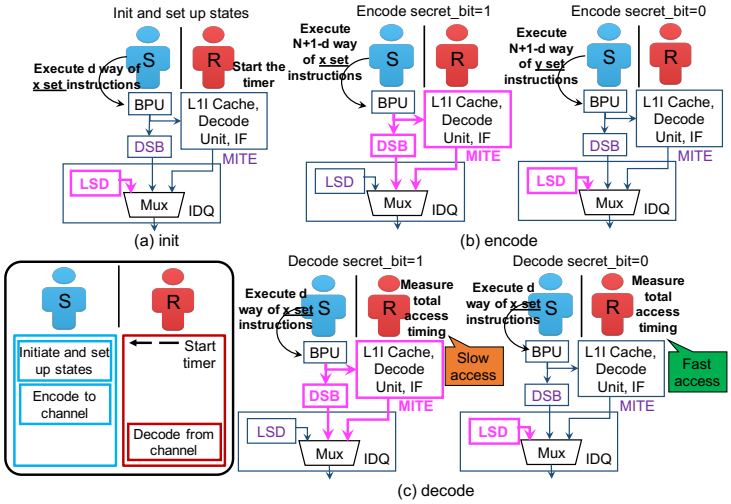


Figure 6.6: Overview of Non-MT Stealthy Eviction-Based Attack.

starts the timer in order to measure total time of the sender. The sender then executes d ($d \leq N$) instructions mix blocks that map to DSB set x. The instructions will be processed by the LSD. In the Encode Step, When sending $m = 1$, the sender will execute $N + 1 - d$ instruction mix blocks that map to the same DSB set as the receiver. When sending $m = 0$, the sender will execute the same number of instruction mix blocks but ones that map to a different DSB set y. (stealthier for security) or do nothing (faster for bandwidth). In the Decode Step, the sender will access the same number d of instruction mix blocks accessed in the Init Step. Then the receiver will end the timer and calculate the total timing of the sender's accesses to derive the information sent. If the Encode Step's access causes evictions, sender's micro-ops in the Decode Step will be delivered from DSB and MITE, where longer timing will be measured, indicating $m = 1$ was sent from the sender. Otherwise, $m = 0$ was transmitted from the sender.

For example, take $d = 6$ and $N = 8$, the instruction access sequences when sending $m = 1$ and $m = 0$ are as follows:

- **Init**: access instruction mix blocks $1 - 6$ mapping to set x
- **Encode**: access instruction mix blocks $7 - 9$ mapping to set x (if $m = 1$); $7 - 9$ of set y (if $m = 0$) (Stealthy) / no access (Fast)
- **Decode**: access instruction mix blocks $1 - 6$ mapping to set x (if $m = 1$, DSB and MITE are used; if $m = 0$, LSD access is used)

6.3.4 Non-MT Misalignment-Based Attack without Multi-Threading

Similar to eviction-based non-MT attack shown in Section 6.3.3, misalignment can also be used to generate interference without multi-threading.

For example, take $d = 5, N = 8, M = 8$, the instruction access sequences when sending $m = 1$ and $m = 0$ are as follows:

- **Init**: access instruction mix blocks $1 - 5$ mapping to set x
- **Encode**: access *misaligned* instruction mix blocks $6 - 8$ mapping to set x (if $m = 1$); *aligned* instruction mix blocks $6 - 8$ mapping to set x (Stealthy) / no access (Fast) (if $m = 0$)
- **Decode**: access instruction mix blocks $1 - 5$ mapping to set x (if $m = 1$, DSB access is used; if $m = 0$, LSD access is used)

6.3.5 Slow-Switch Attack without Multi-Threading

We now also present a covert-channel attack making use of LCP instructions, which we call the slow-switch attack. For slow-switch attack, the receiver (attacker) starts and ends the timer in the Init and Decode Steps. Meanwhile, in the Encode Step, within the loop, there will be in total r number of LCP instructions being executed and the number of loops is p (or q, $p = q$ as the same setting for non-MT eviction-based attacks). When sending $m = 1$, the sender will alternatively execute one normal *add* instruction followed by one *add* instruction with length changing prefix; this is repeated for r times. This new type of instruction mix can enlarge the LCP stall cycles and maximize the LSD-to-DSB switches. When sending $m = 0$, the sender will execute r normal *add* instructions and then execute r *add* instruction with length changing prefixes. This instruction mix has fewer LCP stalls, thus minimizing the LSD-to-DSB switch penalties.

For example, take $r = 16$, the instruction access sequences when sending $m = 1$ and $m = 0$ are as follows:

- **Init**: start the timer.
- **Encode**: access $r = 16$ groups of instructions, where each group has an *add* instruction with a length changing prefix and then a normal *add* instruction (if $m = 1$); or access

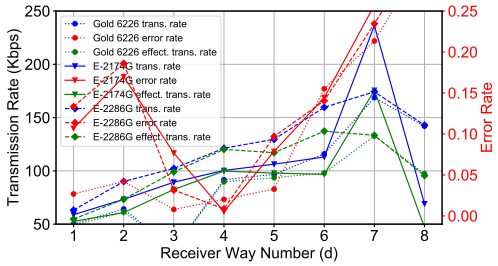


Figure 6.7: Evaluation of MT Eviction-Based Attack for different values of parameter d.

Table 6.2: Transmission rates and error rates of the covert-channel MT Eviction-Based Attack when setting $d = 1$ for for different message patterns: all 0s, all 1s, alternating 0s and 1s, and random.

	All 0s Message			**All 1s Message**			**Alternating 0s and 1s**			**Random Message**		
	G6226	2174G	2286G	2288G	G6226	2174G	2286G	2288G	G6226	2174G	2286G	2288G
Trans. Rate (Kbps)	42.66	49.53	87.33	55.28	61.17	102.39	50.21	58.86	64.96	18.28	21.80	25.61
Error Rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2.68%	10.69%	12.56%	22.57%	18.53%	19.83%

16 normal *add* instruction and then 16 *add* instruction with length changing prefixes (if $m = 0$);

- **Decode**: stop the timer.

6.4 Evaluation of Timing-Channel Attacks

In this section, we evaluate the transmission rates and error rates of all the timing covert-channel attacks discussed in Section 6.3. Power attacks, SGX attacks, use of new covert channels in Spectre, microcode patch fingerprinting, and new side-channel attack are evaluated later.

The evaluation is conducted on 4 recent $x86\_64$ processors from Intel Skylake's family. The specifications of the processors is shown in Table 6.1. For each covert channel, the transmitted data is compared with the received data to compute the error rates. To evaluate the error rates of the channel, the Wagner-Fischer algorithm [113] is used to calculate the edit distance between the sent string and the received string.

6.4.1 Number of Iterations (p, q) for Attack Steps

After careful tuning of the configurations, when sending each bit m of message, non-MT attacks can have $p = q = 10$ (to repeat initialize, encode, and decode steps and still reliably

Table 6.3: Transmission rates and error rates of all the eviction-based and misalignment-based attacks when setting $d = 6$ for eviction-based attacks and $d = 5$, $M = 8$ for misalignment-based attacks. The transmitted message is alternating pattern of 0s and 1s. Transmission rates for the fastest attack are shown in bold. Intel Xeon E-2288G machine we tested has hyper-threading disabled so there is no MT attack possible.

	Non-MT Stealthy Eviction.				**Non-MT Stealthy Misalign.**				**MT Eviction.**			
	G6226	2174G	2286G	2288G	G6226	2174G	2286G	2288G	G6226	2174G	2286G	2288G
Trans. Rate (Kbps)	419.67	851.81	1182.55	1356.43	713.01	466.02	723.15	1094.39	115.97	113.02	161.63	—
Error Rate	6.48%	3.43%	3.45%	0.36%	22.56%	11.34%	16.56%	10.08%	15.52%	14.44%	13.93%	—
	Non-MT Fast Eviction.				**Non-MT Fast Misalign.**				**MT Misalignment.**			
	G6226	2174G	2286G	2288G	G6226	2174G	2286G	2288G	G6226	2174G	2286G	2288G
Trans. Rate (Kbps)	501.06	977.68	1205.90	1399.96	500.90	959.45	1228.35	**1410.84**	129.36	152.44	200.37	—
Error Rate	6.09%	0.00%	0.00%	0.00%	0.16%	0.00%	0.16%	**0.00%**	7.85%	2.77%	4.62%	—

observe result with low error rates). To transmit each bit, the sender does one encoding step and receiver does one decoding step and this pattern of activity is repeated in total 10 times, hence $p = q = 10$. For MT attacks, for each bit to be transmitted the receiver does 10 decoding measurements for each encoding step, while each encoding step has to be repeated 100 times, hence $p/q = 10$, where $q = 100$ (total encoding steps), $p = 1000$ (total decoding steps). The $q = 100$ is due to more noise in the MT setting, compared to $q = 10$ for the non-MT setting.

6.4.2 Threshold for Detecting Transmitted Bit

To establish decoding threshold for timing measurements, to determine $m = 1$ vs. $m = 0$, an alternating pattern of 0s and 1s is sent, and the timing (measured in cycles using the `rdtscp` instruction) is averaged for 0s and 1s to establish the threshold. Based on different covert channels, if a measurement is $30 - 70\%$ or more above the threshold, it is judged to be a "1", otherwise it is judged to be a "0". The simple encoding can be in future replaced with other channel coding methods [114] for possibly faster transmission.

6.4.3 Influence of (d, M) Parameters

To help find the ideal transmission rate, we evaluate the influence of d (number of DSB ways accessed by the receiver) and its impact on the transmission rate and error rates.[1]

1. This work is not aimed at achieving the highest bandwidth covert channel. To fully optimize the transmission rate and error rate, techniques such as the ones used in [115] can be further exploited.

Table 6.4: Transmission rates and error rates of Slow-Switch Attacks. The transmitted message is alternating 0s and 1s.

	Non-MT Slow-Switch-Based	
	G6226	2288G
Tr. Rate (Kbps)	678.11	1351.43
Error Rate	6.74%	0.64%

The results of changing d for MT Eviction-Based Attack is shown in Figure 6.7. When increasing d from 1 to 8 (DSB has $N = 8$ ways), the number of ways accessed by the sender will decrease (number of sender's ways accessed is $N + 1 - d$). Receiver's observation will then become less stable (error rate increases) while on the other hand transmission rate increases. Error rates of small d (e.g., $d = 1, 2$) are also large because when the number of ways accessed by the receiver is small, timing difference of sending 0 and 1 is small, which can be disrupted by the system noise. To find a balance between the transmission rate and error rate, we choose $d = 6$ for eviction-based attacks. For misalignment-based attacks, we choose $d = 5$, $M = 8$ (M is the total number of ways accessed by the sender and receiver for misalignment-based attacks).

6.4.4 Influence of Message Patterns

A sample evaluation of MT Eviction-Based Attack for the four different message patterns with $d = 1$ is shown in Table 6.2. From the results it can be seen that better transmission rate and error rate are derived for all 0s and all 1s. This is possibly because when not changing the bits (as is case for all 0s or all 1s), the frontend path used by the sender accesses remains the same, generating less noise. The random messages are the worst due to the frequent and unstable frontend path changes.

6.4.5 Transmission Rates and Error Rates

The bit transmission rates and error rates for all types of the timing attacks are presented in Table 6.3 and Table 6.4, with $d = 6$ for eviction-based attacks, $d = 5$ for misalignment-based attacks and $r = 16$ for slow-switch attacks. For the best attack, which is the Non-MT Fast Misalignment-Based Attack, the transmission rate can be as high as 1410 Kbps (1.41 Mbps) with almost 0% error rate. Slow-switch attacks have generally similar transmission

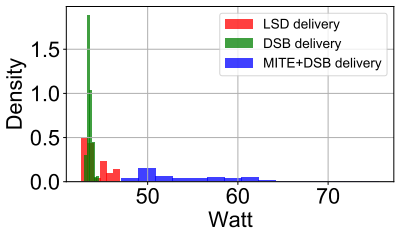


Figure 6.8: Example histogram of power consumption when different frontend paths are used to process micro-ops in Intel Xeon Gold 6226 processor.

rate compared with the non-MT misalignment-based attacks. Non-MT attacks have better transmission rate than MT attacks due to smaller noise.

6.4.6 Power-Channel Attack Evaluation

Switching between LSD or DSB and the MITE will not only cause timing changes for instruction processing, but also power changes. The power changes can be measured by abusing unprivileged access to Intel's Running Average Power Limit (RAPL) interface [116].[2]

Figure 6.8 shows example histogram of the power consumption of utilizing different frontend paths for the micro-ops in Intel Xeon Gold 6226 processor. Based on the power differences, we demonstrate a non-MT attack that can detect LSD or DSB vs. MITE frontend path power differences caused by eviction or misalignment through observing the power changes in RAPL. Configuration of the attack is similar to the non-MT attack demonstrated in Section 6.3.3. To observe the power differences, for each bit transmission the initialize, encode, and decode steps have to be iterated for $p = q = 240,000$ times since RAPL interface update interval is around 20kHz [117]. The power attack's bandwidth is limited by the update interval of RAPL, and is less than for the timing attacks.

Table 6.5 shows the evaluation results of two power-based non-MT attacks on Intel's Xeon Gold 6226 processor. The bandwidth of the power attacks is around 0.6 – 0.7 Kbps. The transmission is still above 100 bps which is considered a high-bandwidth channel by TCSEC [118]. The power attack bandwidth can possibly be further improved using techniques such as the ones shown in recent PLATUPUS work [117].

2. In power attacks, if unprivileged RAPL accesses are prevented, we can still potentially use privilege access and use power to attack SGX enclaves.

Table 6.5: Evaluation of Non-MT Power-Based attacks on Intel Xeon Gold 6226 processor when setting $d = 6$.

	Eviction-Based	**Misalignment-Based**
Tr. Rate (Kbps)	0.66	0.63
Error Rate	18.87%	9.07%

Table 6.6: Transmission rates and error rates of covert channels for leaking information from an SGX enclave when setting $d = 6$ for eviction-based attacks and $d = 5$, $M = 8$ for misalignment-based attacks. The transmitted message is alternating 0s and 1s. Intel Xeon E-2288G machine we tested has hyper-threading disabled so no MT attack data is provided for this machine.

SGX Attacks	**Non-MT Stealthy Eviction.**			**Non-MT Stealthy Misalign.**			**MT Eviction.**		
	E-2174G	E-2286G	E-2288G	E-2174G	E-2286G	E-2288G	E-2174G	E-2286G	E-2288G
Trans. Rate (Kbps)	18.96	19.56	21.20	23.93	24.70	27.10	7.85	14.89	—
Error Rate	0.16%	1.33%	2.18%	0.32%	0.76%	0.76%	6.74%	8.02%	—
SGX Attacks	**Non-MT Fast Eviction.**			**Non-MT Fast Misalign.**			**MT Misalign.**		
	E-2174G	E-2286G	E-2288G	E-2174G	E-2286G	E-2288G	E-2174G	E-2286G	E-2288G
Trans. Rate (Kbps)	29.35	32.01	34.48	30.36	31.18	35.20	6.39	13.62	—
Error Rate	0.04%	1.40%	0.40%	0.08%	1.08%	0.68%	2.56%	12.95%	—

6.5 SGX Attack Evaluation

The goal of Intel Software Guard Extension (SGX) is to protect sensitive data against the untrusted user, even on already compromised system, with the help of hardware-implemented security and cryptographic mechanism inside the processor [19]. Unfortunately, as we demonstrate, SGX is also vulnerable to frontend-related attacks.[3]

To demonstrate our attacks in an SGX environment, we assume a sender program is running inside the SGX enclave and manipulates the use of the frontend paths to communicate to a receiver outside of the SGX. We consider both non-MT and MT SGX attacks, but for both there is only one SGX entry and one SGX exit, while attacker measures the execution time from the outside. Consequently, instruction TLB flushing upon entry and exit does not impact our attacks.

6.5.1 MT Timing SGX Attacks

For MT timing SGX attacks, the sender maintains its own thread and performs the covert transmission from within the enclave. Meanwhile, the receiver decodes bits of the sender by measuring the timing of its own operations. Under this scenario, the receiver is able to

3. We demonstrate attacks on SGX, although there is a newer SGX2 which extends SGX with dynamic memory management and other features, we believe these features will not affect our attacks and our attacks can be applied to SGX2 in future when machines with SGX2 are available.

detect the performance difference of its own instruction access based on the activity inside the SGX. If SGX thread is running, then the receiver will observe the partitioned DSB. If the SGX thread is idle, whole DSB is dedicated to the receiver thread. Receiver can observe its own internal-interference and deduce the DSB state.

Evaluation of the MT timing SGX attacks is shown in Table 6.6. It can be seen from the table that the transmission rates of SGX attacks can be roughly 6 Kbps – 15 Kbps with iteration numbers $p = 1,000$, $q = 10,000$, while maintaining the similar error rates as the MT non-SGX attacks.

6.5.2 Non-MT Timing SGX Attacks

For non-MT timing SGX attacks, the sender program is still inside the enclave, while the receiver derives the information by measuring the timing of SGX operation from outside of the enclave. Under this scenario, the receiver's observations depend on ability to detect the internal interference of the sender's accesses within the enclave, to detect whether there are frontend path changes caused by the eviction or misalignment of the micro-ops or not. The non-MT SGX attacks, because they do not leverage multi-threading, are possible even when multi-threading is disabled for security.

In the non-MT setup, we assume the attacker (receiver) is able to trigger the sender and they both execute on the same hardware thread. To reduce overhead and noise of enclave exits and entrances, for each transmission of a bit, there is only one entrance and exit. Effectively the receiver starts time measurement, then allows the enclave to run, and then finally measures the timing of the enclave as it was affected by the frontend paths. Compared to non-SGX attacks, more iterations of initialization, encoding, and decoding are necessary ($p = q = 1,000 - 5,000$ iterations for the SGX attack compared to $p = q = 10$ iterations for non-SGX attacks) in order to transmit one bit.

Evaluation of the non-MT timing SGX attacks is shown in Table 6.6. As the table shows, the transmission rates of non-MT SGX attacks are roughly 1/25 to 1/30 of non-MT non-SGX attacks, while still maintaining acceptable and even lower error rates.

Table 6.7: L1 miss rates of our Spectre v1 version attack (run on Intel's Xeon Gold 6226 processor) with variants of Spectre v1 that use different covert channels. MEM F+R, L1D F+R, and L1D LRU attacks are from work [119]. L1 miss rates in [119] are L1D miss rates.

	Others			**Our**		
	MEM F+R [119]	**L1D F+R** [119]	**L1D LRU** [119]	**L1I F+R**	**L1I P+P**	**Frontend**
L1 Miss Rate	2.81%	4.79%	4.48%	0.45%	0.48%	0.21%

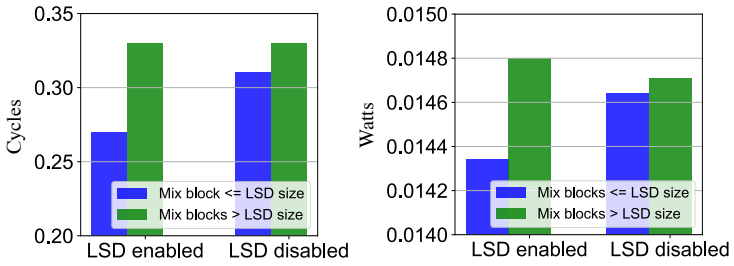
6.5.3 Power-Based SGX Attacks

Power-based attacks are also possible, but not discussed and leave for future work. We remark, however, that even if RAPL is disabled for user-level code, power-based SGX attacks are possible because RAPL can be accessed from the privileged, malicious OS.

6.6 Frontend and Instruction Cache-Based Spectre Attack Evaluation

Speculative attacks leverage transient execution to access secret and then a covert channel to pass the secret to the attacker [3, 4, 120]. In this section, we demonstrate our new variants of Spectre v1. In our Spectre attacks, we assume an in-domain attack where the victim and attacker code are in the same thread, so only one thread is running on the processor core. The secret message is represented by 5 bit chunks (each chunk can have value from 0 to 31). We then use one of the 32 DSB sets to represent each value. Similar to cache-based channels, during the speculative execution, secret value is encoded by accessing the corresponding set. Unlike other cache attacks, to access a DSB set, instruction mix block mapping to that set has to be executed. We also implemented Spectre v1 attacks using L1I cache Flush + Reload attack and L1I Prime + Probe attack, to compare to our frontend attacks.

Table 6.7 shows the L1 miss rate when using our channels compared to other channels. While our Spectre v1 attacks have lower bandwidths than data cache-based Spectre attacks, we are able to achieve lowest L1 miss rates. Especially, compared with recent cache-based LRU [119] covert channels which target stealthy attacks without causing high data cache miss rates, our frontend attack does not cause any cache misses at all, making the L1 miss rate the smallest.



(a) Average Timing (b) Power

Figure 6.9: Example comparison of frontend timing and power for executing instruction mix blocks less or greater than LSD capacity. All mix blocks map to the same DSB set. If LSD is disabled execution falls back to DSB and MITE.

6.7 Microcode Patch Detection Evaluation

When evaluating the behavior of the processor frontend, we also found a new type of attack where performance of the frontend can be used for fingerprinting the microcode updates of the processor. In particular, we evaluated our Intel Xeon Gold 6226 test machine under older `3.20180312.0ubuntu18.04.1` (patch1) and newer `3.20210608.0ubuntu0.18.04.1` (patch2) Intel microcode patches. While neither patch explicitly mentions LSD, we found that with the newer patch2 LSD is disabled while with older patch1 the LSD is enabled. To switch between the patches, the processor has to be restarted so the microcode in the CPU can be updated.

To detect the changes in the LSD behavior, we can use both the timing difference and the power difference when testing code sequences with number of instruction mix blocks less than LSD capacity (so they would fit in LSD and be processed by LSD) or sequences with number of instruction mix blocks greater than LSD capacity (so micro-ops would be forced to be handled by DSB and MITE instead). The average timing and power difference for LSD enabled (patch1) vs. disabled (patch2) are shown in Figure 6.9. Attackers can clearly differentiate which patch has been applied, with timing being a more reliable indicator.

Attackers can leverage this to learn of vulnerabilities of the processor. For example, patch2 protects against CVE-2021-24489: potential security vulnerability in some Intel Virtualization Technology for Directed I/0 (VT-d) products that allows for escalation of privilege.[4] Knowing the patch is applied or not allows the attacker to exploit VT-d related

4. The patch2 also adds protections against CVE-2021-24489, CVE-2020-24511, CVE-2020-24512, and

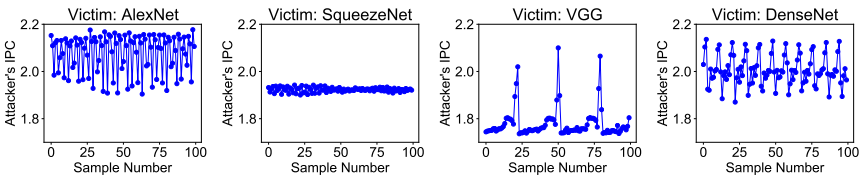


Figure 6.10: Fingerprinting results of machine learning model using frontend side-channel attacks. Baseline IPC of the attacker program is 3.58. With two threads the IPC is roughly halved. Furthermore, due to different patterns of the victim it fluctuates between the 1.8 and 2.2.

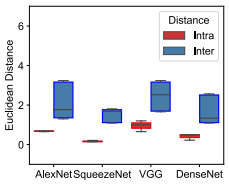


Figure 6.11: Inter-distance and intra-distance of all the models.

attacks. The frontend timing thus cannot only be the target of attack itself, but help attacker discover other vulnerabilities in the system.

6.8 Evaluation of Side-Channel Attack and Fingerprinting of Applications

Based on the frontend characteristics, we developed a new frontend-based fingerprinting technique utilizing a side-channel attack to demonstrate that frontend can be not only used for covert communication, but also for side-channel information leakage. Our fingerprinting technique is able to identify what type of workload a victim is running on a co-located SMT thread. Moreover, our technique can achieve fingerprinting using low-frequency timing measurements, therefore, it works on platforms where access to high-precision timers is limited. The approach does not use any performance counters or privileged access, and depends only on the attacker (receiver) measuring their own instructions per cycle (IPC). The IPC is affected by the shared frontend, especially the shared MITE, and interference between attacker and victim in the frontend are the sources of the information leakage. The attacks were tested on same CPUs as the covert channels and work with current Intel processors where DSB and LSD are partitioned between threads (but MITE is not).

When compared with previous fingerprinting techniques [121, 122], which are mostly based on using performance counters or contention in the backend of the processor, our side-channel attack has number of advantages. Our method 1) does not need to measure the

CVE-2020-24513.

performance of the victim workload, 2) does not require usage of any performance counters but only a low-precision timer, 3) does not depend on eviction of lines in instruction and data caches so it is robust against the existing defense measures on caches, and 4) it is also robust against existing frontend resource hardware partitioning, including DSB partitioning and LSD partitioning implemented on Intel microarchitectures.

6.8.1 Side Channel Design

To develop the side channel, we designed a modified receiver that uses a new mix block of *nop* instructions instead of the prior instruction mix blocks used in the covert channels. We use *nop* instructions in the x86 ISA to construct our attacker thread, which naturally triggers frontend resources to decode the *nops*, but it does not generate any traffic in the backend. The attacker thread used to perform fingerprinting loops through 100 *nop* instructions which will not fit in LSD but are able to fit in DSB. The loop takes two cache lines, which never get evicted from the cache because of the repeated loop access within the attacker program. Victim program will slow down the decoding process of the MITE for the attacker which causes timing variation of the attacker program, and when the attacker measures its own performance variation, it is able to observe patterns that reveal type of victim application. The attacker measures its own performance by computing the IPC based on the number of *nops* executed and time reading from the *rdtsc*.

We measure only the instruction per second at a low frequency of 10Hz because existing platforms limit the usage of high-precision timers [121]. Euclidean distance [123] is used to calculate the distance of IPC measurement traces of two test results. If these two tests of the attacker program run with the same victim benchmark, intra-distance is derived. Otherwise, inter-distance is derived. Furthermore, we verified that the contention indeed happens in the frontend by monitoring the performance counter changes. Note that the actual attack does not use performance counters. They were only used to validate the results.

6.8.2 Fingerprinting of Mobile Applications

To demonstrate the fingerprinting and the side-channel attack on mobile application usage, we performed the experiments using a popular Geekbench5 benchmark suite [124]. It consists

of a wide range of workloads including camera, navigation, speech recognition, etc.

We run the attacker thread along with a Geekbench5 thread on a single SMT-enabled core. Unique IPC waveforms of the attacker are derived when running with different benchmarks. We observe an average 0.232 intra-distance vs. 4.793 inter-distance for the 10 benchmarks tested. Our results indicate that the IPC changes of the attacker thread can be used directly to distinguish the type of the victim application that is running.

6.8.3 Fingerprinting of Machine Learning Algorithms

We also demonstrate the fingerprinting of different machine learning algorithms from the TVM framework. Figure 6.10 shows the average IPC traces of the attacker program thread when running with different CNN model inference threads on the same SMT core. Clear differences in the traces are shown and these can be used to distinguish different machine learning models based on the traces using different convolution layers. A set of traces can thus be compared to reference traces to distinguish a network. Because of the frontend contention in the MITE, even with partitioned LSD and DSB, the attacker can leak information about type of victim machine learning model. As can be seen in Figure 6.11, the inter distance and intra distance can be clearly differentiated. This shows that the fingerprinting results can clearly differentiate machine learning model architectures. We observe an average 0.550 intra-distance vs. 1.937 inter-distance for tested 4 CNN models.

6.8.4 Defense about Frontend Attacks

The frontend vulnerabilities do not involve interference in traditional instruction or data caches, and they do not involve speculation. Therefore, a large set of existing defense mechanism will not be able to prevent them [61, 63, 125]. The major difficulty of dealing with the security vulnerabilities of the frontend paths is that the frontend is designed to give better performance or lower power for different execution scenarios, which inevitably creates inherent timing or power signatures. Eliminating these timing or power signatures would reduce the performance or power benefits. Since frontend components such as the MITE, DSB, and LSD are widely used in modern architecture designs. Defending the frontend vulnerabilities will require new approaches for the design of the frontend.

At the system-level, the SMT can be always disabled for security-critical applications, which would eliminate the MT attacks. This should be probably already done due to other prior attacks on caches, for example.

Even with SMT disabled, the non-MT attacks are possible. Defending these would require careful design of the code so that there is no secret-dependent timing. This requires writing of the code to make sure that the frontend switching or timing is always the same, regardless of the secret data being processed. Instruction alignment, as shown by our misalignment-based attacks, can also cause timing differences, so not just the code, but its location in the address space needs to be considered.

Regarding Spectre attacks, the frontend state should not be updated due to speculative execution. Existing defenses such as buffering cache updates could be applied to the DSB.

For power-based attack, the ability to monitor power of other users or SGX enclaves needs to be disabled. For user-level code, existing patches from Intel should be applied to disable access to the power monitors. For SGX, the power monitors can be enabled in debug mode for development, but disabled in production mode.

Since patch detection is based on timing observation of whether some components are enabled or disabled, there does not seem to exist an easy solution (unless all frontend paths have same timing, which defeats the purpose of having different paths to get better performance). System administrator should assume that potential attackers know exactly which patches have been applied, and the patch level of the system should not be considered a secret.

Although a number of attacks have been demonstrated in our work, we do note that to perform some of the attacks we need to find specific instruction mix blocks to minimize the contention in the backend to allow the attacks to be effective. The attacks may be difficult to deploy in practice, for example, if the right instruction mix block is not available in the code. Nevertheless, our other attacks such as the side-channel and application fingerprinting do not depend on specific instruction mix blocks, but overall operation of the victim program. The frontend then can impact the system security, and more evaluation of the defenses and how to deploy them are needed.

Chapter 7

Preliminary Study of Vulnerabilities in Accelerators Beyond CPUs

This chapter presents preliminary work exploring security of accelerators, such as GPUs or cloud-based quantum computers. As much computation is done on accelerators, not on just the main processors, the security of these accelerators needs to be analyzed, attacks studied, and defenses proposed.

7.1 GPU Covert-Channel Attacks

A GPU, or Graphics Processing Unit, is a specialized designed electronic circuit for rapid manipulating and altering memory to accelerate the image creation in a frame buffer for output to a display device. GPUs are broadly used in accelerating security and efficiency-conscious systems, e.g., autonomous vehicles (AV), datacenters, game consoles, and cloud gaming services.

7.1.1 Parallelism Features of GPU

Compared with CPU, GPU makes use of highly parallel structure to more efficiently process large blocks of data in parallel. Take the most recent GPU microarchitecture Ampere [126]

Figure 7.1: A100 Streaming Multiprocessor (SM). The figure is a screen-capture from [126].

as an example. The A100 Tensor Core GPU implementation of GA100 GPU contains 7 GPU Processing Clusters (GPC), each GPC contains 7 or 8 Texture Processing Clusters (TPC), and each TPC contains 2 Streaming Processors (SM). In this case, each full A100 GPU contains 108 SMs.

For each SM, each SM contains 4 Tensor Cores, 64 INT32 (integer 32-bit), 64 FP32 (floating point 32-bit), and 32 FP64 (floating point 64-bit) CUDA (Compute Unified Device Architecture, a parallel computing platform and application programming interface of GPU) cores, etc. These units can process the data in parallel to increase performance. However, if we are able to establish covert channels in GPUs, these units could help boost the bandwidth of data transmissions through the covert channel, forming a high-speed and high-fidelity transmission channel.

7.1.2 GPU Covert Channels

In order to protect the security and increase performance, for the new generations of GPUs, a new Multi-Instance GPU (MIG) capability is able to provide enhanced client and application isolation for multi-tenant and virtualized GPU environments. This will be beneficial to

cloud service providers. On the other hand, this provides chances of running the victim (sender) and the attacker (receiver) in parallel, where covert channels can be established.

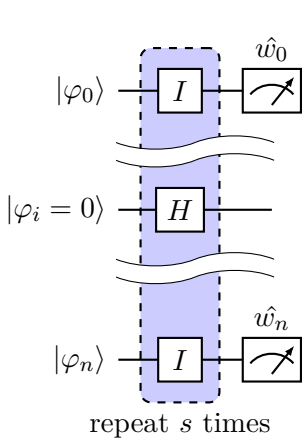
As the microarchitecture shows in Figure 7.1, L0 instruction cache, Warp Scheduler, Special Function Units (SFU), and L1 Data Cache is shared. The address space is shared and the GPU programming model allows the creation of thousands of threads to execute the same code. This provides the chance to set up the sender the receiver to different threads that share the same functional units, where performance characteristics such as timing and power can be observed through side effects of running the functional units. The preliminary work is a collaborated with NVIDIA and the attack details are not further disclosed here.

7.2 Quantum Computing Crosstalk Attacks

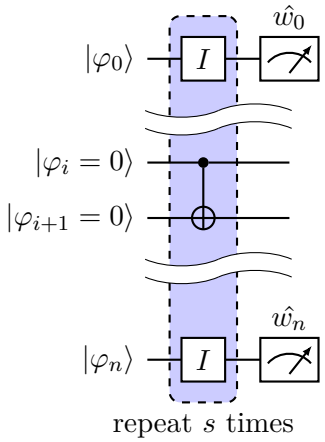
Small quantum computers are already available today as cloud-based accelerators for classical computers to use. Fingerprinting of such quantum computer accelerator devices is a new threat that poses a challenge to shared, cloud-based quantum computers. Fingerprinting can allow adversaries to map quantum computer infrastructures, uniquely identify cloud-based devices which otherwise have no public identifiers, and it can assist other adversarial attacks. This work shows idle tomography-based fingerprinting method based on crosstalk-induced errors in NISQ quantum computers.

Quantum Computer Accelerators in a Cloud Setting

Quantum computers are machines that make use of the properties of quantum physics to perform computations and store data. Especially for certain tasks, they can vastly outperform the current best supercomputers and solve certain computational problems, such as integer factorization (which is the key computation step of the RSA (Rivest–Shamir–Adleman) encryption), in a substantially fast speed. In the next few years, expansion is expected as the quantum computation shifts toward real-world use in the field of pharmaceutical, data security and other applications. Quantum computer power scales exponentially with quits. As the quantum hardware development is accelerating, larger quantum machines are introduced, the throughput and utilization can be improved by multi-programming or



(a) Single-qubit drive on q_i with Hadamard gates, other gates are idle (I).



(b) Two-qubit drive on q_i, q_{i+1} with CNOT gates, other gates are idle (I).

Figure 7.2: Circuit schematic of idle tomography circuits with single- and two-qubit drive. Each spectator qubit q_j is initialized and measured accordingly.

running programs from different users in parallel through the server cloud. With cloud-based access, the provider can decide which quantum computer to schedule the programs on, or it can put two or more programs (users) on the same computer if the resources allow. Time sharing of resources is not possible in quantum computers yet, but spatial sharing for the quantum computers is possible.

However, the dominant user case of providing cloud-based access for remote users to rent quantum computers lead to new security and privacy threats. Under the cloud-based setting, researchers have been exploring how to allow the computers to be shared between different users or tasks to improve the utilization of the resources and eventually provide lower costs for users. However, allowing multiple users to share the quantum computers provide the attacker to be co-located with the victim program, where crosstalk can be exploited to perform fault injection. The remote users can in this case, e.g., try to learn the infrastructure, and attack other users, or leak information from other users.

7.2.1 Crosstalk and Idle Tomography

Noise in quantum computers can be attributed to different types of errors, including gate errors, decoherence errors, readout errors, and crosstalk errors. Among them, crosstalk errors refers to that gate operations on one or two qubits (depending on the gate type)

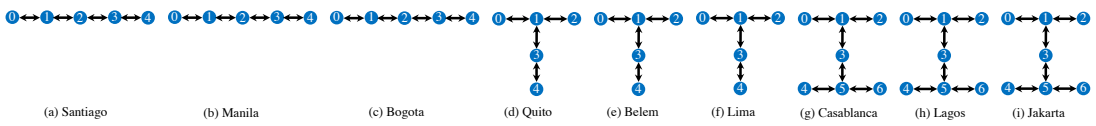


Figure 7.3: The 9 IBM Q machines (backends) used in the evaluation. The figure shows the qubits and physical topologies for each backend. The fingerprinting circuits are mapped onto these topologies, or subgraphs of the topologies if not all qubits are used.

affect other, nearby qubits or gates. The crosstalk can be qubit-to-qubit, coupling-to-qubit, qubit-to-coupling, or coupling-to-coupling. As this work shows, crosstalk is a feature of NISQ quantum computer hardware that allows adversarial threats.

In order to measure the crosstalk, there are two different methods: Simultaneous Randomized Benchmarking (SRB) [127] and Idle Tomography (IDT) [128]. We chose IDT since it uses a comparably smaller number of circuits and relatively shorter circuits [128]. IDT characterizes the error accumulated by idle qubits over time. Figure 7.2 displays a typical IDT setup, where one or two qubits are selected as *drive qubits*. These qubits are prepared in the $|0\rangle$ state in the logical basis. The remainder of the qubits are *spectator qubits*. After preparation, gate operations start to work on the drive qubits. In the typical case, the Hadamard gate H and the controlled-not gate CNOT are used for single- and two-qubit drive cases, respectively. In the meantime, spectator qubits are kept at idle. Finally, we measure each spectator qubit and output the results, which are further used for characterizing the error channels.

7.2.2 Fingerprinting Attack

In this work we utilize crosstalk to fingerprint the quantum computers in a multi-programming setting. Given a subgraph typology, we can obtain its fingerprint without knowing the actual device using certain quantum computing features. In that case, we can construct a classifier to predict with high accuracy which device the subgraph belongs to and what is the location the subgraph in the quantum device. Like the ML model extraction attack in classical computers, we believe this fingerprinting attack can also serve as the foundation of further quantum computing based attacks.

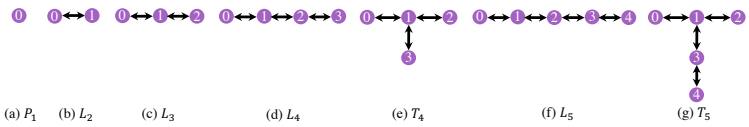


Figure 7.4: Topologies of the 7 tomography circuits used in the evaluation. These represent attackers $\mathscr{A}$ circuits. These circuits are mapped onto the physical topologies of the backends shown in Figure 7.3, by the provider $\mathscr{P}$.

Threat Model

The threat model consists of an attacker $\mathscr{A}$ and a cloud provider $\mathscr{P}$. What the attacker $\mathscr{A}$ targets is to gather one or more full-device fingerprints for each directed graph of the quantum machine during 1) enrollment. Later in the 2) inference phase, based on this information, $\mathscr{A}$ will collect new fingerprint data and attempt to match it to specific devices or localities on specific devices.

Fingerprint Enrollment

During the enrollment stage, the attacker $\mathscr{A}$ will run a set of full-device idle tomography circuits to single- and two-qubit drive. For each qubit, set rest of single qubits as spectator qubits to measure crosstalk. For each coupling between two arbitrary qubit in the circuit, set rest of the coupling qubits as spectator qubits used for measuring crosstalk. Finally, two sets of control-group experiments are performed with all qubits as spectator qubits and idle gate delays corresponding to the single- and two-qubit drive respectively.

Fingerprint Matching

During the inference stage, the attacker requests the cloud provider to run a circuit with topology dependency. If there exists some topology isomorphic to a subgraph derived during the enrollment stage, the topology is determined to be *satisfiable*. The attacker will proceed to run idle tomography circuits on the topology, while the cloud provider translates the circuits on the topology. After the cloud provider returns the circuit measurement results to the attacker, the latter computes a fingerprint.

The attacker $\mathscr{A}$ then attempts to infer if the typology belongs to a circuit or specific location. To do this, $\mathscr{A}$ iterates through every enrollment result and identifies the set

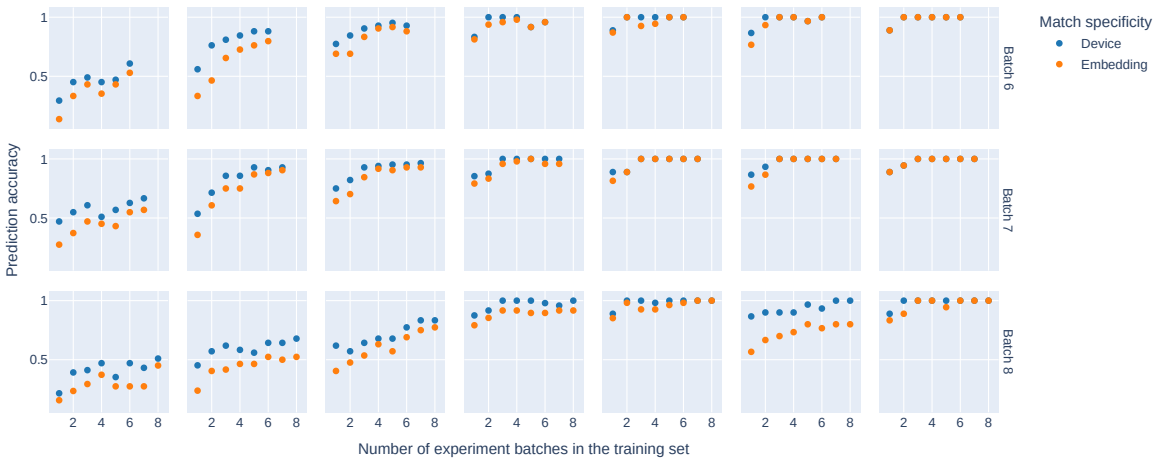


Figure 7.5: Device- and locality-specific prediction accuracy on the last 3 batches, when the training set contains the first n batches for $n \in [1, 8]$.

isomorphic to the topology. The set is then used as a training set to train a classifier sensitive to fingerprints of isomorphisms of the topology. Finally, $\mathscr{A}$ takes the prediction as the inferred locality.

Evaluation Setup

The effectiveness of the fingerprinting scheme is evaluated on 9 IBM Q machines (backends) shown in Figure 7.3. The machines were used to run 9 batches of tomography experiments over 12 days. For idle tomography, we examine the idle sequence lengths 1, 2, 4 and 8. All circuits are run and measured for 2048 shots. Each batch generates one full-device fingerprint for each backend. Generating one full-device fingerprint takes less than an hour on 5-qubit devices, and less than two hours on 7-qubit devices.

We evaluated 7 different subgraph topologies: P_1, L_2, L_3, L_4, T_4, L_5, T_5, as is shown in Figure 7.4. L_5 and T_5 are full-device topologies (i.e., they occupy whole backend on L_5 and T_5 devices respectively). For each subgraph topology, we consider all of its possible embeddings across all devices. Each subgraph topology can be embedded (i.e., mappped to the physical machines) in many ways, and the attacker will find out where their circuit was mapped to.

Prediction Accuracy

To evaluate the prediction accuracy, we vary the size of the training set and derive the corresponding prediction accuracy for various subgraph topologies on two levels of specificity:

- Device-specific. A prediction succeeds if and only if the predicted locality exists on the same device.
- Embedding-specific. A prediction succeeds if and only if the predicted locality exactly matches the true locality of the exact device.

As can be seen in Figure 7.5, as the number of batches in the training set increases, the prediction accuracy values increase substantially. Complex subgraph topologies are easier to pinpoint regardless of specificity apart from an outlier of L_5 in batch 8. Observe that accuracy values for most of the 4-qubit and 5-qubit topologies reach $\sim 100\%$ when at least 3 batches are in the training set. The device- and location-specific fingerprinting are demonstrated with accuracy to be 99.1% and 95.3%, respectively. The excellent fingerprinting abilities across machines are showed.

Chapter 8

Hardware Security Verification

Apart from microarchitectural attacks and defenses presented earlier in this dissertation, this chapter introduces formal methods which can be used for hardware security verification, e.g., verify if circuit implementation violates security and privacy rules or not.

8.1 SecChisel Security Verification Framework

This work presents a design-time security verification framework for secure processor architectures. Our new *SecChisel* framework is built upon the Chisel hardware construction language and tools, and uses information flow analysis to verify the security properties of an architecture at design-time. To enforce information flow security, the framework supports adding security tags to wires, registers, modules, and other parts of the design description, as well as allows for defining a custom security lattice and custom information flow policies. The framework performs automatic security tag propagation analysis in a new SecChisel parser and information flow checking using the Z3 SMT solver. The same SecChisel codebase is used to design hardware modules as well as to verify the security properties, ensuring that the verified design directly corresponds to the actual design. This framework is evaluated on RISC-V Rocket Chip expanded with AES and SHA modules. The framework was able to capture information leaks in the hardware bugs or Trojans that it was tested with.

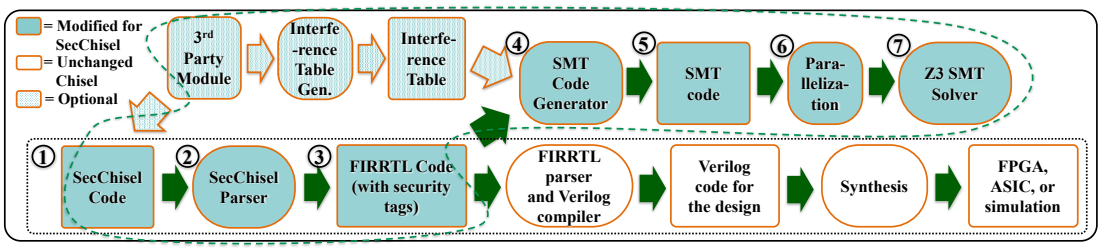


Figure 8.1: SecChisel verification workflow. Square boxes represent files or data, ovals represent tools or processes. The unmodified Chisel tools (in white) can be used to generate the hardware design, while the new SecChisel components (in turquoise blue) perform the security verification. Black dotted line circles pre-existing baseline Chisel. Green dashed line includes whole SecChisel verification flow. Because the security tags are embedded in the source code of the design, single codebase can be used for both security verification as well as to generate the hardware design. The use of third-party modules and interference table is optional in addition to help support use of third-party IP within a design.

8.1.1 Verification Methodology

The goal of this work is to provide a design-time methodology to formally prove security properties of a secure processor architecture (this Section) and a practical framework using the methodology (Section 8.1.2). Works on run-time security checks, e.g., Sapper [129] or GLIFT [130], are complementary to this work.

Assumptions and Threat Model

Either through a bug in a hardware code, or due to a malicious designer or an adversary, some sensitive data may leak out to untrusted low-security outputs, a so-called "information leak." The framework checks, at design-time, if there are any such buggy or malicious flows of information. Using the information flow tracking, policy violations such as confidentiality violation or integrity violation can be detected. Information leaks via physical channels, such as EM radiation, are not considered as they cannot be expressed in today's hardware description languages. Hardware bugs or Trojans at design time are considered, but after-manufacturing bugs [131] are orthogonal to this work. The framework assumes a trusted compiler and toolchains to convert Chisel to an HDL and then the actual hardware.

Information Flow Tracking Approach

Our work uses information flow tracking (IFT) approach. Information flow refers to the transfer of information between different entities. Information flow can be explicit, e.g.,

$a = b$; where data or information in b goes to a; or it can be implicit, e.g., $b = 0$; if (a) then $b = 1$; where the value of b reflects whether a is true, but there is not a direct assignment, or copying of data, from a to b. Typically, when discussing information flow there are different security levels, e.g., a lower-security level ("Low") such as public data, a higher-security level ("High") such as secret key. Each data is associated with a security level, and information flow tracking can be used to check the security properties, e.g., no transfer of "High" to "Low" information (for confidentiality) or "Low" to "High" information (for integrity). Since information flow tracking has inherent presence of false positive, the SecChisel framework supports tagging at very fine granularity (individual bits) and using declassification and dynamic tags to minimize the false positives.

8.1.2 The SecChisel Framework

The proposed methodology is realized in a new SecChisel framework. The framework extends the existing Chisel language and tools with new security verification functionality. The SecChisel workflow is shown in Figure 8.1. SecChisel extends data variables (e.g., various wires, registers, or other parts of the design) of Chisel with security tags, allowing designers to annotate the design with the security tags associated with variables. During compilation, the SecChisel code is converted to a modified FIRRTL (Flexible Intermediate Representation for RTL) [132] and then translated to logical statements that can be used with the Z3 SMT solver [133], which checks for information flow violations based on the security tags. The SMT solver is used to assert that there are no data transfers between variables that could violate the security policy. The security verification steps can be done in parallel with compilation and simulation of a Chisel design. The whole SecChisel workflow consists of:

1. **SecChisel Code** – hardware description, including the security lattice description, the new security tags, the dynamic tag-range functions, and declassification.
2. **SecChisel Parser** – tool for generating the modified FIRRTL that contains both functional description of the design and information about the security tags.
3. **FIRRTL Code** – intermediate representation of the design with information about the security tags.
4. **SMT Code Generator** – tool for parsing FIRRTL into a FIRRTL statement/expression

tree, which is then processed into SMT statements used by an SMT solver.

5. **SMT Code** – code describing the security lattice, the tags, the dynamic tag-range functions, data flows, and the assertions for information flow checking.
6. **Parallelization** – tool that parallelizes the SMT code according to the number of processor cores available.
7. **Z3 SMT Solver** – tool that does the actual information flow checking and generates satisfiable or unsatisfiable result from the SMT code.
8. **Interference Table** – an optional step where third-party black-box modules can be used as part of the verification.

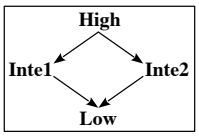
SecChisel Code (① in Figure 8.1)

SecCoreModule class extension and security policy. There is a new `SecCoreModule` class that extends the `CoreModule` class from Chisel and allows modules to have security lattices bound to them. Figure 8.2a shows the sample SecChisel code of a SHA-256 engine realized as a Rocket Chip RoCC. The *io.addend* and *io.accum* are input ports. The module is extended from the new base module `SecCoreModule` to allow its components to be tagged (line 1), i.e., each basic variable can be associated with a security tag for information flow analysis. All variables that are not tagged or within normal `CoreModule` modules have their security tags set to undefined by default. Undefined tags are resolved in the SMT Code Generator step.

The default security policy is for enforcing confidentiality: it is not allowed that variables bound to higher security tags leak their data to the variables with lower security tags. The policy is checked in Z3 SMT Solver, thus does not require additional specification. Integrity can be verified in a similar manner.

Security lattice definition. The base `Lattice` class contains the simplest possible security lattice with two levels: "High" and "Low", where "High" has greater security level than "Low". Any new security lattice can be created by extending a new object with the base `Lattice` class. Any two tags' values in the security lattice have a greatest lower bound (the meet) and least upper bound (the join). The meet or join operations are used to calculate the resulting tag value when variables are processed (Case 2 of Section 8.1.2).

```
1. class SHA256_init (implicit p: Parameters)
                                  extends SecCoreModule()(p){
2.  val io = IO (new Bundle {
3.    val addend = UInt(64.W).asInput
4.    val accum = UInt(64.W).asInput
5.    val initReady = Bool()
6.    val message_output = Vec(16, UInt(32.W)).asOutput})
7.  val message_size = io.accum(SIZE_MSB,SIZE_LSB)
8.  val process_id = io.accum(ID_MSB,ID_LSB)
9.  val message_in = Cat(io.addend(M_PART_ONE_MSB,
     M_PART_ONE_LSB),io.accum(M_PART_TWO_MSB,M_PART_TWO_LSB))
10.
11. // Specify use of custom lattice
12. override val lattice = SHA_Lattice
13.
14. // Define lattice structure
15. object SHA_Lattice extends Lattice {
16.  val Inte1 = NewLatticeElement()
17.  val Inte2 = NewLatticeElement()
18.  LOW < Inte1 < HIGH
19.  LOW < Inte2 < HIGH }
20.
21. // Dynamic value tag function
22. val SHA_TagRange =
23.  createTagRange(lattice.LOW).add(0, 130, lattice.Inte1)
24. .add(131, 500, lattice.HIGH).add(501,1000,lattice.Inte2)
25.
26. // Dynamic value tags
27. message_in :> (SHA_TagRange, process_id)
28.
29. for (i <- 0 to 16-1){
30.  io.message_output(i) :> (SHA_TagRange, process_id)}
31. …
32. }
```



(a) Example from SHA RISC-V Rocket Chip RoCC with custom security lattice shown in the sqaure box.

```
1. class KeyExpansion(implicit p: Parameters)
                                  extends SecCoreModule()(p){
2.  …
3.  val roundkey = Mem(16, UInt(width = 8))
4.  roundkey(0):= "h2b".U; …
5.  roundkey :> lattice.HIGH
6.  …
7. for(i<-0 to 176-1){
8.   io.data_output(i) :> (roundkey_propagate, lattice.LOW)}
```

(b) Example from AES RISC-V Rocket Chip RoCC.

Figure 8.2: Example from SHA and AES RISC-V Rocket Chip RoCC written in SecChisel code, new additions compared to base Chisel code are in bold. For the Chisel code not in bold, please refer to Chisel specification [134]. A custom security lattice is shown, of which "Inte1" and "Inte2" are extra security tags that have security level between "High" and "Low".

For each `SecCoreModule`, the designer can define the module's own security lattice or implicitly use the default `Lattice` class. Sub-modules are allowed to have different security lattices rather than use top module's security lattice. Within the object, new security lattice elements are defined, and the relationship among the elements is defined using the overloaded less-than $<$ operator to show security tag relations between each other. Figure 8.2a shows an example of a designer-defined new security lattice (lines 14 - 19) including a visualization of the security lattice for that class. The custom lattice will overwrite the default one (line 12).

Static tags. Static tags of variables in the design do not change their values throughout the verification process and always have the fixed value assigned by the designer. They are used when the designer is certain about a variable's security level for the whole life cycle of the system. In line 5 of Figure 8.2b, which shows SecChisel code for an AES engine realized as a Rocket Chip RoCC, the encryption key of AES RoCC is tagged as "High" security using the :> operator.

Dynamic tags. Dynamic tags are tags of which the value depends on the data value of other variables (wires, registers, etc.), which are known as the dependent variables. The value of the dynamic tag is a function of the dependent variables. The function outputs one of the values from the security lattice, based on rules specified by the designer, called "tag-range functions." A tag-range function for dynamic tags in SHA RoCC is defined in lines 21 - 24 of Figure 8.2a. The dynamic tags of *message_input* and *message_output* is determined by *process_id*, and is assigned by the overloaded :> operator (line 26 - 30 of Figure 8.2a) combined with the usage of tag-range function. In this case, the tag of the input message and output hash value will be either "High" or "Low" determined by *process_id* and all of the undefined tags in the sub-modules will resolve to have the same dynamic tags. The tag-range function can also be made to support multiple dependent variables by defining sets of ranges.

Declassification. Information flow will always report a violation if there is information flow from "High" to "Low" (for confidentiality). Sometimes, however, this kind of information flow should be allowed. For example, in lines 7 - 8 of Figure 8.2b, the final output *data_output* of AES RoCC is declassified to be "Low" using :> operator. Although the output depends on the secret key and will be tagged as "High", since strong encryption is assumed, the

output conveys no information to the attacker, and thus, it can be *declassified* to "Low" value output.

When using declassification, the system designer might overwrite some rules and use declassification improperly, causing a false negative. Our tool reports the number of declassifications used (to allow users to compare it with the expected number) and also gives warnings about declassification.

Nested modules. Chisel and SecChisel both support describing a design with nested modules. To analyze the information flow, the nested modules will be resolved in SMT Code Generator described in Section 8.1.2.

SecChisel Parser (② in Figure 8.1)

Given SecChisel code, it needs to be parsed into the modified FIRRTL code. The parser is based on Chisel parser, but it includes the tags information for the variables (especially, it marks untagged variables as undefined). Moreover, Chisel sometimes transparently defines new temporary variables during compilation which are not in the original Chisel source code. These will be tagged as undefined in the modified FIRRTL code.

FIRRTL Code (③ in Figure 8.1)

FIRRTL language was created to represent the standardized, elaborated circuit produced from Chisel code [132]. It can be efficiently used to analyze the information flow. SecChisel does not modify FIRRTL language's syntax. Instead, the security information is embedded in the comments section of each line of FIRRTL code. When analyzing the FIRRTL, Chisel's default tools will ignore the comments so that the hardware can be generated directly from the SecChisel code without any changes to the back-end of the Chisel tool-chain. Meanwhile, when FIRRTL is analyzed by the SMT Code Generator, the security information is included to generate the SMT code. So the verification and the final hardware are based on the same design in FIRRTL.

SMT Code Generation (④ in Figure 8.1)

SMT Code Generator converts FIRRTL into an expression/statement tree and then generates SMT code. Processing the FIRRTL tree to SMT statements is the key part of the SecChisel framework, especially when dealing with nested modules. The four phases for transforming FIRRTL code into SMT code are:

(Phase 1) Parse the FIRRTL file and create $L_{taggedVariable}$ structure to store variables and the corresponding explicit tag information in the structure, untagged variables will have no tags associated with them yet.

(Phase 2) Create tags for all variables: apart from variables explicitly tagged by the designer in SecChisel, variables with no tags are tagged with $UndefinedTag$, and all data is stored in new $L_{default}$ structure.

(Phase 3) Resolve all undefined tags in $L_{default}$ through nested modules of the circuit and store the data in the $L_{redefine}$ structure.

(Phase 4) Output SMT code, F_{SMT}, based on security lattice, tag-range functions and tag information in $L_{redefine}$.

In Phase 1, data structure $L_{taggedVariable}$ is generated to store security information derived from FIRRTL file. The data structure $L_{taggedVariable}$ contains the variables and their tags' information: "statically tagged variable" has explicit security tag defined by the system designer, "dynamically tagged variable" has tag-range function and the dependent variable(s) defined by the system designer, and "untagged variable" does not have any tags assigned, i.e., such variables have no defined tags in the SecChisel code.

In Phase 2, tags for variables associated with the left-hand side (lhs), or the right-hand side (rhs), of statements are created based on information from variables already tagged in $L_{taggedVariable}$. After this phase, all the tag information will be stored in $L_{default}$ structure. Especially, there are seven types of FIRRTL statements. In order to simplify tag assignment, these seven types of FIRRTL statements can be classified into the following three cases:

(Case 1) Definitions: a variable is defined to be a constant,

e.g., $a = 12$.

(Case 2) Assignments: a variable is assigned the results of some operations of other variables,

Algorithm 6 *redefineTags* (*statement*, *variable*, *curModule*)

Input: *statement*: a line of FIRRTL code containing the variable of $L_{default}$, whose tag needs to be redefined
variable: the variable (e.g., Reg, Wire, etc.) whose tag needs to be redefined
curModule: the module that is being checked
Output: redefined tag for *variable* of $L_{redefine}$

```
1: if variable has been assigned defined tag then
2:     return defined tag
3: else
4:     for each statement x ∈ L_default of curModule do
5:         if x.lhs == variable then
6:             if tag of x.lhs is defined (or tag of x.rhs is defined) then
7:                 tag of statement.rhs ⇐ tag of x.lhs (⇐ tag of x.rhs)
8:                 return tag of statement.rhs
9:             else
10:                find submoduleList of current module
11:                tag of statement.rhs ⇐ joinRedefineTags (statement, rhs, submoduleList, curModule)
12:                tag of x.rhs ⇐ tag of statement.rhs
13:                tag of x.lhs ⇐ tag of x.rhs
14:                return tag of statement.rhs
15:            end if
16:        end if
17:    end for
18: end if
19: if cannot find statement.rhs then
20:     if curModule has outer module then
21:         tag of statement.rhs ⇐ redefineTags (statement,
            statement.rhs, outer module)
22:         return tag of statement.rhs
23:     else
24:         tag of statement.rhs ⇐ lowest tag of its security lattice
25:         return tag of statement.rhs
26:     end if
27: end if
```

e.g., $b = c + d$.

(Case 3) Connections: a variable is assigned to have the same value as a different variable,

e.g., $e = f$.

For Case 1 and Case 3, if the tag already has an static or dynamic tag value assigned, the tag will be kept; otherwise, the $UndefinedTag$ value will be assigned to the tag. Exceptionally, port variables of top modules without tags are assigned to the default lowest security level to guarantee no information will implicitly leak outside the circuit. For Assignments (Case 2), rather than generate a specific static or dynamic tag, a join statement following three ResRules defined next using tags of the rhs variables in the statement is generated. The tag resolution rules (ResRule) for statements of variables A and B on the right-hand side are:

(ResRule 1) $(Tag_A, Tag_B) \Rightarrow (join\ Tag_A\ Tag_B)$

(ResRule 2) $(Tag_A, UndefinedTag_B) \Rightarrow UndefinedTag$

(ResRule 3) $(UndefinedTag_A, UndefinedTag_B) \Rightarrow$

$UndefinedTag$

Here the $(join\ Tag_A\ Tag_B)$ does not compute the join, but is an SMT statement that will be evaluated in the SMT solver. The Tag_A or Tag_B could end up being resolved as join of some other tags following tag relations defined in security lattice, so all the join operations are computed in the SMT solver at the very end. They can be either static or dynamic.

In Phase 3, $L_{default}$ will be used to generate list $L_{redefine}$, where all the $UndefinedTag$s stored in list $L_{default}$ will be checked recursively using *Algorithm* 6 to go through nested modules until there is an assignment statement that assigns some defined tag to the currently $UndefinedTag$; this can be a static tag, a dynamic tag, or a generated join statement following ResRules.

Nested modules support in SecChisel will resolve all the $UndefinedTag$ first in the current module and then in different sub-modules and outer modules. One variable can be referenced in different layers of module hierarchy. In addition, dynamic tags are resolved to support dependent variables which are in other parent modules or in sub-modules. Specifically, in "redefineTags" – *Algorithm* 6, function "joinRedefineTags" will go through nest modules ("submoduleList") of the current module ("$curModule$"), find and calculate the tag of the variable using "redefineTags" function recursively. Recall that the top-most `SecCoreModule` must have its inputs and outputs explicitly tagged, so eventually, all variables that have some connection to input or output will be assigned a definite tag. Only variables unconnected to the rest of the circuit will remain with $UndefinedTag$, and these will be later synthesized away anyway.

In Phase 4, the security lattice structure, tag-range functions and the $L_{redefined}$ are used to generate SMT format rules and assertions and output an SMT file F_{SMT}.

SMT Code (⑤ in Figure 8.1)

The SMT code file, F_{SMT}, contains the information flow assertions generated based on the FIRRTL code following previous steps. To enable the assertions to work, it also needs the security lattice and tag-range functions expressed as SMT statements. It contains tag

propagation rules which use join of multiple security tags when computing the right-hand side variable's security tag of statements discussed in Section 8.1.2. In order to speed up verification in SMT solver, the SMT Code Generator pre-computes "*join*" results for the variable pairs in the security lattice. Therefore, SMT solver can directly fetch the result when "*join*" operation happens.

For dynamic tags, each tag-range function has a unique ID used to look up the function in the SMT code. SMT solver will enumerate all the possible output (tag) values that the tag-range function can generate for a dynamic tag, based on the dependent variable. Thus, a *join* Tag_A Tag_B statement may resolve to many possible tag values, if either Tag_A or Tag_B are dynamic tags.

SMT Code Parallelization (⑥ in Figure 8.1)

The Z3 SMT solver does the actual verification. Because of the very structured nature of the SMT code the framework generates, it is possible to parallelize the assertion checking. For the SMT code there are two parts in each SMT file: 1) the predefined rules for the security tags, tag-ranges, security lattice, and 2) actual assertions used for checking every operation's information flow. The rules are needed for all assertions, but the assertions can be checked independently of other assertions. The SMT code can be then parallelized by converting source SMT file into n different files, where each file has the same rules, etc., but the assertions are evenly split into the n files and can be processed in parallel.

Z3 SMT Solver (⑦ in Figure 8.1)

The SMT file is used as input to the Z3 SMT solver, but other solvers could be used as long as they can parse the same SMT-lib syntax. The assertions check for violations of information flow policy, thus somewhat counter-intuitively, if an assertion is "satisfied" there is a violation of information flow policy. If an assertion is "unsatisfied" there is no violation of information flow policy. The goal is to have all assertions be "unsatisfied". E.g., Chisel code "c := io.a", assertions corresponded can be "(assert (< c io.a)) (check-sat)" to ensure that assigned variable "c" should not have lower security level than "io.a", in which case information will be leaked.

Interference Table

Sometimes the design will make use of black-box third-party modules. It may not be possible to directly analyze the information flow inside such modules (e.g., no source code is given). The trusted creator of the third-party module can generate an *interference table* which lists how the inputs interact with the outputs for the module, i.e., the information flow from inputs to outputs. The interference table can then be used for the designers to reason about information flow across black-box third-party modules included by the designer in his or her design. The interference table can be generated directly using SecChisel code and done in parallel with SecChisel Parser without influencing the main SecChisel flow. The table and FIRRTL can both be used as input to SMT Code Generator.

8.1.3 Evaluation of the Framework

SecChisel framework is implemented upon Chisel [134][1] and the system complexity of the new framework is shown in Table 8.1 for each of the core parts of SecChisel.

To evaluate the effectiveness and performance of SecChisel, an AES-128 and a SHA-256 accelerators were implemented as Rocket Custom Coprocessor Interface (RoCC) within the Rocket Chip [135] RISC-V processor. The functionality and interoperability of AES RoCC and SHA RoCC within the Rocket Chip were tested to ensure functional tests pass. The SecChisel framework can process the whole Rocket Chip as it can handle both `SecCoreModule`s and the unmodified `CoreModule`s. Since SecChisel is a superset of Chisel, most of the code of Rocket Chip is unmodified, only the two accelerators and corresponding sub-modules are written as `SecCoreModule`s. We evaluate RoCC cores within a RISC-V core. Our framework works with the whole Rocket Chip and can find improper information flows due to bugs or hardware Trojans. The evaluation was done using a server with two Intel Xeon E5 CPUs (total of 24 processor cores) running at 2.90GHz, with 64GB of memory.

1. Commit id: bb12fe7 from Chisel repository at `https://github.com/ucb-bar/chisel`.

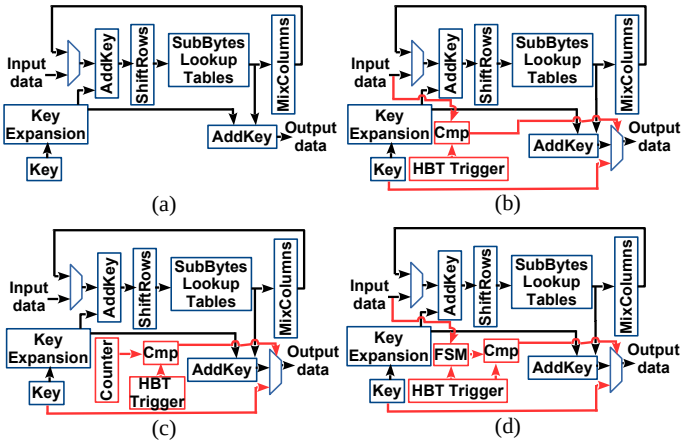


Figure 8.3: Block diagrams of AES-128 RoCC encryption modules without and with hardware bugs or Trojans. HBT components are shown in lighter color in the figures. (a) Basic AES encryption module. (b-d) Encryption module with HBT1, HBT2, and HBT3 hardware bugs or Trojans.

AES RoCC Implementation & Verification

Firstly, a full 10-round AES-128 was implemented as an RoCC of Rocket Chip. AES-128 RoCC encryption block diagram is shown in Figure 8.3a, note decryption process is symmetric to encryption. In our sample AES RoCC implementation, security lattice structure used can be seen in Figure 8.2a. In the first sub-module – $KeyExpansion$ of encryption process, the encryption Key is bound to have tag "High". Other variables in the AES RoCC are untagged in the design.

Without use of declassification ("AES RoCC v1" in Table 8.2), running the whole verification of AES RoCC results in detection of a possible information leak, where the encrypted output is tagged "High" because of the interaction with the secret key, but connects to the "Low" output of the RoCC module. In order to remove this false positive, declassification is used ("AES RoCC v2" in Table 8.2). Especially, the encrypted output can be declassified from "High" to "Low", because assuming AES is a cryptographically strong algorithm, the encrypted data cannot be used to learn the plaintext. Now, there will be no more violation of information flow policy and there will be no false positives. Verification results illustrated above are shown in Column "Formal Verification Result" of Table 8.2. The decryption module can be verified similarly.

In order to further prove the effectiveness of our framework, we insert three kinds of hardware bugs or hardware Trojans into AES RoCC (denoted as "HBT"), as shown in

Figure 8.3 (b-d) and "AES RoCC v2 with HBT1/HBT2/HBT3" in Table 8.2. HBT1 outputs the key when a special input data trigger is sent to the AES RoCC. HBT2 inserts a register and a time counter inside AES RoCC and the key will be output when counter is added to some trigger value. HBT3 inserts a finite state machine inside AES RoCC: when a series of special input data triggers is processed by AES RoCC in a specific order, HBT3 will output the key. These HBTs all send "High" security key to the output. Table 8.2 shows that SecChisel is able to detect all of the HBTs as it finds information flows from "High" data to "Low" outputs of the modules.

SHA RoCC Implementation & Verification

SHA-256 RoCC was implemented in Rocket Chip to test static tags ("SHA RoCC v1" in Table 8.2) and dynamic tags ("SHA RoCC v2" in Table 8.2) of SecChisel. SHA-256 is a secure hash algorithm that is used to generate digests of messages to detect if the message has been changed. The inputs are message, message size and *process_id*. Initialization vector and other constants are hardcoded in the SHA-256 RoCC. The output is the hash value that is computed.

SHA can process both secret and public information (there is no secret key, just hash function). Therefore, if only using static tags for SHA RoCC, there are possibilities that input message is tagged "High" and output hash value is tagged "Low", where false positives will be detected (Table 8.2). Using dynamic tags, the input and output of the SHA RoCC are both tagged with a dynamic tag, which depends on a *process_id* sent from input. This *process_id* represents the ID provided by the system and will determine whether this message can be open to public or not (*process_id* is assumed to be securely provided). In this case the tag of input message and output hash value will be either "High" or "Low" determined by *process_id*. When propagating tags in the inner sub-modules, all of the undefined tags will resolve to have the same dynamic tags, ensuring no false positives as shown in Table 8.2.

Table 8.1: System complexity of the SecChisel framework in terms of lines of code.

	Modified	Added	Total
SecChisel Parser	377	77	454
SMT Code Generator	4	2442	2446
Interference Table Generation	—	239	239
SMT Code Parallelization	—	28	28

Table 8.2: Effectiveness and designer effort in terms of lines of code of AES RoCC and SHA RoCC within Rocket Chip. "Formal Verification Result" shows effectiveness. FP represents False Positive verification result. "Chisel" column shows complexity of design in Chisel, without any security features. "SecChisel" column shows extra lines of code added to include information of security tags.

Module	SecChisel Features Used			Formal Verification	Chisel	SecChisel
	Static Tag	Dynamic Tag	Declass-ification			
AES RoCC v1 w/ static tags	✓	×	×	found FP	1097	+14
AES RoCC v2 w/ declassification	✓	×	✓	verified	1097	+17
AES RoCC v2 w/ HBT1	✓	×	✓	found HBT	1107	+17
AES RoCC v2 w/ HBT2	✓	×	✓	found HBT	1110	+17
AES RoCC v2 w/ HBT3	✓	×	✓	found HBT	1121	+17
SHA RoCC v1 w/ static tags	✓	×	×	found FP	1127	+25
SHA RoCC v2 w/ dynamic tags	×	✓	×	verified	1127	+28

Designers' Effort

SecChisel requires the system designers to add extra code to describe information flow policy and tags, as illustrated in Section 8.1.2. Table 8.2 Column "Chisel" shows the lines of code designer needs to write to implement AES RoCCs (including ones with HBTs) and SHA RoCC. Column "SecChisel" shows that tested modules require only tens of lines of code in order to do verification based on the original Chisel design. SecChisel implementation requires system designer to explicitly add security tags to critical variables, however, most of the variables in a module will have default (undefined) tags and the tags will be automatically resolved in SMT code generation step, and require no designer effort to specify such tags.

Security Verification Performance

Figure 8.4 shows compilation plus simulation runtime and verification time as well as different parts of verification runtime for AES RoCC and SHA RoCC using different SecChisel features, and compares it with the runtime of unmodified Chisel tools for reference. Compilation includes all original Chisel flow before simulation shown in Figure 8.1. Verification consists of whole SecChisel verification flow circled by green dashed line in Figure 8.1. The verification runtime is in all cases slightly less than the compilation plus simulation runtime and the two can be done in parallel. The plain AES and SHA RoCC written in Chisel have no

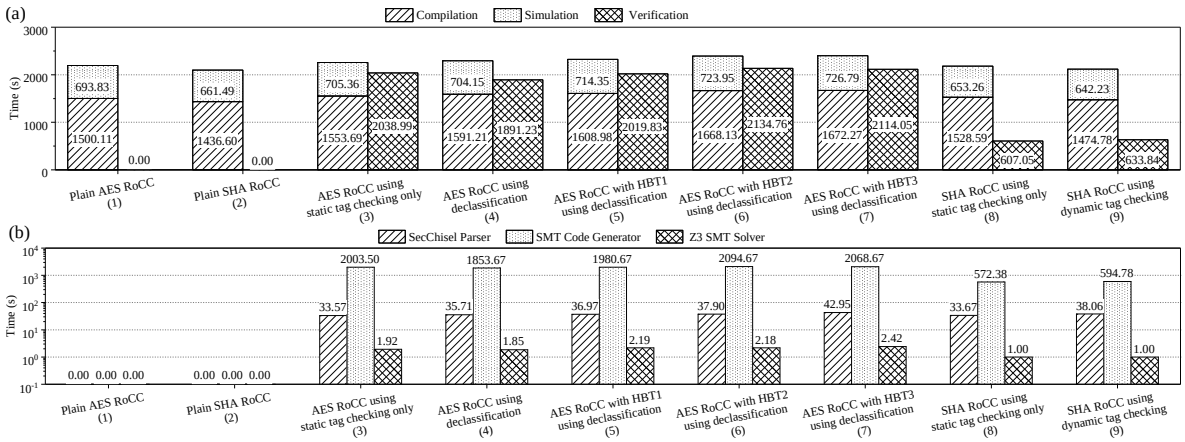


Figure 8.4: Evaluation of runtime of the SecChisel framework, times shown are averages of multiple runs. (a) Comparison of total compilation and simulation time vs. verification time for the different designs. (b) Runtime of SecChisel parser, SMT Code Generator, and Z3 SMT solver for the different designs.

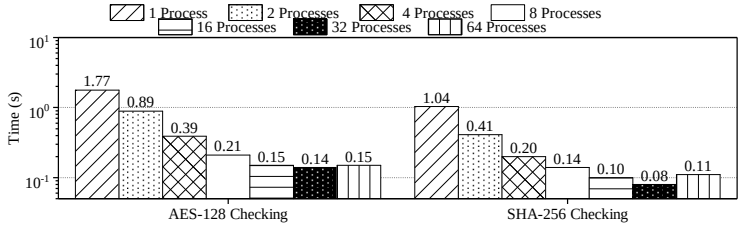


Figure 8.5: Runtime evaluation of effects of parallelizing the SMT code for different number of processors, on a server with 24 cores.

SecChisel features, so no related verification performance. Some small differences are due to the variation in performance of the server.

Shown in Figure 8.4a, SecChisel verification will not cause extra overhead to the original Chisel design. Verification can be hidden in normal compilation and simulation since the total verification time is always smaller than the compilation plus simulation time and the verification can be done in parallel with compilation and simulation. SecChisel's performance time overhead of changing RoCC to a secure module class is relatively small (AES RoCC's is around 3.8%, SHA RoCC's is around 2.5% by comparing Design 1, 2 with the other Designs in Figure 8.4a, respectively). As shown in Figure 8.4b, there is no significant performance time difference for the different cases. Finding the existence of an information leak by HBTs (Design 5, 6, 7 in Figure 8.4b) only introduced small time overhead comparing with Design 4 that does not have HBTs in it, for example.

Interference Table Evaluation

Interference table (IT) can be created during FIRRTL generation and will on average require 0.41s for AES RoCC and SHA RoCC, which is negligible compared with FIRRTL generation time shown in the SecChisel parser time in Figure 8.4b. SMT code generation with and without IT do not change much on the total runtime, but is offered a solution for use of trusted third-party modules for which source code is not available.

Z3 SMT Solver and Parallelization

Verifying the AES RoCC v2 and SHA RoCC v2 in Z3 SMT solver without parallelization cost 1.77s, 1.04s, respectively (Figure 8.5). For reference, AES RoCC generates 20832 lines of SMT code and SHA RoCC generates 9044 lines of SMT code, which is fully automatically generated based on the SecChisel code. Therefore, there is an approximately linear relationship between lines of SMT code and run time, including using different SecChisel features like dynamic tags, as shown in Figure 8.4b. Based on our experience with AES and SHA RoCC, we can estimate that one million of lines of Chisel code (far more than the current Rocket Chip code) will require around acceptable 2000s SMT runtime.

Parallelizing the SMT checks and running on multi-core system, as discussed in Section 8.1.2, can further reduce the run time of the SMT solver, as shown in Figure 8.5. For evaluation on a 24-core processor, the average improvement is about 20x.

Hardware Performance

The final design generated using the SecChisel code is identical to a design that would not contain any security tags or other SecChisel modifications. Specifically, SecChisel does not add any run-time components to the design as all the security verification is done at design-time. Therefore, the verified design will not add any performance overhead compared with the original design.

Chapter 9

Conclusion and Future Directions

This dissertation presented research on processor microarchitecture security, with a special focus on covert-channel and side-channel attacks and defenses.

9.1 Conclusion

Chapter 3 first proposed a new three-step model in order to model all possible cache timing vulnerabilities. It further provided a cache three-step simulator and reduction rules to derive effective vulnerabilities, allowing us to find ones that have not been exploited in literature. With the exhaustive effective vulnerability types listed, this work presented analysis of 18 secure processor cache designs with respect to how well they can defend against these timing vulnerabilities. It showed that vulnerabilities based on internal interference of the victim application are difficult to protect against and many secure cache designs fail to protect against them. A summary of secure processor cache features may also provided that could be integrated to make an ideal secure cache that is able to defend timing attacks.

Chapter 4 combined three works that put theoretical modeling approach into the practice. The first work further improved the three-step model and provided the first benchmark suite for evaluating all 88 possible Strong cache timing attacks types in processors. The model allowed us to find 32 new timing attack types. Further, scripts were implemented to auto-generate the 1094 benchmark tests from our three-step model's 88 theoretical attack types for testing different combinations and types of instructions that can lead to attacks

on real processors. The benchmarks were run on a number of commodity processors to gave each machine the Cache Timing Vulnerability Score (CTVS) to measure the degree of the machine's robustness against cache timing vulnerabilities. The three-step model, benchmarks, and the CTVS can be used to measure existing systems and help design future secure caches and other defense mechanisms. The second work presented for the first time a large-scale evaluation of 34 Arm devices against the 88 types of vulnerabilities. In total, three different cloud platforms were leveraged for the evaluation, and *gem5* was used for further analysis of certain microarchitectural features. Based on the evaluation results, the work uncovered a number of components of the microarchitectual design that influence the effectiveness of different types of the vulnerabilities. Further, sensitivity tests were used to understand impacts of possible misconfiguration on the outcome of the benchmarks, and also showed that even with uncertain cache configuration, number of attack types can be successful. To help defend the attacks, the PL and RF secure caches were implemented and evaluated on *gem5*. Based on the benchmarking results of the secure caches, a new attack on PL cache, and possible issues due to small window size in the RF cache were uncovered. The third work further extended the three-step modeling approach to exhaustively enumerate all possible TLB timing vulnerabilities. It showed how to automatically generate micro security benchmarks that test for the TLB vulnerabilities. It gave details of two new hardware secure TLB designs: a Static-Partition (SP) TLB and a Random-Fill (RF) TLB. The simulations confirmed the theoretical channel capacity calculations and full system performance on FPGA showed that the new secure TLBs are as good as regular TLBs, while protecting against the various attacks. The proposed secure TLBs can defend not only against the previously publicized attacks, but also other possible timing attacks in TLBs found using our three-step modeling approach.

Chapter 5 demonstrated microarchitectural attacks beyond caches and TLBs. The attacks focused on understanding a special type of predictor, the value predictor, and demonstrated new security attacks on these predictors. The systematic model for analyzing value predictor attacks demonstrated different variants of attacks utilizing value predictors to leak information. Security techniques were discussed for securing value predictors. Defenses were suggested to be used when value prediction is implemented in real processors.

Chapter 6 demonstrated attacks on the processor frontend. It evaluated new security threats due to the processor frontend in modern Intel processors. Each frontend path has its own unique timing and power signatures, which lead to the side- and covert-channel attacks presented in this work. Especially, the switching between the different paths lead to observable timing or power differences which, as this work demonstrated, could be exploited by attackers. Because of the different paths, the switching, and way the components are shared in the frontend between hardware threads, two separate threads are able to be mutually influenced and timing or power can reveal activity on the other thread. The security threats are not limited to multi-threading, and this work further demonstrated new ways for leaking execution information about SGX enclaves or a new in-domain Spectre variant. Finally, this work demonstrated a new method for fingerprinting the microcode patches and applications running in parallel of the processor by analyzing the behavior of different paths in the processor frontend.

Chapter 7 introduces two microarchitectural attacks that exists in accelerators beyond the classical processors. The first type of attack targeted GPUs. Because of the hardware parallelism features and multi-context sharing setting, GPUs are also under the threats of side- and covert-channel attacks. The second type of attack focused on quantum computers, used as cloud-based accelerators, where potential security threats were also detected. It demonstrated the new threat of fingerprinting of quantum computers using crosstalk, and evaluated the approach on IBM Q cloud-based quantum computers. The device- and location-specific fingerprinting were demonstrated with accuracy to be 99.1% and 95.3%, respectively. We showed excellent fingerprinting abilities across many machines and across different calibration periods.

Chapter 8 showed the hardware security verification framework called SecChisel, which is the first hardware security verification framework based on Chisel that is also the first hardware security verification framework supporting nested modules, without having to check individual module separately. SecChisel was tested by implementing and verifying AES-128 and SHA-256 RoCC accelerators within a Rocket Chip RISC-V processor. We showed that SecChisel is able to detect information leaks due to hardware bugs or Trojans, and that SecChisel is fast and scalable when verifying designs.

9.2 Future Directions

To achieve the vision of designing high performance and secure computer architectures, the work on systematic hardware security analysis and the microarchitectural vulnerability study of various structures is only an initial step. Looking forward, there are a wider range of hardware security problems to be explored.

Further customization of three-step model for other structures. As the next future step in this direction, it is possible to customize the model to other microarchitectures for systematic analysis of the side-channel vulnerabilities, such as the branch target buffer or cache directory. Since more and more microarchitectures are shown to be vulnerable to side-channel attacks, it is important to consider *the security holistically among different microarchitectures*. An interesting problem is that, when combining different security protection schemes, how to guarantee these schemes do not interference and backfire, jeopardizing system security? For example, recent Spectre LVI attack makes use of line fill buffer to load value injection. Line fill buffer, on the other hand, inclusively or exclusively work with the cache to store data and instruction for different machines. In this case, line fill buffer and cache should be analyzed together to systematically explore the related side-channel vulnerabilities.

Design of frontend defence techniques. Based on the current understanding of real processor frontend, the next research step is to propose effective defense solutions to mitigate all frontend attacks known to date. Defending the frontend vulnerabilities will require new approaches for the frontend design. Common techniques such as partitioning to prevent interference among SMT threads could be a way to begin designing a more secure frontend. In principle, LSD, DSB, and MITE pipeline should all be partitioned to ensure isolation, however, it is not done in the current commercial processors. The likely reason is that partitioning would largely wipe out the performance benefits that are brought by these components in the first place. As a step to preserve performance benefits while ensuring security, it may be possible to introduce randomness into MITE mapping or add random operations to MITE. For the side-channel fingerprinting attack we developed, randomly inserting instructions into MITE pipeline could de-correlate the access patterns.

Study of side-channel vulnerabilities in GPUs. GPUs are used to accelerate security-critical and efficiency-conscious systems, including data centers, autonomous vehicles (AV), cloud gaming services, and game consoles. System isolation guarantees that different users and programs cannot extract information or data from another without explicit communication through supported channels. However, GPU partitioning features, e.g., Multi-instance GPU or MiG, can be potentially exploited to enable the transparent co-operation among multi-process GPU applications, making GPUs vulnerable to covert-channel attacks. It is possible to keep exploring covert channels constructed over memory systems of GPU and functional unit contention. It is an important problem because, with the parallelism feature of GPU hardware, the bandwidth of covert channel can be generally increased much more than the CPU covert-channel attacks to create *high-speed and high-fidelity covert channels*. Specifically, it could be possible to extend the three-step model to GPUs as well and find new timing channels, attacks, and develop defenses.

Securing integrated and heterogeneous systems. The current trend in systems-on-chip (SoCs) designs are system-level integration of heterogeneous components onto the same chip in mobile devices and other systems, including different types of processors such as CPU, GPU, and Accelerated Processing Unit, known as APU. Such diverse processing elements may come from different providers and vendors, e.g., Google, Microsoft, and Intel. On the other side, the application executable codes typically have varying levels of security and trust. Thus, executing them together on the same compute platform with many shared resources creates an extremely fertile attack ground. Nowadays, the conventional approaches and software-only add-on schemes have failed to provide sufficient security protections and trustworthiness for such integrated and heterogeneous systems. Without the protection, while the user applications are running on the platform, the offloaded data from the cloud to the devices can lead to potential side-channel issues. The future research can extend the research methodology and construct benchmarks to systematically analyze timing attacks in the integrated and heterogeneous systems.

Securing AI accelerators. Many companies such as Google and NVIDIA are deploying domain-specific accelerators for AI. The most well-known instance is TPU [136], which can outperform GPU in the inference stage of DNN execution. Despite the intensive interests from

both academia and industry, there has been no existing systematic approach to analyzing the side channels of AI accelerators. Similar to GPU, TPU also requires high bandwidth for use of very large effective batch sizes and current TPU does not have local caches. In this case, it is expected that the similar high-speed and high-fidelity timing channels will be applicable in TPU and require systematic analysis and defense. One particular challenge is to study the SoC of AI ASICs. For example, network-on-chip communication can potentially introduce different types of timing variances compared with timing channels within the processor. One may need to develop a theoretically different modeling approach in order to systematically study the timing channels of SoC, for example.

Securing quantum computing systems. In the current Noisy Intermediate-Scale Quantum (NISQ) era, the capability of a quantum machine is limited by the decoherence time, gate fidelity and the number of qubits. As NISQ machines exhibit high error-rates, only programs that require a few qubits can be executed reliably. Therefore, NISQ machines tend to underutilize its resources. As larger NISQ machines are introduced, the throughput and utilization can be improved with multi-programming or running programs from different users. However, the dominant use-case of providing cloud-based access for remote users to rent quantum computers opens up new security and privacy threats. Future work can further explore the crosstalk influence upon quantum computer architectures. Apart from that, the preliminary analysis of quantum computer control logic [137], typically implemented using classical FPGAs, reveals that there are potential security issues with the design. Timing channels seem to exist regarding different qubit reset behavior for the quantum computer control units. Future research can focus on the control algorithms running on the classical FPGAs, which configure and measure the quantum qubits. It would be an important topic to study since qubit reset is generally used for normal quantum circuits and hard to directly avoid or mitigate.

Appendix A

List of Acronyms

AES Advanced Encryption Standard.

AI Artificial Intelligence.

ALU Arithmetic Logic Units.

ASIC Application-Specific Integrated Circuit.

ASLR Address Space Layout Randomization.

CNN Convolutional Neural Network.

CNOT Controlled-NOT gate.

CPU Central Processing Unit.

CTVS Cache Timing Vulnerability Score.

CUDA Compute Unified Device Architecture.

CVE Common Vulnerabilities and Exposures.

DRAM Dynamic Random-Access Memory.

DSB Decode Stream Buffer.

EdDSA Edwards-curve Digital Signature Algorithm.

EEPROM Electrically Erasable Programmable Read-Only Memory.

EM Electro-Magnetic.

EPROM Erasable Programmable Read-Only Memory.

FIRRTL Flexible Intermediate Representation for RTL.

FP Floating Point.

FPU Floating Point Units.

FPGA Field Programmable Gate Arrays.

GPC GPU Processing Cluster.

GPU Graphics Processing Unit.

HBT Hardware Bugs or Trojans.

HDL Hardware Description Language.

IDQ Instruction Decode Queue.

IDT IDle Tomography.

IDQ Instruction Decode Queue.

IFT Information Flow Tracking.

INT Integer.

IP Intellectual property.

IPC Instructions Per Cycle.

ISA Instruction Set Architecture.

IT Interference table.

L1 Cache Level-1 Cache.

L2 Cache Level-2 Cache.

LCP Length-Changing Prefix.

LLC Last-Level Cache.

LRU Least Recently Used.

LSD Loop Stream Detector.

LVI Load Value Injection.

MIG Multi-Instance GPU.

MITE Micro-Instruction Translation Engine.

ML Machine Learning.

MT Multi-Threading.

NISQ Noisy Intermediate-Scale Quantum.

OS Operating System.

PC Program Counter.

PL Partition-Locked.

PROM Programmable Read-Only Memory.

RAPL Running Average Power Limit.

RAT Register Alias Table.

RF Random Fill.

RISC-V Reduced Instruction Set Computer, fifth generation.

RoCC Rocket Custom Coprocessor.

ROM Read-Only Memory.

RSA Rivest–Shamir–Adleman, is a public-key cryptosystem.

RTL Register Transfer Level.

SFU Special Function Units.

SGX Software Guard Extensions.

SHA Secure Hash Algorithms.

SIMD Single Instruction Multiple Data.

SM Streaming Processor.

SMT Simultaneous MultiThreading.

SMT Solver Satisfiability Modulo Theories Solver.

SoC Systems-on-Chip.

SP Static-Partition.

SRAM Static Random-Access Memory.

SRB Simultaneous Randomized Benchmarking.

SSD Solid State Drives.

TLB Translation-Lookaside Buffer.

TPC Texture Processing Cluster.

TPU Tensor Processing Unit.

VHist Value History.

VPS Value Prediction System.

Appendix B

List of Publications

- **Shuwen Deng**, Bowen Huang, and Jakub Szefer. "Leaky Frontends: Micro-Op Cache and Processor Frontend Vulnerabilities". Accepted by the the 28th IEEE International Symposium on High-Performance Computer Architecture (**HPCA**), 2022.
- **Shuwen Deng**, Nikolay Matyunin, Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. "Evaluation of Cache Attacks on Arm Processors and Secure Caches". Accepted by IEEE Transactions on Computers (**IEEE TC**), 2021.
- **Shuwen Deng** and Jakub Szefer. "New Predictor-Based Attacks in Processors". Accepted by the 58th Design Automation Conference (**DAC**), 2021. Acceptance rate of **23%**.
- **Shuwen Deng**, Wenjie Xiong, and Jakub Szefer. "Understanding Insecurity of Processor Caches Due to Cache Timing-Based Vulnerabilities". In Journal of IEEE Security & Privacy, 2021.
- **Shuwen Deng**, Wenjie Xiong, and Jakub Szefer. "A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks". Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**), 2020. Acceptance rate of **18.10%**.
- **Shuwen Deng**, Wenjie Xiong, and Jakub Szefer. "Secure TLBs". Proceedings of the 46th International Symposium on Computer Architecture (**ISCA**), 2019. Acceptance

rate of **16.98%**. **(Top Picks in Hardware and Embedded Security 2021)**

- **Shuwen Deng**, Wenjie Xiong, and Jakub Szefer. "Analysis of Secure Caches using a Three-Step Model for Timing-Based Attacks". In Journal of Hardware and Systems Security (**JHSS**), 2019.

- **Shuwen Deng**, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y. Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. "SecChisel Framework for Security Verification of Secure Processor Architectures". Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (**HASP**), 2019.

- **Shuwen Deng**, Wenjie Xiong, and Jakub Szefer. "Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic". Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (**HASP**), 2018.

- Allen Mi, **Shuwen Deng**, and Jakub Szefer. "Device- and Locality-Specific Fingerprinting of Shared NISQ Quantum Computers". the 10th International Workshop on Hardware and Architectural Support for Security and Privacy (**HASP**), 2021.

- Ferhat Erata, **Shuwen Deng**, Faisal Zaghloul, Wenjie Xiong, Onur Demir, and Jakub Szefer. "Survey of Approaches for Security Verification of Hardware/Software Systems." Cryptology ePrint Archive (2022).

- Wen Wang, Bernhard Jungk, Julian Walde, **Shuwen Deng**, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. "XMSS and Embedded Systems - XMSS Hardware Accelerators for RISC-V". Proceedings of the 34th International Conference on Selected Areas in Cryptography (**SAC**), 2019.

Bibliography

[1] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[2] F. Liu and R. B. Lee. Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 203–215. IEEE, 2014.

[3] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, January 2018.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[5] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[6] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical Report*, 2018. See also USENIX Security paper Foreshadow [138]. `http://ForeshadowAttack.com`.

[7] M. Yan, C. Fletcher, and J. Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. *arXiv preprint arXiv:1808.04761*, 2018.

[8] S. Deng, W. Xiong, and J. Szefer. Analysis of secure caches using a three-step model for timing-based attacks. *Journal of Hardware and Systems Security*, Nov 2019.

[9] S. Deng, W. Xiong, and J. Szefer. A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks. In *Proceedings of 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 683–697, 2020.

[10] S. Deng, W. Xiong, and J. Szefer. Secure TLBs. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 346–359. IEEE, 2019.

[11] R. Sheikh, H. W. Cain, and R. Damodaran. Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores. In *International Symposium on Microarchitecture*, pages 423–435, 2017.

[12] R. Sheikh and D. Hower. Efficient Load Value Prediction using Multiple Predictors and Filters. In *International Symposium on High Performance Computer Architecture*, pages 454–465, 2019.

[13] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. *White paper*, 2016.

[14] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[15] S. Deng, D. Gümüşoğlu, W. Xiong, Y. S. Gener, O. Demir, and J. Szefer. Secchisel framework for security verification of secure processor architectures. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, HASP, June 2019.

[16] J. Atalla and D. Kahng. Metal oxide semiconductor (mos) transisto r demonstrated. the silicon engine. *Computer History Museum*, 1960.

[17] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *International Conference on Architectural Support for Programming Languages and Operating System*, pages 138–147, 1996.

[18] A. Perais and A. Seznec. BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction. In *International Symposium on High Performance Computer Architecture*, pages 13–25, 2015.

[19] Intel 64 and ia-32 architectures software developer's manual: Volume 3. `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html`.

[20] D. Gullasch, E. Bangerter, and S. Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505. IEEE, 2011.

[21] C. Percival. Cache missing for fun and profit, 2005.

[22] D. J. Bernstein. Cache-timing attacks on aes. 2005.

[23] J. Bonneau and I. Mironov. Cache-collision timing attacks against aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[24] O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[25] J. Daemen and V. Rijmen. AES Proposal: Rijndael. 1999.

[26] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ Flush: a Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[27] Y. Yarom and K. Falkner. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[28] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[29] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-Cores Cache Covert Channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64, 2015.

[30] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

[31] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[32] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. In *International Symposium on Microarchitecture*, 2016.

[33] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *International Symposium on High Performance Computer Architecture*, 2014.

[34] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Conference on Computer and Communications Security*, 2016.

[35] S. Deng and J. Szefer. New predictor-based attacks in processors. In *Proceedings of the Design Automation Conference*, DAC, December 2021.

[36] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. Microscope: enabling microarchitectural replay attacks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 318–331. IEEE, 2019.

[37] R. Paccagnella, L. Luo, and C. W. Fletcher. Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical. In *USENIX Security Symposium*, 2021.

[38] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. *ACM SIGPLAN Notices*, 47(6):99–110, 2012.

[39] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 503–516. ACM, 2015.

[40] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83–93. IEEE, 2008.

[41] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[42] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[43] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 2–13. IEEE Computer Society, 2005.

[44] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[45] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 347–360. ACM, 2017.

[46] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Conference on Computer and Communications Security*, 2017.

[47] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security Symposium*, pages 937–954, 2018.

[48] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, pages 955–972, 2018.

[49] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[50] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[51] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. In *International Symposium on Computer Architecture*, 2021.

[52] J. Kim, H. Jang, H. Lee, S. Lee, and J. Kim. Uc-check: Characterizing micro-operation caches in x86 processors and implications in security and performance. In *International Symposium on Microarchitecture*, 2021.

[53] F. Yao, M. Doroslovacki, and G. Venkataramani. Are Coherence Protocol States Vulnerable to Information Leakage? In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 168–179. IEEE, 2018.

[54] D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, pages 897–912, 2015.

[55] C. Trippel, D. Lustig, and M. Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[56] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 38–55. IEEE, 2016.

[57] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[58] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, 2008.

[59] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. RIC: relaxed inclusion caches for mitigating LLC side-channel attacks. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[60] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors.

[61] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[62] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas, et al. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. *arXiv preprint arXiv:1812.09822*, 2018.

[63] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787. IEEE, 2018.

[64] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.

[65] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[66] Z. He and R. B. Lee. How secure is your cache against side-channel attacks? In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–353. ACM, 2017.

[67] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[68] G. Taylor, P. Davies, and M. Farmwald. The TLB slice-a low-cost high-speed address translation mechanism. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 355–363. IEEE, 1990.

[69] C. Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[70] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 208–225. Springer, 2012.

[71] A. Seznec. A case for two-way skewed-associative caches. *ACM SIGARCH computer architecture news*, 21(2):169–178, 1993.

[72] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.

[73] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. 2018.

[74] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-inclusive World. In *USENIX Security Symposium (USENIX)*, page 0, 2019.

[75] W. Xiong and J. Szefer. Leaking information through cache lru states. In *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2020.

[76] S. Deng, W. Xiong, and J. Szefer. Secure TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 346–259, New York, NY, USA, 2019. ACM.

[77] Speculative Store Bypass Bug CVE, 2018. CVE 2018-3639., 2018.

[78] T. Zhang and R. B. Lee. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 96–105. ACM, 2014.

[79] B. L. Welch. The Generalization of Student's Problem When Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28–35, 1947.

[80] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *Symposium on Security and Privacy (S&P)*, 2019.

[81] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross Processor Cache Attacks. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 353–364, 2016.

[82] J. Nomani and J. Szefer. Predicting program phases and defending against side-channel attacks using hardware performance counters. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, June 2015.

[83] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 549–564, 2016.

[84] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth. Autolock: Why cache attacks on {ARM} are harder than you think. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1075–1091, 2017.

[85] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[86] X. Zhang, Y. Xiao, and Y. Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 858–870. ACM, 2016.

[87] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[88] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. Smotherspectre: exploiting speculative execution through port contention. *arXiv preprint arXiv:1903.01843*, 2019.

[89] S. Bhattacharya and I. Verbauwhede. Exploring micro-architectural side-channel leakages through statistical testing. *yet to receive the details*, 2020.

[90] S. A. Crosby, D. S. Wallach, and R. H. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):1–29, 2009.

[91] N. V. Smirnov. On the estimation of the discrepancy between empirical curves of distribution for two independent samples. *Bull. Math. Univ. Moscou*, 2(2):3–14, 1939.

[92] Microsoft corp. Visual Studio App Center. `https://appcenter.ms/`, accessed online.

[93] Amazon.com Inc. AWS Device Farm. https://aws.amazon.com/device-farm/, accessed online.

[94] Google LLC. Google Firebase. https://firebase.google.com/, accessed online.

[95] Android Open Source Project. Security-Enhanced Linux in Android. https://source.android.com/security/selinux, accessed online.

[96] Android Issue Tracker. [Android Q Beta] Apps can no longer execute binaries in their home directory. https://issuetracker.google.com/issues/128554619, accessed online.

[97] Amazon.com. Elastic Compute Cloud (EC2). Cryptology ePrint Archive, Report 2019/167. http://aws.amazon.com/ec2.

[98] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[99] B. Burgess. Samsung exynos m1 processor. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–18. IEEE, 2016.

[100] J. Borghoff, A. Canteaut, T. Güneysu, E. Kavun, M. Knezevic, L. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, et al. A low-latency block cipher for pervasive computing applications-extended abstract. Asiacrypt, 2012.

[101] M. K. Qureshi. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 360–371. IEEE, 2019.

[102] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 675–692, 2019.

[103] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[104] P. Guide. Intel® 64 and ia-32 architectures software developer‘s manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[105] A. Virtualization. Amd-v nested paging. *White paper*, 2008.

[106] A. J. Goldsmith and P. P. Varaiya. Capacity of fading channels with channel side information. *IEEE Transactions on Information Theory*, 43(6):1986–1992, 1997.

[107] E. Rotem and S. P. Engineer. Intel architecture, code name skylake deep dive: A new architecture to manage power performance and energy efficiency. In *Intel Developer Forum*, 2015.

[108] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar. Next generation intel® micro-architecture (nehalem) clocking architecture. In *Symposium on VLSI Circuits*, pages 62–63. IEEE, 2008.

[109] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE Computer Society, 2012.

[110] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[111] A. Perais and A. Seznec. Practical Data Value Speculation for Future High-End Processors. In *International Symposium on High Performance Computer Architecture*, pages 428–439, 2014.

[112] W. S. Gosset. The Probable Error of a Mean. *Biometrika*, pages 1–25, 1908.

[113] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[114] J. L. Massey. Foundation and methods of channel encoding. In *International Conference on Information Theory and Systems*, volume 65, pages 148–157. NTG-Fachberichte, 1978.

[115] G. Saileshwar, C. W. Fletcher, and M. Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[116] C. Gough, I. Steiner, and W. Saunders. *Energy efficient servers: blueprints for data center optimization*. Springer Nature, 2015.

[117] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. Platypus: Software-based power side-channel attacks on x86. In *Symposium on Security and Privacy*, 2021.

[118] DoD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria, 1983. `http://csrc.nist.gov/publications/history/dod85.pdf`.

[119] W. Xiong and J. Szefer. Leaking information through cache lru states. In *International Symposium on High Performance Computer Architecture*, 2020.

[120] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, 2019.

[121] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, 2019.

[122] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Conference on Computer and Communications Security*, 2015.

[123] P.-E. Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[124] Introducing Geekbench 5, 2021. https://www.geekbench.com/.

[125] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[126] N. NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. *Volume 1.0: Whitepaper, Part*, 1:82, 2020.

[127] J. M. Gambetta, A. D. Córcoles, S. T. Merkel, B. R. Johnson, J. A. Smolin, J. M. Chow, C. A. Ryan, C. Rigetti, S. Poletto, T. A. Ohki, M. B. Ketchen, and M. Steffen. Characterization of addressability by simultaneous randomized benchmarking. *Physical Review Letters*, 109(24):240504, 2012.

[128] R. Blume-Kohout, E. Nielsen, K. Rudinger, K. Young, M. Sarovar, and T. Proctor. Idle tomography: Efficient gate characterization for n-qubit processors. In *APS March Meeting Abstracts*, 2019.

[129] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A language for hardware-level security policy enforcement. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 97–112. ACM, 2014.

[130] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, volume 44, pages 109–120. ACM, 2009.

[131] R. S. Chakraborty, S. Narasimhan, and S. Bhunia. Hardware trojan: Threats and emerging solutions. In *International High-Level Design Validation and Test Workshop*, pages 166–171. IEEE, 2009.

[132] P. S. Li, A. M. Izraelevitz, and J. Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[133] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[134] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[135] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[136] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[137] A. Mi, S. Deng, and J. Szefer. Device- and locality-specific fingerprinting of shared nisq quantum computers. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, HASP, October 2021.

[138] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also Technical Report Forshadow-NG [6]. `http://ForeshadowAttack.com`.