© 2024 Zirui Zhao

YOU SHARE, YOU LEAK: PRACTICAL SIDE-CHANNEL ATTACKS AND DEFENSES IN MODERN CLOUDS

BY

ZIRUI ZHAO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair and Director of Research
Associate Professor Christopher W. Fletcher
Professor Darko Marinov
Associate Professor Adam Morrison, Tel Aviv University
Professor Moinuddin K. Qureshi, Georgia Institute of Technology
Associate Professor Mohit Tiwari, The University of Texas at Austin

Abstract

Over the past few decades, the pursuit of higher computational density and resource sharing has resulted in substantially improved performance and efficiency of modern computer systems. However, this shift has also introduced serious security concerns, notably side-channel attacks. Public cloud computing, with its ever-growing market size and extensive hardware resource sharing among mutually-distrusting tenants, stands out as a prime target for these attacks. Recognizing these threats, this thesis delves deeply into both side-channel vulnerabilities and defenses in public cloud environments.

On the attack front, this thesis examines the intricacies of conducting end-to-end side-channel attacks in modern public clouds, including how to co-locate with the victim program and set up side channels to extract information in a noisy, dynamic production cloud environment. This thesis introduces methods to increase the likelihood of an attacker co-locating with a target victim, filling a critical gap for side-channel attacks in public clouds. Additionally, the thesis presents novel techniques for setting up and monitoring cache-based side channels in a noisy public cloud environment. The result of both works is the first demonstration of cross-tenant information leakage in the Google Cloud.

On the defense front, this thesis introduces *Untangle*, a framework to quantify information leakage in schemes that perform dynamic partitioning of hardware resources, which are promising side-channel defenses. Using Untangle, the thesis proposes design principles and defense mechanisms to tightly bound and reduce the leakage, resulting in low-leakage high-performance dynamic partitioning schemes. Besides defending against conventional side-channel attacks, this thesis also develops both hardware-only and hardware-software co-design mechanisms to substantially reduce the execution overhead of transient execution defenses.

Finally, this thesis also explores new side channels in modern Intel processors and develops defenses for microarchitectural replay attacks, an emerging type of attack.

To my parents and those who offered me an opportunity when no one else did.

Acknowledgments

As I sat down to write my thesis on a sunny early summer afternoon in central Illinois, I could not help but retrace my entire PhD and undergraduate journey. Looking back, the journey feels both rewarding and surreal. Ten years ago, I dreamed of studying and contributing to theoretical physics. Now, I am writing my PhD thesis in Computer Science on the other side of the earth from where I grew up. Along this long "student arc," so many people offered their invaluable help and guidance, for which I am sincerely grateful.

First and foremost, I want to express my deepest gratitude to my dear PhD advisor, Prof. Josep Torrellas, for his guidance and support throughout my PhD study. I particularly appreciate Josep's patience. I still remember my early years as a PhD student, passionate but inexperienced in research. Often, I would rush into premature ideas and conclusions, exceeding the "speed limit" of research. My meetings with Josep were not helped by my often vague explanations. However, Josep, being a patient advisor, always carefully listened, gently slowed down my thought process, and guided me to meticulously examine each aspect of my ideas. Over the years, Josep taught me how to become a mature researcher with patience, knowing when to slow down on the F1 racing track that is Computer Science research. I also appreciate Josep's patience when I hit roadblocks and made little progress, as well as the research freedom he provided. I truly could not have found a better advisor than Josep.

Next, I want to thank my long-term collaborators, Prof. Christopher W. Fletcher and Prof. Adam Morrison, for their inspiration and support over the years. Working with them has been a joy. As a student with little security background initially, both Chris and Adam taught me how to think like a security researcher. I still remember our inspiring discussions in the InvarSpec project about optimizations that leak information in unexpected ways. Moreover, Chris taught me how to be a great communicator. When it came to paper writing, Chris showed me how to effectively communicate the novelty of our ideas to readers and how these ideas fit into the bigger picture. Chris also helped me tremendously in structuring my presentations, which significantly improved my job talk.

I also appreciate Adam's broad knowledge and his ability to bring refreshing viewpoints to our projects. His well-articulated insights on many complex technical problems were critical to unblocking our projects. Additionally, Adam provided timely help when I was deep in the weeds, whether it was meeting a paper deadline or dry-running a presentation, and I truly appreciate his help. Finally, Chris's and Adam's work on Speculative Taint Tracking (STT) and Adam's work on speculative type confusion have been great sources of inspiration for this thesis and are my favorite works in the grand space of microarchitectural side channel research.

I would also like to thank my good friends and mentors, Prof. Darko Marinov and Prof. Tianyin Xu. My PhD journey would have been incomplete without them. Although Darko is not officially my advisor, he has always been there for me, offering many easy-to-follow and critical pieces of advice. For example, Darko emphasized the importance of non-technical skills like communication during my first semester. He also stressed the need for automation and reducing the design iteration time. His suggestion motivated me to invest a few months in setting up a gem5 SimPoint infrastructure, which had a huge return on investment in time during my later PhD years. Finally, I want to thank Darko for planting the seed of side-channel attacks in public clouds during an elevator conversation four years ago, which eventually became a major part of this thesis (Chapters 3–4).

Tianyin and I started our UIUC journey around the same time, he as an Assistant Professor and me as a graduate student. Throughout these years, Tianyin has always been like a big brother to me, offering much advice and help. For example, he shared many graduate school wisdoms with me in my early PhD years, invited me to insightful mock PC meetings to help me understand and experience the peer-review process, and once revised my entire presentation slide by slide. Additionally, we often went hiking on nearby trails to relax, during which we had many good conversations about research, life, turtles, and citrus trees. Last but not least, I also want to thank Tianyin's wife, Fei, for her help over the years.

Next, I want to express my special gratitude to Prof. Michael R. Lyu and Prof. Tao Xie for being my Bo Le (伯乐)[1]. Both of them gave me priceless opportunities to start a CS career. During my junior year, I decided to apply for PhD programs in CS instead of physics and was desperate for any CS research opportunity. After sending out hundreds of cold emails and getting few responses, Michael was the only professor who gave me a chance. That same year, I spent a productive summer at CUHK, working with Michael and his then-PhD student, Hui Xu, who is now an Associate Professor at Fudan University. I am forever grateful for this invaluable experience and their help, which opened the door to CS for me. Both Michael and Hui also offered tremendous support during my graduate school application later that year. I also want to thank Kai Xing, my undergraduate advisor at USTC, and Yongqiang Xiong, my mentor at Microsoft Research Asia, for their help during my undergraduate study.

Tao was my PhD advisor when I joined UIUC, and I am forever grateful to him for admitting me into the PhD program. Tao has many research wisdoms that I still follow today. For example, the motto of Tao's group is "Work hard, work smart, and work wise," which I found to be so true in retrospect. He taught me how to manage and reflect on projects. His blogs and slides from his advice portal also built my foundation as a researcher. It was also Tao who introduced me to a research opportunity in hardware security, which eventually became my thesis topic. Besides Tao,

[1] `https://en.wikipedia.org/wiki/Bo_Le`

I would also like to thank the other two anonymous UIUC faculty members who believed in me and signed the form admitting me into the PhD program.

I would also like to thank Prof. Moinuddin K. Qureshi and Prof. Mohit Tiwari for being the external members of my thesis committee and for their generous support during my job search. Moin is an expert in memory systems, and I greatly admire his prior work in dynamic cache partitioning and randomized cache, which has been a great source of inspiration for this thesis. Additionally, I want to thank Moin for his invaluable feedback during the writing of this thesis. I also look up to Mohit's work in information flow and hardware security. His insights on much of my work, especially the Untangle project, made me think deeper about the project. It was a great honor to have both Moin and Mohit on my thesis committee.

During my PhD program, I had several memorable internships at Intel Labs and Lyft, for which I am truly grateful. Specifically, I want to thank my manager Carlos V. Rozas and my mentor Fangfei Liu at Intel. Their work on Intel SGX and last-level cache Prime+Probe attacks are great sources of inspiration for this thesis. I also want to thank Scott D. Constable, Thomas Unterluggauer, Frank McKeen, Alaa R. Alameldeen, and Yuan Xiao at Intel Labs for our insightful discussions. Besides internships, I would like to thank Intel Labs for their funding support during my PhD study. Last but not least, my summer internship at Lyft was also very enjoyable, and I want to thank my manager Ryan Cox and my mentor Tony Allen at Lyft for that.

The i-acoma group is full of inspiring and fun people, and it is a fond memory of my PhD journey. I would like to thank all my lab mates: Mengjia Yan, Yasser Shalabi, Thomas Shull, Raghavendra Pothukuchi, Dimitrios Skarlatos, Azin Heidarshenas, Antonio Maria Franques Garcia, Zhangxiaowen (Andy) Gong, Apostolos Kokolis, Serif Yesil, Hyoungwook Nam, Jovan Stojkovic, Antonis Psistakis, Dingyuan Cao, Yao Yao, Gerasimos (Makis) Gerogiannis, Charles (Chase) Blocks, Nikoleta Iliakopoulou, Jakob Sidney Arend, Burak Ocalan, Abraham Farrell, Chloi Alverti, Houxiang Ji, and Namrata Mantri, for embarking on this memorable UIUC journey with me. Specifically, I would like to thank Mengjia for her help with my first project, InvarSpec, and for being a role model for me in hardware security research. I would also like to thank Houxiang and Dingyuan for working with me. I want to thank Jovan, Makis, and Chase for their deep technical insights. Finally, I want to thank Antonis for his enthusiasm in organizing group activities.

Next, I would like to especially thank Prof. Dimitrios Skalartos, a former i-acomer. Dimitrios is the most energetic and cheerful person I have ever worked with, and I want to thank him for our memorable collaboration experience. I still remember that great summer when we worked on our project, Jamais Vu, which is one of the most enjoyable research projects I have been part of. Even after his graduation, our collaboration continued. He kindly invited me to be part of the Perspective project, from which I gained a lot of experience with operating system kernels and student mentoring. In retrospect, we are similar enough to efficiently understand each other, yet different enough to bring refreshing perspectives to the project. Besides research collaboration, I also want to thank Dimitrios for his help and advice on the PhD program and job search. Last but not least, I also want to thank my two undergraduate mentees at CMU, Tae Hoon Kim and David Rudo, for their hard work on the Perspective project and for tolerating my often unreasonable expectations.

I also want to express my appreciation to all my friends outside of the i-acoma group for being part of my PhD journey. I would like to thank all my friends from Prof. Tao Xie's group, Prof. Darko Marinov's group, Prof. Christopher W. Fletcher's group, Prof. Tianyin Xu's group, Prof. Sasa Misailovic's group, and Prof. Tarek Abdelzaher's group. Specifically, I would like to especially thank Jinyang Li for being such a wonderful roommate for the past six years. I felt extremely lucky to have you as my roommate. I also want to thank his golden retriever, Max, for bringing joy to my daily life. I would like to express my gratitude to Linyi Li, who offered important advice on the Untangle project.

The Brett Daniel Software Engineering (SE) seminar at UIUC is the best seminar I have ever attended. It is a seminar that often runs over time, but people do not mind. I would like to specially thank its founder Prof. Darko Marinov, the organizers, and frequent attendees for making the seminar such a pleasure to attend and for building a strong, inclusive SE community at UIUC.

Being able to find a job at the end of the PhD journey is important. I would like to thank all the people who helped me in my academic job search process: Prof. Josep Torrellas, Prof. Darko Marinov, Prof. Christopher W. Fletcher, Prof. Adam Morrison, Prof. Moinuddin K. Qureshi, Prof. Mohit Tiwari, Prof. José F. Martínez, Prof. Dimitrios Skalartos, Prof. Tianyin Xu, Prof. Lingming Zhang, Prof. Sasa Misailovic, Prof. Saugata Ghose, Prof. Ling Ren, Prof. Mengjia Yan, Prof. Riccardo Paccagnella, Prof. Nikita Borisov, Prof. Varun Chandrasekaran, Prof. Owolabi Legunsen, Prof. Daniel Genkin, Xudong Sun, Bingzhe Liu, Jai Arora, Russel Arbore, and all my practice talk attendees. I would like to specially thank Prof. Darko Marinov for starting and organizing the Job Search Seminar, which tremendously helped my job search. I would like to thank all the presenters, panelists, and participants of the Job Search Seminar. I would also like to thank the Mavis Future Faculty Fellows Program, which generously allowed my participation even though I was not part of the program. Last but not least, I would like to thank all the people I met during the interviews and campus visits.

Last but not least, I would like to express my heartfelt thanks to my parents. Due to many reasons, we could not meet each other for the past five years. Even though we are separated by the great Pacific Ocean, your unconditional love and support have helped me endure many difficult times. No words can truly express how much I love you two.

TABLE OF CONTENTS

CHAPTER 1: Introduction

Over the past decades, the pursuit of higher computational density and resource sharing has resulted in substantially improved performance and efficiency of modern computer systems. However, this evolution has also introduced serious security concerns, notably *microarchitectural side-channel attacks* [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. These attacks exploit hardware resources shared between a victim and an attacker. Observing changes in the victim's utilization of these shared resources, the attacker can exfiltrate secret information from the victim, such as cryptographic keys [3, 15, 17].

In 2018, the disclosure of Spectre [18] and Meltdown [19] elevated the threat of side-channel attacks to the next level, leading to the emergence of *speculative side-channel attacks*. Following Spectre and Meltdown, numerous speculative execution attacks [20, 21, 22, 23, 24, 25, 26, 27, 28] were discovered. These attacks combine conventional microarchitectural side channels with out-of-order execution, a fundamental performance optimization in modern processors. Speculative side-channel attacks can bypass both software checks (e.g., bounds checks) and hardware isolation mechanisms (e.g., the privileged bit in a page-table entry) to access a wide range of secrets, including kernel data.

Public cloud computing, which is fundamentally based on sharing hardware resources among mutually distrusting tenants, stands out as a prime target for these attacks. Making matters worse, cloud vendors are shifting towards emerging cloud computing paradigms, such as Function-as-a-Service (FaaS) computing [29, 30, 31], aiming to encourage more resource sharing between tenants. Additionally, cloud vendors are developing optimizations to harvest under-utilized resources of other tenants [32, 33, 34, 35]. Although these emerging paradigms and optimizations promise greater efficiency, they also open up more opportunities for side-channel attacks.

1.1 THE PROBLEMS

1.1.1 Problem 1: Are Side-Channel Attacks Practical in Modern Public Clouds?

Despite the threats of microarchitectural side-channel attacks, cloud vendors are skeptical about the practicality of these attacks [36]. For example, AWS dismisses certain side channels—such as last-level cache (LLC) side channels [3, 4, 15]—as impractical due to noise in the production environment [36]. Indeed, most prior side-channel attacks have been developed and evaluated in local, quiescent, lab environments. Demonstrating these attacks in public clouds requires non-trivial efforts and new attack techniques to overcome the noise and other practical challenges.

Generally speaking, side-channel attacks in public clouds comprise two steps: (i) *co-location* and (ii) *extraction*. In the co-location step, the attacker needs to launch containers or virtual machines (VMs) such that they are scheduled to run on the same physical machine as the victim container or VM. This step is challenging due to the vast scale of data centers and cloud vendors' opaque host selection policies. Although there have been some previous studies [37, 38, 39] on the risk of co-location in public cloud from 2009 to 2015, the evolution of cloud computing has rendered techniques proposed in previous studies mostly ineffective. For example, Ristenpart et al. [37] and Xu et al. [38] used network information, such as host IP address or packet routing information, to detect co-location. However, such information is now hidden by cloud vendors. Varadarajan et al. [39] used a pairwise method to test if two VMs are co-located. However, pairwise methods scale poorly to the number of VMs or containers under testing. Scalability matters because the attacker may need to launch many VMs or containers due to the growth of the data center size in recent years.

Even if the co-location can be reliably achieved, performing the extraction step is not an easy task either. Taking the LLC side channel as an example, it requires three substeps to extract information from the victim [3] (detailed in Section 2.2.3). First, the attacker sets up LLC side channels through a process of constructing LLC eviction sets. Each eviction set can be used to monitor memory accesses to a specific LLC set. Second, the attacker identifies which eviction set corresponds to the LLC set that is accessed by the victim in a secret-dependent manner. Finally, the attacker monitors the LLC set of interest and extracts information from the victim. All of these three steps need to be completed in a noisy production environment. If the target victim runs in a dynamic environment like FaaS, where user workloads are frequently spawned and terminated, these steps also have to finish within a short time window of co-location between the attacker and victim. Additionally, the wide adoption of non-inclusive LLC and the fact that huge pages are inaccessible in containerized environments have made this attack process even harder. Thus, while İnci et al. [15, 40] conducted an LLC Prime+Probe attack on AWS EC2 in 2015, their techniques are incompatible with modern clouds, as their techniques relied on long-running attack steps, huge pages, and inclusive LLCs.

Given these practical challenges of performing side-channel attacks in public clouds, this thesis tries to answer the following questions: (1) Should side-channel attacks be a security concern of public cloud vendors? (2) What techniques can a determined attacker use to make side-channel attacks in public clouds more practical?

1.1.2 Problem 2: How to Mitigate Side Channels in Public Clouds with Low Overhead?

Under the premise that side-channel attacks are practical, many defenses have been proposed to mitigate these attacks. These defenses range from software solutions such as data-oblivious programming [41, 42, 43], to hardware solutions such as randomizing resource usage patterns [44, 45, 46, 47, 48, 49, 50, 51, 52] or partitioning resources [53, 54, 55, 56, 57, 58, 59, 60, 61].

Adopting these solutions for general-purpose applications is challenging. Randomization-based schemes usually offer high performance but not comprehensive security guarantees. Partition-based schemes that use a fixed partition size throughout application execution—i.e., static resource partitioning—provide comprehensive security guarantees, but they can lead to significant performance overhead and resource under-utilization. Dynamically changing partition sizes can improve performance, but such schemes can reintroduce information leakage and hence have weaker security guarantees. While data-oblivious programming practices are increasingly adopted by cryptographic libraries, applying these practices to software outside of cryptographic libraries is limited by programming difficulty and performance overhead.

Given these challenges, this thesis also seeks to answer the following questions: (1) How to analyze the security guarantees of high-performance but leaky solutions like dynamic partitioning? An understanding of their security guarantees allows the user to make informed security-performance trade-off decisions when using these solutions. (2) How can the performance overhead of defenses be reduced without compromising their security guarantees?

1.2 CONTRIBUTIONS OF THIS THESIS

To answer the questions in Section 1.1, this thesis makes contributions towards *practical side-channel attacks and defenses in public clouds*. Specifically, this thesis has the following contributions.

- **Co-location in public cloud FaaS (Chapter 3).** As discussed in Section 1.1.1, the first step of side-channel attacks in public clouds is to co-locate the attacker program with the victim program on the same physical host. This step is challenging due to the vast scale of data centers and cloud vendors' opaque host selection policies. To address these challenges, we devised a novel technique for *fingerprinting physical hosts* in container environments. Our key insight is that the attacker can bypass software countermeasures and learn sensitive host information by directly interacting with the underlying shared host hardware. We demonstrated that these fingerprints identify individual hosts with more than 99.9% accuracy. Using this, we discovered an exploitable host selection behavior in Google Cloud Run [30], a production serverless platform. We then demonstrated an attack with a 100% success rate of co-locating the attacker containers with at least one victim container. Moreover, the attacker is co-located with 61%–100% of all the victim containers across three major data centers, costing only about 25 USD.
- **Last-level cache Prime+Probe attack in the modern public cloud (Chapter 4).** Once co-

located with a victim, the attacker then sets up a side channel and monitors the victim's activities. This step is challenging, as the channels are inundated with background noise in a production environment. Overcoming this challenge, we developed a series of noise-resilient attack techniques. Using these techniques, attackers can quickly set up LLC side channels in just 2.4 minutes on average in the production Google Cloud Run environment, enabling the monitoring of all memory accesses in the system. The same process would take more than ten hours using prior techniques, which are susceptible to noise. This efficiency in channel setup is vital because attackers might have only a few minutes to retain co-location with their target, due to the dynamic nature of clouds. Finally, we showcased, for the first time, an end-to-end attack in Google Cloud Run that extracts the secret nonce from a vulnerable ECDSA implementation in a mere 17 seconds after setting up the channel.

Impact: Our works (Chapters 3–4) highlight the realistic threats posed by side channels in production cloud environments, refuting the belief among cloud vendors that such attacks are impractical. Following our report, Google filed a critical-level bug report to their product team and AWS revised their security whitepaper.

- **High-performance, low-leakage dynamic resource partitioning (Chapter 5).** A principled approach to defend against side-channel attacks at some performance cost is to statically partition shared hardware resources among mutually distrusting tenants. Dynamic partitioning, which dynamically adjusts partition sizes based on program demand, tries to minimize performance loss. However, dynamic partitioning can reintroduce side-channel leakage.

To address this challenge, we proposed *Untangle*, the *first* framework to tightly quantify and reduce the information leakage in dynamic partitioning. Untangle's main contribution is a formal model that cleanly splits the information leakage into two categories: (1) *action leakage* from *how* the victim resizes and (2) *scheduling leakage* from *when* the victim resizes. Using Untangle, we introduced design principles for partitioning techniques that help eliminate the action leakage altogether. We also developed a model to conservatively bound and reduce the scheduling leakage without the impractical analysis of program timing. The versatility of Untangle makes it suitable for partitioning various hardware resources, and its application to a conventional dynamic LLC partitioning scheme achieved a 78% reduction in information leakage without performance loss.

Impact: Untangle stands out as a pioneering framework for tightly measuring and reducing information leakage in dynamic partitioning. It offers a balanced trade-off: minimal and controlled information leakage while retaining high performance.

- **High-performance speculative side-channel defenses (Chapters 6–7).** The general approach to defend against speculative side-channel attacks is to delay the execution of vulnerable instructions until they are no longer speculative. However, this can create high execution overhead.

Our work (Chapter 6) highlights that the primary contributor to overhead is delaying execution until no memory consistency violations (MCVs) are possible. As a solution, we proposed a general hardware mechanism, *Pinned Loads* [62], that extends the cache coherence protocol so that it prevents MCVs from occurring as early as possible. In our experiments, Pinned Loads reduced the execution overhead of three popular defenses by around 50%.

Furthermore, we developed a hardware-software co-design framework named *InvarSpec* (Chapter 7). InvarSpec statically analyzes the data and control flows of a program, identifying "safe instructions" that cannot influence whether vulnerable instructions should execute and their operand values. This unique insight arises from a static program analysis pass that accounts for all potential program paths—unlike hardware, whose knowledge is limited to the current execution path. Using this information, InvarSpec hardware allows the execution of vulnerable instructions earlier, without waiting for the completion of these "safe instructions", thereby substantially reducing execution overhead.

Impact: Intel showed substantial interest in both approaches, leading to two internships there for technology transfer.

• **Contention-based side-channel attacks exploiting the page walker (Appendix A).** We introduced *Binoculars* [16], the first indirect and stateless side channel. It exploits resource contention during page-walk operations, which defenses typically overlook. Remarkably, Binoculars produces timing perturbations of up to 20k cycles from just a *single* dynamic instruction. These perturbations are at least two orders of magnitude greater than those in other known microarchitectural side channels. With Binoculars, we showcased an attack that extracts the secret nonce from a vulnerable ECDSA implementation in just *one* victim run, thanks to the channel's exceptional signal-to-noise ratio. We also demonstrated an attack that compromises Linux's kernel address-space layout randomization (KASLR) even with existing software defenses turned on. As this attack requires the attacker and the victim to run on the same physical core but different hyperthreads, it is not applicable to the modern public cloud [36] and may only concern client environments with hyperthreading enabled. As a result, this contribution is presented as an appendix of the thesis.

Impact: The discovery of Binoculars instigated in-depth discussions within Intel. In addition, the approach we used to determine its root cause inspired the work of other researchers.

• **Thwarting microarchitectural replay attacks (Appendix B).** Microarchitectural Replay Attacks (MRAs) is a new class of attacks that can de-noise arbitrary side channels. MRAs exploit transient instructions, that is, instructions that are executed but do not retire, to replay vulnerable instruction execution to amplify side-channel signals. Dimitrios Skarlatos and I developed the first defense named *Jamais Vu* that mitigates MRAs. From a high level, Jamais Vu detects when an instruction is squashed. Then, as the instruction is reinserted into the pipeline, Jamais Vu automatically places a fence before it to prevent the attacker from speculatively executing it again. This work presents several Jamais Vu designs that offer different trade-offs between security, execution overhead, and implementation complexity. One design, called Epoch-LoopRem, effectively mitigates MRAs, has an average execution time overhead of 13.8% in benign executions, and only needs counting Bloom filters. An even simpler design, called Clear-on-Retire, has an average execution time overhead of only 2.9%, although it is less secure. My contributions in this work are a security analysis of Jamais Vu, a static program analysis pass used by one of the Jamais Vu designs, and implementing and evaluating various Jamais Vu designs.

Impact: Jamais Vu is the first defense scheme against Microarchitectural Replay Attacks (MRAs).

CHAPTER 2: General Background

This chapter provides a general background to this thesis. Additional background information specific to an individual chapter is provided within that chapter.

2.1 FUNCTION-AS-A-SERVICE

2.1.1 Overview

Function-as-a-Service (FaaS) [29, 30, 31] is an emerging cloud computing paradigm that disaggregates a large monolithic application into many small standalone components called *functions*. Each function typically has a single independent functionality. To perform the task of the original monolithic application, these functions collaborate and communicate with each other over the network. Consequently, functions are usually implemented as web services that can be invoked through various means, such as HTTP requests, WebSockets, or remote procedure calls [63]. In this thesis, we use *function* and *service* interchangeably.

Function deployment and management. To simplify the deployment and management of functions, each function is packed with its dependencies into a lightweight, self-contained *container image*. The containerization process ensures a consistent execution of the function in various environments.

The FaaS platform orchestrator fully manages function instances at container-level granularity. When a function is invoked by a user or another function, the orchestrator launches a new container instance of the requested function to process the incoming request. After serving the request, the instance enters an idle state, releasing its CPU and awaiting further incoming requests. Idle instances are typically charged minimally or not at all. If an instance remains idle for an extended period of time (e.g., 15 minutes), it is terminated and destroyed [64]. As a result, user instances often have a short lifetime [64, 65, 66].

Autoscaling. A FaaS platform can dynamically adjust the number of instances of a function based on its demand, a feature known as *autoscaling* [67]. When there is a surge in requests for a function that exceeds its current capacity, the orchestrator *scales out*, deploys additional instances to accommodate the increase in demand. Conversely, when the demand for a function declines, the orchestrator *scales in* by terminating excess instances, thus freeing up resources for instances of other functions.

Instance placement. When selecting a host to place a new container instance, a typical FaaS orchestrator first identifies all the hosts that meet specific constraints. Then, among such hosts, it selects the one with the highest score, based on criteria such as resource utilization and load balancing [68]. The orchestrator can tweak the instance placement algorithm by incorporating additional policies, such as *affinity* and *anti-affinity* rules. Affinity rules aim to place instances from functions that frequently interact with each other on the same host to reduce communication overhead. In contrast, anti-affinity rules attempt to distribute instances of the same function across different hosts for fault tolerance.

2.1.2 The Cloud Run Platform

In this thesis, we focus mainly on the Cloud Run platform [30] from Google Cloud as our target. Cloud Run is a fully-managed serverless computing platform designed for containers, and it powers Google Cloud's FaaS platform. Users of Cloud Run can deploy services using either predefined container templates or custom-built container images. As user containers can run arbitrary programs on Cloud Run, the platform offers two types of sandboxed execution environments to ensure software security.

First generation environment (GEN 1) [69]. Cloud Run uses gVisor [70] to sandbox Linux containers in its first-generation environment *without* host hardware virtualization. Figure 2.1 shows an overview of gVisor. At a high level, gVisor runs as a userspace kernel that intercepts and emulates normal system calls. This design prevents the untrusted application from directly interacting with the host kernel, reducing the attack surface. Consequently, the user application cannot access sensitive host information. For example, gVisor conceals the host CPU model name and cache sizes by emulating `/proc/cpuinfo`. Additionally, gVisor also virtualizes the host's runtime states, such as its IP address and uptime.

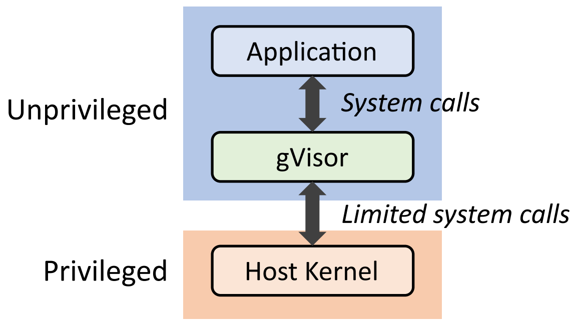


Figure 2.1: Overview of gVisor container sandbox [70].

Second generation environment (GEN 2) [69]. Cloud Run uses lightweight virtual machines (VMs) to sandbox user programs in its second-generation environment, which was introduced in December 2022 [71]. In GEN 2, the untrusted user program runs inside a guest VM on the virtualized host hardware. Using hardware virtualization, the hypervisor can trap and emulate certain x86 instructions like `cpuid`, thus creating an illusion of the hardware on which the user runs. As a result, the user has no access to sensitive host information.

Comparison between GEN 1 and GEN 2. Both execution environments have their pros and cons, which means that they are complementary rather than substitutional. Since GEN 1 uses Linux containers, it has a small resource footprint and features fast start-up time [69]. This feature is crucial for user-facing web applications that are latency-critical [72, 73, 74], such as web search [74], online collaborative document editing [75], and key-value stores [74, 76, 77]. Increases in latency can negatively impact advertisement revenue [78]. Yet, a limitation of GEN 1 lies in its potential compatibility issues stemming from system call emulation.

Conversely, GEN 2 provides full Linux compatibility, and in a steady state, it performs better than GEN 1. However, its large resource footprint results in longer start-up times [69]. At the time of this writing, Cloud Run uses GEN 1 for services by default [69]. Moreover, GEN 1 is used in other Google Cloud products like Cloud Function [79] (Google Cloud's equivalent of AWS Lambda [29]). Therefore, in this thesis, we primarily focus our exploration on GEN 1 and demonstrate the transferability of our results to GEN 2.

2.2 MICROARCHITECTURAL SIDE-CHANNEL ATTACKS AND DEFENSES

2.2.1 Overview

In a side-channel attack, the attacker exploits hardware resources that are shared with the victim. Specifically, when a program's execution *modulates* (i.e., changes the utilization of) hardware resources (i.e., *channels*) as a function of its secret data, an attacker can measure these modulations and from them infer the secret.

The ways in which hardware channels can be modulated to pass information can be broken down along two axes. To start, channel modulations are *stateful* if they leave a persistent state change (e.g., the eviction of a line from the cache) [1, 2, 3, 6, 7, 80, 81, 82, 83, 84, 85, 86], or alternatively *stateless* if they create only temporary contention on a resource (e.g., on an execution unit) [9, 10, 12, 13, 87, 88, 89, 90, 91]. Orthogonally, channels can be modulated *directly* by the execution of the victim instruction's micro-ops, or *indirectly* by operations that occur outside the purview of the instruction's micro-ops.[1] The large majority of channel modulations are direct (e.g., all of the above works). An example is a cache attack due to the execution of a victim memory instruction that evicts a line from the cache. On the other hand, there are a handful of

[1]Indirect modulations have been called "implicit" modulations by prior work [92].

channels involving indirect operations [8, 92, 93, 94, 95, 96, 97]. Examples are “implicit” memory operations due to hardware prefetchers or page walkers that occur beyond the purview of micro-ops.

Historically, stateless channels have been considered more difficult to exploit than stateful channels. Indeed, it is relatively easy to monitor a stateful channel because its effect persists after the victim instruction modulating it has retired. Further, the contention effects of stateless channels are typically small, which exacerbates measurement noise.

2.2.2 Cache-Based Side-Channel Attacks

Since caches are shared between processes in different security domains, they provide an opportunity for an attacker to exfiltrate sensitive information about a victim process by observing their cache utilization. This constitutes a *cache-based side-channel attack*. Such attacks can be classified into *reuse-based attacks* and *contention-based attacks* [50].

Reused-based attacks rely on shared memory between attacker and victim, often the consequence of memory deduplication [98]. In such attacks, the attacker monitors whether shared data are brought to the cache due to victim accesses. Notable examples of such attacks include Flush+Reload [6], Flush+Flush [7], and Evict+Reload [85]. However, as memory deduplication across security domains is disabled in the cloud [36, 99], these attacks are inapplicable.

Contention-based attacks, such as Prime+Probe [1, 2], do not require shared memory between attacker and victim. The core attack primitive of Prime+Probe is called an *eviction set*. An eviction set for a cache set s is a set of cache lines that, once accessed, can evict any cache line mapped to s by fully occupying s [3, 4]. Section 4.2.1 provides detailed background information on constructing eviction sets. Using an eviction set for s, the attacker can monitor the victim’s memory accesses to cache set s with Prime+Probe. During the attack, the attacker first *primes* s by filling all its ways with cache lines from an eviction set for s. Subsequently, the attacker continuously *probes* these lines, measuring the latency of accessing them. If the victim accesses s, it evicts one of the attacker’s cache lines, which the attacker can detect through increased probe latency. The attacker then re-primes s and repeats the probing process to continue monitoring.

Cloud vendors generally prevent processes of different tenants from sharing the same physical core at the same time [36, 100]. Therefore, the attacker has to perform a *cross-core* attack targeting the shared LLC. On modern processors, the LLC is split into multiple slices. Each physical address is hashed to one of the slices.

Table 2.1: Steps of an LLC Prime+Probe attack in clouds.

Step	Description	Discussed in
Step 1. Co-location	Co-locate the attacker program on the same physical host as the target victim program	Chapter 3
Step 2. Prepare LLC side channels	Construct numerous eviction sets, each corresponding to a potential target LLC set	Sections 4.4–4.5
Step 3. Identify target LLC sets	Scan LLC sets to identify those that the victim accesses in a secret-dependent manner	Sections 4.6–4.7
Step 4. Exfiltrate information	Monitor the target LLC sets and extract information	Sections 4.6–4.7

2.2.3 Last-Level Cache Prime+Probe Attack in Public Cloud

Mounting LLC Prime+Probe attacks in the modern public cloud requires several steps [3, 4, 15, 37, 101], as listed in Table 2.1. First, the attacker *co-locates* their program with the target victim program on the same physical machine (Step 1) [37, 38, 39, 101]. Second, the attacker prepares LLC channels by constructing LLC eviction sets (Step 2) [3, 4]. In practice, the attacker generally does not know the *target LLC sets*, which are LLC sets accessed by the victim in a secret-dependent manner. Hence, in Step 2, the attacker needs to build *hundreds to tens of thousands* of eviction sets, each corresponding to a potential target LLC set [3, 4, 102]. Then, the attacker scans through the potential target LLC sets and identifies the actual target LLC sets (Step 3). Finally, the attacker monitors the target LLC sets with Prime+Probe and exfiltrates the secret (Step 4).

2.2.4 Microarchitectural Side-Channel Defense Through Resource Partitioning

A popular way to defend against side-channel attacks is to partition the shared resource among different security domains. The partition can be *spatial* or *temporal*. Spatial partitioning divides the resource into non-overlapping sections used by different domains (e.g., way-partitioning in caches [61]). A temporal partitioning scheme splits the time into non-overlapping slices, and only one domain is allowed to use the resource in each time slice (e.g., interconnect traffic shaping [103]). When it is not ambiguous, we use the term *partition size* as the portion of the total resource assigned to one domain, regardless of spatial or temporal partitioning.

Depending on the partitioning policy, a scheme can either fix the partition size or dynamically resize it to adapt to a program's demand. The former schemes are *static*, while the latter are *dynamic* (e.g., [104, 105]).

2.2.5 Microarchitectural Side-Channel Leakage Detection

Information leakage through side channels can be detected using a variety of techniques. One approach is to leverage taint analysis [106, 107]. In this case, secret data are annotated as taint sources. Then, taint propagation is used to detect instructions that have secret-dependent usage of the resource of interest, or instructions that are control-dependent on secrets. Other approaches include symbolic execution [108, 109, 110] or abstract interpretation [111, 112, 113]. These specific works formally model the behavior of a cache and find instructions that put the cache into a secret-dependent state.

2.3 SPECULATIVE SIDE-CHANNEL ATTACKS AND DEFENSES

2.3.1 Out-of-Order Execution

Dynamically-scheduled processors execute data-independent instructions in parallel [114], out of program order. Instructions are dispatched to reservation stations (RS) in program order, where they await execution. An instruction becomes ready to execute once its input operands have been computed. In each cycle, a hardware scheduler picks a subset of ready instructions and issues them to execution units. After they execute, their outputs become available to dependent instructions. Instruction retirement, where the instruction finally frees up its pipeline resources, is done in program order. In-order retirement is implemented by queuing instructions into a FIFO queue called a reorder buffer (ROB) [115] in program order, and retiring an instruction once it reaches the ROB head.

2.3.2 Attack Overview

Attack structure. In out-of-order processors, some instructions may execute but later get squashed and not commit. These bound-to-squash instructions are called *transient instructions*. As a result, speculative side-channel attacks is sometimes called *transient execution attacks*. This thesis uses these two terms interchangeably. In a transient execution attack, an attacker exploits the side-effects of transient instructions to learn information it would not be able to learn from a non-transient correct execution. A typical attack consists of a transient load accessing some secret value, which is then forwarded to *transmitter* instruction(s) (or *transmitters*) that leak the secret over a covert channel [54, 116]. These steps are collectively referred to as a disclosure gadget [117].

In general, a transmitter is any instruction whose execution creates operand-dependent microarchitectural resource usage that reveals the operand (even if only partially) [54, 116, 117]. The prototypical example is a load instruction, which causes address-dependent changes to the state of the cache hierarchy by filling and evicting cache lines. As a result, the cache line accessed by the load can be inferred using techniques such as Flush+Reload [6] or Prime+Probe [1].

Figure 2.2 shows Spectre v1 [18], an example of a transient execution disclosure gadget. It exploits the misprediction of a bounds-checking branch to perform an out-of-bounds array load (Line 2), which can read a secret from any memory location. A transmit load then leaks the secret (Line 3).

```
1  if (x < array1_size) { // mispredicted branch
2    uint8 s = array1[x]; // access load
3    uint8 y = array2[s * 4096]; // transmit load
4  }
```

Figure 2.2: Spectre V1.

Security violations. Attacks are categorized by the relationship between the hardware protection domains of the disclosure gadget and the victim [117]. In a *domain-bypass* attack, the gadget and victim are in different domains. An example is Meltdown [19], where a userspace process reads OS kernel memory. In a *cross-domain* attack, the gadget resides in the victim's domain (which differs from the attacker's domain, which is from where the attacker monitors the covert channel). An example is a network server whose code inadvertently contains a Spectre gadget that can be passed a malicious input [18]. Finally, in an *in-domain* attack, the attacker circumvents software sandboxing. For example, an array access in JavaScript (compiled by a browser) is subject to a bounds check, producing code such as in Figure 2.2. Mispredicting the bounds check allows the attacker to circumvent the bounds check.

2.3.3 Hardware Defenses to Speculative Side-Channel Attacks

To defend against speculative execution attacks, researchers have proposed hardware-based schemes [116, 118, 119, 120, 121, 122, 123]. These schemes share a common general approach. First, they deploy a hardware mechanism that protects the relevant transmitter instructions. This protection prevents a transmitter from leaking its operands, blocking the side channel. However, it imposes a performance cost. Later, the protection is lifted when the transmitter's operands become safe to reveal. This execution point is called the instruction's *Visibility Point* (VP) [118]. When an instruction reaches the VP depends on the scheme's threat model, i.e., which types of transient instructions it considers.

Threat models. A popular but weak threat model is the *Spectre* model. It only considers transient instructions caused by incorrect control flow. An instruction reaches its VP when all of its older control-flow instructions have resolved. Another model is the Futuristic model [118], which we rename to the more descriptive name *Comprehensive* model. This model considers transient instructions caused by all types of squashes. An instruction reaches its VP only when it cannot be squashed anymore, which is most often when it reaches the ROB head. In this thesis, we use the Comprehensive model.

Protection mechanisms. Most defense schemes target cache and TLB-based side channels. They typically apply a variety of protection mechanisms to loads. For example, InvisiSpec [118] and SafeSpec [119] issue speculative loads invisibly. CleanupSpec [124] records the state generated by speculative loads, to be able to undo it on a squash. Delay-On-Miss (DOM) [120, 121] delays speculative loads that miss in the L1 cache, but allows L1-hitting speculative loads to execute. CSF [125] prevents speculative loads from changing visible cache state by inserting stalling fences. All of these mechanisms introduce performance overhead.

CHAPTER 3: Co-Location in the Modern Public Cloud

3.1 INTRODUCTION

As discussed in Section 1.1.1 and Section 2.2.3, the first step of side-channel attacks in public cloud is to ensure that the attacker's processes are *co-located* with the target victim's process on the same physical host [37] (Step 1 in Table 2.1). This chapter focuses on how an unprivileged malicious cloud user can co-locate their processes with a target victim process in a modern public cloud environment.

Attaining co-location in modern public clouds is challenging for several reasons. First, due to the widespread adoption of the virtual private cloud (VPC) [126], modern cloud infrastructures have become resistant to prior network-based co-location attack techniques [37, 38, 39]. Second, with the rapid expansion of cloud computing and the ever-growing sizes of data centers, the likelihood of attacker-victim co-location has been reduced.

Adding to these challenges, cloud computing is gradually shifting towards the emerging paradigm of *Function-as-a-Service (FaaS)*, exemplified by platforms like AWS Lambda [29], Google Cloud Run [30], and Azure Functions [31]. A detailed introduction to FaaS can be found in Chapter 2.1. Co-location attack techniques in these FaaS environments are relatively unexplored and present new challenges for attackers. Unlike conventional virtual machine (VM) environments, where users can specify the placement of VMs in availability zones and choose hosts with certain CPU models, FaaS platforms abstract away these operational details and fully manage the FaaS container placement. Furthermore, due to the dynamic nature of FaaS environments, container instances are frequently launched and soon terminated to accommodate dynamic workload demands. Such characteristics make achieving co-location in FaaS settings particularly difficult.

In this chapter, we present the first comprehensive study on risks of and techniques for co-location attacks in modern public FaaS environments. Since public FaaS platforms do not disclose their instance placement policies, reverse engineering these policies is crucial to understand the co-location risk and develop efficient co-location attacks.

To study how container instances are scheduled to physical hosts, we first develop two novel *host fingerprinting* techniques. We show that, despite the use of sandboxing and virtualization technologies [70, 127, 128, 129] in the cloud, the attacker can still learn sensitive host information by directly interacting with the host hardware—specifically, the timestamp counter. Our techniques are applicable to both non-virtualized Linux containers (e.g., Docker [127]) and lightweight VMs (e.g., Firecracker [129]), which are the two mainstream containerization technologies used in FaaS platforms. Armed with host fingerprints, we propose a new method to inexpensively *verify instance* *co-location* on a large scale. This is essential in vast modern data centers, where the attacker needs to launch numerous instances to achieve co-location.

Using the host fingerprints and our scalable co-location verification methodology, we perform a large-scale study on Google Cloud Run [30] to analyze its instance placement strategy. Our investigation uncovers exploitable instance placement behaviors in Cloud Run. In particular, Cloud Run appears to employ a load-balancing mechanism that distributes instances of a function to numerous hosts when the function experiences a high demand within a short time window. We then develop an instance launching strategy that exploits this behavior to deploy attacker instances onto a significant portion of Cloud Run hosts within a data center—drastically increasing co-location efficacy and reducing the financial cost of the attack.

We demonstrate the ability of our attack strategy to achieve 100% probability of co-locating the attacker with at least one victim instance in three major Cloud Run data centers in the US: *us-east1*, *us-central1*, and *us-west1*. Moreover, our strategy effectively co-locates the attacker with 100% of victim instances in *us-west1*, nearly 100% in *us-east1*, and between 61% and 90% in *us-central1*, depending on the victim account. In addition, we observe at least 1702 hosts in the largest data center, and show that our strategy successfully deploys attacker instances that reside on 904 hosts *at once*, with an estimated expense of only 23 USD—showcasing the practicality of co-location attacks in large modern data centers.

This chapter makes the following contributions:

• We introduce two effective host fingerprinting techniques as a primitive to study the instance placement policies of modern public FaaS platforms.

• We propose a scalable and inexpensive methodology for instance co-location verification assisted by host fingerprints.

• We systematically study the instance placement policies of Google Cloud Run and identify behaviors exploitable for co-location attacks.

• We devise an efficient attack strategy that achieves high co-location rates with different victim accounts on Google Cloud Run.

Disclosure to Google. We reported our findings to Google in early August 2023. Google identified our findings as an abuse risk and assigned the issue to their Trust & Safety team.

Availability. We open sourced our implementations at `https://github.com/zzrcxb/EAAO`.

3.2 BACKGROUND: TIMEKEEPING IN X86

In recent years, the timestamp counter (TSC) has become a preferable timekeeping option on x86 platforms, as CPU vendors increasingly support *invariant TSC*. An invariant TSC is reset to zero at boot time and increments at a *fixed* rate, irrespective of the CPU's frequency scaling and power state [130]. Compared to other clock sources, TSC offers greater time resolution and lower time retrieval cost. TSC can be conveniently accessed using the unprivileged instructions `rdtsc` and `rdtscp`.

To use the TSC as a clock source in Linux, the kernel needs to determine its frequency. Since the actual TSC frequency usually deviates from the frequency reported by `cpuid` by a constant value (which can be up to a few MHz), the kernel refines the TSC frequency using other hardware clocks in the system and utilizes the refined frequency for more accurate timekeeping [131]. On multi-core systems, Linux also verifies TSC synchronization across cores to prevent time anomalies. Generally, TSC is synchronized among cores across sockets on Intel platforms.

3.3 THREAT MODEL

Recall that a generalized microarchitectural side-channel attack consists of two steps: co-location and extraction (Section 2.2). In this chapter, we consider an attacker aiming to co-locate with instances of a target victim service on a public FaaS platform (STEP 1). We assume that the victim service processes sensitive information, such as a login service that performs authentication. Since the victim service is usually part of a large web application with public interfaces, we assume that the attacker can either directly or indirectly invoke the victim service through those interfaces. Finally, we assume that, once co-located with the victim, the attacker can detect when the victim program is running and exfiltrate the said sensitive information through techniques discussed in prior work [15, 17, 37, 39, 132].

We assume an unprivileged attacker who is a standard user of a public FaaS platform (e.g., Cloud Run). We also assume that the FaaS platform is trusted and does not collude with the attacker. These two assumptions imply that the attacker can only interact with the platform through standard FaaS interfaces that are available to all platform users, such as deploying custom services and sending requests to services. Using these interfaces, the attacker can execute arbitrary programs on the platform *inside their containers*, and the attacker can launch new container instances through autoscaling (Section 2.1). Additionally, we assume that the attacker has no knowledge of the *exact* host selection policies employed by the platform orchestrator and can only observe their behavior using black-box methods.

3.4 HOST FINGERPRINTING IN THE WILD

Public FaaS platforms, such as Cloud Run [30], do not reveal their instance placement policies. In this section, we propose novel, highly accurate physical host fingerprinting techniques suitable for both non-virtualized Linux containers (e.g., the Gen 1 environment) and lightweight VMs (e.g., the Gen 2 environment). Using our fingerprints, attackers can gain insights into the placement policy of the cloud platform, allowing them to develop launching strategies that drastically boost the efficacy of co-location attacks.

As the focus of this chapter is on the Gen 1 environment, we organize this section as follows: Section 3.4.1 provides an overview of host fingerprinting in Gen 1; Section 3.4.2 discusses two possible implementations to obtain host fingerprints in Gen 1; Section 3.4.3 proposes a new methodology to verify instance co-location in a scalable manner; Section 3.4.4 evaluates our fingerprinting for Gen 1 in the wild; and Section 3.4.5 extends the Gen 1 fingerprinting technique to the Gen 2 environment and evaluates its accuracy.

3.4.1 Overview

Recall that, in the Gen 1 environment, gVisor sandboxes user programs and hides the host information (Section 2.1.2), thereby blocking host fingerprinting through IP addresses or statistics in the `/proc` filesystem [37, 39]. However, we find that we can bypass gVisor's software countermeasures to learn sensitive host information by directly interacting with the non-virtualized host hardware.

For example, the attacker can use the unprivileged instruction `cpuid` to extract information like the CPU model and cache hierarchy structure, which are essential for many cache-based side-channel attacks [1, 2, 3, 133][1]. Similarly, the attacker can use the unprivileged instructions `rdtsc` and `rdtscp` to read the host's timestamp counter (TSC). The TSC is reset to 0 on host boot and increments at a fixed rate non-stop (Section 3.2). Therefore, the attacker can use the value of TSC to infer the uptime of the host, which in turn can be used to determine its boot time.

Based on this insight, we propose to use the host's CPU model (*model*) and the host's boot time in real-world time (T_{boot}) to fingerprint a host in the Gen 1 execution environment. The intuition of using T_{boot} is that different hosts very likely have different boot times due to system maintenance, hardware failures, and power management (e.g., powering off the host when the computation demand is low). As a result, T_{boot} can accurately differentiate physical hosts. Since it is trivial to read the CPU model through `cpuid`, we focus on deriving the host's T_{boot}.

[1]Intel introduced the Processor Serial Number (PSN) in the Pentium III processor [134]. The PSN uniquely identifies an individual processor and can be queried through `cpuid`. However, the PSN is discontinued in recent Intel processors due to privacy concerns.

3.4.2 Deriving the Boot Time from the TSC Value

To derive the host's boot time T_{boot}, the attacker can read the host's TSC value (denoted by tsc) through `rdtsc` or `rdtscp`, and simultaneously record the real-world time of this measurement (denoted by T_w) through a system call. Then, the host's boot time is calculated as follows:

$$T_{boot} = T_w - tsc/f, \tag{3.4.1}$$

where f is the TSC frequency measured in Hz. Eq. 3.4.1 assumes that the host CPU supports an invariant TSC (Section 3.2), which holds for all CPU models we observed in Cloud Run. However, even on the *same* host, the derived T_{boot} can exhibit small variations across measurements due to noise. Consequently, we round T_{boot} to a certain precision p_{boot} (e.g., 1 s). With the rounded value, measurements from the same host consistently produce the same fingerprint.

To obtain f, we propose two methods. Neither method relies on any features from gVisor or Cloud Run, making them applicable to other Linux container-based environments.

① **Using the reported TSC frequency.** In this method, the attacker uses the TSC frequency reported by `cpuid`. If `cpuid` does not report the TSC frequency, which is the case on Cloud Run, the attacker can use the labeled base frequency found in the model name. Empirically, this base frequency is equal to the TSC frequency that the clock is supposed to operate on [130]. For example, CPU model "Intel Xeon CPU @ 2.00GHz" has a base frequency and TSC frequency of 2.00 GHz. We refer to a TSC frequency obtained through either way as the *reported TSC frequency*.

Unfortunately, the reported TSC frequency is often slightly inaccurate, deviating from the actual TSC frequency by a constant value [131] (Section 3.2). This inaccuracy can cause the derived T_{boot} to drift over time, causing fingerprinting false negatives. To understand why, let us denote the reported TSC frequency as $f_r = f^* + \epsilon$, where f^* is the actual frequency and ϵ is the constant error. Suppose we collect two fingerprints from the *same* host at two different real-world times T_{w_1} and T_{w_2}, with TSC values tsc_1 and tsc_2, respectively, as illustrated in Figure 3.1.

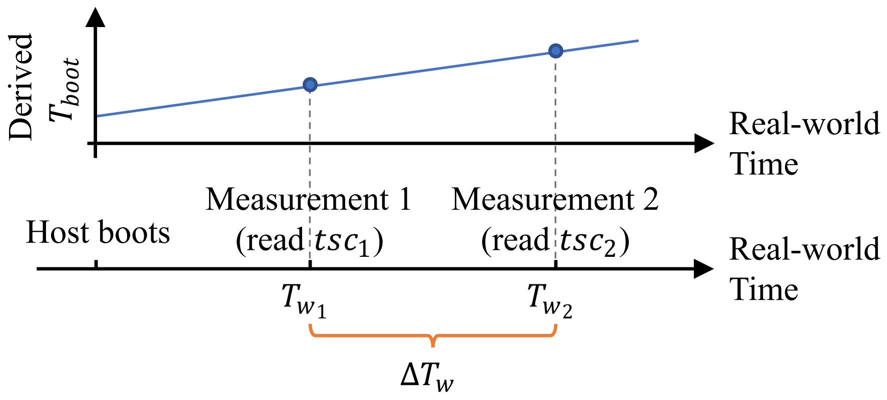


Figure 3.1: Illustration of drifting in the derived T_{boot} over time.

Using Eq. 3.4.1, the T_{boot} derived from the two measurements differs by

$$\begin{aligned}\Delta T_{boot} = T_{boot_2} - T_{boot_1} &= (T_{w_2} - tsc_2/f_r) - (T_{w_1} - tsc_1/f_r) \\ &= \Delta T_w - \Delta tsc/f_r \\ &= \Delta T_w - \Delta T_w f^*/f_r \\ &= \Delta T_w \epsilon / f_r. \end{aligned} \tag{3.4.2}$$

Since both ϵ and f_r are constant, $|\Delta T_{boot}|$ increases linearly as ΔT_w grows, where ΔT_w is the time elapsed between two measurements. If ΔT_w is sufficiently large, $|\Delta T_{boot}|$ will exceed the rounding precision p_{boot}, causing the rounded T_{boot} to differ and leading to a false negative. As a result, we say that the fingerprint exhibits an "*expiration time*" that depends on the frequency error ϵ.

② **Using measured TSC frequency.** An alternative approach is to measure the actual TSC frequency. This approach mitigates the drifting problem. Similar to how Linux refines the TSC frequency at boot time [131], the attacker can read the TSC twice, waiting a real-world time ΔT_w in-between. The TSC frequency can then be calculated as $\Delta tsc/\Delta T_w$. However, unlike the Linux kernel, the attacker cannot access other *hardware* clocks to obtain an accurate ΔT_w in the sandboxed container, as accessing those clocks requires privileged instructions. Consequently, the attacker can only rely on system calls to obtain ΔT_w, which may be subject to noise caused by interrupts and context switches.

We tested this approach on Cloud Run with $\Delta T_w \approx 100$ ms and found that the measured TSC frequency exhibits standard deviations of less than 100 Hz after 10 repetitions on most Cloud Run hosts. However, on a small yet significant portion of the evaluated hosts, we observed large standard deviations ranging from 10 kHz to a few MHz, even after 100 repetitions with an increased ΔT_w. As a result, two co-located instances on such problematic hosts can measure the TSC frequency as two significantly different values—and the T_{boot} derived by the two instances will not match, leading to declaring that the two instances are on two different hosts (i.e., a false negative).

Notably, during the experiment of validating fingerprinting accuracy in Section 3.4.4, we found that 58 out of the 586 evaluated hosts (or about 10%) exhibited such problematic behavior. These affected hosts were largely the same across measurements conducted at different times. Therefore, in the rest of the chapter, we obtain f using the first method (i.e., the reported TSC frequency) and, in Section 3.4.4, evaluate the expiration time of fingerprints due to the inaccurate f.

3.4.3 Verifying Instance Co-location in a Scalable Manner

In Section 3.4.4, we will evaluate the accuracy of our fingerprints by launching multiple instances and measuring whether instances that obtain the same fingerprints are indeed co-located, and vice versa. To do that, in this section, we first develop a new methodology to generate the ground truth of instance co-location in a scalable and inexpensive manner.

To understand our methodology, consider first the conventional approach to test instance co-location using a covert channel. The process involves picking two container instances at a time and instructing both of them to simultaneously put high pressure on a shared resource in the host, such as the random number generator (RNG) [90] or memory bus [89]. If both instances observe resource contention above a certain threshold, then we conclude that they are co-located. The drawback of this naive pairwise approach is that it has a time complexity of $O(N^2)$, where N is the total number of instances under test.

We propose a new approach to test n instances at once, where $n > 2$. Specifically, consider a covert-channel test primitive for n instances

$$CTest(i_1, i_2, ..., i_n) \to \{b_1, b_2, ..., b_n\},$$

that takes as input a list of n container instances under test ($\{i_1, i_2, ..., i_n\}$) and instructs all n instances to simultaneously put pressure on the shared resource. This primitive returns a list of boolean values ($\{b_1, b_2, ..., b_n\}$) that indicate whether each corresponding instance observes a contention level above a certain threshold. We assume that it only takes two co-located instances to generate enough contention to go over the contention threshold. In this case, if an instance A does not see contention over the threshold (i.e., it tests negative), then we can conclude that A is not co-located with any of the other $n-1$ instances. Furthermore, if all n instances test negative, then, in a single test, we conclude that no instance is co-located with any other instance.

If an instance A tests positive, we *may* know which instances are co-located with A, depending on the total number of positive instances in the test. To see why, consider as an example a test with four instances $\{A, i_1, i_2, i_3\}$. If three instances, including A, are positive, e.g., $CTest(A, i_1, i_2, i_3) \to \{T, T, T, F\}$, then we can conclude that $\{A, i_1, i_2\}$ must be co-located, as it takes at least two co-located instances to test positive. However, if all four instances are positive, we *cannot* conclude that they are co-located on the same host; it is possible that these four instances reside on two hosts, e.g., $\{A, i_1\}$ are co-located and $\{i_2, i_3\}$ are co-located. As a result, we can only test $n \leq 3$ instances at once without the confusion of whether they share one or multiple hosts.

We can further improve the test efficiency by (i) either raising the contention level threshold for an instance to test positive, or (ii) reducing the amount of pressure each instance generates. For example, if each instance generates a contention of 1 unit and we set the threshold to m units ($m > 2$), then it takes at least m co-located instances for each one of the m instances to test positive. As a result, if m, $m+1$, ..., or $2m-1$ instances test positive, we verify that these positive instances share the same host in a single test.

Our approach. Based on the above discussion, our approach hierarchically generates the ground truth for fingerprinting validation. The approach is illustrated in Figure 3.2, where each symbol represents a container instance, and truly co-located instances have the same shape. Assume that we have nine instances and that, using our fingerprinting method, conclude that there are three different fingerprints ($\mathbb{F}_1$, $\mathbb{F}_2$, and $\mathbb{F}_3$), and three instances per fingerprint. To validate our findings, we first group the instances based on their fingerprints (dashed lines in ①). If our fingerprints have a high accuracy, instances in the same group are likely to be indeed co-located, while those in different groups are likely not.

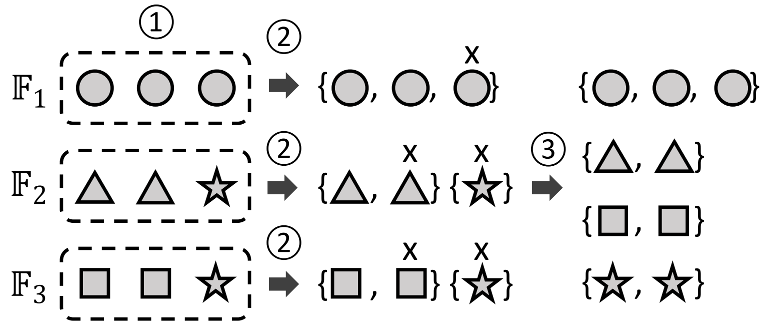


Figure 3.2: Overview of our fingerprint validation methodology. Each symbol represents a container instance. Instances with the same shape are truly co-located.

Next, we use an appropriate m to test likely co-located instances from each group at once. In the example, we use $m = 2$ (②). After this test, if a group had any false positive (e.g., the group with $\mathbb{F}_2$ and $\mathbb{F}_3$ in Figure 3.2), it is divided into several clusters, where each cluster includes instances that are verified to be co-located. Otherwise, the entire group remains intact and is considered as a single cluster (e.g., the group with $\mathbb{F}_1$ in Figure 3.2). Step ② has identified all the false positives. In the best-case scenario where no false positives existed, each fingerprint is verified with one single test. In this case, the total number of tests is the number of fingerprints under validation, which is the number of hosts—except for potential false negatives, as we will consider next.

Note that Step ② tests each fingerprint group in sequence, to avoid interference. We can further reduce the execution time of Step ② by concurrently verifying fingerprints that are *guaranteed* to belong to different hosts, such as those with different CPU models. As will be discussed in Section 3.4.5, the GEN 2 fingerprint makes crucial use of this optimization.

After this, we want to find the false negatives—i.e., two instances with different fingerprints that are actually co-located. Since instances from the same cluster are verified to be co-located, we pick one instance from each cluster to represent the host they reside on. In Figure 3.2, we pick the five instances that are decorated with an ×. These selected instances are unlikely co-located. Hence, we set $m = 2$ and test these selected instances all at once (③). Those that test positive are false negatives. In our example, they are the two stars. Then, we further refine our tests on the positive instances to identify co-located instances and merge the clusters they represent. In the example, we end up with four clusters, as the figure shows. In the best-case scenario where no false negatives existed, Step ③ only requires one test.

If an initial group in Step ① is large, we split it into several smaller groups with no more than $2m - 1$ instances each, where m is small (in our implementation, we use $m = 2$). Then, we test each group individually. If each small group is verified to be co-located, we pick one instance from each group to hierarchically test their co-location. If some tests in this process turn out negative, for simplicity, we fall back to pairwise tests within the initial large group.

In summary, our approach's best-case time complexity is $O(M)$, where M is the number of hosts occupied by the instances under validation. This best-case scenario is common if fingerprints are accurate, which we will see is true for our Gen 1 fingerprints in Section 3.4.4.

Comparison with conventional pairwise covert-channel testing [15, 37, 39]. The main goal of our host fingerprinting technique is to improve the attacker's ability to achieve co-location. Remarkably, while conventional pairwise covert-channel testing only confirms co-location at a particular moment, our host fingerprinting allows an attacker to track a host over time. Our technique thus enables the attacker to comprehensively study how container instances are placed onto specific hosts at different times, which can be leveraged to develop efficient instance launching strategies to significantly increase the probability of co-location with a target victim (Section 3.5). We will see that, if an attacker simply launches instances without insight into the placement policy, the rate of co-location is often zero, whereas our technique attains a co-location rate of 61%–90% in *us-central1* and near 100% in *us-east1* and *us-west1* with minimal cost.

Our co-location verification method is both faster and financially cheaper than conventional pairwise covert-channel testing. In Section 3.4.4, we verify the co-location of 800 container instances. Using pairwise testing, this verification process requires $319,600$ pairwise tests. Moreover, these tests are *serialized* to avoid interference. Assuming an optimistic execution time of 100 ms per test, finishing these pairwise tests would take 8.9 hours. In contrast, we find that our approach only takes about 1 to 2 minutes to validate all 800 instances.

To estimate the financial cost of performing these tests on Cloud Run, we use the Cloud Run pricing model [135]. For a standard instance requesting 1 vCPU and 0.5 GB memory, the cost is estimated using the formula $N \times t \times (R_{cpu} + 0.5R_{mem})$. In this equation, N is the number of active instances, t is the active time of these instances in seconds, R_{cpu} is the CPU time cost per vCPU-second in USD, and R_{mem} is the memory cost per GB-second in USD. At the time of this writing, $R_{cpu} = ¢0.0024$/vCPU-second and $R_{mem} = ¢0.00025$/GB-second in *us-east1*, *us-central1*, and *us-west1*.

Based on this pricing model and rates, performing the pairwise tests would cost about 645 USD. This cost would be even higher if we used the pairwise test method discussed by Varadarajan et al. [39], which takes several seconds to complete one pairwise test. In contrast, our approach costs about 1 to 3 USD. Importantly, the time and financial cost of pairwise testing grows quadratically with the number of instances being verified for co-location.

Note that prior work [15, 39] speeds up pairwise testing by filtering out instances that do not co-locate with any other instance. İnci et al. [15] call this filtering step Single Instance Elimination (SIE). During SIE, the attacker tests all instances simultaneously and removes instances that test negative. However, SIE is ineffective in a FaaS environment. This is because the FaaS orchestrator tends to place multiple instances onto the same host, as will be discussed in Section 3.5.2. Consequently, every instance is co-located with some other instances and SIE will fail to remove any instance.

3.4.4 Evaluating Fingerprinting

Accuracy Results. We evaluate fingerprint accuracy at a large scale on the public Cloud Run platform in three different data centers: *us-east1*, *us-central1*, and *us-west1*. In each data center, we deploy a service and launch 800 concurrently-running container instances. Although a user can launch up to 1000 container instances from the same service on Cloud Run, new instance creation slows down as the instance count approaches 1000, escalating the financial cost. As a result, we launch 800 instances. We accomplish this by configuring each instance to just handle one connection and then establishing 800 WebSocket connections to these instances.

For each instance, we collect its host CPU model name, the TSC value, and the real-world time of the measurement. We also record the true co-locations of the instances (i.e., the ground truth) using the scalable validation methodology discussed in Section 3.4.3. Our implementation of the methodology utilizes a low-noise covert channel based on contention on the random number generator (RNG) [90]. Because the RNG is rarely used [90], we find that the likelihood of observing RNG contention due to background activities is less than 1%. We require the presence of contention in at least 30 measurements out of 60 to confirm co-location. Hence, the risk of false positives is extremely small. Finally, we evaluate the fingerprint accuracy while varying the rounding precision of T_{boot} (i.e., p_{boot}). We repeat our experiments 5 times at different days and different times of day, totaling 15 measurements across three data centers.

To measure fingerprint accuracy, we examine all unique pairs of instances. For each pair of matching fingerprints, if the instances are indeed co-located, it is a true positive; otherwise, it is a false positive. For each pair of mismatching fingerprints, if the instances are not co-located, it is a true negative; otherwise, it is a false negative. We call the number of true and false positives TP and FP, respectively, and the number of true and false negatives TN and FN, respectively. Then, we compute the Fowlkes-Mallows index (FMI) [136], which is a common metric of clustering performance. FMI is calculated by

$$FMI = \sqrt{precision \cdot recall} = \sqrt{\frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}}.$$

FMI ranges from 0 to 1, with 1 indicating that the fingerprints are perfect (i.e., there are no false positives or false negatives).

Figure 3.3 shows the accuracy results averaged across all measurements and the three data centers under evaluation. The top plot of Figure 3.3 shows the FMI for different values of p_{boot}; the bottom plot shows the recall and the precision for the same values of p_{boot}. In the figure, p_{boot} values are measured in seconds. The error bars represent the standard deviations. From Figure 3.3, we observe that when p_{boot} is small and, therefore, the rounded T_{boot} has many significant digits (left end of Figure 3.3), fingerprints suffer from low recall (i.e., many false negatives) and have a low FMI. The reason is that a small p_{boot} cannot overcome the noise in measuring T_{boot}. Hence, the same host gets different fingerprints in different co-located instances. On the other hand, if p_{boot} is very large and, therefore, the rounded T_{boot} has few significant digits (right end of Figure 3.3), different hosts with similar boot time are rounded to the same value. The result is many false positives, which reduce the precision and the FMI.

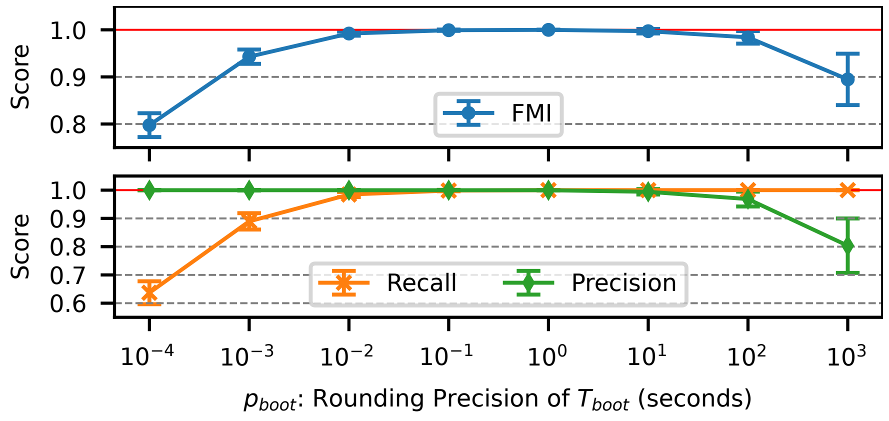


Figure 3.3: Average fingerprint accuracy with respect to the rounding precision of T_{boot} (i.e., p_{boot}). Error bars show standard deviations. Our Gen 1 fingerprints show near-perfect accuracies with 100 ms $\leq p_{boot} \leq$ 1 s.

The sweet spot for p_{boot} ranges from 100 ms to 1 s, where we reach an average FMI of 0.9999, which is nearly perfect. Since using a large p_{boot} can extend the expiration time of fingerprints, we use p_{boot} = 1 s by default. With this value, our fingerprints are highly accurate: among the 15 experiments conducted across three data centers, we find that 14 generate *perfect* fingerprints, while one generates nearly perfect fingerprints.

Fingerprint Expiration Time Results. Our fingerprints are subject to drifting because we use the *reported* TSC frequency, which is inaccurate (Section 3.4.2). However, having long-lived fingerprints is crucial for the attacker to track hosts and understand instance placement behavior across many measurements over the time. Hence, in this subsection, we evaluate the expiration time of GEN 1 fingerprints.

To observe how the T_{boot} of a host drifts over the time, we launch a group of 50 long-running container instances and continuously record their hosts' fingerprints every hour for one week. Note that these instances can still be terminated and restarted on a different host over the course of our measurement. When this happens, we conservatively assume that the restarted instance runs on a different host. Consequently, most hosts have a fingerprint history shorter than a week. We filtered out fingerprint histories that are shorter than 24 hours.

We conducted our experiment in the *us-east1*, *us-central1*, and *us-west1* data centers. After filtering, we obtained 66, 67, and 79 fingerprint histories in each data center, respectively. Since we hypothesize that T_{boot} drifts linearly in Eq. 3.4.2, we use linear regression to fit T_{boot} as a function of real-world time for each fingerprint history and examine the r-value of the linear regression. An r-value with an absolute value close to 1 corresponds to a strong linear correlation [137]. We found that the *minimum* absolute r-value across all histories is 0.9997, suggesting that T_{boot} indeed drifts linearly.

Based on the T_{boot} of a fingerprint and the slope of its drifting obtained from the linear regression, we can use linear interpolation to accurately estimate the fingerprint expiration time. The expiration time is the amount of time it takes for T_{boot} to drift across a rounding boundary and result in a different rounded value. Figure 3.4 shows the CDF of the estimated expiration time of fingerprints in each of the three data centers. From the plot, it is clear that most fingerprints can last a few days before they expire. The average estimated time for 10% of the fingerprints to expire is about 2 days.

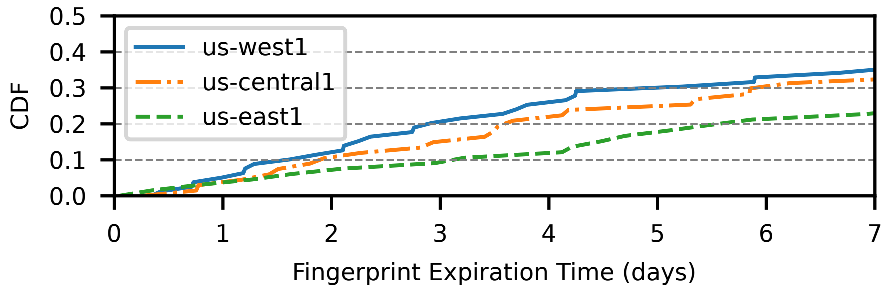


Figure 3.4: CDF of the estimated fingerprint expiration time. Most GEN 1 fingerprints take many days to expire.

3.4.5 Host Fingerprinting in the Gen 2 Environment

Our fingerprinting technique for Gen 1 is not directly applicable to the Gen 2 environment due to a hardware virtualization feature known as *TSC offsetting* [130]. With TSC offsetting, the hypervisor can configure the hardware to add an offset to the host TSC when it is read by the guest VM. Typically, when the hypervisor boots a guest VM, it saves the current value of the host TSC (call it tsc_0). Then, when the guest VM asks for a TSC value, the host returns the current TSC value minus tsc_0. This creates the illusion to the guest VM that the TSC was zero when the guest VM booted. Therefore, using Eq. 3.4.1, one can only derive the boot time of the guest VM instead of the host's.

To circumvent this challenge, we point out that, although the TSC value read by the guest VM has an unknown offset, the guest TSC still increments at the same rate as the host's. As a result, the guest VM can observe the host's actual TSC frequency, which deviates from the reported frequency and is likely to be unique among hosts. Therefore, we propose to use the actual host TSC frequency to fingerprint a host in the Gen 2 execution environment.

Surprisingly, obtaining the actual host TSC frequency is easier in the VM-based Gen 2 environment than in the Linux container-based Gen 1 environment. This is despite the fact that VMs typically offer good isolation between the guest and host. In the Gen 2 environment, KVM exports the refined host TSC frequency to the guest VM for timekeeping. Since the attacker program has the root privilege within the guest VM, it can simply read the refined frequency from the guest kernel. However, this approach cannot be used to obtain the refined host TSC frequency in the Gen 1 environment and use it as f in Eq. 3.4.1, as the sandboxed Linux container can only interact with gVisor.

Using the same setup for validating Gen 1 fingerprints, we evaluate the accuracy of Gen 2 fingerprints in *us-east1*, *us-central1*, and *us-west1*. Our evaluation results show that the Gen 2 fingerprint is less accurate than the Gen 1 one, due to its low precision (i.e., its many false positives). Averaged across all measurements in the three data centers, the Gen 2 fingerprint has an FMI of 0.66, and a precision of 0.48. The reason for the low precision is that Linux only refines the TSC frequency to a precision of 1 kHz, causing multiple hosts to share the same refined frequency. In our experiments across three data centers, we find that, on average, 2.0 hosts have the same fingerprint.

Although the Gen 2 fingerprint has relatively low precision, it cannot produce false negatives. This is because the host TSC frequency is refined only once at host boot time, which means that co-located instances must have the same host TSC frequency. Because Gen 2 fingerprints cannot have false negatives, when we use the covert-channel approach of Figure 3.2 to verify fingerprints, we can perform the tests in Step ② *in parallel* without worrying about interference between tests. Also, we can skip Step ③, which finds false negatives. Consequently, even if the Gen 2 fingerprint is less accurate, we can still efficiently generate the co-location ground truth for numerous container instances.

3.5 CLOUD RUN ORCHESTRATOR AND CO-LOCATION

Leveraging our host fingerprinting and co-location verification techniques proposed in Section 3.4, we can accurately identify physical hosts within a data center and verify instance co-location inexpensively. In Section 3.5.1, we employ these two techniques to systematically study Cloud Run's instance placement policy, uncovering exploitable behaviors. In Section 3.5.2, we propose adversarial instance launching strategies that exploit these uncovered behaviors, along with an evaluation of their efficacy and financial cost.

We set up our investigation using three standard Google Cloud accounts: ACCOUNT 1, ACCOUNT 2, and ACCOUNT 3. ACCOUNT 1 is designated as the attacker account, while ACCOUNT 2 and ACCOUNT 3 serve as victim accounts. Following our threat model of Section 3.3, we assume that once attacker and victim are co-located, the attacker can detect victim program execution and exfiltrate sensitive information using prior techniques [3, 15, 37, 39, 132].

When we rely on fingerprints to identify hosts without verifying them using a covert channel, we refer to these hosts as the *apparent hosts*. As GEN 1 fingerprints are nearly perfect and long-lived (Sections 3.4.4 and 3.4.4), apparent hosts identified by GEN 1 fingerprints should closely match real physical hosts. Lastly, our primary focus is on the GEN 1 environment, unless specified otherwise.

3.5.1 Understanding the Instance Placement Policy

We perform a set of experiments to study the instance placement policy of Cloud Run. Through these experiments, we seek to answer the following questions about a user launching numerous container instances: (i) How are the instances distributed across hosts and managed by Cloud Run (Experiment 1); (ii) Does the orchestrator exhibit a consistent behavior across launches (Experiment 2); and (iii) What are the major factors that affect the orchestrator's behavior (Experiments 3 and 4). In the remainder of this subsection, we primarily focus on the GEN 1 environment in the *us-east1* data center. We will discuss the different execution environments and data centers at the end of this subsection.

Experiment 1: Instance distribution. In this experiment, we launch 800 instances of the same service, and record the set of hosts they are placed onto (i.e., their *host footprint*). We use the covert-channel approach to generate the co-location ground truth. We observe that these 800 instances are placed onto 75 hosts. Moreover, we see that the instance distribution across the hosts used is close to uniform, with the majority of hosts running 10 or 11 instances. This behavior is different from

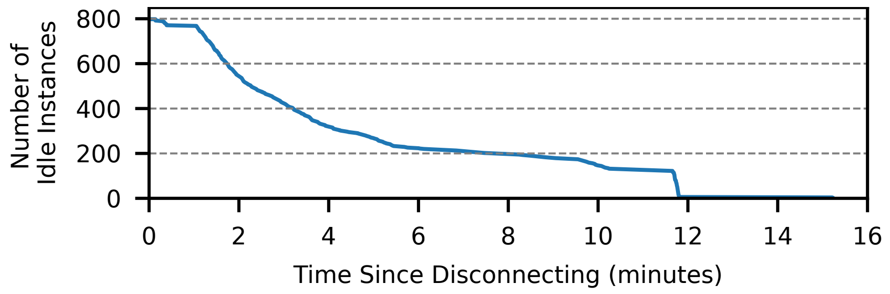


Figure 3.5: Number of idle instances after disconnecting from 800 instances, as measured in *us-east1*. Practically all instances are terminated in 12 minutes after disconnecting.

that of a VM environment (e.g., AWS EC2 [138]), where it is observed that instances from the same account do not share a host [37, 38].

Observation 1. Container instances of the same service share hosts, and instance distribution across the hosts used is close to uniform.

Next, we disconnect from these 800 instances, leaving them in an idle state, and observe when they are terminated by the orchestrator (Section 2.1). To record the termination time, we capture the `SIGTERM` signal sent by the orchestrator before it terminates an instance [64]. Upon capturing `SIGTERM`, the container reports the current time to a separate server and then terminates. Figure 3.5 shows the number of idle instances as a function of time since disconnecting. From the plot, we can see that these idle instances are preserved in the first minute. After that, the orchestrator starts to gradually terminate idle instances. After about 12 minutes, almost every instance is terminated. This behavior matches Cloud Run's documentation [64], which states that idle instances are preserved for at most 15 minutes.

Observation 2. Cloud Run gradually terminates idle instances over an approximate period of 12 minutes.

Experiment 2: Behavior across launches. We study if this scheduling behavior changes across launches. In this experiment, we repeat six times the launch of 800 instances of the same service, and compare the host footprint of each launch. After each launch, we immediately disconnect from the 800 instances, putting them into an idle state. Then, we wait for 45 minutes before the next launch to make sure that all the old instances are terminated and the service enters a "cold" state. This cold state will be discussed in Experiment 4.

Upon analyzing the experiment results, we observe that the instance distribution remains consistent across launches. We then examine whether instances from different launches share any hosts.

Since the old instances from a previous launch are terminated before the next launch, it is impossible to use a covert channel to verify co-location of instances across launches. We rely solely on fingerprints to identify hosts in this experiment, thus reporting the apparent hosts.

Figure 3.6 shows the number of apparent hosts in each launch (identified by the Launch ID). It also shows the cumulative number of apparent hosts since the first launch. We observe that each launch occupies a similar number of apparent hosts. Moreover, the growth of the cumulative number of apparent hosts is minimal, suggesting that the apparent host footprints are highly overlapped across launches. This behavior can be caused by a data locality optimization, as the orchestrator may prefer hosts that already have the container image from previous launches.

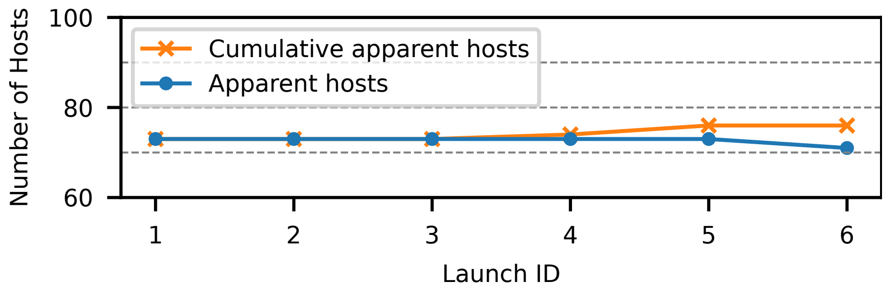


Figure 3.6: Number of apparent hosts occupied by 800 instances of the same service, as measured in *us-east1*. The footprints of apparent hosts are highly overlapped across launches.

To test the hypothesis of the data locality optimization, we repeat this experiment using a *different* service in each launch. These services are owned by the same account. Before the experiment, we rebuild the container images of the services that will be invoked, to ensure that the images of the services are not cached in any host. Under this configuration, we still observe a pattern that closely resembles Figure 3.6.

These experiment results suggest that the orchestrator tends to place instances from the *same account* onto a specific set of hosts. This behavior can be explained by affinity scheduling (Section 2.1). Affinity scheduling aims to reduce communication overhead by co-locating instances that frequently interact with each other, which is a likely scenario for services originating from the same account.

Observation 3. Cloud Run exhibits a consistent behavior across launches. It prefers a specific set of hosts for container instances owned by the same account. We refer to these preferred hosts as the ***base hosts***.

Experiment 3: Different accounts. In this experiment, we modify Experiment 2 slightly: the services used in launch 1 and 2 are owned by ACCOUNT 1, the services used in launch 3 and 4 are owned by ACCOUNT 2, and the services used in launch 5 and 6 are owned by ACCOUNT 3. Figure 3.7

illustrates the results in a manner similar to Figure 3.6. We observe that the cumulative number of apparent hosts establish a step pattern. When a launch uses a service owned by an account different from the accounts in previous launches, we see a large growth in the cumulative number of apparent hosts; otherwise, the growth is minimal. This observation suggests that the orchestrator uses different base hosts for different accounts.

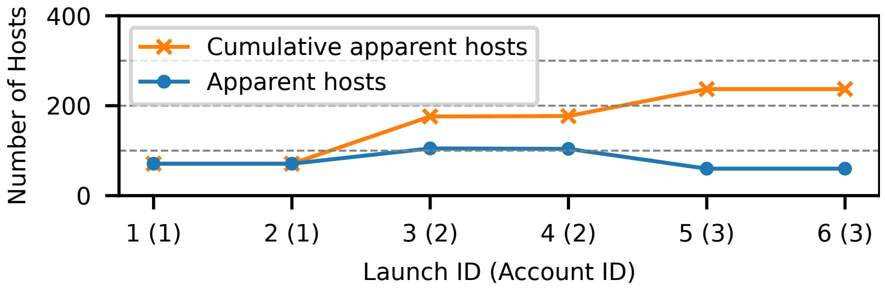


Figure 3.7: Number of apparent hosts occupied by 800 instances from three different accounts, as measured in *us-east1*. The numbers in parenthesis are the account IDs.

Observation 4. The Cloud Run orchestrator uses different base hosts for different accounts.

Experiment 4: Short launch interval. In this experiment, we repeat Experiment 2 with a short time interval between launches of 10 minutes. Under this configuration, we see an interesting orchestrator behavior, as illustrated in Figure 3.8. Unlike Figure 3.6 from Experiment 2, we observe that both the number of apparent hosts and the cumulative number of apparent hosts drastically increase after each of the first three launches. Moreover, the difference between the two curves is small. These results suggest that the orchestrator places instances to both the hosts used in previous launches and the new hosts. As a result, after six launches, we have a host footprint of 264 apparent hosts, which is far higher than the number of base hosts.

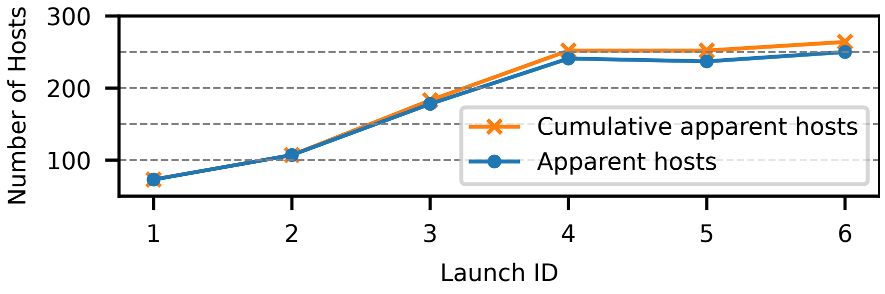


Figure 3.8: Experiment 2 repeated with a time interval between launches of 10 minutes, as measured in *us-east1*. We observe drastic increases in both the number of apparent hosts and cumulative number of apparent hosts.

To further investigate this behavior, we repeat this experiment with different launch intervals. We observe that this behavior only occurs with an interval smaller than 30 minutes. Furthermore, when the interval is too short, the number of new hosts is small. For example, with the 10-minute interval shown in Figure 3.8, we observe 177 more apparent hosts after launch 6 than after launch 1. However, with a 2-minute interval, we observe only 12 more apparent hosts. We also repeat the experiment with a different service used in each launch, but we do not observe this behavior.

Based on the aforementioned observations, we hypothesize that this behavior is induced by a load-balancing mechanism of Cloud Run. The mechanism considers the usage of the *same* service within approximately the past 30 minutes. If the service exhibits a high demand (i.e., repeatedly running 800 concurrent instances in our case), then the orchestrator will attempt to place some of the instances of the same service onto hosts that are not base hosts. As a result, it reduces the load on the base hosts. We refer to these extra hosts that are used as *helper hosts*. After a certain number of repeated launches, this behavior saturates.

Observation 5. For a service that has a high demand within less than 30 minutes, Cloud Run appears to use a load balancer that places instances of the service onto hosts that are not base hosts (i.e., ***helper hosts***).

Under this hypothesis, in the first launch, since there is no usage history of the service in the past 30 minutes, instances are placed onto the base hosts. As we wait for some time before the next launch, some of the idle instances are terminated. Therefore, in the next launch, the orchestrator has to create new instances to compensate for the terminated ones. Since the previous launch has primed the service into a high-demand state, the orchestrator starts to place the newly-created instances on the helper hosts to relieve pressure on the base hosts. This cycle repeats for a few iterations. When the wait interval between launches is small, the number of terminated idle instances before the next launch is also small. Consequently, the orchestrator creates fewer new instances, thereby occupying fewer helper hosts.

We consistently observe this behavior when repeating the experiment at different times of the day or using a different service in the experiment. We also observe that different services use different sets of helper hosts; these sets are not mutually exclusive and do overlap. We demonstrate this fact by repeating Experiment 4 for six episodes; in each episode, we use a different service that is launched six times with 800 instances every time. In each episode of Experiment 4, we measure the helper host footprint by computing the difference between the host footprint after the sixth launch and after the first launch.

Figure 3.9 shows the results of the experiment. It shows the number of apparent helper hosts and the cumulative footprint of apparent helper hosts after each of the episodes. We see that the cumulative footprint of apparent helper hosts expands after each episode. This expansion suggests that each episode uses new helper hosts not seen in previous episodes. The increase in the cumulative footprint of helper hosts after a single episode is less than the number of helper hosts in that episode, indicating overlaps in helper hosts across different services. As will be discussed later, an attacker can exploit this behavior by repeatedly launching instances of multiple services and therefore obtaining residence on a substantial portion of Cloud Run hosts within a data center.

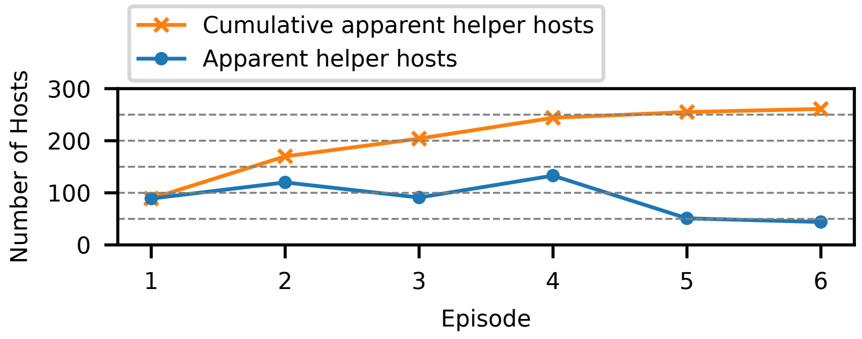


Figure 3.9: Experiment 4 repeated in six episodes, with a different service used per episode, as measured in *us-east1*. We see growth in the cumulative *helper* host footprint after each episode.

Observation 6. Different Cloud Run services use different but overlapping sets of helper hosts.

Other factors. We investigate other factors that influence the orchestrator's behavior. We report four findings. First, we observe similar placement behavior when launching on different dates and at different times of day. Second, container instances with different resource specifications (such as CPU and memory) share the same base hosts. Third, all nine Cloud Run data centers in the US exhibit similar placement behavior except for *us-central1*, where instance placement is more dynamic. In *us-central1*, many instances are placed onto different hosts across launches, even if we launch from a cold service in each launch. Fourth, the Gen 2 execution environment shows similar placement behavior, and Gen 2 instances can share hosts with Gen 1 instances.

Implications. The existence of base hosts is a double-edged sword for the attacker. On the one hand, it reduces the uncertainty on where the victim instances are likely to reside, making co-location with the victim easier. On the other hand, base hosts limit the set of hosts where the attacker can reside and, therefore, the set of hosts that the attacker can explore. As a result, naively launching attacker instances has a low chance of co-locating with the victim (Section 3.5.2), as different accounts often use different base hosts. In practice, the load-balancing behavior of Cloud Run helps the attacker to overcome this challenge. For example, the attacker can prime their services into a high-demand state through repeated launches, which help spread attacker instances onto many helper hosts. This strategy drastically improves the efficacy of co-location attacks (Section 3.5.2).

3.5.2 Co-location with Victims

In this subsection, we evaluate two instance launching strategies for co-location with victims. Our primary metric is the *victim instance coverage*, which is the percentage of victim instances that are co-located with the attacker. Then, we report the financial cost of the attack, the estimated size of the Cloud Run data centers, and the transferability of our results to the GEN 2 environment. We conclude with a discussion on potential optimizations that can enhance attack efficacy.

Evaluation setup. For this evaluation, ACCOUNT 1 is designated as the attacker, while ACCOUNT 2 and ACCOUNT 3 serve as the victims. We conduct the evaluation across three data centers: *us-east1*, *us-central1*, and *us-west1*. For each combination of data center and victim account, we repeat the measurement three times on different days and at different times of day. Co-location between attacker and victim instances is verified using the covert-channel method described in Section 3.4.3.

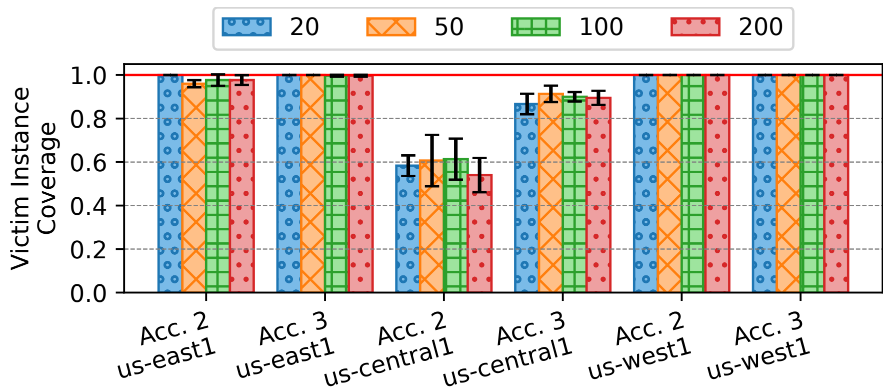
In each experiment, we vary the number of victim instances. As the default configuration for Cloud Run services allows a maximum of 100 instances, we assess configurations with 20, 50, 100, and 200 victim instances, setting the 100-instance configuration as the default. Prior work [139] suggests that orchestrators might prefer co-locating instances with similar resource specifications (such as CPU and memory) in the same nodes. Accordingly, we vary the size of the victim instances, using the sizes outlined in Table 3.1. We choose SMALL as the default victim size since it is the standard configuration for Cloud Run services. We also fix the attacker instance size to SMALL.

Table 3.1: Various container sizes used in our evaluation. Note that we define these four container sizes for the purpose of this study; a user can use a size different than these four.

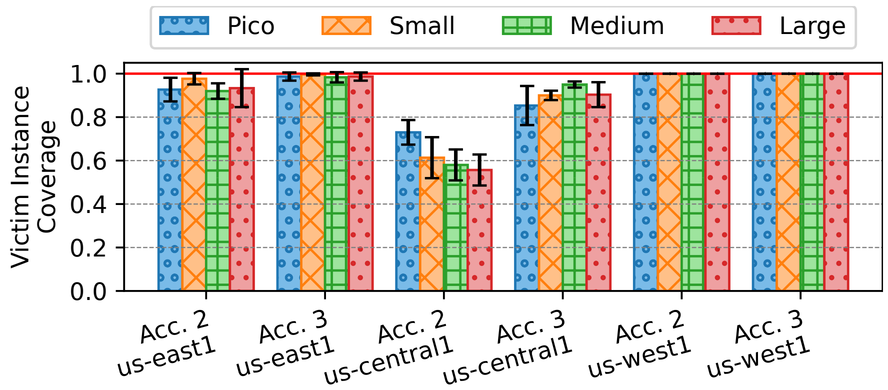
Size	# of CPUs	Memory
PICO	0.25	256 MB
SMALL (Default)	1	512 MB
MEDIUM	2	1 GB
LARGE	4	4 GB

Strategy 1: Naive instance launching. Here, the attacker simply launches numerous instances from services in a cold state. This strategy represents a naive attacker who has no insight into the Cloud Run's instance placement behavior. In our experiment, this naive strategy launches $4,800$ instances from six services.

Despite the large number of attacker instances, we observe *zero* co-location with ACCOUNT 2 in *us-east1* and *us-central1*, or with ACCOUNT 3 in *us-east1* and *us-west1*. We see high average victim instance coverage only with ACCOUNT 2 in *us-west1* (100.0%) and with ACCOUNT 3 in *us-central1* (81.0%), as the base hosts of the attacker and victim happen to be highly overlapped in the corresponding data centers. Changing the number of victim instances or their size does not yield



(a) Varying the number of victim instances (20, 50, 100, and 200). The victim instance size is fixed to SMALL.



(b) Varying the victim instance size (Pico, Small, Medium, and Large). The number of victim instances is fixed to 100.

Figure 3.10: Average victim instance coverage across three measurements. Error bars represent standard deviations. "Acc." is an abbreviation for "Account". Our optimized launching strategy can achieve a high victim coverage in different evaluated data centers.

significant variations. This is consistent with our observation that services from the same account share the same base hosts, even when they have different resource specifications (Section 3.5.1). Overall, the data indicates that a naive launching strategy without any insight into Cloud Run's placement behaviors is often ineffective.

Strategy 2: Optimized instance launching. This strategy exploits the load-balancing behavior of Cloud Run. The high-level idea is to prime the attacker service into a high-demand state by repeatedly launching many instances with an appropriate time interval. This action enables the attacker to deploy instances onto numerous helper hosts. Given our observation that different services use different but overlapping sets of helper hosts (Section 3.5.1), attacker instances can reside on more helper hosts if they utilize multiple services. In our experiment, the attacker deploys six services. Similar to Experiment 4 in Section 3.5.1, the attacker repeatedly launches 800 instances of each service at a 10-minute interval, killing the instances after each launch except after the last one.

Figure 3.10 shows the average victim instance coverage using the optimized launching strategy, with the error bars indicating standard deviations. In Figure 3.10a, we vary the number of victim instances while fixing the victim size to SMALL. Conversely, in Figure 3.10b, we vary the victim size while keeping the number of victim instances set to 100.

Figure 3.10a illustrates that our optimized instance launching strategy is highly effective. With the default configuration of 100 SMALL victim instances, we observe high victim instance coverage. From left to right, we see, in *us-east1*, victim instance coverages of 97.7% and 99.7% with ACCOUNT 2 and ACCOUNT 3, respectively. In *us-central1*, we see lower coverages of 61.3% with ACCOUNT 2 and 90.0% with ACCOUNT 3. Finally, in *us-west1*, we see 100.0% coverage with both ACCOUNT 2 and ACCOUNT 3. One potential reason for the reduced coverage in the *us-central1* data center is its vast size, as will be shown later in this section. Another factor might be that *us-central1* has a more dynamic instance placement behavior, as discovered in Section 3.5.1.

Figure 3.10b shows a similar behavior, as we vary the victim size while keeping the number of victim instances set to 100. Overall, considering the data from both Figure 3.10a and 3.10b, we conclude that, in the large majority of cases, the number or the size of victim instances has no significant influence on the average victim instance coverage.

Financial cost of the attack. FaaS platforms such as Cloud Run only charge users for the active time of instances. Since we disconnect from all the instances after each launch, these instances are in the idle state between launches and do not contribute to the cost during this time. The main cost comes from launching instances. On average, to set up a co-location attack with our configuration (six attacker services, six launches per service, and 800 instances per launch), the estimated average costs are 24 USD, 23 USD, and 27 USD in *us-east1*, *us-central1*, and *us-west1*, respectively. These costs are small.

Scale of Cloud Run clusters. To estimate the size of a Cloud Run cluster, we deploy eight services from each of the three accounts (ACCOUNT 1, ACCOUNT 2, and ACCOUNT 3) and use the total 24 services to explore hosts that run Cloud Run services in the data center. We use the optimized strategy to launch instances of these services and record the apparent host footprint of each launch. We launch each service four times. Then, the size of the Cloud Run cluster is estimated by counting the number of unique host fingerprints across *all* launches. The intuition for using services from different accounts instead of more services from the same account is that we can start exploration from different base hosts, and thus discover new hosts more efficiently.

Figure 3.11 shows the cumulative number of unique apparent hosts as we aggregate apparent hosts across launches. In total, these launches found 474 apparent hosts in *us-east1*, 1702 apparent hosts in *us-central1*, and 199 apparent hosts in *us-west1*. Since the growth of the cumulative number of unique apparent hosts gradually flattens out in all three data centers as we include more launches, it is reasonable to use the total number of unique apparent hosts that we found to estimate the size of the Cloud Run cluster in each data center. Using this estimation, in the co-location experiment that we performed using Strategy 2, the attacker (ACCOUNT 1) covered 59%, 53%, and 82% of the hosts in *us-east1*, *us-central1*, and *us-west1* on average, respectively.

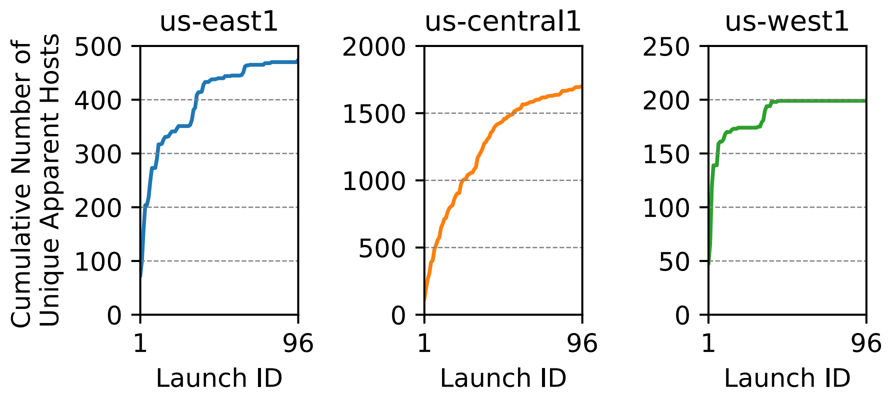


Figure 3.11: Cumulative number of unique apparent hosts across launches.

Co-location in the GEN 2 environment. We evaluate our optimized launching strategy in the GEN 2 environment, with both attacker and victims launching GEN 2 instances. Averaging three measurements in each data center, we observe victim instance coverage of 87.3% with ACCOUNT 2 and 88.7% with ACCOUNT 3 in *us-east1*; 40.7% with ACCOUNT 2 and 75.3% with ACCOUNT 3 in *us-central1*; and 96.0% with ACCOUNT 2 and 97.3% with ACCOUNT 3 in *us-west1*. No significant coverage differences arise when varying the number of victim instances or size. These results indicate that our launching strategy is highly effective in the GEN 2 environment as well.

Potential attack optimizations. To occupy an even larger fraction of Cloud Run hosts within a data center, the attacker can create more accounts and deploy more services per account. This approach is similar to the experiment where we measured the scale of Cloud Run clusters. However, a challenge arises as cloud providers often cap new accounts to limited resources—e.g., allowing a maximum of only 10 instances per service. For an attacker to be eligible for higher quotas, the new account needs to sustain a consistent usage *over several months*. This limitation results in additional time and financial costs.

If the attacker intends to repeatedly attack services from the same victim account, an optimization is to record the fingerprints of hosts used by the victim during the first attack. These hosts can be the base hosts preferred by the victim. Therefore, in the subsequent attacks targeting the same victim, the attacker can focus side-channel attack efforts on hosts with fingerprints that match the fingerprints recorded in the first attack.

3.6 POTENTIAL MITIGATIONS

Our two fingerprinting techniques exploit the fact that the timestamp counter (TSC) value or its frequency are shared between the host and untrusted user containers. Therefore, to defend against our techniques, one could mask both the value and frequency of the host's TSC through either TSC emulation or hardware-assisted TSC virtualization.

In the non-virtualized GEN 1 environment, the host can disable `rdtsc` and `rdtscp` in Ring-3 (i.e., userspace) by configuring the `CR4` model specific register [130]. By doing so, the kernel can trap and emulate both instructions. However, this mitigation also forces user applications to switch to kernel space when accessing high-precision timers, incurring high timer access overhead.

The impact of the slower timer accesses depends on the specific application and its use case. Hence, the actual end-to-end execution overhead in an application can only be determined through benchmarking. We identify some applications where this added end-to-end execution overhead is likely to matter: (1) real-time systems that process high-frequency events such as live media or financial data, (2) database systems using fine-grained timestamps for concurrency control, (3) distributed systems employing fine-grained timestamps for synchronization, and (4) applications characterized by intensive logging and journaling. For instance, Cassandra's [140] write latency is reportedly reduced by 43% on Amazon EC2 instances after switching from the `xen` clock source to TSC (the `xen` clock source requires a context switch to kernel space to access) [141].

In the virtualized GEN 2 environment, the hypervisor can also trap and emulate both `rdtsc` and `rdtscp`, which, as in GEN 1, leads to significant timer access overhead. An alternative that does not add overhead is for the host to support hardware-assisted TSC virtualization features, such as TSC offseting and scaling [130, 142]. These features are available on modern Intel and AMD processors and were primarily developed for live VM migration.

Besides emulating or virtualizing the TSC, cloud vendors can also adopt scheduling algorithms that reduce the chance of co-location [143, 144] or mitigate the risk of side-channel attacks after co-location is achieved [145]. Finally, they can detect and stop ongoing side-channel attacks [146, 147, 148, 149].

3.7 RELATED WORK

Co-location attacks in the public cloud. In 2009, Ristenpart et al. [37] conducted the first study of VM co-location attacks on Amazon's EC2 service using network probing. To assess Amazon's defenses in response to this work, Xu et al. [38] investigated VM co-location attacks on Amazon EC2 service in 2015, by employing network scanning. However, these network-based techniques have become obsolete in the modern cloud environment, due to the widespread adoption of the virtual private cloud (VPC) [126], which logically isolates the networking environments of different accounts. To overcome the challenge posed by VPC, Varadarajan et al. [39] employed a pairwise covert-channel test, based on memory bus contention explored by Wu et al. [89], to investigate VM co-location attacks. However, due to the scalability issue of pairwise testing, their approach is unsuitable for the modern FaaS environment, where it is necessary to verify co-location of thousands of instances.

Exploiting cloud orchestrators. Makrani et al. [150] proposed an attack named Cloak & Co-locate, which employs adversarial machine learning to generate fake resource usage traces, fooling machine learning-based resource provisioning systems [151, 152] to co-locate attacker instances with the victim. However, their evaluation is limited to a private mini-cloud running on local clusters. Concurrent to Cloak & Co-locate, Fang et al. [139] proposed Repttack, suggesting that the attacker can launch instances with requirements and preferences that replicate the victim's to increase the likelihood of co-location. However, on Cloud Run, we do not observe any significant increase in co-location rate when the attacker instance has the same configuration as the victim (Section 3.5.2).

Remote device fingerprinting. Kohno et al. [153] exploited the clock skew in system time that is accumulated over time to fingerprint remote physical devices. They monitor such skew through timestamps included in TCP packets. However, for contemporary systems where clocks are well synchronized to the real-world time through the network time protocol (NTP) [154], such accumulated clock skew is not detectable using coarse-grained TCP timestamps [155], which have the resolution of only one millisecond [156]. Compared to their approach, our fingerprinting method for Gen 1 relies on the host's boot time instead of clock skew. Further, our fingerprinting method for Gen 2 detects clock *frequency* skews, making it effective even if clocks are well synchronized through NTP.

3.8 CONCLUSION

In this chapter, we presented the first comprehensive study on risks of and techniques for co-location attacks in modern public cloud FaaS environments. We introduced two novel physical host fingerprinting techniques based on the timestamp counter to aid in reverse engineering instance placement policies. Using host fingerprints, we proposed a cost-effective methodology for large-scale instance co-location verification. With these techniques, we conducted an extensive study on Google Cloud Run, discovering an exploitable instance placement behavior. Based on our findings, we devised an efficient instance launching strategy that deploys attacker instances across a large portion of Cloud Run cluster hosts within a data center. Our strategy attains high attack efficacy at minimal financial cost.

CHAPTER 4: Last-Level Cache Prime+Probe Attack in the Modern Public Cloud

4.1 INTRODUCTION

In Chapter 3, we showed how an unprivileged cloud user can co-locate their containers with a target victim's containers on the same physical hosts. In this chapter, we focus on how to extract information from the victim container with side-channel attacks after co-location.

Among all side-channel attacks, a particularly dangerous class of attacks is Prime+Probe attacks on the last-level cache (LLC) [3, 4, 5, 15, 157, 158, 159] (Section 2.2.2). This is because these attacks do not require the attacker to share a physical core or memory pages with the victim program. As a result, this chapter focuses on LLC Prime+Probe attacks in public clouds.

Despite the potency of LLC Prime+Probe attacks, executing them in a modern public cloud environment is challenging for several reasons. First, the modern cloud is *noisy*, as the hardware is shared by many tenants to attain high computation density [34, 35, 160]. In particular, the LLC is flooded with noise created by activities of other tenants. This noise not only interferes with eviction set construction (Step 2 in Table 2.1), but also poses challenges to identifying the target LLC sets (Step 3 in Table 2.1) and exfiltrating information (Step 4 in Table 2.1).

Second, the modern cloud is *dynamic*. With cloud computing paradigms like Function-as-a-Service (FaaS) [29, 30, 31], user workloads are typically short-lived on a host (e.g., they last only from a few minutes to tens of minutes [64, 65, 66, 161, 162]). As a result, the attacker has a *limited time window* to complete *all* the attack steps while co-locating with the victim. This challenge is exacerbated by the increased number of LLC sets in modern processors—which require preparing more eviction sets and monitoring more cache sets.

Third, the lack of huge pages in some containerized environments [30] and the wide adoption of non-inclusive LLCs increase the effort to execute LLC Prime+Probe attacks in clouds [5]. Thus, while İnci et al. [15, 40] conducted an LLC Prime+Probe attack on AWS EC2 in 2015, their techniques are incompatible with modern clouds, as they relied on huge pages, long-running attack steps, and inclusive LLCs.

As a result of the aforementioned challenges, cloud vendors believe that LLC Prime+Probe attacks are not a threat "in the wild." For instance, the security design whitepaper of Amazon's Elastic Compute Cloud (EC2) [138] explicitly rules out LLC Prime+Probe attacks as impractical [36].

This chapter refutes the belief that LLC Prime+Probe attacks are impractical in the noisy modern public cloud. We demonstrate, for the first time, an end-to-end, cross-tenant attack on cryptographic code (a vulnerable ECDSA implementation [163]) on Cloud Run [30], an FaaS platform from Google Cloud [164]. Every step of the attack requires new techniques to address the practical challenges posed by the cloud. While our demonstrated attack targets Google Cloud Run, the techniques that we develop are applicable to any modern Intel server with a non-inclusive LLC. Therefore, we believe that multi-tenant cloud products from other vendors, such as AWS [165] and Azure [166], may also be susceptible to our attack techniques.

This chapter makes the following contributions:

① **Existing Prime+Probe approaches fail in the cloud.** We show that STEPS 2–4 of Prime+Probe in Table 2.1 are made harder in the Cloud Run environment. In particular, we show that state-of-the-art eviction set construction algorithms, such as group testing [47, 133, 167] and Prime+Scope [159], have a low chance of successfully constructing eviction sets on Cloud Run. Due to the noise present, they take 10× to 24× as much time as when operating in a quiescent local setting. Since the attacker needs to construct eviction sets for up to tens of thousands of LLC sets within a limited time window, this low performance makes existing eviction set construction algorithms unsuitable for the public cloud.

② **Effective construction of eviction sets in the cloud.** To speed-up the generation of eviction sets in STEP 2, we introduce: (1) a generic optimization named *L2-driven candidate address filtering* that is applicable to all eviction set construction algorithms, and (2) a new *Binary Search-based* eviction set construction algorithm. By combining these two techniques, it takes only 2.4 minutes on average to construct eviction sets for all the 57,344 LLC sets of an Intel Skylake-SP machine in the noisy Cloud Run environment, with a median success rate of 99.1%. In contrast, utilizing the well-optimized state-of-the-art eviction set construction algorithms, this process is expected to take at least 14.6 hours.

③ **Techniques for victim monitoring and target set identification.** We develop two novel techniques for STEPS 3–4. The first one, called *Parallel Probing*, enables the monitoring of the victim's memory accesses with high time resolution and with a quick recovery from the noise created by other tenants' accesses. The technique probes a cache set with overlapped accesses, thus featuring a short probe latency and a simple high-performance prime pattern.

The second technique identifies the target LLC sets in a noise-resilient manner. This technique leverages *power spectral density* [168] from signal processing to detect the victim's periodic accesses to the target LLC set in the frequency domain. It enables the attacker to identify the target LLC set in 6.1 s, with an average success rate of 94.1%.

④ **End-to-end attack in production.** Using these techniques, we showcase, for the first time, an *end-to-end*, *cross-tenant* attack on a vulnerable ECDSA implementation [163] on Cloud Run. We successfully extract a median value of 81% of the secret ECDSA nonce bits from a victim container. The LLC Prime+Probe attack, which includes STEPS 2–4 from Table 2.1, takes approximately 19 seconds on average after co-locating on the victim's host.

Availability. We open sourced our implementations at `https://github.com/zzrcxb/LLCFeasible`.

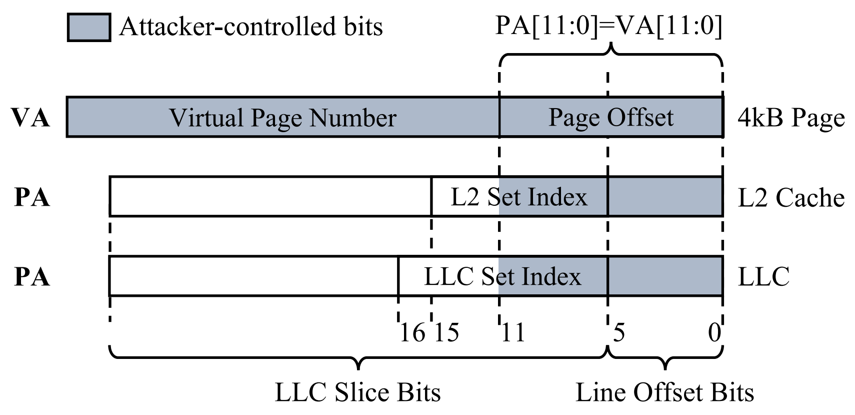


Figure 4.1: Mapping addresses to Skylake-SP's L2 and LLC.

4.2 BACKGROUND: CONSTRUCTING EVICTION SETS ON MODERN INTEL PROCESSORS

4.2.1 Eviction Sets

A key step in Prime+Probe is the construction of an *eviction set* [3, 133]. An eviction set for a specific cache set s is a set of addresses that, once accessed, evict any cache line mapped to s [3, 133]. Given a W-way cache, an eviction set needs to contain at least W addresses that are mapped to s. These addresses are referred to as *congruent addresses* [133]. An eviction set is *minimal* if it has only W congruent addresses.

Eviction set construction algorithms. Building a minimal eviction set for a cache set s generally consists of two steps [3, 133]. The first step is to create a *candidate set* that contains a sufficiently large set of *candidate addresses*, of which at least W addresses are congruent in s. The second step is to prune the candidate set into a minimal set.

① **Candidate set construction.** When a program accesses a virtual address (VA), the address is translated to a physical address (PA) during the access. Part of the PA is used to determine to which cache set the PA maps. For example, Figure 4.1 illustrates the address mapping of Intel Skylake-SP's L2 and LLC. The L2 uses PA bits 15–6 as the set index bits to map a PA to an L2 set. The LLC uses PA bits 16–6 as the set index bits. All the PA bits except for the low-order 6 bits are used to map a PA to an LLC slice [169]. The low-order 6 bits of the PA and VA are shared and are the line offset bits. The low-order 12 bits of the PA and VA are also shared and are the page offset bits. This is because the standard page size is 4 kB.

An unprivileged attacker can control only the page offset of a PA. They lack control and knowledge of the higher-order PA bits. As a result, the attacker has only partial control and knowledge of the set index bits of the L2 and LLC, as well as of the slice index bits of the LLC. Therefore, for a given attacker-controlled VA, there are a number of possible L2 or LLC sets to which it may map. We refer to this number as the *cache uncertainty*, denoted by U.

In general, the set index bits are directly used as the set number. Therefore, the L2 cache's uncertainty is $U_{L2} = 2^{n_{uc}}$, where n_{uc} is the number of uncontrollable L2 set index bits. For the sliced LLC, its uncertainty depends on the slice hash function. On modern processors, LLC slice bits usually map to individual slices via a complex, non-linear hash function [5, 169]. As a result, partial control over the slice index bits is not enough to reduce the number of possible slices that a VA might hash to. Hence, the LLC's uncertainty is $U_{LLC} = 2^{n_{uc}} \times n_{slices}$, where n_{uc} is the number of uncontrollable LLC set index bits and n_{slices} is the number of LLC slices. In the Skylake-SP's address mapping shown in Figure 4.1, there are 5 uncontrollable LLC set index bits and 4 uncontrollable L2 set index bits. Hence, a 28-slice Skylake-SP has an LLC uncertainty of $U_{LLC} = 2^5 \times 28 = 896$ sets and an L2 uncertainty of $U_{L2} = 2^4 = 16$ sets.

When constructing a candidate set for a target cache set s at page offset o, the set needs to contain a large number of addresses with page offset o due to cache uncertainty. Intuitively, the cache uncertainty U indicates how *unlikely* a candidate address maps to s. Therefore, the greater the value of U, the larger the candidate set needs to be [133, 167].

② **Pruning the candidate set into a minimal eviction set.** Given a candidate set, there are several algorithms [3, 47, 51, 133, 159, 170] to build a minimal eviction set with W congruent addresses. We briefly describe two state-of-the-art algorithms [47, 133, 159]. To simplify the discussion, we assume that there is an address T_a that is mapped to cache set s and accessible by the attacker. Consequently, the attacker can determine if a set of addresses forms an eviction set for s by testing whether they evict T_a after being accessed.

Algorithm 1: Group testing [47, 133]. Group testing splits the candidate set into G groups of approximately the same size. A common choice of G is $G = W + 1$ [47, 133]. After the split, the algorithm withholds one group from the candidate set and tests whether the remaining addresses can still evict T_a. This process involves first loading T_a into the cache, traversing the remaining addresses, and timing an access to T_a to check if it remains cached. If T_a is evicted, the withheld group is discarded and the candidate set is reduced; otherwise, the withheld group is added back to the candidate set and the algorithm withholds a different group. Overall, with $G = W + 1$, group testing has a complexity of $O(W^2N)$ memory accesses [133], where W is the associativity of the target cache and N is the candidate set size.

Algorithm 2: Prime+Scope [159]. Prime+Scope first loads T_a. Then, it *sequentially* accesses each candidate address from the list. After each candidate address is accessed, the algorithm checks

whether T_a is still cached. If it is not, that indicates that the last accessed candidate address is congruent, and it is added to the eviction set. This search for congruent addresses is repeated until W different congruent addresses are identified, which form a minimal eviction set for s.

Number of Eviction Sets. In practice, a victim often accesses only a few target cache sets in a secret-dependent manner. An unprivileged attacker, however, generally has limited or no information about the locations of these target cache sets. Consequently, in STEP 2 of Table 2.1, the attacker needs to build eviction sets for all possible cache sets that might be the targets. Subsequently, in STEP 3, the attacker uses Prime+Probe to monitor each of these possible cache sets to identify the actual target cache sets.

The number of eviction sets that the attacker needs to build and monitor depends on how much information about the target cache sets the attacker has. If the attacker knows the page offset of a target cache set, they only need to build eviction sets for cache sets corresponding to that page offset and monitor such sets [3, 102]. We refer to this scenario as PAGEOFFSET. Conversely, if the attacker has no information about the target sets, they must construct eviction sets for all cache sets in the system and monitor them [3, 102]. We refer to this scenario as WHOLESYS. Considering the standard 4 kB page size and 64 B cache line size, the attacker in the WHOLESYS scenario needs to build and monitor 64× as many eviction sets as in the PAGEOFFSET scenario. For a 28-slice Skylake-SP CPU, the attacker needs to build $U_{LLC} = 896$ eviction sets for the LLC sets at a give page offset and $U_{LLC} \times 64 = 57{,}344$ eviction sets for all LLC sets in the system.

Bulk Eviction Set Construction. The process of constructing eviction sets for the PAGEOFFSET or WHOLESYS scenarios is based on the procedure to build a single eviction set. Because one can construct eviction sets for the WHOLESYS scenario by repeating the process for the PAGEOFFSET scenario at each possible page offset, we only explain the generation of eviction sets for the PAGE-OFFSET scenario.

First, we build a candidate set containing addresses with the target page offset. The candidate set needs to contain enough congruent addresses for *any cache set* at that page offset. Second, we pick and remove one address from the candidate set and use it as the target address T_a. Third, we use either of the address pruning algorithms in Section 4.2.1 to build an eviction set for the cache set to which T_a maps. The constructed eviction set is removed from the candidate set and saved to a list L containing all the eviction sets that have been built so far. Fourth, we pick and remove another address A from the reduced candidate set. If A cannot be evicted by any eviction set in L, we use A as the target address T_a and proceed to the third step to construct a new eviction set; otherwise, we discard A and repeat the fourth step. We stop when either we run out of candidate addresses or enough eviction sets have been built.

Table 4.1: Parameters of the Skylake-SP cache hierarchy.

Structure	**Parameters**
L1	Data/Instruction: 32 kB, 8 ways, 64 sets, 64 B line
L2	1 MB, 16 ways, 1,024 sets, non-inclusive to L1
LLC Slice	1.375 MB, 11 ways, 2,048 sets, non-inclusive to L1/L2
SF Slice	12 ways, 2,048 sets
Num. Slices	Up to 28 slices. A 28-slice LLC and SF is the most common configuration in Cloud Run datacenters

4.2.2 Non-Inclusive LLC in Intel Server CPUs

Beginning with the Skylake-SP microarchitecture [171], Intel adopted a non-inclusive LLC design on their server platforms. Under this design, cache lines in private caches may or may not exist in the LLC. The Snoop Filter (SF) [171] tracks the ownership of cache lines present only in private caches, serving as a coherence directory for such cache lines. Similar to the LLC, the SF is shared among cores and sliced. The SF has the same number of sets, number of slices, and slice hash function as the LLC. Therefore, if two addresses map to the same LLC set, they also map to the same SF set.

The interactions among private caches, SF, and LLC are complex and undocumented. Based on prior work [5] and our reverse engineering, we provide a brief overview of these interactions, acknowledging that our descriptions may not be entirely accurate or exhaustive.

Lines that are in state Exclusive (E) or Modified (M) in one of the private caches are tracked by the SF; we call these lines *private*. Lines that are in state Shared (S) in at least one of the private caches are tracked by the LLC (and, therefore, are also cached in the LLC); we call these lines *shared*.

When an SF entry is evicted, the corresponding line in the private cache is also evicted. The evicted line may be inserted into the LLC depending on a reuse predictor [172, 173]. When a line cached in the LLC needs to transition to state E or M due to an access, it is removed from the LLC and an SF entry is allocated to track it. When a private line transitions to state S, it is inserted into the LLC, and its SF entry is freed.

In this chapter, we use the FaaS Google Cloud Run platform [30]. In our experiments, we find that the CPU microarchitecture used in Cloud Run datacenters is dominated by Intel Skylake-SP and Cascade Lake-SP. Since these two microarchitectures have similar cache hierarchies, we focus our discussion on Skylake-SP. Table 4.1 lists the key parameters of Skylake-SP's cache hierarchy.

4.3 THREAT MODEL

In this chapter, we assume an attacker who aims to exfiltrate sensitive information from a victim containerized service running on a public FaaS platform such as Cloud Run [30] through LLC side channels. In Chapter 3, we have demonstrated how an attacker can co-locate with a target victim container on Cloud Run. If the victim runs container instances on multiple hosts, our techniques can co-locate attacker containers with a large portion of the victim instances. Therefore, we assume the co-location step is completed and focus on Steps 2–4 from Table 2.1.

We assume that the attacker is an unprivileged user of a FaaS platform. The attacker's interaction with the FaaS platform is limited to the standard interfaces that are available to all platform users. Using these interfaces, the attacker can deploy services that contain attacker-controlled binaries. Finally, we assume that the attacker can trigger the victim's execution by sending requests to the victim service, either directly or through interaction with a public web application that the victim service is part of.

Since cloud vendors typically prevent different users from simultaneously using the same physical core via Simultaneous Multithreading (SMT) [36, 100], the attacker must perform a *cross-core* cache attack. Similar to prior work [5, 159] that targets Intel Skylake-SP, we create eviction sets for the SF and monitor the SF for the victim's accesses. Note that an SF eviction set is also an LLC eviction set, as the SF and LLC share the same set mapping and the SF has more ways.

Lastly, we found that user containers on Cloud Run are unable to allocate huge pages. Therefore, we assume that the attacker can only rely on the standard 4 kB pages to construct eviction sets. This assumption is consistent with other restricted execution environments [102, 133, 174, 175] and places fewer requirements on the attacker's capability.

4.4 EXISTING EVICTION SETS FAIL IN THE CLOUD

In this section, we show that existing algorithms to construct eviction sets fail in the cloud. This is because of the noise in the environment and the reduced time window available to construct the eviction sets. In the following, we first examine the resilience to environmental noise of a core primitive used by all address pruning algorithms (Section 4.4.1). Then, we evaluate the success rate and execution time of the two state-of-the-art address pruning algorithms on Cloud Run (Section 4.4.2), and investigate the reasons why they fall short in the cloud (Section 4.4.3).

4.4.1 TestEviction Primitive & Its Noise Susceptibility

All the address pruning algorithms require a primitive that tests whether a target cache line is evicted from the target cache after a set of candidate addresses are accessed [3, 5, 47, 133]. We refer to this generic primitive as *TestEviction*. Specifically, group testing uses *TestEviction* to prune away non-congruent addresses, while Prime+Scope employs it to identify congruent addresses.

Due to environmental noise, *TestEviction* can return *false-positive* results—i.e., the target cache line is evicted by accesses from other tenants and not by the accesses to the candidate addresses. When this occurs in group testing, the algorithm may discard a group of addresses with congruent addresses, falsely believing that the remaining addresses contain enough congruent addresses. Similarly, Prime+Scope can misidentify a non-congruent address as a congruent one, incorrectly including it in the eviction set. In both cases, the algorithms may fail to construct an eviction set.

In general, the longer the execution time of *TestEviction* is, the more susceptible it becomes to noise, due to the increased likelihood of the target cache set being accessed by other tenants during its execution. Thus, the execution time of *TestEviction* not only affects the end-to-end execution time of the algorithm, but also the algorithm's resilience to noise.

Prior work [133] that uses the group testing algorithm implements *TestEviction* with linked-list traversal [176]. As this implementation serializes memory accesses to candidate addresses, we refer to this type of implementation as *sequential TestEviction*. Prime+Scope also uses sequential *TestEviction*, as it tests whether a target line is still cached after *each* access to a candidate address. Since sequential *TestEviction* does not exploit memory-level parallelism (MLP), it has a long execution time.

In our work, we find that overlapping accesses to candidate addresses to exploit MLP can significantly reduce the execution time of *TestEviction*. We refer to this implementation as *parallel TestEviction*. It is based on a pattern proposed by Gruss et al. [174], and our implementation can be found online [177]. However, as will be shown in Sections 4.4.2 and 4.4.3, even though parallel *TestEviction* is significantly faster than sequential *TestEviction*, it alone is not enough to overcome the noise in the cloud. In the rest of this chapter, we use parallel *TestEviction* in all algorithms except for Prime+Scope, which is incompatible with parallel *TestEviction* due to its algorithm design.

4.4.2 Noise Resilience of Existing Algorithms

In this section, we implement both the group testing and Prime+Scope algorithms for Skylake-SP's SF. We then evaluate their success rates and execution times in a local environment with minimal noise, as well as in the Cloud Run environment, which features a significant level of environmental noise from other tenants.

Implementation. Following prior work that builds SF eviction sets [5, 159], we first construct a minimal LLC eviction set comprising 11 congruent addresses, and then expand it to an SF eviction set by finding one additional congruent address. To insert cache lines into the LLC, we use a helper thread running on a different physical core that repeats the accesses made by the main thread. These repeated accesses turn the state of the cache lines to *S*, and thus cause the lines to be stored in the LLC (Section 4.2.2). Similar techniques are used in prior work [5, 159]. Finally, as per Section 4.4.1, our group testing implementation uses parallel $TestEviction$, while our Prime+Scope implementation uses sequential $TestEviction$.

To ensure a fair comparison among algorithms, we re-implement group testing and Prime+Scope using the same data structures to store candidate sets and eviction sets, and the same primitives to test whether a set of addresses is an eviction set. We call these algorithms GT and PS, respectively. In addition, we also implement optimized versions of these algorithms for Skylake-SP. We call these optimized algorithms GTOP and PSOP.

Optimizations to group testing. In the baseline group testing algorithm [133], the candidate set is divided into *G* similarly-sized groups. The algorithm temporarily withholds a group and tests whether the remaining addresses can still evict a target address T_a (Section 4.2.1). Once such a removable group is found, the algorithm prunes the identified group, stops searching the remaining groups, and re-partitions the remaining candidates into *G smaller* groups. This approach is called early termination. Prior work [167] suggests that this approach improves performance. However, we find that an alternative approach that continues to search the remaining groups without early termination offers better performance and higher success rate on Skylake-SP. We believe that not performing early termination helps the algorithm to prune away addresses more quickly by pruning groups of larger sizes. We call the alternative approach without early termination GTOP.

We also implement a group testing variant suggested by Song et al. [167]. This variant randomly withholds n/W candidate addresses and tests whether they are removable, where n is the remaining number of addresses in the candidate set. Based on our evaluation, this variant has similar performance and success rate as GTOP in both local and Cloud Run environments. Therefore, we omit the discussion of this variant in the chapter.

Optimizations to Prime+Scope. In the baseline Prime+Scope [159], when the algorithm finds a congruent address, the address is removed from the candidate set. This depletes the congruent addresses that are near the head of the candidate list, causing the algorithm to search deep into the list. In PSOP, we gradually move candidate addresses from the back of the list to the a near front position after we identify a congruent address. This optimization "recharges" the front of the list with more congruent addresses and reduces the number of candidates being checked.

Experiment setup. We evaluate these algorithms in both a cloud setup and a local setup.

Cloud setup. We deploy our attacker service to the *us-central1* data center, where we observe the largest Cloud Run cluster. Since our setup requires a concurrently running helper thread, each attacker instance requests 2 physical cores. In *us-central1*, the predominant CPU model used by Cloud Run is the Intel Xeon Platinum 8173M, which is a Skylake-SP processor with 28 LLC/SF slices.

During each experiment, we launch 300 attacker instances and retain only one per host. We then use each algorithm to build SF eviction sets for 50 random cache sets. To measure the effects of environmental noise fluctuations due to computation demand changes, we repeat our experiments for five days and at four different periods each day, namely, morning (9–11am), afternoon (3–5pm), evening (8–10pm), and early morning (3–5am). Altogether, we conducted 1,767 experiments on Cloud Run, totaling 88,350 eviction set constructions for each algorithm.

Local setup. Our local setup uses a Skylake-SP processor with the Intel Xeon Gold 6152, which has 22 LLC/SF slices. During the experiment, the system has minimal activity beyond the running attacker container instance. We employ each algorithm to construct 1,000 SF eviction sets.

Algorithms. For each SF eviction set, we allow each algorithm to make at most 10 construction attempts. If the algorithm fails these many times or it takes more than 1,000 ms to complete, we declare its failure. For group testing, which uses backtracking to recover from erroneous $TestEviction$ results, we permit at most 20 backtracks per attempt.

In our experiments, we need to start by generating a set of candidate addresses for a given page offset. Empirically, we find that a set with $3UW$ candidate addresses is enough for Skylake-SP's LLC/SF, where U and W are the cache uncertainty (Section 4.2.1) and associativity, respectively.

Results. Table 4.2 shows the effectiveness of the state-of-the-art algorithms to construct an eviction set for SF in different environments: quiescent local, Cloud Run, and Cloud Run from 3am to 5am, which are typically considered "quiet hours". The metrics shown are the success rate, average execution time, standard deviation of execution time, and median execution time. The success rate is the probability of successfully constructing an SF eviction set. The execution time measures the real-world time that it takes to reduce a candidate set to an LLC eviction set and then extend it with one additional congruent address to form an SF eviction set.

We see that all algorithms achieve very high success rates and good performance in the quiescent local environment. However, on Cloud Run, where there is substantial environmental noise from other tenants, all algorithms suffer significant degradation in both success rate and performance. Moreover, we do not observe significant variations in success rate or execution time across different periods of a day, including the 3am to 5am quiet hours. We believe this could be due to certain server consolidation mechanisms that adjust the number of active hosts based on demand [178, 179, 180], leading to a relatively constant load level on active hosts throughout the day.

Table 4.2: Effectiveness of the state-of-the-art address pruning algorithms in different environments. The metrics shown are: success rate, average execution time, standard deviation of execution time, and median execution time.

Env.	Metrics	GT	GTOP	PS	PSOP
Quiescent Local	Succ. Rate	97.0%	98.8%	98.5%	98.2%
	Avg. Time	32.9 ms	21.1 ms	55.9 ms	54.9 ms
	Stddev Time	72 ms	35 ms	166 ms	156 ms
	Med. Time	18.5 ms	13.7 ms	23.8 ms	21.7 ms
Cloud Run	Succ. Rate	39.4%	56.0%	3.2%	6.9%
	Avg. Time	714 ms	512 ms	580 ms	572 ms
	Stddev Time	476 ms	457 ms	329 ms	331 ms
	Med. Time	1,015 ms	384 ms	504 ms	495 ms
Cloud Run (3-5am)	Succ. Rate	41.4%	57.2%	3.7%	7.6%
	Avg. Time	693 ms	499 ms	581 ms	576 ms
	Stddev Time	482 ms	456 ms	327 ms	332 ms
	Med. Time	1,009 ms	350 ms	509 ms	502 ms

Implications. As discussed in Section 4.2.1, an unprivileged attacker needs to construct eviction sets for all SF sets at a given page offset (PAGEOFFSET) or in the system (WHOLESYS). We estimate the time to construct many eviction sets as $n_{sets} \times t_{avg}/SR$, where n_{sets} is the number of eviction sets we need to build, t_{avg} is the average execution time of attempting to construct one eviction set, and SR is the success rate. Similar metrics are also used in prior work [167].

For the Skylake-SP processor that we are targeting, the attacker needs to build 896 and 57,344 SF eviction sets in the PAGEOFFSET and WHOLESYS scenarios respectively (Section 4.2.1). Hence, on Cloud Run, GTOP, the fastest and most noise-resilient of the evaluated algorithms, would take 13.7 minutes and 14.6 hours to construct eviction sets required in the PAGEOFFSET and WHOLESYS scenarios, respectively.

We performed two additional small-scale experiments to validate our estimation. In the first experiment, which is conducted on 95 hosts, GTOP attempts to construct the 896 eviction sets required in the PAGEOFFSET scenario. GTOP takes, on average, 9.9 minutes to complete the task, and it only succeeds in 37.3% of the sets. In the second experiment, which is conducted on 69 hosts, GTOP tries to construct the 57,344 eviction sets required in the WHOLESYS scenario. Due to the timeout constraint of Cloud Run [181], we can only run GTOP for one hour and thus report the number of eviction sets it constructs under the constraint. Our best outcome is constructing 3,741 eviction sets in one hour, with an average number of 1,074 sets in one hour. This means that building eviction sets for the system's 57,344 SF sets would take GTOP over 57,344/3,741 $\approx$ 15 hours even in the best case.

This performance is unsatisfactory for a practical attack on FaaS platforms for several reasons. First, on some popular FaaS platforms [29, 31], the attacker can only execute for 10 to 15 minutes before timeout [65, 66]. Even on a more permissive platform like Cloud Run, the maximum timeout is just one hour [181]. After a timeout, the attacker might *not* reconnect to the same instance [181], thus losing the attack progress. Second, container instances usually have a short lifetime before being terminated [161, 162]. Hence, the long eviction set construction time means that the co-located victim instance may get terminated before eviction sets are ready. Finally, as FaaS platforms charge customers by the CPU time, the long execution time can cause significant financial cost to the attacker. This is especially the case if the attacker is launching many attacker instances on different hosts to increase the chance of a successful attack.

4.4.3 Explaining the Results

Compared to a quiescent local environment, we find that the cloud environment has a drastically higher rate of LLC accesses made by other tenants, and that *TestEviction*'s execution is slower. These two factors contribute to why the state-of-the-art algorithms are ineffective in a cloud environment. Our conclusion is based on the following two experiments. To ensure meaningful comparisons, both experiments are conducted using the container instances of Section 4.4.2.

Experiment 1: LLC set access frequency. In this experiment, we measure how frequently an LLC set is accessed by background activities, such as system processes and processes of other tenants. The reason why we focus on the access frequency of the LLC instead of the SF is because address pruning algorithms build eviction sets in the LLC and then expand them to SF eviction sets (Section 4.4.2).

During the experiment, we first construct an eviction set for a randomly chosen LLC set. Then, we detect background LLC accesses with Prime+Probe [1]. We record the timestamp of each LLC access. Each experiment trial collects the timestamps of 1,000 back-to-back LLC accesses. On Cloud Run, we perform 50 trials per host (88,350 trials in total). In the local environment, we carry out 1,000 trials.

Experiment 2: TestEviction execution duration. In this experiment, we measure the execution duration of both the parallel and sequential *TestEviction* when testing varying numbers of candidate addresses. We perform the measurement in both the Cloud Run and local environments. For each host and candidate set size, we measure *TestEviction*'s execution time for 100 times after 10 warm-ups.

Results. Figure 4.2 shows the cumulative distribution function (CDF) of the time between LLC accesses by background activity to a randomly chosen LLC set in both environments. On Cloud Run, the average LLC access rate is 11.5 accesses per millisecond per set. In the local environ-

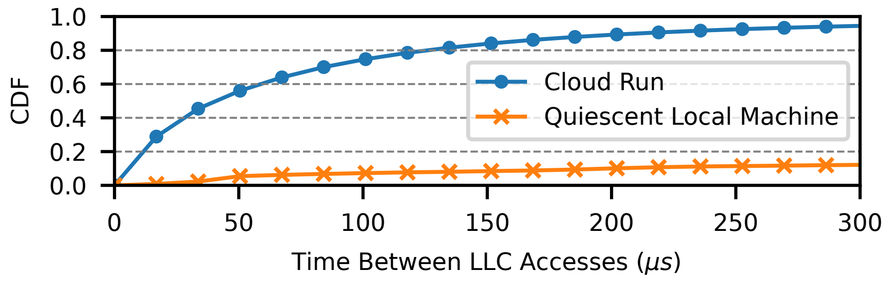


Figure 4.2: CDF of the time between accesses by background activity to a randomly chosen LLC set.

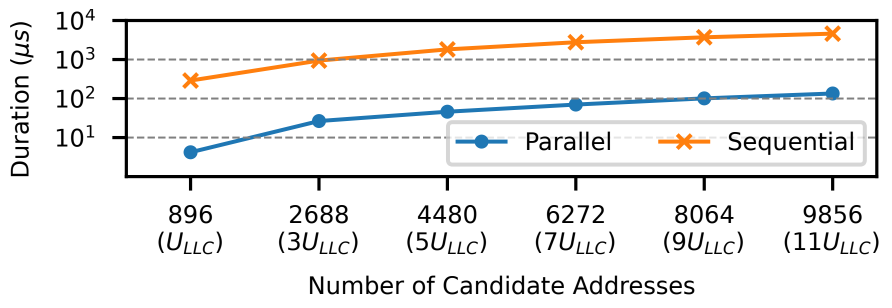


Figure 4.3: Different *TestEviction*'s execution times on Cloud Run under various number of candidate addresses.

ment with minimal noise, the average access rate is merely 0.29 accesses per millisecond per set. Figure 4.3 shows the execution time of the parallel and sequential *TestEviction*, for varying numbers of candidate addresses on Cloud Run. As the execution times of *TestEviction* in the local environment follow similar trends, we omit them in the plot. It can shown that, on average, the execution times of the sequential and parallel *TestEviction* are 26.9% and 42.1% lower in the local environment compared to Cloud Run, respectively.

These results explain why existing address pruning algorithms show unsatisfactory effectiveness on Cloud Run. For Prime+Scope, when using sequential *TestEviction* to identify the first congruent address, it is expected to test $11U_{LLC}$ candidate addresses. This takes approximately 4.6 ms on average. However, during this time, the target LLC set is expected to experience 53.0 background LLC accesses. Consequently, Prime+Scope's *TestEviction* very likely reports erroneous results under this level of noise.

As for group testing, its parallel *TestEviction* executes an order of magnitude faster than the sequential *TestEviction*. For example, it takes only 134.8 µs to test $11U_{LLC}$ candidates. Given the background LLC access rate, the probability of the set *not* being accessed during the *TestEviction* execution is about 18.4%. This permits the parallel *TestEviction* a reasonable chance to complete without experiencing interference from background accesses. In combination with the backtracking mechanism [133], group testing has a substantially higher probability of success compared to

Prime+Scope on Cloud Run. Still, both GT and GTOP experience a large number of backtracks due to erroneous $TestEviction$ results and are drastically slowed down on Cloud Run. For example, the optimized GTOP performs an average number of 32.2 backtracks per eviction set on Cloud Run, while it only needs 4.0 backtracks on average in the local environment.

4.5 CONSTRUCTING EVICTION SETS IN THE CLOUD

Based on the insights from Section 4.4, we propose two techniques that enable *fast* (and therefore also noise-resilient), eviction set construction in the cloud: *L2-driven candidate address filtering* (Section 4.5.1) and a *Binary Search-based* algorithm for address pruning (Section 4.5.2).

4.5.1 L2-driven Candidate Address Filtering

To speed-up the eviction set construction, we propose to reduce the candidate set size with an algorithm-agnostic optimization that we call *candidate address filtering*. Our insight is that the L2 set index bits are typically a subset of the LLC/SF set index bits. For example, Skylake-SP uses PA bits 15-6 as the L2 set index and PA bits 16-6 as the LLC/SF set index (Figure 4.1). Hence, if addresses A and B are not congruent in the L2, then A and B have different PA bits 15-6 and, therefore, they must *not* be congruent in the LLC/SF.

Based on this insight, we introduce a new *candidate filtering step* after candidate set construction and before address pruning. Assume that we want to construct an eviction set for an LLC/SF set to which an attacker-accessible address T_a maps. To perform the candidate filtering, we first construct an L2 eviction set for T_a. Then, using the L2 eviction set, we test whether it can evict each address from the candidate set. If a candidate address A cannot be evicted by the L2 eviction set, then it implies that A and T_a are not congruent in either the L2 or the LLC/SF. Consequently, A is removed from the candidate set. After candidate filtering, the candidate set contains only addresses that are congruent with T_a in the L2. These filtered addresses are passed to the address pruning algorithm to find a minimal LLC/SF eviction set.

As Skylake-SP has an L2 uncertainty of $U_{L2} = 16$, only about $1/16$ of the candidate addresses are congruent with T_a in L2. Therefore, the size of the filtered candidate set is only about $1/16$ of the original set size. On a common 28-slice Skylake-SP CPU, we expect to find one congruent address every $U_{LLC} = 896$ candidates in the candidate set before filtering. With candidate filtering, we now expect to find one congruent address every $896/16 = 56$ candidates.

Since the candidate set is universally used by different address pruning algorithms, including both group testing and Prime+Scope, our candidate filtering is a generic optimization. Moreover, in modern processors, the number of L2 sets is typically smaller than the number of LLC sets in one LLC slice. Hence, the property that the L2 set index bits are a subset of the LLC set index bits generally holds for other processors as well, Therefore, the candidate filtering optimization also applies to them. The idea of candidate filtering can also be applied to a more restricted environment where the attacker cannot even control the page offset bits [133]. In such an environment, the attacker can hierarchically construct L1 and L2 eviction sets to gradually filter candidates for the next lower cache level. Finally, the candidate filtering optimization cannot be applied to building eviction sets for randomized LLCs, as whether two addresses conflict in a randomized LLC is independent of whether they conflict in the L2.

4.5.2 Using Binary Search for Address Pruning

To further speed-up eviction set construction in the cloud, we propose a new address pruning algorithm based on binary search. Our algorithm uses *parallel* $\mathit{TestEviction}$.

Algorithm design. Given a list of candidate addresses, we test whether the first n addresses can evict a target address T_a. For a W-way cache, increasing n from zero will result in a negative test outcome until the first n addresses include W congruent addresses. We define the *tipping point*, denoted by τ, as the smallest n for which the first n addresses evict T_a. Therefore, τ is the index of the W-th congruent address in the list, assuming that the indexation begins from 1. For a given n, if the first n addresses evict T_a, it means that $n \geq \tau$; otherwise, $n < \tau$. Our main idea is to use binary search to efficiently determine τ and thus identify one congruent address. Then, we exclude the congruent address from any future search, and repeat the binary search process until W different congruent addresses are found.

Figure 4.4 shows the pseudo code of the algorithm. It takes as inputs a target address T_a, an array of addresses $addrs$ representing the candidate set, and the array size N. The array $addrs$ should contain at least W congruent addresses, and thus $N \geq W$. The algorithm iteratively finds W congruent addresses by finding the tipping point at each iteration (Lines 5–16 in Figure 4.4). Within each iteration, the algorithm tests in a loop if the first $n = \lfloor (LB + UB)/2 \rfloor$ addresses from $addrs$ can evict T_a (Line 9). The variables LB and UB are then updated in a manner that LB always represents the largest n such that the first n addresses *cannot* evict T_a and UB always represents the smallest n such that the first n addresses *can* evict T_a. Therefore, when $UB = LB + 1$, UB is the tipping point of iteration i, denoted by τ_i. Consequently, the τ_i-th address of the array is a congruent address. The algorithm then swaps the just-found congruent address with the i-th address in $addrs$ and proceeds to the next iteration (Line 15).

Before the binary search in the next iteration starts, LB is reset to $i - 1$ (Line 6), as the first $i - 1$ addresses are the congruent addresses found in previous iterations and are thus excluded from the search. In contrast, UB needs *not* to be reset to N, as the first UB addresses always contain

```
1  // T_a: target address
2  // addrs: an array of candidate addresses
3  // N: size of the addrs array (N >= W)
4  size_t LB, UB = N;
5  for (size_t i = 1; i <= W; i++) {
6     LB = i - 1;
7     while (UB - LB != 1) {
8        n = (LB + UB) / 2;
9        if (TestEviction(T_a, addrs, n))
10          UB = n; // T_a can be evicted
11       else
12          LB = n; // T_a cannot be evicted
13    }
14    size_t tau_i = UB;
15    swap(addrs[i], addrs[tau_i]);
16 } // addrs[1]~addrs[W] form an eviction set
```

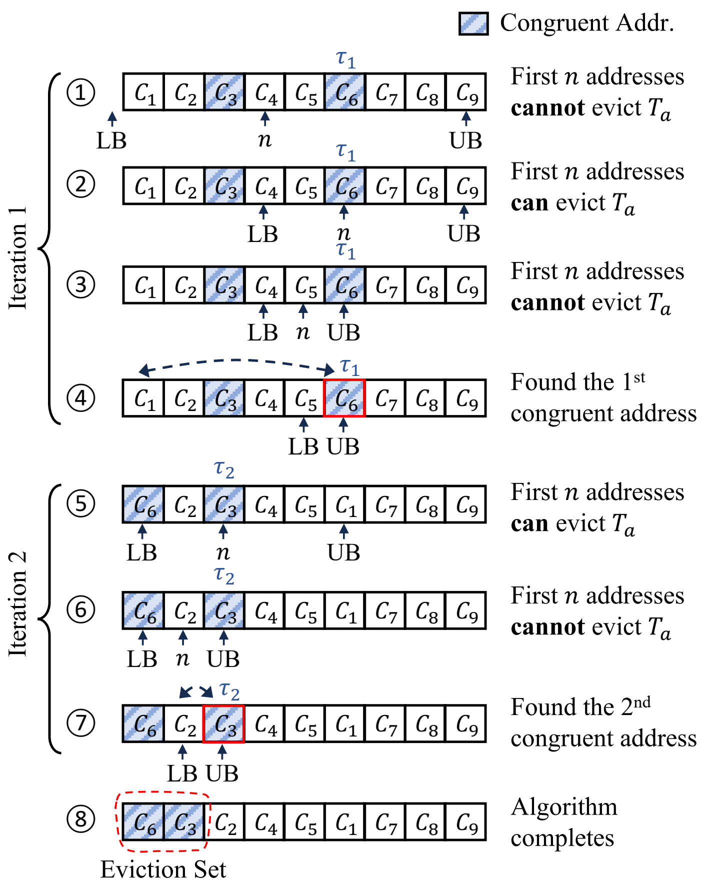
Figure 4.4: Pseudo code of our proposed algorithm. All array indexes start from 1. Parallel $TestEviction(T_a, addrs, n)$ returns a boolean value that indicates whether the first n candidate addresses from array $addrs$ can evict the target T_a.

W congruent addresses due to the swapping. Finally, after W iterations, the first W addresses in $addrs$ form a minimal eviction set for T_a (Line 16).

Example. Figure 4.5 demonstrates the algorithm with a nine-address candidate set $(C_1, C_2, \dots, C_9)$ and a target cache with associativity $W = 2$. Initially, we set $i = 1$, $LB = 0$, $UB = N = 9$, and $n = \lfloor (UB + LB)/2 \rfloor = 4$ (Step ①). Because the first $n = 4$ addresses cannot evict T_a, we set $LB = n = 4$ and update n to $\lfloor (UB + LB)/2 \rfloor = 6$ (Step ②). With the updated n, the first $n = 6$ addresses now can evict T_a, so we set $UB = n = 6$ and update $n = 5$ (Step ③). This process is repeated until $UB = LB + 1 = 6$ (Step ④). At this point, C_6 is found to be a congruent address, which is saved to the front of the list by swapping it with C_1. Then, we increment i to 2, set $LB = i - 1 = 1$ without changing UB (Step ⑤), and repeat the binary search (Steps ⑤–⑦). The algorithm finishes once W congruent addresses are found (Step ⑧), which form a minimal eviction set for T_a.

Backtracking mechanism. When the $TestEviction$ returns a false-positive result due to environmental noise, our algorithm can incorrectly set UB to a value smaller than τ. As a result, the binary search may incorrectly identify a non-congruent address as a congruent one. This erroneous state is detected if the first UB addresses cannot evict T_a after the binary search for the iteration finishes. To recover from this state, we gradually increase UB with a large stride until the first UB addresses can evict T_a and restart the binary search.

Comparison to existing algorithms. Unlike Prime+Scope, our algorithm uses parallel $TestEviction$. As discussed in Section 4.4.3, parallel $TestEviction$ is at least an order of magnitude faster than



Congruent Addr.
Iteration 1
Iteration 2
① First n addresses **cannot** evict T_a
② First n addresses **can** evict T_a
③ First n addresses **cannot** evict T_a
④ Found the 1st congruent address
⑤ First n addresses **can** evict T_a
⑥ First n addresses **cannot** evict T_a
⑦ Found the 2nd congruent address
⑧ Algorithm completes
LB
n
UB
τ_1
τ_2
C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9
Eviction Set

Figure 4.5: Illustration of our proposed binary search-based algorithm (assuming $W = 2$). Blocks with shaded pattern represent congruent candidate addresses.

sequential $TestEviction$. Therefore, our algorithm is faster than Prime+Scope.

Compared to group testing, both our algorithm and group testing can use parallel $TestEviction$. Assume that we use the number of memory accesses as a proxy for execution time. Using our algorithm, it takes $O(\log N)$ parallel $TestEviction$ executions to find a tipping point, where N is the candidate set size. Since each parallel $TestEviction$ needs to make $O(N)$ memory accesses, it takes $O(N \log N)$ accesses to find one congruent address. As we need to find W congruent addresses, the end-to-end execution requires $O(WN \log N)$ accesses. In contrast, group testing requires $O(W^2N)$ accesses. Therefore, whether group testing or our algorithm makes fewer accesses, and consequently executes faster, depends on the specific values of W and $\log N$.

As an intuitive comparison, the ratio of the number of accesses made by group testing over our algorithm is estimated by $O(W/\log N)$. Since we use $N = 3UW$ (Section 4.4.2), we rewrite the ratio as $O(W/\log(UW))$. This suggests that in caches with high associativity (i.e., a large W), group testing *tends* to make more accesses than our algorithm. This is supported by our experiments in Section 4.5.3.

4.5.3 Evaluating Our Optimizations

We evaluate group testing, Prime+Scope, and our binary search-based algorithm with candidate filtering in both the Cloud Run and local environments. We use the same methodology as the experiment in Section 4.4.2, except for reducing the time limit of constructing one eviction set to 100 ms (because of candidate filtering). Each algorithm is evaluated in three scenarios: (1) SINGLESET, where we construct a single eviction set for a randomly chosen SF set; (2) PAGEOFFSET, where we construct eviction sets for all SF sets at a randomly chosen page offset; and (3) WHOLESYS, where we construct eviction sets for all SF sets in the system. Our experiments include 88,350 and 1,000 measurements per algorithm in the cloud and local environments, respectively, in SINGLESET; 8,835 and 100 in PAGEOFFSET; and 1,767 and 20 in WHOLESYS.

Table 4.3: Eviction set construction effectiveness of various algorithms under SINGLESET.

Env.	**Metrics**	**SINGLESET** # Ev sets: Local=1, Cloud=1			
		GT	**GTOP**	**PSBST**	**BINS**
Quiescent Local	Succ. Rate	99.3%	99.5%	99.2%	99.9%
	Avg. Time	15.2 ms	14.7 ms	14.7 ms	14.1 ms
	Stddev Time	3.1 ms	2.6 ms	0.8 ms	2.2 ms
	Med. Time	14.7 ms	14.4 ms	14.5 ms	13.9 ms
Cloud Run	Succ. Rate	96.7%	97.7%	97.2%	98.1%
	Avg. Time	28.8 ms	27.2 ms	33.2 ms	26.6 ms
	Stddev Time	14.4 ms	10.8 ms	21.4 ms	11.6 ms
	Med. Time	25.1 ms	24.7 ms	26.7 ms	23.9 ms

Table 4.4: Eviction set construction effectiveness of various algorithms under PAGEOFFSET. The number of eviction sets may vary between local and cloud because the experiments use machines with different number of slices.

Env.	**Metrics**	**PAGEOFFSET** # Ev sets: Local=704, Cloud=896			
		GT	**GTOP**	**PSBST**	**BINS**
Quiescent Local	Succ. Rate	98.6%	99.2%	99.4%	99.5%
	Avg. Time	1.95 s	1.48 s	3.02 s	1.04 s
	Stddev Time	0.72 s	0.17 s	2.48 s	0.16 s
	Med. Time	1.77 s	1.43 s	1.39 s	1.00 s
Cloud Run	Succ. Rate	95.6%	97.4%	98.4%	98.0%
	Avg. Time	5.51 s	3.95 s	4.51 s	2.87 s
	Stddev Time	2.62 s	1.90 s	2.72 s	1.58 s
	Med. Time	4.94 s	3.52 s	3.85 s	2.53 s

Table 4.5: Eviction set construction effectiveness of various algorithms under WholeSys. The number of eviction sets may vary between local and cloud because the experiments use machines with different number of slices.

Env.	**Metrics**	**WholeSys** # Ev sets: Local=45,056, Cloud=57,344			
		Gt	**GtOp**	**PsBst**	**BinS**
Quiescent Local	Succ. Rate	99.0%	99.1%	99.5%	99.5%
	Avg. Time	103.6 s	79.6 s	175.0 s	50.1 s
	Stddev Time	16.1 s	7.9 s	72.7 s	5.5 s
	Med. Time	96.8 s	76.9 s	185.6 s	48.9 s
Cloud Run	Succ. Rate	88.1%	90.5%	91.7%	92.6%
	Avg. Time	301.1 s	212.6 s	244.4 s	142.4 s
	Stddev Time	63.0 s	52.1 s	58.9 s	34.8 s
	Med. Time	290.1 s	200.4 s	229.6 s	134.2 s

Results. Tables 4.3–4.5 list the success rate and execution time of each algorithm under different scenarios in both the Cloud Run and local environments. The execution time measures *both candidate filtering and address pruning*. As we find that Ps and PsOp have similar success rates and execution times after applying candidate filtering, Tables 4.3–4.5 only show the one with the shortest average execution time, and call it PsBst. We call our binary search-based algorithm BinS.

The SingleSet scenario in Table 4.3 is directly comparable to the scenario in Table 4.2. Table 4.3 shows the effectiveness of candidate filtering on Cloud Run, as it leads to significantly shortened execution times. For example, the average execution time of GtOp is reduced from 512 ms to 27.2 ms. The resulting success rate also increases substantially. Indeed, for GtOp, it goes from 56.0% to 97.7%.

Recall that the average execution time comprises both candidate filtering and addresses pruning. In the SingleSet scenario, it can be shown that candidate filtering on Cloud Run takes on average 22.3 ms, which dominates the execution time. As a result, the average execution times are similar across all algorithms. As will be shown in Section 4.5.3, the portion of the execution time spent on candidate filtering drastically decreases when building numerous eviction sets in the PageOffset and WholeSys scenarios.

Next, consider PageOffset (Table 4.4). All the algorithms experience increases in average execution times as they go from the local to the Cloud Run environments. Comparing group testing to our algorithm on Cloud Run, we see that Gt and GtOp take 92% and 38% more time to build eviction sets on average, as we find that Gt and GtOp make 162% and 52% more memory accesses than BinS. As for PsBst, it takes on average 57% more time than BinS, due to its use of the sequential *TestEviction*.

The results for WholeSys (Table 4.5) are qualitatively similar to PageOffset, except for larger drops in success rates as we go from the local to the Cloud Run environments. Still, while the average success rates of GT, GTOP, PSBST, and BINS on Cloud Run are 88.1%, 90.5%, 91.7%, and 92.6%, respectively, the medians are 96.7%, 98.5%, 99.4%, and 99.1%, respectively.

To summarize, the combination of candidate filtering and our binary search-based algorithm offers significant performance improvements over the *well-optimized* state-of-the-art algorithms. On Cloud Run, they reduce the time to construct eviction sets for all SF sets in the system from an expected duration of 14.6 hours (Section 4.4.2) to a mere 2.4 minutes (last column of Table 4.5), with a median success rate of 99.1%. These improvements make the LLC Prime+Probe attack in the cloud feasible.

Overhead of Candidate Filtering. As indicated before, it takes 22.3 ms to complete one candidate filtering on Cloud Run. This time includes constructing one L2 eviction set and using it to filter candidates. While this time dominates the execution time when constructing a *single* eviction set (Section 4.5.3), the same filtered candidates can be reused to construct many eviction sets for LLC/SF sets that are mapped to the same L2 set. For example, in the 28-slice Skylake-SP processor used in our Cloud Run evaluation, constructing the 896 LLC/SF sets in the PAGEOFFSET scenario requires only 16 candidate filtering executions, which takes 435 ms on average. This execution time makes up a small portion of the total execution time in PAGEOFFSET (2.87 s in Table 4.4).

In the WHOLESYS scenario, a naive process would build eviction sets for all 1,024 L2 sets and execute candidate filtering 1,024 times. We optimize the process by exploiting the following property of the L2: if addresses A and B are congruent in L2, then $A' = A + \delta$ and $B' = B + \delta$ are also congruent in L2—as long as the δ is small enough such that A and A' belong to the same page, and B and B' belong to the same page [3, 4, 102].

Exploiting this property, we first construct 16 eviction sets for all L2 sets at page offset `0x0`. Then, we use each eviction set to generate a filtered candidate set at page offset `0x0`. Finally, we can derive a new filtered candidate set at page offset δ by adding δ to each candidate address of the filtered candidate set at page offset `0x0`. As a result, the WHOLESYS scenario requires only 16 L2 eviction set constructions and candidate filtering executions. The time of completing candidate filtering (435 ms) is negligible compared to the total execution time in WHOLESYS (142.4 s in Table 4.5).

Other Intel Server Platforms and Target Caches. As discussed in Section 4.5.2, group testing tends to incur a higher execution overhead over our binary search-based algorithm when the cache associativity increases. To illustrate this trend, we measure the performance of eviction set construction on Ice Lake-SP, which features caches with higher associativity than in Skylake-SP. Specifically, Ice Lake-SP has a 16-way SF and a 20-way L2 cache, whereas Skylake-SP has a 12-way SF and a 16-way L2 cache.

Because we do not see Ice Lake-SP being used on Cloud Run, we measure the performance on local quiescent Skylake-SP and Ice Lake-SP machines. The Skylake-SP machine utilized is the same as in prior experiments. The Ice Lake-SP machine uses an Intel Xeon Gold 5320, which has 26 LLC/SF slices. For each machine and algorithm, we measure the time to construct a single SF or L2 eviction set 1000 times. Candidate filtering is enabled for SF eviction set construction, but its time is *not* included in our measurements.

First, we consider constructing eviction sets for the SF. Gt, GtOp, and BinS take, on average, 2.23 ms, 1.77 ms, and 1.17 ms, respectively, to construct a single eviction set for the 12-way SF of Skylake-SP. The same process takes Gt, GtOp, and BinS on average 3.81 ms, 3.07 ms, and 1.68 ms, respectively, for the 16-way SF of Ice Lake-SP. As we go from Skylake-SP to Ice Lake-SP, the ratio Gt/BinS and GtOp/BinS changes from 1.91 and 1.51 to 2.27 and 1.83, respectively.

Similarly, Gt, GtOp, and BinS take, on average, 2.49 ms, 1.90 ms, and 1.33 ms, respectively, to construct a single eviction set for the 16-way L2 of Skylake-SP. The same process takes Gt, GtOp, and BinS on average 14.48 ms, 8.16 ms, and 2.28 ms, respectively, for the 20-way L2 of Ice Lake-SP. As we go from Skylake-SP to Ice Lake-SP, the ratio Gt/BinS and GtOp/BinS changes from 1.87 and 1.43 to 6.35 and 3.58, respectively.

4.6 MONITORING MEMORY ACCESSES & IDENTIFYING TARGET CACHE SETS

Eviction set construction is the first step of an end-to-end LLC attack after co-location (Step 2 in Table 2.1). In this section, we improve the remaining steps with two new techniques. First, Section 4.6.1 introduces *Parallel Probing*, which enables the monitoring of victim memory accesses with high time resolution. This technique optimizes Step 3 (identify target sets) and Step 4 (exfiltrate information) in Table 2.1 for the noisy cloud environment. Second, Section 4.6.2 leverages *Power Spectral Density* [182] from signal processing to easily identify the victim's target cache set. This technique optimizes Step 3 in Table 2.1 for the noisy cloud environment.

4.6.1 Parallel Probing for Memory Access Monitoring

Given a cache set, the attacker can detect memory accesses to that set with Prime+Probe (Section 2.2.2). It is vital that *both* prime and probe latencies are short. A short probe latency enables the attacker to monitor when accesses occurs at a high time resolution [159]. A short prime latency allows the attacker to quickly prepare the monitored cache set for detecting the next access. In a noisy cloud environment, where a cache set may be frequently accessed by processes of other tenants, failure to prime the set in a timely manner can increase the chance of missing the victim's accesses.

To minimize the probe latency, Prime+Scope [159] primes a specific line from the eviction set to become the *eviction candidate (EVC)*, which is the line to be evicted when a new line needs to be inserted into the set. This method enables the attacker to check only if the EVC remains cached. Further, since the EVC can be cached in L1, the probe latency becomes minimal, leading to a high time resolution. However, this comes at the cost of using a slower and more complex priming pattern to prepare the replacement state [159], which can reduce monitoring effectiveness in a noisy environment.

Our solution. We discover that, due to the high memory-level parallelism supported by modern processors, simply probing *with overlapped accesses* all the W lines of a minimal eviction set (Section 4.2.1) results in a probe latency only slightly higher than that of Prime+Scope. The advantage of this *parallel probing* method is that it allows us to prime the cache set without preparing any replacement state. Therefore, parallel probing works irrespective of the replacement policy used by the target cache, which can be unknown or quite complex [133, 183, 184, 185].

Evaluating Parallel Probing. We conduct a covert-channel experiment similar to the one done by Purnal et al. [159] to evaluate two different Prime+Scope strategies and our parallel probing. In the experiment, we create a sender and a receiver thread that agree on a target SF set. The sender thread accesses the target set at a fixed time interval, while the receiver thread uses Prime+Scope or parallel probing to detect accesses to the target set. For a sender's access issued at time t, if the receiver detects an access at time $t' \in (t, t + \epsilon)$, where ϵ is an error bound, we say that the sender's access is detected by the receiver. We use $\epsilon = 500$ cycles (or 250 ns).

We conduct this experiment on Cloud Run with varying access intervals. In each experiment, the sender thread accesses the target SF set 2,000 times. We measure the percentage of the sender's accesses that are detected by the receiver—i.e., the *detection rate*. We also collect the probe and prime latencies and exclude outliers that are above 20,000 cycles, as an interrupt or context switch likely occurred during the operation. The experiment is done on different hosts on different days and at different times of day. We repeat the experiment 10 times on each host, totaling 4,070 measurements.

For Prime+Scope, we evaluate two prime strategies discussed by Purnal et al. [159]. The first strategy (PS-Flush) is to load, flush, and sequentially reload the eviction set. The second strategy (PS-Alt) is to perform an alternating pointer-chase using *two* eviction sets. More details of these strategies is found in [159]. For our parallel probing technique (Parallel), we use a prime strategy that simply traverses the eviction set 12 times with overlapped accesses.

Table 4.6 lists the prime and probe latencies of each strategy. The table reveals that the average probe latency of Parallel is only 24 cycles higher than that of Prime+Scope, yet Parallel exhibits a substantially lower prime latency.

Table 4.6: Prime and probe latencies of two Prime+Scope strategies and parallel probing on Cloud Run. The host processors' frequency is 2 GHz.

Strategy	**Prime Latency** (mean ± std. deviation)	**Probe Latency** (mean ± std. deviation)
PS-FLUSH	6,024 ± 990 cycles	94 ± 0.7 cycles
PS-ALT	2,777 ± 735 cycles	
PARALLEL	1,121 ± 448 cycles	118 ± 0.7 cycles

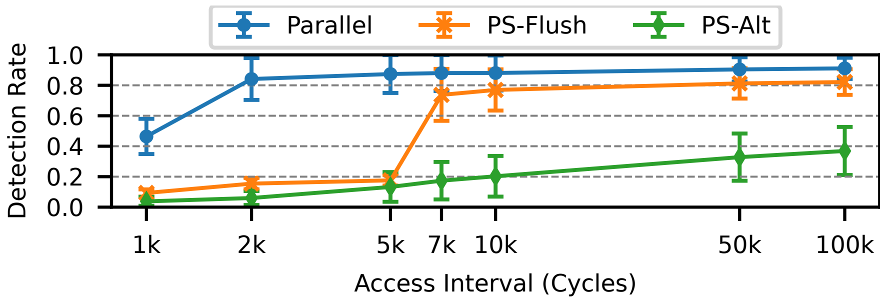


Figure 4.6: Detection rate of each monitoring strategy with various access interval. The x-axis employs a logarithmic scale. The error bars represent the standard deviations.

The benefit of this reduced prime latency is depicted in Figure 4.6, which shows the average detection rate for different access intervals. With a 2k-cycle access interval, PARALLEL achieves an average detection rate of 84.1%, while PS-FLUSH and PS-ALT reach average detection rates of 15.4% and 6.0%, respectively. The low detection rates of PS-FLUSH and PS-ALT are primarily due to their long prime latencies.

Even when the access interval is sufficiently long for all strategies to complete priming, PARALLEL still maintains the highest detection rate. With a 100k-cycle access interval, PARALLEL, PS-FLUSH, and PS-ALT attain average detection rates of 91.1%, 82.1%, and 36.9%, respectively. To understand why, we inspected a random subset of the detected memory access traces. In PS-FLUSH, we observe that missed detections mainly result from noisy accesses made by other tenants to the monitored cache set, occurring just before the sender's access. After the receiver detects the noisy access, it is unable to finish priming before the sender accesses the set.

In PS-ALT, although the receiver initially detects the sender's accesses, it often later fails to prime the monitored line as the EVC, leading to many missed detections. We believe this might be due to the SF replacement states being altered by background accesses, resulting in failing to prepare the EVC.

4.6.2 Power Spectral Density for Set Identification

To identify the target cache sets (STEP 3 in Table 2.1), the attacker can collect a short memory access trace from each potential target cache set *while the victim is executing*. The attacker then applies signal processing techniques to determine whether a given memory access trace has any characteristic that resembles what is expected from a given target cache set. Prior work has considered characteristics such as the number of accesses in the trace or the access pattern [3, 4]. These characteristics can be hard to identify in the cloud due to the high level of environmental noise.

Our solution. Our insight is that a victim program's accesses to the target cache set are often periodic in a way that the attacker expects, while this is not the case for the background accesses. Therefore, we propose to process the access traces in the frequency domain, where it is easier to spot the expected periodic patterns. Specifically, we estimate the *Power Spectral Density* (PSD) [182] of each memory access trace using Welch's method [168]. PSD measures the "strength" of the signal at different frequencies [182]. If the access trace is collected from the target set where the victim makes periodic accesses, we will observe peaks in the trace's PSD around the expected victim-access frequencies. If, instead, the trace is not collected from the target set, it will have a PSD without the expected peaks.

Example. To demonstrate our proposal, we collect an access trace from a target SF set of a victim program and another trace from a non-target SF set, and compare the PSD of both traces. In this example, the victim executes an ECDSA implementation [163] that will be described in Section 4.7.1. In this implementation, the victim processes each individual secret bit in a loop. The victim accesses the target SF set when an iteration starts and, if the secret bit being processed in the iteration is zero, it also accesses the set in the midpoint of the iteration. The execution of each iteration takes a mostly fixed time duration of about 9,700 cycles on a 2 GHz Skylake-SP machine on Cloud Run. Because of the access that may occur in the midpoint of an iteration, the victim's accesses to the target set have a period of about 4,850 cycles. Therefore, we expect to observe a peak in the PSD at the frequency of $f = 2\,\text{GHz}/4{,}850 \approx 0.41\,\text{MHz}$.

The top plot of Figure 4.7 shows two 100 µs memory access traces collected on a 2 GHz Skylake-SP machine on Cloud Run. The blue dots at the top are the observed accesses to the target SF set; while the orange dots at the bottom are the observed accesses to the non-target SF set. For both traces, we see similar numbers of accesses: 50 accesses to the target set and 48 to the non-target set. It is difficult to interpret these two patterns.

The bottom plots show the PSD of the access traces collected from the target set (left) and the non-target set (right). In the PSD for the target set, we clearly see a peak at the base frequency $f = 0.41\,\text{MHz}$ and at multiples of f. In contrast, in the PSD for the non-target set, we see no significant peaks at the expected frequency.

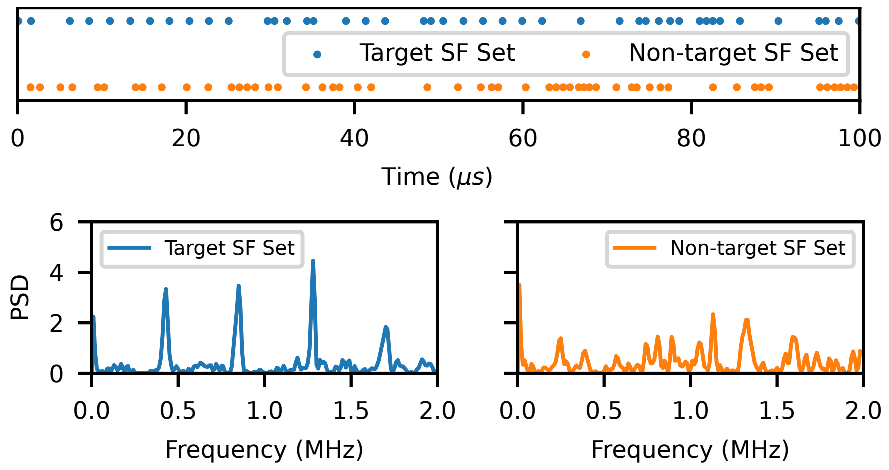


Figure 4.7: The top plot shows traces of memory access to the target SF set (top trace) and the non-target SF set (bottom trace) collected on Cloud Run. The two bottom plots show the power spectral density of the two traces.

4.7 DEMONSTRATING AN END-TO-END ATTACK

In this section, we demonstrate the combination of our techniques discussed in Sections 4.5 and 4.6 by mounting an end-to-end, cross-tenant attack in Cloud Run. Our demonstration uses a vulnerable implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) [186] from OpenSSL 1.0.1e [163] as an example victim. While this implementation is deprecated, we use it solely as a vehicle to illustrate our techniques.

4.7.1 Attack Outline

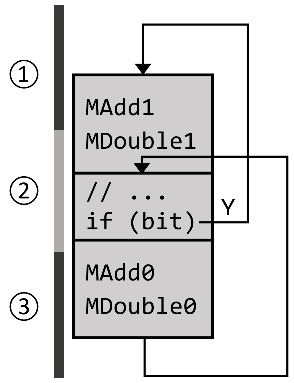
The vulnerable ECDSA implementation that we target uses the Montgomery ladder technique [187] to compute on the nonce k, an ephemeral key that changes with each signing. The attacker can derive the private key used for signing by extracting some bits of k across multiple signing operations [188, 189, 190, 191, 192, 193, 194]. Thus, the attacker's goal is to learn as many bits of k as possible. Our demonstration targets curve `sect571r1`, which uses a 571-bit nonce.

Similar to prior work [3, 15, 189], we assume the attacker knows the memory layout of the library used by the victim. This assumption generally holds, as victims often install and use libraries whose binaries are publicly released. Moreover, as we are targeting a victim web service (Section 4.3), we assume the library is loaded once at the victim container startup time and uses the same VA-PA mapping throughout the container's lifetime.

Figure 4.8a shows a simplified version of the Montgomery ladder implementation [163] that we are targeting. The code iterates through each bit of the nonce k and calls functions `MAdd` and `MDouble` with different arguments depending on the value of the bit. This implementation is re-

```
for bit in k {
  if (bit) {
    MAdd(x1,z1,x2,z2); // MAdd1
    MDouble(x2,z2); // MDouble1
  } else {
    MAdd(x2,z2,x1,z1); // MAdd0
    MDouble(x1,z1); // MDouble0
  }
  // ...
}
```

(a) Simplified code snippet.



(b) Memory layout.

Figure 4.8: Simplified vulnerable code snippet (left) and its memory layout in VA space (right). Each thick vertical line represents a cache line. The control-flow edge that exits the loop is omitted in the right figure.

silient to end-to-end timing, as it executes the same sequence of operations regardless of the bit value. However, it has secret-dependent control flow. Since each side of the branch resides on a different cache line, the program fetches different cache lines based on the value of the nonce bit. As a result, the attacker can infer each individual nonce bit by monitoring code fetch accesses to these cache lines tracked by the SF.

Figure 4.8b shows the memory layout of the vulnerable code snippet in VA space, compiled with the default build options and static linkage. Each thick vertical line represents a different cache line. Given this layout, one approach is to monitor accesses to cache line ②. Line ② is used by the `if` statement, which is executed at the beginning of an iteration. As a result, the code fetch accesses made by the `if` statement serve as a "clock" and mark the iteration boundaries.

Cache line ② is also utilized by the true direction of the branch. When the control flow takes the true direction and `MAdd1` is executing, Prime+Probe will evict line ②. As the control flow returns from `MAdd1` and is about to call `MDouble1`, the program needs to fetch line ②, creating one access in the midpoint of the iteration. Then, while `MDouble1` is executing, Prime+Probe evicts cache line ② again, triggering a code fetch access when returning from `MDouble1` and executing the `if` statement.

Therefore, we observe two accesses to line ② per iteration if the bit value is 1, and one access to line ② if the bit value is 0. It should be noted that, although line ② slightly overlaps with the beginning of the `else` block, we will *not* observe an extra access if the bit value is 0. This is because the overlapped region is executed immediately after the `if` statement, and the interval is too brief to be detected.

In practice, when we collect a trace of the memory accesses to the target SF set to which cache line ② maps, we also want to collect the ground truth of nonce bit *k* and iteration boundaries for

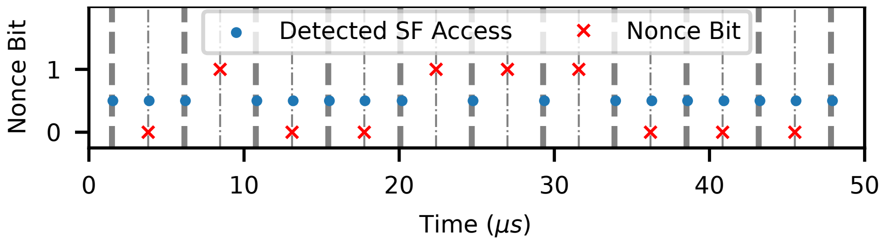


Figure 4.9: A snippet of memory accesses to the target SF set collected on Cloud Run. Dots are detected accesses, and crosses are the nonce bit k values (1 or 0).

validation purpose. This requires some slight instrumentation of the binary, a practice also seen in prior work [8, 16]. The instrumentation is purely for validation purpose and it is not necessary for the attack. However, due to the instrumentation, the layout of the code changes, and it is easier to monitor the cache line corresponding to the `else` direction. The reasoning is similar to the explanation for line ②, but we now observe the additional memory access at the midpoint of an iteration when the bit value is 0, not 1.

We collect the trace of memory accesses to the target SF set (using the techniques of Section 4.6), the ground truth of nonce bit k, and iteration boundaries on Cloud Run, while the victim code is executing. Figure 4.9 shows a short snippet of the trace that happens to contain no noisy accesses made by other tenants. In the figure, thick dashed vertical lines represent the ground truth for iteration boundaries, and thin dashed vertical lines represent halves of iterations. Dots are detected accesses, and crosses are the nonce bit k values (1 or 0). Iterations where bit k value is 0 have two accesses. From the trace, we can easily read the nonce bits.

It takes only about 9,700 cycles on Cloud Run to execute one iteration of the Montgomery ladder loop that we target. Thus, when the nonce bits have a sequence of continuous zeros, the attacker needs to detect a sequence of accesses that are 4,850 cycles apart. As shown in Table 4.6, the prime pattern of Prime+Scope's [159] PS-Flush takes on average 6,024 cycles to complete, while the PS-Alt pattern has a low detection rate (Figure 4.6). As a result, the Prime+Scope versions either frequently miss memory accesses or report an access as occurring at a time different from when the actual access occurs. In contrast, our *Parallel Probing* strategy takes on average only 1,121 cycles to execute (Table 4.6) and thus accurately detects the memory accesses in ECDSA.

4.7.2 Finding the Target Cache Set with PSD

We apply our PSD method to identify the victim's target SF set on Cloud Run. To obtain the ground truth, we run the victim and attacker programs in the same container. The attacker `mmaps` the victim program so that the attacker can access the target line. Then, when the attacker identifies an eviction set that might correspond to the target SF set, the attacker can validate it by checking whether the eviction set indeed evicts the target line.

Scanning strategy. Since the attacker knows the VA of the target cache line of the ECDSA victim, they only need to construct eviction sets for SF sets at the page offset of the target line and scan only those sets—i.e., it is the PAGEOFFSET scenario. To approximate the WHOLESYS scenario, we also measure the effectiveness of our approach by scanning cache sets at every page offset in a *random* order.

The ECDSA victim program spends only about 25% of its execution time running the vulnerable code. Therefore, there is a high chance that the attacker cannot detect the target set, as they may collect the traces while the victim is not executing the vulnerable code—a problem known as de-synchronization. Hence, the attacker repeatedly scans all possible sets until detecting the target set or timeout. We set the timeouts for PAGEOFFSET and WHOLESYS to 60 s and 900 s, respectively. Time spent on eviction set construction is not counted towards the timeout.

Scanner implementation. To automatically detect the target set with the PSD method, we train a supporting-vector machine (SVM) to predict if a trace is from the target set based on the trace's PSD. The SVM model uses a polynomial kernel and is trained using scikit-learn [195]. To train the model, we collect 2,266 traces from monitoring the target set and 120,103 traces from non-target sets from different Cloud Run hosts. We randomly withhold 30% of the traces as the validation set and use the remaining traces to train the model. Our model has a 1.02% false-negative rate and a 0.01% false-positive rate on the validation set.

During an attack, the attacker program first builds eviction sets for the SF sets at the target page offset or in the whole system. It then collects a 500 µs access trace from each SF set. In our implementation, the attacker program running on Cloud Run needs to stream access traces back to a local machine to be processed by a Python program. To reduce the data transmission overhead, the attacker program performs a preliminary filtering and only sends back traces containing 50-400 accesses. This filtering range is empirically determined based on the victim's behavior and the access trace length. For every filtered trace, the local Python program computes its PSD and uses the SVM model to predict if it is from the target set.

Note that our PSD method may falsely identify a non-target set as a target set. This can occur when scanning an SF set corresponding to the data or instruction accesses performed by `MAdd` or `MDouble`, as those accesses may occur at a frequency that is similar to that of the target cache line. To filter out these false positive results, we attempt to extract the nonce bits from a positive access trace using the method detailed in Section 4.7.3. If we fail to extract enough nonce bits, or if the extracted "nonce bits" are heavily biased towards 0 or 1, we disregard this access trace and continue searching. As we find that the risk of false positives is low for PAGEOFFSET, we only apply this technique to WHOLESYS.

Evaluation setup. We conduct this experiment on Cloud Run at different times of day, totaling 357 measurements for PageOffset and 207 measurements for WholeSys. For WholeSys, we deem the scan successful if it manages to locate the cache set accessed by the either side of the branch, as accesses made by either side can disclose the nonce k.

Results. Table 4.7 lists the key metrics of finding the target cache set using the PSD method. Given our timeout configurations, 94.1% and 73.9% of the scanning attempts find the target set under PageOffset and WholeSys, respectively. The lower success rate under WholeSys is mainly because we can only scan each SF set fewer times within the timeout period, leading to more failures due to the de-synchronization problem. Averaged among successful scans, it takes 6.1 s and 179.7 s to find the target set under PageOffset and WholeSys, respectively. Finally, we scan from 762 sets/s to 831 sets/s. The scanning speed can be improved by using multiple threads to scan cache sets in parallel.

Table 4.7: Performance of identifying the target cache set.

Metric	**PageOffset**	**WholeSys**
Success Rate	94.1%	73.9%
Average Success Time	6.1 s	179.7 s
Std. Deviation of Success Time	6.9 s	177.4 s
95% Percentile Success Time	16.1 s	546.6 s
Average Scan Rate	831 sets/s	762 sets/s

4.7.3 End-to-End Nonce Extraction

Putting all the pieces together, we demonstrate end-to-end, *cross-tenant* nonce k extractions on Cloud Run. In this demonstration, the attacker first successfully co-locates their attack container with the victim container [101]. Then, the attacker builds the eviction sets and finds the target set using the PSD method, while sending requests to trigger victim executions. Once the target set is identified, the attacker triggers the victim execution 10 more times to steal the different nonces used in each execution.

To process the memory access trace, we train a random forest classifier [195, 196] to predict if a detected memory access corresponds to an iteration boundary. To filter out false-positive boundary predictions, we consider only boundary pairs that are $8k$ to $12k$ cycles apart, as this is the duration variation that we expect from a single iteration on these hosts. From each pair of predicted neighboring boundaries, we recover the nonce bit in the iteration by checking if there is an extra access in the middle of the iteration.

We attempt end-to-end nonce k extractions under the PageOffset scenario on 52 pairs of co-located containers on Cloud Run. We identify a potential target set and observe a signal in 47 of them. Within the 470 traces collected from these 47 victims, we extract an average of 68% (or a median value of 81%) of the nonce bits. Among these recovered bits, our average bit error rate is 3%. The full attack, which includes constructing eviction sets, identifying the target SF set, and collecting 10 traces, takes an average of 19 seconds.

4.8 RELATED WORK

Side-channel attacks in cloud. Ristenpart et al. [37] examined the placement of virtual machines on physical hosts within AWS and developed techniques to achieve co-location. Zhang et al. [17] employed Flush+Reload for a cross-tenant attack on a Platform-as-a-Service (PaaS) cloud. However, Flush+Reload is no longer feasible in modern clouds [36, 99]. İnci et al. [15] in 2015 conducted a Prime+Probe attack on AWS EC2 to extract RSA keys, using a reverse-engineered LLC slice hash function and huge pages to build eviction sets. Their attack is long running, relies on huge pages, and targets an inclusive LLC—all of which are incompatible with modern cloud environments.

Mitigations to cache-based side-channel attacks. Defenses can be broadly categorized into two types. The first type, partition-based solutions [53, 54, 55, 57, 59, 60, 61, 197], blocks attacks by partitioning the cache between different tenants. However, this approach often requires complex hardware design and results in high execution overhead. The second type, randomization-based defenses [46, 47, 48, 49, 50, 52, 198, 199, 200], focuses on obfuscating the victim's cache usage. While this method offers high performance, it fails to provide comprehensive security guarantees.

Eviction set construction. Algorithms for constructing eviction sets have received significant attention [3, 51, 133, 167, 170, 201]. However, most approaches are developed and evaluated in a quiescent local environment. Besides the group testing [47, 133] and Prime+Scope [159] algorithms discussed in Section 4.2.1, Prime+Prune+Probe (PPP) [51] exploits the LRU replacement policy to defeat randomized caches by minimizing memory accesses. CTPP [170], which is concurrent to our work, builds on PPP by integrating it with Prime+Scope. Based on the evaluation in CTPP [170], the success rates of both PPP and CTPP fall to almost zero when a single memory-intensive SPEC 2006 benchmark [202], such as `mcf`, runs in the background. Using the average LLC access rate as a metric, the cache activity caused by `mcf` is only about 10% of what we observed on Cloud Run. Section 4.10 offers a more detailed discussion of eviction set construction algorithms that exploit cache replacement policy. Lastly, Guo et al. [201] exploited a non-temporal prefetch instruction to accelerate eviction set construction on Intel inclusive LLCs, but found this technique inapplicable to Intel non-inclusive LLCs [201].

Prime+Probe techniques. Prior arts [157, 203] also used parallel probing in their Prime+Probe implementations [204, 205]. However, to our knowledge, we are the first to study the parallel probing strategy to strike a good balance between probe and prime latency. Oren et al. [102] processed memory access traces in the frequency domain to fingerprint websites.

4.9 CONCLUSION

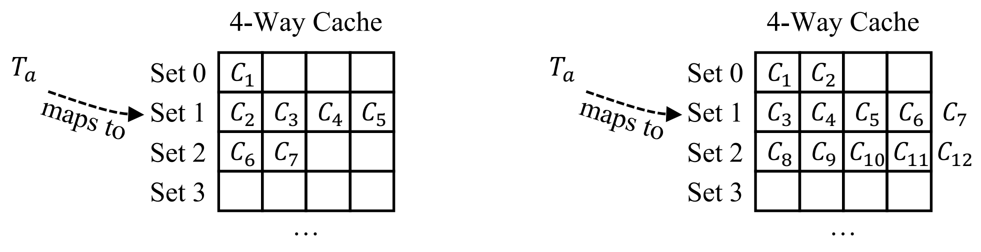
In this chapter, we presented an end-to-end, cross-tenant LLC Prime+Probe attack on a vulnerable ECDSA implementation in the public FaaS Google Cloud Run environment. We showed that state-of-the-art eviction set construction algorithms are ineffective on Cloud Run. We then introduced L2-driven candidate address filtering and a binary search-based algorithm for address pruning to speed-up eviction set construction. Subsequently, we introduced parallel probing to monitor victim memory accesses with high time resolution. Finally, we leveraged power spectral density to identify the victim's target cache set in the frequency domain. Overall, we extracted a median value of 81% of the secret ECDSA nonce bits from a victim container in 19 seconds on average.

Ethical considerations. We limited our attempts to exfiltrate information from only victims under our control. We monitored just one SF set of the host at a time, thus minimizing potential performance interference with other tenants.

4.10 APPENDIX SECTION: COMPARISON TO EVICTION SET CONSTRUCTION ALGORITHMS EXPLOITING REPLACEMENT POLICY

One possible solution to defend against cache-based side-channel attacks is to use randomized caches [46, 47, 52], which randomly but deterministically map physical addresses to cache sets with the option of refreshing the mapping periodically [51]. To evaluate the security of randomized caches, many prior works [47, 51, 170] discussed efficient algorithms that aim to reduce the number of memory accesses to construct an eviction set. These efficient algorithms allow the construction of an eviction set before the mapping is refreshed, thus enabling cache side-channel attacks in randomized caches. For example, Qureshi et al. [47] proposed algorithms that exploit thrashing behaviors in replacement policies such as LRU and RRIP [184].

To understand how to exploit thrashing behavior to build an eviction set, Figure 4.10a shows a 4-way cache using the LRU replacement policy. In this example, we have seven candidate addresses ($\{C_1, ..., C_7\}$) that are mapped to three different cache sets. Set 1 is completely filled with the candidate addresses while the remaining sets are underfilled. The goal in this example is to build an eviction set for a target address T_a that is mapped to the target Set 1. This example is similar to the one used by Qureshi et al. [47].



(a) Only the target set is filled by the candidate set.

(b) Both the target set and a non-target set are overflowed by the candidate set.

Figure 4.10: Different candidate sets for discussing eviction set construction algorithms.

To build an eviction set for T_a, the algorithm finds addresses that are congruent with T_a by exploiting the thrashing behavior of LRU. Specifically, the algorithm primes addresses $C_1, C_2, ..., C_7$ sequentially. Then it accesses T_a to evict C_2, the oldest line in Set 1 after priming. After that, the algorithm times the accesses to (or *probes*) $C_1, C_2, ..., C_7$ sequentially and records addresses that suffer cache misses. Since the probing accesses have the same order as the priming accesses, every probing access to a congruent address that maps to Set 1 will miss in the cache due to thrashing. Meanwhile, accesses to all the non-congruent addresses will hit in the cache. As a result, the addresses that suffered cache misses during the probing are the congruent addresses that form a minimal eviction set for T_a. This algorithm is efficient, as it requires only $2N + 1$ accesses, where N is the size of the candidate set. For caches that use RRIP [184], which is common in modern Intel processors [206], Qureshi et al. [47] proposed to prime the candidate addresses by accessing each address twice to trigger the trashing behavior. As these algorithms find eviction sets by exploiting the thrashing, we refer to this class of algorithms as *thrash-exploiting algorithms* hereafter.

Note that both the discussion in Figure 4.10a and the discussion in [47] make a simplification that the target set (Set 1 in Figure 4.10a) is completely filled by the candidate set while other non-target sets are underfilled. Acknowledging that such a simplification is appropriate when evaluating the security of randomized caches, where an idealized powerful adversary is often assumed, there are many challenges of using the thrash-exploiting algorithms in practice. Several follow-up works [51, 170, 207] pointed out that the candidate set often overflows not only the target set but also many non-target sets, as illustrated in Figure 4.10b. Traversing a candidate set like the one in Figure 4.10b will result in many cache misses due to self-conflicts, such as when probing non-congruent addresses $\{C_8, ..., C_{12}\}$ in Figure 4.10b. As a result, the thrash-exploiting algorithm fails to generate a minimal eviction set for T_a by including non-congruent addresses $\{C_8, ..., C_{12}\}$ in the eviction set.

To overcome the challenge of self-conflicting candidate sets, Purnal et al. proposed a new algorithm named Prime+Prune+Probe [51, 207] that introduces an additional *pruning* step to remove self-conflicting addresses. From a high level, the pruning step repeatedly primes and probes the candidate set and *discards* any address that suffered a cache miss during probing. This iterative pruning process is repeated until no cache misses are observed during probing. Consider again Figure 4.10b, the pruning step will observe misses when probing C_7 and C_{12}, as they do not fit into the cache. As a result, C_7 and C_{12} are removed from the candidate set, resulting in a candidate set that is not self-conflicting.

While not discussed in [51, 207], the pruning step must traverse the candidate set in such a way that *completely avoids* thrashing or *anti-thrashing*. Otherwise, we may find that more than one address suffered a cache miss in the target Set 1 during probing. As a result, we will over-prune more than one address from the target Set 1. When over-pruning occurs, the candidate set contains less than $W = 4$ congruent addresses, and the eviction set construction will fail. Despite the importance of anti-thrashing during pruning, how anti-thrashing can be achieved is not discussed by Purnal et al. [51, 170]. Lastly, while Purnal et al. improved the idea of thrash-exploiting algorithms, their discussion and implementation are limited to a simulated system instead of a commercial processor.[1]

To address the challenge of over-pruning, a follow-up work [170], which is concurrent to the work in this chapter, proposed a new algorithm named Conflict Testing and Probe+Prune (CTPP). The high-level idea of CTPP is to use Prime+Scope for the pruning step. Specifically, CTPP locates the W-th congruent address in the candidate set with Prime+Scope. Then CTPP discards all the candidate addresses that are after the W-th congruent address. As a result, the remaining addresses are guaranteed not to self-conflict *in the target set*. Therefore, one can then use a pruning approach that is not anti-thrashing to prune the remaining addresses. This is because any over-pruning now can only occur in non-target sets, which will not fail the eviction set construction process.

According to the evaluation by Xue et al. [170], CTPP achieves good performance in a local quiescent environment. However, the success rates of both CTPP and Prime+Prune+Probe fall to almost zero when a single memory-intensive benchmark, such as `mcf`, runs in the background. Using the implementation [208] provided by Xue et al., we independently evaluate CTPP on both a local 22-slice Skylake-SP machine and Cloud Run with a setup similar to Section 4.4.2. After 1,000 eviction set constructions in the local environment, CTPP shows a success rate of 37.2% with an average execution time of 2.6 ms. Compared with the local quiescent results using other algorithms in Table 4.2, CTPP has the best performance, but the least success rate. In the Cloud Run environment, we run CTPP on 118 hosts and construct 100 eviction sets on each host. On

[1]To the best of our knowledge, at the time of the work in this chapter is done, there was no implementation of thrash-exploiting algorithms that works on commercial processors. As was shown by a follow-up work [170] that is concurrent to the work in this chapter, it is not trivial to implement thrash-exploiting algorithms even in a local quiescent environment. As a result, the main experiments in this chapter did not consider thrash-exploiting algorithms as practical baselines.

average, CTPP has a success rate of merely 1.3% and an execution time of 15.1 ms. Our results suggest that CTPP, the state-of-the-art thrash-exploiting algorithm, is yet to be practical in the noisy cloud environment. Our observation in Cloud Run is consistent with the results obtained by Xue et al. [170] in a local noisy environment with `mcf` running. Using the average LLC access rate as a metric 4.4.3, the cache activity caused by `mcf` is only about 10% of what we observed on Cloud Run. We refer the reader to [170] for more details on CTPP and its noise susceptibility.

4.10.1 Practical Challenges of Thrash-Exploiting Algorithms

Without being tied to CTPP or Prime+Prune+Probe, we identified several *practical* challenges in using the thrash-exploiting algorithms. Some of the challenges are related to the noise susceptibility of these algorithms, while others are related to implementation. As mentioned earlier, these practical challenges can be ignored when evaluating the security of randomized caches, where we assume an idealized attacker, but they need to be addressed for a practical attack in the noisy cloud.

Noise tolerance of CTPP [170] and Prime+Prune+Probe [51]. We discuss the noise tolerance of the CTPP and Prime+Prune+Probe algorithms. The first negative effect of background cache accesses made by other running applications is that these accesses can destroy the carefully primed cache replacement policy states, leading to no thrashing or no anti-thrashing behaviors. As a result, during the pruning step, Prime+Prune+Probe can over-prune the candidate set due to failed anti-thrashing and CTPP can over-prune the candidate set because Prime+Scope stops before the W-th congruent address is found (similar to how Prime+Scope fails in the cloud, as discussed in Section 4.4.2). Similarly, if thrashing fails, the probing step will not collect all the congruent addresses, leading to the failure of constructing an eviction set.

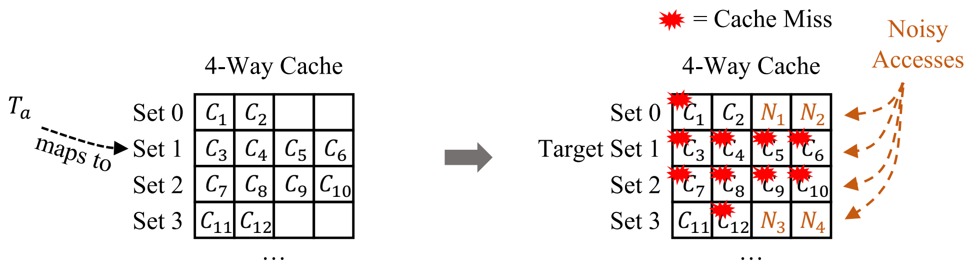


Figure 4.11: An illustration of the priming and probing steps being affected by noisy accesses to multiple cache sets. The left side shows the cache state before priming and probing. The right side shows the cache being filled by cache lines brought by noisy accesses ($N_1, ..., N_4$) and many non-congruent addresses suffered a cache miss during probing.

Now, assuming that thrashing and anti-thrashing can occur perfectly and we can obtain a pruned candidate set without over-pruning, the remaining priming and probing steps are still highly vulnerable to noise. This is because these steps can be affected by noisy accesses to *all* the cache sets to which some candidate addresses are mapped. Consider the example in Figure 4.11. The left side of the figure shows a candidate set that is perfectly pruned and the target Set 1 contains exactly $W = 4$ congruent addresses. These candidate addresses span across Sets 0–3. The desired outcome in this example is that accesses to every line from the target Set 1 miss in the cache during the probing, while accesses to lines from non-target sets all hit in the cache. However, this is unlikely to happen due to the noise, as all Sets 0–3 will experience noisy accesses made by other running applications. These noisy accesses can evict addresses in non-target sets and make them miss in the cache during the probing. The right side of Figure 4.11 illustrates a possible probing result under the noise. Candidate addresses that are missed in the cache are marked with a "blast" sign. As the figure shows, the cache is now filled by cache lines brought by noisy accesses ($N_1, ..., N_4$) and we observe cache misses in both the target set and non-target sets. As a result, the algorithm will include many non-congruent addresses in the eviction set, failing to construct a minimal eviction set. In contrast, algorithms that use the $TestEviction$ primitive are more resilient to noise. This is because $TestEviction$ is affected by noisy accesses to only the target set, while thrash-exploiting algorithms are affected by noisy accesses to hundreds if not thousands of cache sets to which candidate addresses are mapped.

Priming cache replacement policy states. Thrash-exploiting algorithms require careful priming of cache replacement policy states either to trigger thrashing or completely avoid thrashing. In addition to the fact that noisy accesses can destroy carefully primed cache states, crafting a thrashing or anti-thrashing access sequence for caches using complex replacement policies like RRIP [184], which is widely used by Intel processors [206], is also challenging. Since 2-bit RRIP [184] is a common replacement policy used by the LLCs of modern Intel processors [206], we focus on 2-bit RRIP in the following discussion. Using a 2-bit RRIP policy, a cache line can have an age from 0 to 3, with 0 being the youngest and 3 being the oldest [184, 206]. Upon a cache miss, the cache set is searched and a line with age 3 will be evicted. A 2-bit RRIP policy is also called *Quad-age LRU (QLRU)* [209, 210]. We will use 2-bit RRIP and QLRU interchangeably.

Now consider again Set 1 in Figure 4.10a, to which four addresses $\{C_2, ..., C_5\}$ are mapped. Under the QLRU policy, it is believed that thrashing can be triggered by first sequentially priming the set with a pattern of $C_2, C_2, C_3, C_3, ..., C_5, C_5$, then accessing T_a, followed by probing the set with $C_2, C_3, ..., C_5$ [47, 170]. The rationale for using a *repeating-access pattern* is that the first access inserts the cache line into Set 1 with an insertion age, which is usually 1. Then the second access to the same cache line will promote the line into the "hit" age, which is usually 0. After using this pattern, $\{C_2, ..., C_5\}$ will have the same age. Then after accessing T_a to evict C_2, the probe accesses to $C_2, C_3, ..., C_5$ will trigger thrashing.

However, whether the repeating-access pattern can trigger thrashing depends on many factors, such as the initial cache replacement policy states and the exact QLRU variant, which includes the insertion policy, the hit promotion policy, the tie-breaking policy, and the policy of handling the case where no lines have an age 3 [206]. Due to the complexity and large design space of QLRU, we tested the repeating-access pattern using the cache simulation tool provided by nanoBench [206, 211]. This simulation tool has been used to match against Intel's cache replacement behaviors in an effort to recover Intel's cache replacement policies [206]. Therefore, this simulation tool should produce high-fidelity results. Using nanoBench to simulate the repeating-access pattern in an *initially empty* 4-way cache set while varying QLRU variants, we found that thrashing did *not* occur in 232 QLRU variants out of 320 deterministic QLRU variants supported by nanoBench.[2] Starting from a non-empty cache set will further increase the difficulty of causing thrashing, as the initial cache replacement policy states are unknown to the attacker.

Necessity of constructing and using L2 eviction sets. Using thrash-exploiting algorithms on caches with QLRU replacement policy is more expensive than prior work [47] estimated. This is because when using the repeating-access pattern, one needs to guarantee that the repeated access is not filtered by L1 or L2 cache and is received by LLC. As a result, one needs to build an L2 eviction set for each candidate address and evict the candidate address with the L2 eviction set before the repeated access. Even without considering the cost of building L2 eviction sets, the construction of an LLC eviction set would actually require $(3+W_{L2})N+1$ accesses, instead of $3N+1$ accesses estimated by prior work [47], where N is the candidate set size and W_{L2} is the L2 associativity, which is often 16 or higher.

Unknown replacement policy of Intel's non-inclusive LLCs. To the best of our knowledge, the replacement policy of non-inclusive LLCs used by modern Intel server processors is yet to be recovered. The main difficulties in recovering its replacement policy are the undocumented interaction between the LLC and SF and the policy considers the coherence state of the cache lines (Section 4.2.2). Additionally, it has been found that Intel server processors can allocate ways for Direct Cache Accesses (DCAs) [212], which further complicates the recovery and exploitation of the replacement policy in a data center setting, where DCA is frequently used.

[2]To reproduce our simulation, one can download nanoBench [211] and run the `cacheSeq.py` script found in directory `tools/CacheAnalyzer`. For example, command `./cacheSeq.py -sim QLRU_H00_M1_R0_U1 -sets 1 -simAssoc 4 -seq "A A B B C C D D E A? B? C? D?"` simulates a 4-way cache using the `QLRU_H00_M1_R0_U1` variant, which is used by Intel Core i5-1035G1 (Ice Lake) [206], with an access sequence of "A A B B C C D D E A? B? C? D?". In this access sequence, each letter represents an access to a cache line, with different letters representing accesses to different cache lines. Only cache hits from accesses followed by a "?" are reported. For the command above, nanoBench reports one cache hit on the last "D?" access. Note that while this simulation requires no kernel module support, nanoBench will complain if the kernel module is not loaded. This can be solved by commenting out lines 389–404 in `kernelNanoBench.py` except for line 398. Finally, please refer to the nanoBench paper [206] for more details on QLRU variants and their naming conventions.

CHAPTER 5: High-Performance, Low-Leakage Dynamic Resource Partitioning

5.1 INTRODUCTION

As shown in Chapters 3–4, it is feasible for an unprivileged malicious cloud user to exfiltrate sensitive information from a target victim using microarchitectural side-channel attacks, such as last-level cache Prime+Probe, in public clouds. As microarchitectural side-channel attacks rely on the sharing of resources between the attacker and the victim, a principled approach to defend against these attacks is to partition the shared resources among different users.

For example, a shared hardware structure can be spatially partitioned [61]. In such a design, each process gets a static partition for the duration of its execution. Hence, information about a process' use of the shared resource cannot leak to processes that own other partitions of the structure. Taking a shared cache as an example, each process may get one way. While static partitioning is safe, it is undesirable for a dynamic environment where the running processes and the process resource demands change over time. In such an environment, any static partition is suboptimal and can lead to resource wastage or under-provisioning [46, 53].

An intermediate approach that can retain high performance while leaking a limited amount of information is *dynamic partitioning* [104, 105, 213, 214]. Here, individual processes can dynamically increase or decrease the size of their partition. For example, a process may be allowed to resize its partition at certain times and by a certain amount.

It would be useful to formally quantify the leakage of dynamic partitioning. One could then assess the trade-off between security lost and performance gained. Unfortunately, accurately quantifying the leakage with dynamic partitioning is hard. The precise way to do so is to enumerate all the possible inputs that the victim program can take (including their probabilities) and record all the resulting *Resizing Traces*. A resizing trace is the sequence of resizing actions (e.g., expand the partition, shrink it, or maintain it), and the time of each action. Then, the leakage of the program is calculated as the *entropy* (intuitively, the variability) of these resizing traces [215].

Clearly, this approach does not scale. Furthermore, in today's dynamic partitioning schemes, *what* resizing decisions are made (in 'space') and *when* they are made (in 'time') are *entangled*. For example, *when* a program reaches a given phase and triggers a resize does depend on its rate of forward progress up to that point (time), which in turn is based on previous partition decisions (space), and so on. Since program timing depends on low-level effects such as microarchitectural details, it is typically intractable to analyze. By implication, since the sequence of actions is entangled with timing, the sequence of actions is intractable to analyze.[1]

[1]This issue is akin to taint explosion in traditional information-flow systems [216, 217].

As a result, state-of-the-art leakage analysis typically assumes the worst case: all the resizing traces that could theoretically occur are realizable and, therefore, at each resizing decision point, all choices are equally likely. The result is *leakage overestimation*. Then, assuming that a user has a fixed leakage budget and that exhausting the budget at runtime will prohibit further resizings [218, 219, 220], overestimating leakage means that fewer partition resizings are allowed before the budget is reached—unnecessarily hurting performance. Therefore, if the leakage could be bound tightly, it would be possible to improve performance.

In this chapter, we present *Untangle*, a novel framework for constructing low-leakage and high-performance dynamic partitioning schemes. *Untangle* formally splits the leakage into two parts: (i) leakage from deciding *what* resizing action to perform (*action leakage*) and (ii) leakage from deciding *when* each resizing action occurs (*scheduling leakage*).

Based on this breakdown, *Untangle* makes two advances. First, *Untangle* introduces a set of principles for constructing dynamic partitioning schemes that untangle program timing from the action leakage. As a result, the sequence of resizing actions only depends on the retired dynamic instruction sequence of the execution, and not on timing (e.g., the cycle when each instruction retires). Following these principles, the action leakage can be altogether eliminated with the help of annotations.

In a second advance, *Untangle* introduces a novel way to model the scheduling leakage without analyzing program timing. Overall, with these two contributions, *Untangle* is able to quantify the leakage of a dynamic resizing scheme more tightly than prior work.

The main focus of this chapter is on describing the *Untangle* framework rather than presenting a detailed hardware implementation. Still, *Untangle* can be applied to a variety of hardware structures. In our evaluation, we apply it to dynamically partition the last-level cache. For a large set of workloads, we compare *Untangle* to a conventional dynamic partitioning approach. We show that, on average, workloads leak 78% less under *Untangle* than under the conventional dynamic approach, for approximately the same workload performance.

This chapter makes the following contributions:

- Proposes *Untangle*, a novel framework for constructing low-leakage and high-performance dynamic partitioning schemes.
- Presents a set of principles to untangle program timing from action leakage.
- Introduces a way to model scheduling leakage without analyzing program timing.
- Applies *Untangle* to dynamic partitioning of the last-level cache and evaluates its performance and leakage under a large set of workloads.

Table 5.1: Characteristics of some prior dynamic partitioning schemes.

Name	Resource	Utilization Metric	Action Heuristic	Resizing Schedule
UMON [104]	Last-level cache (LLC)	Number of LLC hits under different partition sizes	Pick partition sizes that maximize global LLC hits	Every 5 M cycles
Jigsaw [214]	LLC	Similar to UMON [104]	Peekahead algorithm in software	Every 50 M cycles
Jumanji [105]	LLC	Tail latency of network requests	Compare to static thresholds	Every 100 ms
SecSMT [213]	Pipeline structures	Number of "full" events	Increase the partition that has the most "full" events	Every 100 K cycles

5.2 BACKGROUND: ENTROPY AND MUTUAL INFORMATION

Entropy is a quantitative measure of information, represented by the *uncertainty* of a random variable [215]. Let X be a discrete random variable that takes values in $\mathcal{X}$ and $p(x)$ be the probability of $\{X = x\}, x \in \mathcal{X}$. Then the entropy of X is

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log p(x). \tag{5.2.1}$$

When the log is to the base 2, the entropy is measured in bits. $H(X)$ has the property of $H(X) \leq \log|\mathcal{X}|$, where $|\mathcal{X}|$ is the number of elements in $\mathcal{X}$. The equality is achieved if and only if X follows a uniform distribution over $\mathcal{X}$. Intuitively, the more uniform the distribution of the variable is, the higher the entropy or information carried by the variable is.

The *joint entropy* of two random variables X and Y is

$$H(X,Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log p(x,y), \tag{5.2.2}$$

where $p(x,y)$ is the probability of $\{X = x, Y = y\}, x \in \mathcal{X} \wedge y \in \mathcal{Y}$. The *conditional entropy* of X given Y is

$$H(X|Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log p(x|y). \tag{5.2.3}$$

The *mutual information* between variables X and Y is the amount of information we learn about one of the two variables when observing the other variable. It is defined by

$$I(X;Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x)p(y)}{p(x,y)}. \tag{5.2.4}$$

$I(X;Y)$ is always *non-negative* and $I(X;Y) = 0$ if X and Y are independent. In all cases, $I(X;Y) = I(Y;X)$.

5.3 LEAKAGE OF DYNAMIC PARTITIONING SCHEMES

5.3.1 Generalizing Dynamic Partitioning Schemes

A dynamic partitioning scheme is typically characterized by three components (Table 5.2). One is the *Utilization Metric* for the resource of interest. This metric reflects a program's demand for the resource and guides resizing. For example, for the last-level cache (LLC), one possible utilization metric is the number of LLC misses per thousand instructions. Typically, improving the utilization metric translates into performance improvements.

Table 5.2: Components of a dynamic partitioning scheme.

Component	**Description**
Utilization Metric	Measure of the demand for the resource
Action Heuristic & Resizing Actions	How to pick what resizing action to perform (e.g., EXPAND, SHRINK, MAINTAIN)
Resizing Schedule	When to make a resizing assessment and perform the decided action

Another component is the *Action Heuristic* and the *Resizing Actions* (or *Actions* for short). Resizing actions are scheme-defined operations for adjusting the partition size. Common actions are "expand the partition" (EXPAND), "shrink it" (SHRINK), and "maintain it" (MAINTAIN). More generally, a scheme can define a set of actions, and each action consists of using a given partition size next (e.g., actions can be "set the cache partition size to 1 MB", or "to 2 MB", or "to 4 MB"). The role of the action heuristic is to pick one of the possible actions based on the utilization metric value. For example, an action heuristic is to compare the utilization metric to some utilization thresholds and, based on the result, decide which action to perform. We call this checking and decision process a *Resizing Assessment*.

A final component is the *Resizing Schedule* (or *Schedule* for short). It determines *when* to make a resizing assessment and perform the action. The action decided during the assessment is typically performed immediately, but it can also be performed later. Example schedules are to assess resizing at fixed time intervals or after retiring a fixed number of instructions. The choice of resizing schedule affects a scheme's responsiveness to the program's demands.

Table 5.1 lists the components of our framework for some prior dynamic partitioning schemes.

5.3.2 Leakage with Dynamic Partitioning

Dynamically adjusting the partition size of a program based on the program's resource demands can cause information leakage. Secrets can be leaked through *when* resizing assessments are made and *what* resizing actions are taken. More formally, the victim's *Resizing Trace*, which includes the *sequence of resizing actions* and the *timing of each action*, is secret-dependent and can leak information. Using cache partitioning as an example, Figure 5.1 demonstrates three ways of leaking a secret (similar to the three types of leakage in [221]).

```
if (secret)
   for i in 0..4M // traverse a 4MB array
      access(&arr[i]);
// Resizing assessment, expand?
```

(a) Resizing action depends on the secret through control flow.

```
for i in 0..4M // traverse a 4MB array
   access(&arr[i * secret]);
// Resizing assessment, expand?
```

(b) Resizing action depends on the secret through data flow.

```
if (secret)
   usleep(1000); // sleep for 1ms
for i in 0..4M // traverse a 4MB array
   access(&arr[i]);
// Resizing assessment, will expand
```

(c) Resizing timing depends on the secret.

Figure 5.1: Code snippets that demonstrate the leakage of a dynamic cache-partitioning scheme.

In Figure 5.1a, the secret controls the execution of a large-array traversal. If the secret is non-zero, the array is traversed, increasing the cache utilization and causing a partition expansion. The attacker can observe the expansion, hence exfiltrating the secret (Section 5.4 details our threat model). In Figure 5.1b, the secret influences the indexes used in the array traversal. Depending on the secret value, the array traversal may access a different number of cache lines, resulting in a different cache utilization and, possibly, a different resizing action. Finally, in Figure 5.1c, regardless of the secret value, the array traversal always executes and triggers a partition expansion. However, the secret is leaked based on *when* the expansion occurs.

The most accurate way to measure leakage in a dynamic partitioning scheme is to exhaustively enumerate all possible victim program inputs (including their probability) and the resulting resizing traces under the partitioning scheme. These are the set of resizing traces that are *realizable*. Then, the leakage of the program is calculated as the entropy of these traces using Equation 5.2.1. This is the amount of information that the victim program leaks under this scheme. Unfortunately, although this approach is accurate, it is not feasible in practice.

5.3.3 Limitations of Prior Work

To mitigate the leakage in dynamic partitioning schemes, most prior work either coarsens the granularity of resizing (i.e., resizes less frequently or reduces the number of possible actions), or fixes the timing of resizing actions to a publicly-known schedule [103, 218, 219, 220]. As a result, these approaches reduce the leakage at the cost of losing some of the adaptivity of dynamic schemes. Fundamentally, they trade off performance for better security.

Making matters worse, prior schemes often *overestimate* the leakage from resizing by implicitly assuming that all the resizing traces that could theoretically occur are realizable. For example, consider a dynamic partitioning scheme that makes resizing assessments every one millisecond and supports two resizing actions. In a one-second execution of the program, the scheme will make 1000 resizing assessments. Since the timing of the assessments is fixed at multiples of one millisecond, the leakage purely comes from what action is taken at each assessment. Hence, a common but over-conservative estimation is that the scheme can produce *any* of the 2^{1000} different traces in a one-second execution and that all traces have the same probability of occurring. Hence, the leakage is computed using Equation 5.2.1 to be $\log 2^{1000} = 1{,}000$ bits, which is too conservative for most programs.

Overall, this conservative assumption results in further restricting the adaptivity allowed: given a target leakage budget, the budget will be consumed sooner because of the leakage overestimation, which will prohibit further resizings. The end result is to render dynamic schemes less appealing.

5.3.4 Our Approach and Challenges

An intuitive idea to reduce the leakage of dynamic resizing schemes is to make the scheme aware of which data is public and which is secret. Then, resizing assessments that only depend on public data can be performed without leaking secret information. Consequently, the scheme gains more resizing flexibility—which maximizes performance without impacting security.

Consider Figures 5.1a and 5.1b again. If the scheme knows that the array traversal is secret-dependent (e.g., through annotations inserted by the side-channel detection tools of Section 2.2.5), it can conveniently exclude the *secret-dependent cache demand* when measuring the cache utilization metric. Hence, the resulting resizing trace depends on only public cache utilization and does not reveal the secret.

However, these annotations alone cannot remove the leakage in Figure 5.1c. This is because the secret causes the *public* memory accesses to have *secret-dependent timing*. Note that the secret-dependent timing can manifest not only through *when* a resizing assessment is made, but also through *what* resizing action is taken at an assessment. To see how, consider Figure 5.1c but with a resizing schedule that makes an assessment at 1 ms (in contrast to making an assessment after the array traversal at Line 5). In this case, depending on the secret value, the point of assessment may be before or after the public array traversal, resulting in a different cache utilization and, consequently, a different resizing action.

Unfortunately, extending existing side-channel detection tools for this type of implicit flow through program timing is very hard—given the impracticality of fully determining program timing with modern processors and taint explosion in traditional information-flow systems [216, 217]. As a result, in the case when the victim has secret-dependent timing (which is the general case), it is hard to bound the leakage any tighter than the conservative approach discussed above does.

To address this problem, in Section 5.5, we build a novel framework named *Untangle* that helps reason about this entanglement of action and timing. Based on the framework, we develop design principles for dynamic resizing schemes and mitigations to achieve a tight bound on the leakage. *Untangle* enables us to attain high performance without compromising security.

5.4 THREAT MODEL

We consider a public cloud environment where users are mutually-distrusting peers. Under this mutually-distrusting peer model, the attacker and the victim are in the same security level. Additionally, the attacker and the victim share a hardware resource (e.g., a cache). The system can partition the resource into attacker and victim partitions. The system can dynamically change the partition sizes based on the resource utilization, which can be secret-dependent and reveals sensitive information of the victim.

We assume an idealized attacker that can directly observe the victim's exact resizing trace (i.e., what resizing actions are taken and when). In practice, an attacker can only indirectly estimate the victim's resizing trace by probing its own partition size and observing how it changes over time as a result of victim resizes. This estimation is not completely accurate because neither the resizing actions nor the attacker's probing are instantaneous. Hence, a realistic attacker would be less capable than the idealized attacker that we are assuming. Lastly, we assume that the partition scheme does not change utilization metric, action heuristic, resizing actions, or resizing schedule in the course of the victim's execution.

The victim sets a threshold for how much leakage from the victim program's run or runs is tolerable. The dynamic partitioning scheme (i.e., *Untangle*) measures the runtime leakage and *guarantees* it cannot exceed this threshold. If and when the threshold is reached, the victim is not allowed to perform further resizings—hurting the performance of its subsequent execution, *but not its security*.

We assume that there are sound approaches to annotate programs with secret-dependent usage of the resource being partitioned and secret-dependent control-flow. This is achievable with existing analyses [109, 110, 111, 113], or with a conservative approach that annotates all the instructions from the part of the program that handles secrets. Section 5.6.5 discusses the capabilities of these existing analyses.

5.5 UNTANGLE: SECURE DYNAMIC PARTITIONING

Recall from Section 5.3.1 that a dynamic partitioning scheme uses a resizing schedule to decide when to perform resizing assessments, and an action heuristic to decide what resizing actions to take. As a result, secrets are leaked through the observation of "when" and "what" actions are taken. While annotating instructions that have secret-dependent resource usage can help reduce the leakage in some special cases, annotations have limited use in general programs due to secret-dependent timing (Section 5.3.4).

In this section, we present *Untangle*, a novel framework that quantifies the leakage in a dynamic partitioning scheme with a tight bound. *Untangle* formally splits the leakage into two parts: (i) leakage from deciding what resizing action to perform (*action leakage*) and (ii) leakage from deciding when each resizing action occurs (*scheduling leakage*). Based on this breakdown, *Untangle* makes two contributions. First, it introduces a set of principles to disentangle program timing from the action leakage, and eventually remove the action leakage with annotations. Second, *Untangle* introduces a novel way to tightly-bound scheduling leakage without analyzing program timing.

Figure 5.2 shows a diagram with the action and scheduling leakages, and how both are affected by the same two root causes: secret-dependent demand and secret-dependent timing. In this section, we describe how *Untangle* allows us to untangle the different effects. First, Section 5.5.1 shows how we formally separate the two types of leakage. Then, Section 5.5.2 shows how we can eliminate action leakage. Finally, Section 5.5.3 presents an easy way to bound scheduling leakage. The result is a tight bound estimation of the total leakage in a dynamic partitioning scheme.

5.5.1 Decoupling the Two Types of Leakage

Recall from Section 5.3.2 that a resizing trace is a sequence of tuples, where each tuple contains a resizing action and the time of the action. Further, the leakage of a specific victim program is the

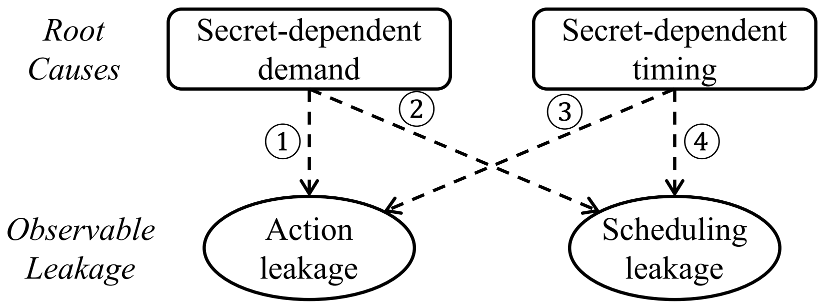


Figure 5.2: Action and scheduling leakages and their root causes. *Untangle* is able to eliminate Edges ① and ③, and simplify the analysis for Edges ② and ④.

entropy of the realizable resizing traces for the program.

To understand how we decouple the leakages for a given program, let S be a discrete random variable that represents a sequence of resizing actions. S takes values in a set $\mathcal{S}$. If we denote the set of supported resizing actions by $\mathcal{A}$, then an action sequence $s \in \mathcal{S}$ is $a_1, a_2, ..., a_n$, where a_i is the ith action in the sequence, and $a_i \in \mathcal{A}$. Note that S only contains what resizing actions are taken but not when.

For an action sequence $s \in \mathcal{S}$, let T_s be a discrete random variable that represents the timing of action sequence s. T_s is a sequence of strictly-increasing timestamps $t_1, t_2, ..., t_n$, where t_i is the timestamp when the ith action occurs. T_s takes values in $\mathcal{T}[s]$. Without loss of generality, we assume that these timestamps have a finite resolution and therefore represent them as integers. Under a fixed time-interval resizing schedule, s has only one possible T_s (i.e., $|\mathcal{T}[s]| = 1$). However, under a more general resizing schedule, s can have many different T_s (i.e., $|\mathcal{T}[s]| > 1$). An example of one such resizing schedules is to make a resizing assessment every N retired instructions. T_s varies because the time to retire N instructions and then trigger an assessment depends on program timing.

We use tuple (S, T_S) to denote a random variable that represents the resizing trace. (S, T_S) takes values in $\{(s, \tau_s) \mid s \in \mathcal{S} \wedge \tau_s \in \mathcal{T}[s]\}$. Therefore, the leakage L, which is equal to the entropy of the realizable resizing traces, is the joint entropy of S and T_S (using Equation 5.2.2):

$$L = H(S, T_S) = -\sum_{s \in \mathcal{S}} \sum_{\tau_s \in \mathcal{T}[s]} p(s, \tau_s) \log p(s, \tau_s), \tag{5.5.1}$$

where $p(s, \tau_s)$ is the probability of following a specific action sequence s with a specific timing sequence τ_s.

By the chain rule of joint entropy [215], we can rewrite Equation 5.5.1 as:

$$\begin{aligned}
L &= H(S, T_S) = H(S) + H(T_S|S) \\
&= H(S) - \sum_{s\in\mathcal{S}} \sum_{\tau_s\in\mathcal{T}[s]} p(s, \tau_s) \log p(\tau_s|s) && (5.5.2) \\
&= H(S) - \sum_{s\in\mathcal{S}} \sum_{\tau_s\in\mathcal{T}[s]} p(s)p(\tau_s|s) \log p(\tau_s|s) && (5.5.3) \\
&= H(S) + \sum_{s\in\mathcal{S}} p(s)\underbrace{(-\sum_{\tau_s\in\mathcal{T}[s]} p(\tau_s|s) \log p(\tau_s|s))}_{\text{denoted by } H(T_s|S=s)} && (5.5.4) \\
&= H(S) + \sum_{s\in\mathcal{S}} p(s)H(T_s|S = s) && (5.5.5) \\
&= H(S) + E[H(T_s|S = s)]. && (5.5.6)
\end{aligned}$$

Equation 5.5.2 applies the definition of conditional entropy from Equation 5.2.3. The term over the bracket in Equation 5.5.4 is the entropy of the timing sequences in a specific action sequence s, which is denoted as $H(T_s|S = s)$ in Equation 5.5.5. The second term in Equation 5.5.5 is the expected value of $H(T_s|S = s)$ for every possible action sequence s.

Equation 5.5.6 shows that the leakage is composed of two simple terms: (1) $H(S)$ is the entropy of resizing action sequences, which we call *action leakage*; and (2) $E[H(T_s|S = s)]$ is the expected value of the entropy of timing sequences T_s for every possible action sequence s, which we call *scheduling leakage*.

Example. Figure 5.3 illustrates the computation of leakage by decoupling action and scheduling leakages. For this example, assume a dynamic partitioning scheme with two supported resizing actions, Expand and Maintain, and three realizable traces. These three traces have two unique action sequences: (1) s_1 = Expand, Maintain (i.e., s_1 performs Expand and then Maintain), and (2) s_2 = Maintain, Maintain. Both sequences are equally likely (i.e., $p(s_1) = 0.5$ and $p(s_2) = 0.5$). Sequence s_1 has two equally probable timing sequences, τ_{s_1} = 100 cycles, 200 cycles and τ'_{s_1} = 150 cycles, 300 cycles. Sequence s_2 has only one possible timing sequence, τ_{s_2} = 120 cycles, 240 cycles.

With our framework, the action leakage is the entropy of the resizing action sequences. Since there are two resizing action sequences in total and they are equally likely, the action leakage $H(S)$ is $-(0.5 \log 0.5 + 0.5 \log 0.5) = 1$ b. As for the scheduling leakage, since sequence s_1 has two equally likely timing sequences, $H(T_{s_1}|S = s_1) = 1$ b; further, since sequence s_2 has only one possible timing sequence, $H(T_{s_2}|S = s_2) = 0$. Therefore, the scheduling leakage is $E[H(T_s|S = s)] = p(s_1)H(T_{s_1}|S = s_1) + p(s_2)H(T_{s_2}|S = s_2) = 0.5$ bits. In total, these three traces leak $L = H(S) + E[H(T_s|S = s)] = 1.5$ bits.

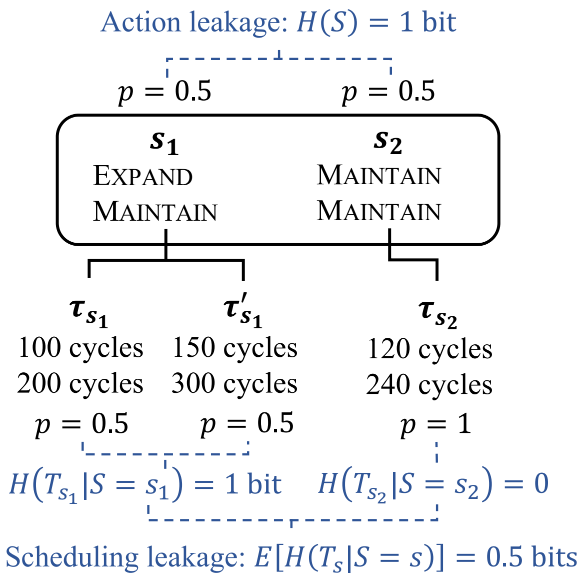


Figure 5.3: An illustration of decoupling the leakage.

5.5.2 Eliminating Action Leakage

The action leakage $H(S)$ can be caused by both secret-dependent demand and secret-dependent timing (Edges ① and ③ in Figure 5.2). Unfortunately, it is challenging to reduce the bounds on the action leakage due to the impracticality of analyzing secret-dependent program timing (Section 5.3.4). To make the action leakage independent of program timing and, therefore, remove Edge ③, we propose two design principles. With these two principles, the action sequence will only depend on the retired dynamic instruction sequence in the execution, but not on program timing. Then, we will discuss how to remove Edge ① with annotations. By removing both edges, we have completely eliminated the action leakage.

Principle 1: Use a *timing-independent metric* to measure the resource utilization. A timing-independent metric means that it only depends on the architectural semantics of the executed program, such as its retired dynamic instruction sequence, and not on the actual instruction timing. An example of what is *not* a timing-independent metric for caches is the number of cache hits in the past T cycles (similar to the metric used in [104]). This metric is not timing-independent for two reasons. First, the performance statistic, i.e., the number of cache hits, is timing-dependent in modern out-of-order processors. The reason is that the program timing can change the order of memory accesses, resulting in different cache states and affecting the number of cache hits. Second, the profiling history included in the window of T cycles is also timing-dependent.

To define a timing-independent metric, we must only use timing-independent performance statistics. Also, if the metric is defined on a window of execution, the history included in the window must not depend on program timing. In the example of cache partitioning, a timing-independent metric can be the memory footprint (i.e., the number of unique memory lines accessed) of the past *N retired* memory instructions, regardless of what level in the cache hierarchy the memory requests were served from.

Principle 2: Use a resizing schedule based on the progress of instruction execution (or "progress-based schedule" for short). This means that we tie the assessment points to when the program has made a certain progress (e.g., after N retired instructions)—not to when a certain time has elapsed. The reason is that if assessment points are tied to elapsed time, e.g., making an assessment after T cycles, then the utilization metric value at the point of assessment depends on what instructions the program can execute in T cycles, which depends on program timing. As a result, secret-dependent timing can still influence the resizing action taken at an assessment, even if a timing-independent metric is used. Figure 5.4 illustrates a time-based schedule that assesses at every T cycles and a progress-based schedule that assesses at every N retired instructions.

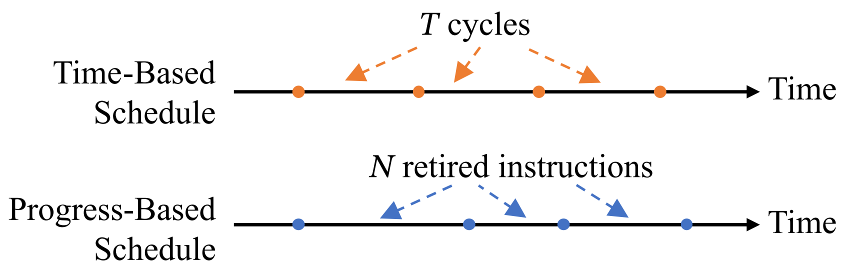


Figure 5.4: Comparison of a time-based schedule (used by prior work [104, 105, 213, 214]) and a progress-based schedule. Dots on the timelines are the times when assessments occur.

By following these two design principles, the resizing action sequence becomes timing-independent—i.e., it only depends on the sequence of retired dynamic instructions in an execution. This removes Edge ③ in Figure 5.2. With this property, if we can additionally ensure that the action sequence only depends on the *public* portion of the dynamic instruction sequence, we can also remove Edge ① and, therefore, completely eliminate the action leakage.

To make the action sequence only dependent on the *public* portion of the instruction sequence, we annotate all the instructions that use the resource under partitioning and are data- or control-dependent on secrets. Then, when measuring the utilization metric, we *exclude their contribution*. We also annotate any instructions that are control-dependent on secrets, irrespective of whether they use the resource. Then, the execution of these instructions is *not counted towards the execution progress.* Prior program analyses [109, 110, 111, 113] can be applied to find and annotate these two kinds of instructions. With this support, regardless of the values of secret inputs, the point in the execution where an assessment is made and the utilization metric value at that assessment point are independent of the said secrets. This removes Edge ①. Overall, the action sequence is now secret-independent. This means that, for a given public input, there is only one possible realizable action sequence s regardless of the secret inputs. Therefore, we have eliminated the action leakage.

5.5.3 Bounding Scheduling Leakage

The scheduling leakage $E[H(T_s|S = s)]$ is the expected value of the entropy of timing sequences T_s, for every possible action sequence s. Since there is only one possible action sequence s for a given public input due to the elimination of the action leakage in Section 5.5.2, then $E[H(T_s|S = s)] = H(T_s|S = s)$. Hence, the following discussion focuses on bounding $H(T_s|S = s)$ for the specific action sequence s that occurs at runtime for the given public input.

If we tied assessment points to elapsed time (e.g., a resize every T cycles), $H(T_s|S = s) = 0$ for any action sequence s and the scheme would not have scheduling leakage. But then it would have timing-dependent action leakage (Section 5.5.2). If, instead, we use a progress-based resizing schedule as discussed in Section 5.5.2, we eliminate the action leakage with the help of annotations. However, we still have timing-dependent scheduling leakage: when an assessment occurs leaks how much time the program takes to make a certain amount of execution progress.

It may seem that no matter whether a scheme ties assessment points to elapsed time or to progress, one cannot avoid analyzing program timing. To solve this problem, we propose a covert channel model that enables us to bound the *worst-case* scheduling leakage in an environment with a progress-based resizing schedule, without analyzing program timing. With this approach then, we have no action leakage and can compute a tightly bound of the scheduling leakage.

A covert channel assumes that both the sender (i.e., victim) and the receiver (i.e., attacker) are cooperative, while a side channel assumes that the sender is non-cooperative. Therefore, computing *the maximum data rate* of the more capable covert channel produces an upper bound of the scheduling leakage that occurs in the real environment with a non-cooperative victim. Overall, with this approach, we do not need to find exact realizable timing sequences nor consider Edges ② and ④ in Figure 5.2.

Next, we describe the model for the covert channel, the bound on the scheduling leakage, and optimizations to reduce the leakage. We assume a progress-based resizing schedule and that we have already eliminated the action leakage with annotations.

Understanding the Covert Channel. We observe that the leaked information is encoded as the *duration* of remaining in a certain observable state (i.e., using a certain partition size). To illustrate this observation, we revisit the code snippet in Figure 5.1c. The code snippet always decides to Expand after finishing the array traversal, but the timing of Expand is secret-dependent, as shown in Figure 5.5. Therefore, the example can be modeled as a covert channel that changes the current

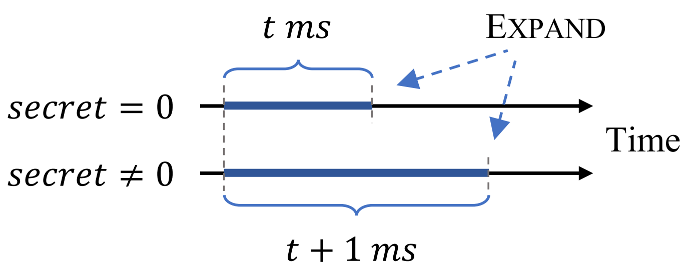


Figure 5.5: Timing of EXPAND for the code snippet in Figure 5.1c.

state (i.e., performs EXPAND) after t ms to transmit a symbol “0”, or after $t + 1$ ms to transmit a symbol “1”.

The sender can try various transmission strategies to amplify the leakage. For example, the sender can use more than two input symbols per transmission to increase the amount of data being transferred each time. Each symbol will be assigned a different time duration. The sender can also increase the time duration differences between symbols, to make the channel more resilient to potential noise—e.g., instead of using t ms and $t + 1$ ms to encode “0” and “1”, one can use t ms and $t + 10$ ms to make “0” and “1” more distinguishable under noise. Finally, the sender can tune the probability distribution of input symbols.

We do not limit the transmission strategy that a sender uses. Even in this case, the maximum data rate through the covert channel is still bounded because of a trade-off between the amount data per transmission and the average transmission time. Intuitively, this is because as the number of symbols increases or the time differences that distinguish these symbols increase, so does the average transmission time. It can be shown that, after a point, increasing the data per transmission results in a lower data transmission rate.

Example. The following two strategies illustrate the trade-off: (i) STRATEGY 1 uses 1 ms, 2 ms, 3 ms, and 4 ms to represent an alphabet of four symbols; and (ii) STRATEGY 2 uses 1 ms, 2 ms, ..., 8 ms to represent an alphabet of eight symbols. To simplify the discussion, we assume that all symbols are equally likely in both strategies. Then, STRATEGY 1 transmits $\log 4 = 2$ bits per transmission (i.e., the entropy of the four symbols), and the average transmission time is $(1 + 2 + 3 + 4)/4 = 2.5$ ms. STRATEGY 2 transmits $\log 8 = 3$ bits per transmission and the average transmission time is $(1 + 2 + ... + 8)/8 = 4.5$ ms. Comparing the data rates for both strategies, we see that STRATEGY 1’s data rate (2 bits/2.5 ms $=$ 800 bits/s) is higher than STRATEGY 2’s (3 bits/4.5 ms $\approx$ 667 bits/s), despite using fewer symbols.

Limiting the Maximum Data Rate.. Based on the previous intuitive explanation of the covert channel, and before presenting a formal model of it, we introduce two mechanisms to reduce the

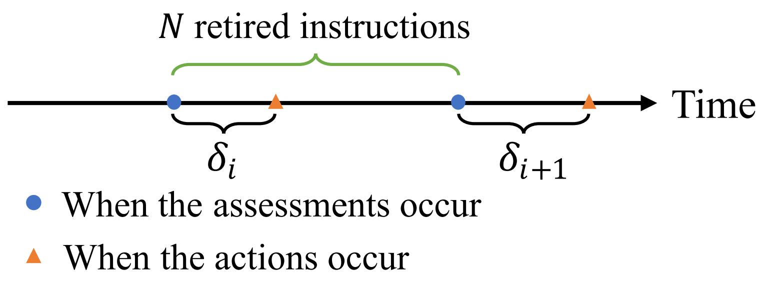


Figure 5.6: Delaying actions by a random amount of time.

maximum data rate of the covert channel (i.e., the upper bound of the scheduling leakage rate). One mechanism lowers the transmission rate and the other reduces the amount of data that the receiver (i.e., the attacker) can learn per transmission.

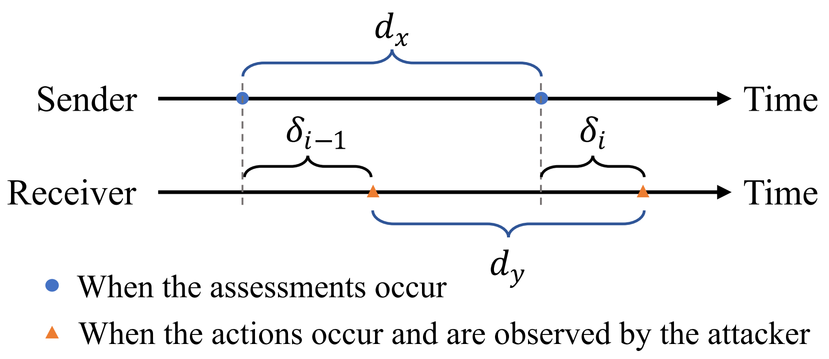
Mechanism 1: Set a minimum wait time, called the *Cooldown Time* (denoted by T_c), between two consecutive resizing assessments. Specifically, if an assessment occurs at t, then the scheme enforces that the next assessment cannot occur before $t + T_c$. This cooldown time helps reduce the transmission rate.

Once T_c is picked, the resizing schedule has to be aware of the value of T_c, and guarantee that the time between two consecutive assessments is never below T_c. For example, using a resizing schedule that makes assessments every N retired instructions, a possible approach is to set N to the maximum number of instructions that the core can possibly retire within T_c. If the core has a commit width of w, then $N = wT_c$ (assuming T_c is measured in cycles).

The cooldown time is set based on the security and performance goals: the longer the cooldown time is, the lower the leakage rate is, and the slower the program execution is.

Mechanism 2: Delay each resizing action by a random time δ after the corresponding assessment point. Intuitively, adding random delays "blurs" the differences between symbols and can introduce bit errors in the channel, thus reducing the amount of information learned by the receiver. This technique is shown in Figure 5.6. On the time axis, blue dots show when the assessments occur, and orange triangles show when the actions take place. Note that, right after Assessment i is made, we start counting progress towards Assessment $i + 1$. This ensures that the *action* taken at Assessment $i + 1$ is not influenced by program timing.

Formal Analysis. This section formalizes the proposed covert channel model and the two data rate reduction mechanisms just described. To be conservative, we reason about the upper bound of $H(T_s|S = s)$ for the worst-case action sequence s. The worst-case action sequence is the one that changes the partition size at every action, thus making the timing of every action visible to the attacker. Later, in Section 5.5.3, we will discuss how the model can be optimized when the sequence includes MAINTAIN decisions—as usual, assuming that only one action sequence is possible.



d_x
Sender
Time
δ_{i-1}
δ_i
Receiver
Time
d_y
● When the assessments occur
▲ When the actions occur and are observed by the attacker

Figure 5.7: Timeline of the sender and the receiver.

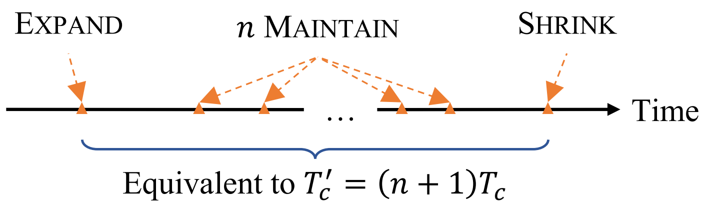


Figure 5.8: Optimizing the covert channel with MAINTAIN.

We assume that the resolution at which the attacker (i.e., receiver) can measure the time is finite. We represent timestamps with unit-less integers, with 1 time unit being the resolution.

As per Section 5.5.3, the information is encoded as the time duration of remaining in a certain partition size, and the sender uses different durations to represent different input symbols. Therefore, let X be a random variable that represents an input symbol. X takes values in a discrete input alphabet $\mathcal{X}$. An input symbol $x \in \mathcal{X}$ follows the input distribution $p(x)$. For each input symbol x, we use a unique time duration d_x to represent it. Since we enforce that two assessments must be at least T_c apart, $d_x \geq T_c$ for any x. Then, the average time for one transmission is:

$$T_{avg} = \sum_{x \in \mathcal{X}} p(x) d_x. \tag{5.5.7}$$

Lastly, if there are multiple transmissions (one per each assessment), we denote the input symbol in the ith transmission as X_i, and the input sequence used in n transmissions as $X^n = X_1, X_2, ..., X_n$.

On the receiver side, let Y be a random variable that represents an output symbol. Note that Y is not the same as X because of the random delay δ. Specifically, Y takes values in a discrete output alphabet $\mathcal{Y}$, which is determined by $\mathcal{X}$ and the distribution of the random delay δ (i.e., $p(\delta)$). For a specific input symbol x represented by time duration d_x, the time duration observed by the receiver, denoted by d_y, can be different from d_x due to the δ. If we denote the random delay in transmission i by δ_i, then

$$d_y = d_x + \delta_i - \delta_{i-1}, \tag{5.5.8}$$

as illustrated in Figure 5.7. Since each possible d_y is mapped to $y \in \mathcal{Y}$, the output distribution $p(y)$ can be computed from $p(x)$ and $p(\delta)$. Lastly, the output sequence received from n transmissions is denoted by $Y^n = Y_1, Y_2, ..., Y_n$.

The maximum amount of information that the receiver learns from n transmissions is $I(X^n; Y^n)$, the mutual information between X^n and Y^n (Equation 5.2.4). Also, the average time for n transmissions is nT_{avg}. Therefore, the data rate R of the covert channel is

$$R = I(X^n; Y^n)/nT_{avg}. \tag{5.5.9}$$

Different input distributions of $p(x)$ result in different values of $I(X^n; Y^n)$ and T_{avg}. Therefore, we are interested in the input distribution of $p(x)$ that produces the maximum data rate (R_{max}) of the covert channel. This value is an upper bound of the scheduling leakage rate of any victim program.

Optimized Covert Channel Model.. In Section 5.5.3, the covert channel model conservatively assumes the worst-case action sequence, where every action changes the partition size, thus making the timing of every action visible to the attacker. However, in practice, most resizing assessments result in MAINTAIN (as shown in Section 5.9), whose timing is invisible to the attacker.

Since the victim only has one possible action sequence s under *Untangle* for a given public input, we can leverage the MAINTAIN actions to optimize the covert channel model to reduce the maximum data rate. The idea is illustrated in Figure 5.8. If the victim chooses MAINTAIN *n consecutive* times, the execution is equivalent to a case when the two visible resizing actions that occur right before and right after these n MAINTAIN actions are separated by a longer cooldown time $T_c' = (n + 1)T_c$. Therefore, the scheduling leakage *during this period* is reduced because of the increased cooldown time. Consequently, we can monitor the number of consecutive MAINTAINS performed during a victim execution and lower the upper bound of the scheduling leakage rate of the execution.

Computing the Maximum Data Rate. Computing a closed-form of R_{max} is complex. Consequently, we show a numerical method that computes a tight upper bound of R_{max}. Recall that the maximum data rate R_{max} is:

$$R_{max} = \max_{p(x)}\{I(X^n; Y^n)/nT_{avg}\}, \tag{5.5.10}$$

where the maximization is taken over all possible input distributions $p(x)$.

The following discussion assumes that the random delay δ across transmissions is *independent and identically distributed (IID)*, and that the input symbol X follows that same input distribution $p(x)$ across transmissions. As a result, the output symbol Y follows the same output distribution $p(y)$ across transmissions.

To find the maximum data rate R_{max}, the first step is to compute the mutual information $I(X^n; Y^n)$. However, computing it directly from the mutual information definition is not feasible, since the number of transmissions n can be unbounded. Hence, we perform the following conservative simplification and approximation. By the definition of mutual information (Equation 5.2.4), we have

$$I(X^n; Y^n) = H(Y^n) - H(Y^n|X^n). \tag{5.5.11}$$

For the first term $H(Y^n)$, which is the joint entropy of $Y_1, Y_2, ..., Y_n$, we apply the chain rule [215]

$$H(Y^n) = H(Y_1) + \sum_{i=2}^{n} H(Y_i|Y^{i-1}) \tag{5.5.12}$$

$$\leq \sum_{i=1}^{n} H(Y_i) = nH(Y). \tag{5.5.13}$$

For the second term $H(Y^n|X^n)$, by the definition of the conditional entropy (Equation 5.2.3), we have

$$H(Y^n|X^n) = -\sum_{x^n \in \mathcal{X}^n} \sum_{y^n \in \mathcal{Y}^n} p(x^n, y^n) \log p(y^n|x^n) \tag{5.5.14}$$

$$= -\sum_{x^n \in \mathcal{X}^n} \sum_{\delta^n \in \Delta^n} p(x^n, \delta^n) \log p(\delta^n|x^n) \tag{5.5.15}$$

$$= -\sum_{x^n \in \mathcal{X}^n} \sum_{\delta^n \in \Delta^n} p(x^n) p(\delta^n) \log p(\delta^n) \tag{5.5.16}$$

$$= -\sum_{\delta^n \in \Delta^n} p(\delta^n) \log p(\delta^n) \tag{5.5.17}$$

$$= H(\delta^n) = nH(\delta), \tag{5.5.18}$$

where Equation 5.5.15 substitutes y^n with δ^n because y^n is a function of δ^n and x^n, Equation 5.5.16 holds because δ^n and x^n are independent, and Equation 5.5.18 holds because random delays are IID.

Therefore,

$$I(X^n; Y^n) = H(Y^n) - H(Y^n|X^n) \leq n(H(Y) - H(\delta)). \tag{5.5.19}$$

Using Equation 5.5.19, we can conservatively approximate $I(X^n; Y^n)$ without considering the whole sequences of X^n and Y^n. Hence, the goal becomes finding

$$R'_{max} = \max_{p(x)} \{(H(Y) - H(\delta))/T_{avg}\} \tag{5.5.20a}$$

$$\text{subject to} \quad \sum_{x} p(x) = 1, p(x) > 0 \tag{5.5.20b}$$

over all possible input distributions $p(x)$. R'_{max} is an upper bound of R_{max}. The optimization problem 5.5.20 fits the standard single-ratio *fractional programming* (FP) problem [222]. *Dinkelbach's transform* [223] can iteratively converge to the optimal input distribution $p(x)$ that achieves R'_{max}.

Dinkelbach's transform [223]. To simplify the discussion, we first consider a general FP problem

$$\begin{aligned} \underset{z}{\text{maximize}} \quad & N(z)/D(z) && (5.5.21a) \\ \text{subject to} \quad & z \in \mathcal{Z}, && (5.5.21b) \end{aligned}$$

where $N(z)$ and $D(z)$ are continuous and real-valued functions of z. Moreover, $D(z) > 0$ for all $z \in \mathcal{Z}$.

To solve Problem 5.5.21, Dinkelbach's transform introduces an auxiliary variable q and a helper function given by

$$F(q) = \max_{z \in \mathcal{Z}} \{N(z) - qD(z)\}. \qquad (5.5.22)$$

The algorithm then iteratively updates q according to the following steps (with the value of q in the ith iteration denoted by q_i):

1. Set $q_1 = 0$ and $i = 1$.
2. Solve $F(q_i)$ for z_i over $z \in \mathcal{Z}$.
3. Update $q_{i+1} = N(z_i)/D(z_i)$, increment i, and go to Step 2.

The algorithm iterates n times until either $F(q_n) < \epsilon$, where ϵ is a positive real number representing the tolerance, or n reaches the maximum number of iterations. Subsequently, z_n can be used as an approximate solution to Problem 5.5.21, and q_n is an approximation of $\max_{z \in \mathcal{Z}}\{N(z)/D(z)\}$.

To find a tight upper bound for $\max_{z \in \mathcal{Z}}\{N(z)/D(z)\}$ using the iterative solution q_n, we observe that $F(q)$ is strictly monotonic decreasing with respect to q [223]. Furthermore, it can be proven that $q^* = \max_{z \in \mathcal{Z}}\{N(z)/D(z)\}$ if and only if $F(q^*) = 0$ [223]. Therefore, we can guess an upper bound $q' = q_n + \delta$, where δ is a small positive real number. If we verify that $F(q') \leq 0$, then we know that $q' \geq q^*$ since $F(q)$ is strictly monotonic decreasing. Otherwise, we increase δ and repeat the process.

Our implementation. We apply Dinkelbach's transform to solve Problem 5.5.20 and find an upper bound of R'_{max}. To do so, we need to find a distribution $p(x)$ that maximizes $(H(Y) - H(\delta)) - q_i T_{avg}$ for each iteration (Step 2 of the algorithm). We begin with analyzing the concavity of the target function by examining its individual components. The first term, $H(Y)$, is the entropy of the output symbols and is a concave function of $p(y)$ [215]. Since $p(y)$ is a linear function of $p(x)$ (i.e., $p(y) = \sum_x p(y|x)p(x)$), $H(Y)$ is also a concave function of $p(x)$. The second term $H(\delta)$ is a constant for a given random noise distribution. The last term, T_{avg}, is a linear function of $p(x)$ by the definition of T_{avg} (Equation 5.5.7). Therefore, the target function $(H(Y) - H(\delta)) - q_i T_{avg}$ is concave and can be optimized with a standard concave programming method.

We implement the optimization using PyTorch's [224] Adam optimizer [225] and have observed convergence. We then guess a tight upper bound of R'_{max} with $q' = q_n + \delta$ and use the same optimizer to empirically verify that $F(q') < 0$ after 10,000 iterations. We leave proving $F(q') < 0$ as an open problem for future work.

5.6 DISCUSSION

In this section, we describe some aspects related to the operation of *Untangle*.

5.6.1 Timing-Dependent Dynamic Instruction Sequences

In Section 5.5.2, we removed Edge ③ in Figure 5.2 and made the resizing action sequence depend not on program timing but only on the retired dynamic instruction sequence of the execution. However, in some cases, this is not enough to eliminate the effect of timing because the dynamic instruction sequence *itself* depends on program timing. This case may happen, e.g., in parallel programs, where the timing of when a thread attempts a synchronization operation may result in different outcomes: repeated spinning or proceeding past the synchronization. It may also happen in single-threaded programs, where the thread may check the current time and take different paths based on the result.

To handle this case, the code regions with these timing-dependent dynamic instruction sequences need to be annotated, so that one can exclude their contribution when measuring the utilization metric and exclude their instructions when quantifying execution progress. The techniques used by existing analysis tools that detect secret-dependent control and data flow in a program can be used as a basis to identify and annotate these timing-dependent sequences. For example, one can treat the data read inside a critical section or the return value of a get-time system call as a secret. We consider any further analysis of this issue the subject of future work.

5.6.2 Other Attacks

A powerful attacker can replay the victim program many times, gaining additional information at every replay from the scheduling leakage. However, the operating system can use the upper bound of the victim program's leakage rate as computed by *Untangle* (Equation 5.5.9) to keep accumulating the victim program leakage across the multiple runs. When the accumulated leakage across runs reaches a user-defined threshold, the system prevents the victim program from performing any further resizes. From then on, the performance of the program will decrease, but there will be no more leakage.

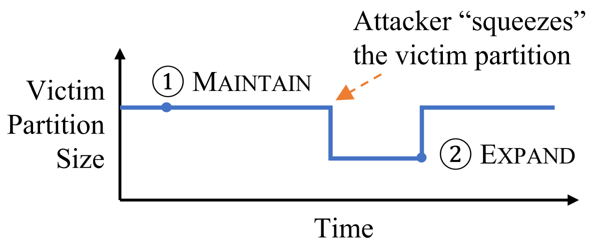


Figure 5.9: An active attacker that interacts with the victim.

The attacker can also actively interact with the victim, as illustrated in Figure 5.9. In this example, the victim is in a steady state and decides to MAINTAIN at assessment ①. Since MAINTAIN is invisible to an attacker, our optimized covert channel model can leverage it to further bound the scheduling leakage (Section 5.5.3). However, before the victim makes the next assessment, the attacker can put a high pressure on the shared resources to "squeeze" the victim partition. This strategy can force the victim to perform an attacker-visible action EXPAND at the next assessment (②), incurring a higher *scheduling* leakage rate. As a result, the user-defined leakage threshold will be reached sooner, which can disable further resizing and hurt execution performance, but cannot violate the security guarantees discussed in Section 5.4, as the leakage will not exceed the threshold. Note that an active attacker cannot cause *action* leakage in *Untangle*. The reason is that even if the action sequence changes, it is not due to secret values and, therefore, there is no action leakage. The change is due to the attacker actions, which cannot affect the victim's timing-independent resource utilization metric at the points of assessment.

5.6.3 Partitioning Other Hardware Resources

In Section 5.3 and 5.5, we mainly use the last-level cache (LLC) as an example of resource of interest. However, *Untangle* is a general framework and it can be applied to different hardware resources. To apply *Untangle* to a new type of resource, we first need to define a timing-independent utilization metric for that resource (Section 5.5.2). For example, we can trivially extend the LLC utilization metric to the TLB. Another example of resource of interest is functional units shared by two SMT threads [213, 226], where we can use the fraction of the retired instructions that utilize a certain type of function unit as a metric. Next, we also need to extend the static analysis to identify the secret-dependent usage of the new resource of interest. For the TLB, we can reuse the static analysis for caches [109, 110, 111, 113]; for function units, an analyzer that detects secret-dependent control flow suffices.

5.6.4 Extending the Threat Model

Untangle assumes a peer security model, where all programs are mutually distrusting and in the same security level (Section 5.4). It is possible to extend *Untangle* to support a more complex security lattice that is also tiered. Under this model, information flow from a lower-tiered program (*L*) to a higher-tiered program (*H*) is allowed, but not vice versa. As a result, program *L* can take resizing actions that claim resources from or free resources to *H* without counting towards the leakage thresholds of both programs. However, there is one caveat: *L*'s resizing can affect the timing of *H* due to the change of available resources. This timing change in *H* can be secret-dependent and observed by *L* through other observable events (e.g., termination of *H*). *Untangle*'s covert channel model can be adapted to measure this type of leakage.

5.6.5 Using Existing Static Analyses for Annotation

Recall from Section 5.5.2 that we need to annotate all instructions that have secret-dependent resource usage and all instructions that are control-dependent on secrets. The main challenge of using existing analyses [109, 110, 111, 113] for annotation is their scalability, since they use static analysis to ensure soundness. According to the literature, all these analyses can analyze cryptography libraries. Besides that, Cacheaudit [111] also analyzes sorting primitives and Casym [109] analyzes database applications (e.g., PostgreSQL [227]). These libraries and applications can process sensitive information. For applications that are beyond the capability of these tools, one can use manual inspection assisted by these tools to generate conservative but sound results for *Untangle*—e.g., by applying analyses on a manually-selected portion of the program. In this case, the performance of applications may decrease due to the conservativeness of the analysis.

5.7 HARDWARE IMPLEMENTATION

This section discusses a potential implementation of key aspects of the *Untangle* hardware. Similar to the previous discussion, we use LLC partitioning as an example. We do not explore a full implementation because the focus and the novelty of this chapter is in the *Untangle* framework.

Transmitting the Annotations to the Hardware. There are several possible ways to transmit the annotations to the underlying *Untangle* hardware. Intuitively, we can re-purpose a currently-unused instruction prefix to mark the relevant instructions. A similar approach is used by Intel for lock elision [130]. However, this approach can generate bloated binaries if many instructions are annotated. An alternative approach is to introduce two new instructions that flag the start and the end of a secret-dependent code region. Finally, we can also introduce a special bit in the page table to coarsely annotate pages that contain secret-dependent code [228]. The latter approach does not require recompilation and can be applied to legacy programs.

Monitoring LLC Utilization. Many prior works have proposed various ways of monitoring LLC utilization [104, 105, 214]. We describe one possible mechanism that we use in the evaluation. The mechanism is similar to UMON [104], which assumes that the partition size is chosen from a pre-defined list of supported sizes. At a high level, for each domain, at runtime, the mechanism simulates memory accesses with each possible partition size, and measures the corresponding number of LLC hits. Then, during a resizing assessment, the monitor picks the size for each domain that maximizes the number of LLC hits across all domains. To satisfy the requirements of a timing-independent utilization metric, the monitor does not consider memory instructions that are data- or control-dependent on secrets. In addition, it only considers the memory accesses resulting from retired memory instructions and in program order (Section 5.5.2).

The proposed mechanism can be implemented with a set-associative hardware table that selectively simulates memory accesses to only certain cache sets. This hardware table only contains tags but not data. When a public load or store to one of the monitored sets retires, the table is accessed. Memory accesses that would hit in the private caches are filtered out.

Measuring Scheduling Leakage at Runtime. Recall from Section 5.5.3 that we leverage consecutive MAINTAIN actions, which are invisible to the attacker, to further reduce the bound on scheduling leakage rate. However, it is impractical to compute the optimized scheduling leakage rate at runtime, since it needs to generate a new R_{max}, and this involves a computation-intensive algorithm (Section 5.5.3). Therefore, we use a small hardware table that stores pre-computed leakage rates. Specifically, table entry i stores the leakage rate R_{max_i}, corresponding to when i consecutive MAINTAINS occur. At runtime, if the victim chooses MAINTAIN m consecutive times, we conservatively assume that the next action is not MAINTAIN and use the rate R_{max_m} to compute the leakage for that resizing. If the next action turns out to be another MAINTAIN, we switch to the lower rate $R_{max_{m+1}}$. Finally, if m exceeds the table capacity, we conservatively use the rate of the entry for the maximum number of MAINTAINS considered.

5.8 EXPERIMENTAL METHODOLOGY

We use last-level cache (LLC) partitioning as an example to demonstrate that *Untangle* can offer flexibility and better performance than static partitioning, and significantly less leakage than prior dynamic partitioning schemes. We choose the LLC as the resource of interest because it is a commonly-exploited hardware resource, and there are many prior LLC partitioning schemes [57, 59, 60, 61, 104].

Table 5.3: Parameters of the simulated architecture.

Parameter	Value
Architecture	8 out-of-order x86 cores at 2.0 GHz
Core	8-issue, 4-commit, no SMT, 72 load queue entries, 56 store queue entries, 224 ROB entries, LTAGE branch predictor
Private L1-I & L1-D cache	32 kB, 64 B line, 8-way, 2 cycle round trip (RT) latency
Shared L2 cache (LLC)	16 MB (2 MB per slice), 64 B line, 16-way, 8 cycles RT latency
DRAM	50 ns RT latency after L2
Supported partition sizes for a domain	128 kB, 256 kB, 512 kB, 1 MB, 2 MB, 3 MB, 4 MB, 6 MB, 8 MB
Monitor window M_w	1 M LLC references

Table 5.4: Partitioning schemes evaluated.

Scheme	Description
STATIC	Static partitioning. Each domain uses a 2 MB partition
TIME	Dynamic partitioning. Assessing resizing every 1 ms
UNTANGLE	Dynamic partitioning. Assessing resizing every 8 M retired instructions with a cooldown time of 1 ms
SHARED	No partitions. All domains share the 16 MB LLC

Following prior work [57, 59, 60], we use set partitioning. We assume a simple design where the size of a partition is chosen from a pre-defined list of 9 choices. We use the mechanism discussed in Section 5.7 to monitor LLC utilization and guide resizes. The monitor only considers the past M_w LLC references made by retired memory instructions at the time of an assessment, to focus on the program's most recent LLC utilization.

Configurations & Schemes. We model an 8-core system (Table 5.3) using cycle-level simulations with gem5 [230]. We consider four LLC partitioning schemes (Table 5.4). The baseline scheme is STATIC, which partitions the LLC statically, giving 2 MB to each domain. We evaluate two dynamic partitioning schemes: TIME is similar to previous ones [104, 105, 213, 214] that make resizing assessments at a fixed time interval (1 ms interval in our configuration); UNTANGLE applies our mitigations described in Section 5.5. UNTANGLE makes resizing assessments every 8 M retired instructions and its minimum wait time between resizes is 1 ms. We use this configuration for UNTANGLE to match the performance of TIME by performing resizing assessments at a similar frequency. The random delay in UNTANGLE follows a uniform distribution between $[0, 1\,\text{ms})$. Both TIME and UNTANGLE start with a partition size of 2 MB. Finally, SHARED is an insecure configuration that uses a shared LLC without partitioning.

Table 5.5: OpenSSL [229] cryptographic benchmarks.

Name	**Description**
Chacha20	Stream cipher, encrypt 10 kB payloads
AES-128	Block cipher with a 128-bit key, encrypt 10 kB payloads
AES-256	Block cipher with a 256-bit key, encrypt 10 kB payloads
SHA-256	Digest function, compute 10 kB payloads
RSA-2048	RSA signing with a 2048-bit key
RSA-4096	RSA signing with a 4096-bit key
ECDSA	ECDSA signing using curve Secp256k1
EdDSA	EdDSA signing using curve Ed25519

Workloads. To evaluate workloads that have both secret-related and public parts, we build workloads composed of one SPEC17 benchmark [231] and one cryptographic benchmark from OpenSSL 3.0.5 [229] (Table 5.5). Both benchmarks share the same domain and hence use the same LLC partition. Since we target a typical workload that spends most of its execution time in the public part, we repeatedly run in a loop 1 M instructions from the cryptographic benchmark and then 10 M instructions from the SPEC17 benchmark. Both benchmarks make forward progress. We conservatively assume that all instructions from the cryptographic benchmark are secret-dependent. We do not set a leakage threshold for a workload; we allow it to freely resize and then measure its leakage.

For SPEC17, we use the reference input size. We simulate all 36 SPEC17 benchmarks.[2] For each SPEC17 benchmark, we use SimPoint [232] to select a representative slice of 500 M instructions. We study each SPEC17 benchmark's sensitivity to LLC size by running it with every supported partition size and normalize its instruction-per-cycle (IPC) to the IPC with an 8 MB partition (i.e., the maximum partition size). The study is detailed in Section 5.12. We neglect the crypto benchmarks because they have much smaller LLC use. We then define the *adequate LLC size* of a benchmark as the *minimal* LLC size that allows the benchmark to reach a normalized IPC of at least 0.9. If a benchmark has an adequate LLC size higher than 2 MB (i.e., the STATIC partition size), we classify it as *LLC-sensitive* (8 benchmarks in total); otherwise, it is *LLC-insensitive* (28 benchmarks in total).

Since we simulate an 8-core system, we first randomly select a mix of eight workloads (2 LLC-sensitive and 6 LLC-insensitive). Then, from this base mix, we randomly replace two LLC-insensitive workloads with two LLC-sensitive ones to generate a new mix. We repeat this change until there are no LLC-insensitive workloads in the mix. We run our experiments on each mix. Next, we repeat this process with different base mixes to cover all possible workloads. For each experiment, we warm up the system for 5 ms. Then, we simulate the mix of workloads until each

[2]The same SPEC application with another input is a different benchmark.

workload finishes its slice (500 M instructions from SPEC and 50 M from crypto). When a workload finishes, if there are other running workloads in the system, the finished workload maintains its pressure on the LLC, but does not update the statistics that we collect.

Measuring the Leakage. We measure the leakage in TIME with $\log |\mathcal{A}|$ bits per assessment, where $|\mathcal{A}|$ is the number of supported resizing actions (Section 5.3.3). For UNTANGLE, we use the leakage model proposed in Section 5.5 with the optimization discussed in Section 5.5.3.

We compare TIME with 1 ms assessment interval against UNTANGLE with $T_c = 1$ ms, which corresponds to 8 M retired instructions between assessments. We report the *leakage per assessment* of a workload under each scheme. Leakage per assessment determines the number of assessments that a workload is allowed under a leakage threshold. The lower the leakage per assessment is, the more assessments the scheme can make. Because TIME and UNTANGLE use different resizing schedules, the same workload can have different number of resizing assessments under different schemes, in spite of running the same number of instructions. The total leakage from an execution is proportional to the number of resizing assessments during the execution.

5.9 EVALUATION

We evaluate 16 workload mixes in total. Due to the similarity between mixes, we only show in Figure 5.10 the results of 4 selected mixes. Section 5.12 includes the results of the rest of mixes.

In Figure 5.10, each group of three charts corresponds to one mix. The top-left group (Mix 1) is for a mix with 2 LLC-sensitive workloads (shown in bold). As we move from left to right (Mix 2), and then from top to bottom (Mix 3 and Mix 4), we replace 2 non LLC-sensitive SPEC17 benchmarks with 2 LLC-sensitive SPEC17 benchmarks, until we reach a mix with all 8 LLC-sensitive benchmarks. In the title of each group, we show the total *LLC demand* as the sum of the adequate LLC size of all the workloads in the mix. In a given group, the bottom-most chart shows the IPC of every workload and scheme—normalized to STATIC. The middle chart shows the leakage per assessment in bits for TIME and UNTANGLE. Finally, the topmost chart shows the distribution of partition size measured at intervals of 100 µs. In that chart, the thick short bar covers the first to third quartile range; the thin long bar is the minimum and maximum range; the white dot is the median of the partition sizes. Note that the figure has a non-linear y-axis, and each dashed horizontal line corresponds to a supported partition size listed in Table 5.3.

Consider the top-left group of charts in Figure 5.10. It shows Mix 1, which has 2 LLC-sensitive workloads and a total LLC demand from all the 8 workloads equal to 14.6 MB. There is enough LLC for every workload. Under both TIME and UNTANGLE, the two LLC-sensitive workloads,

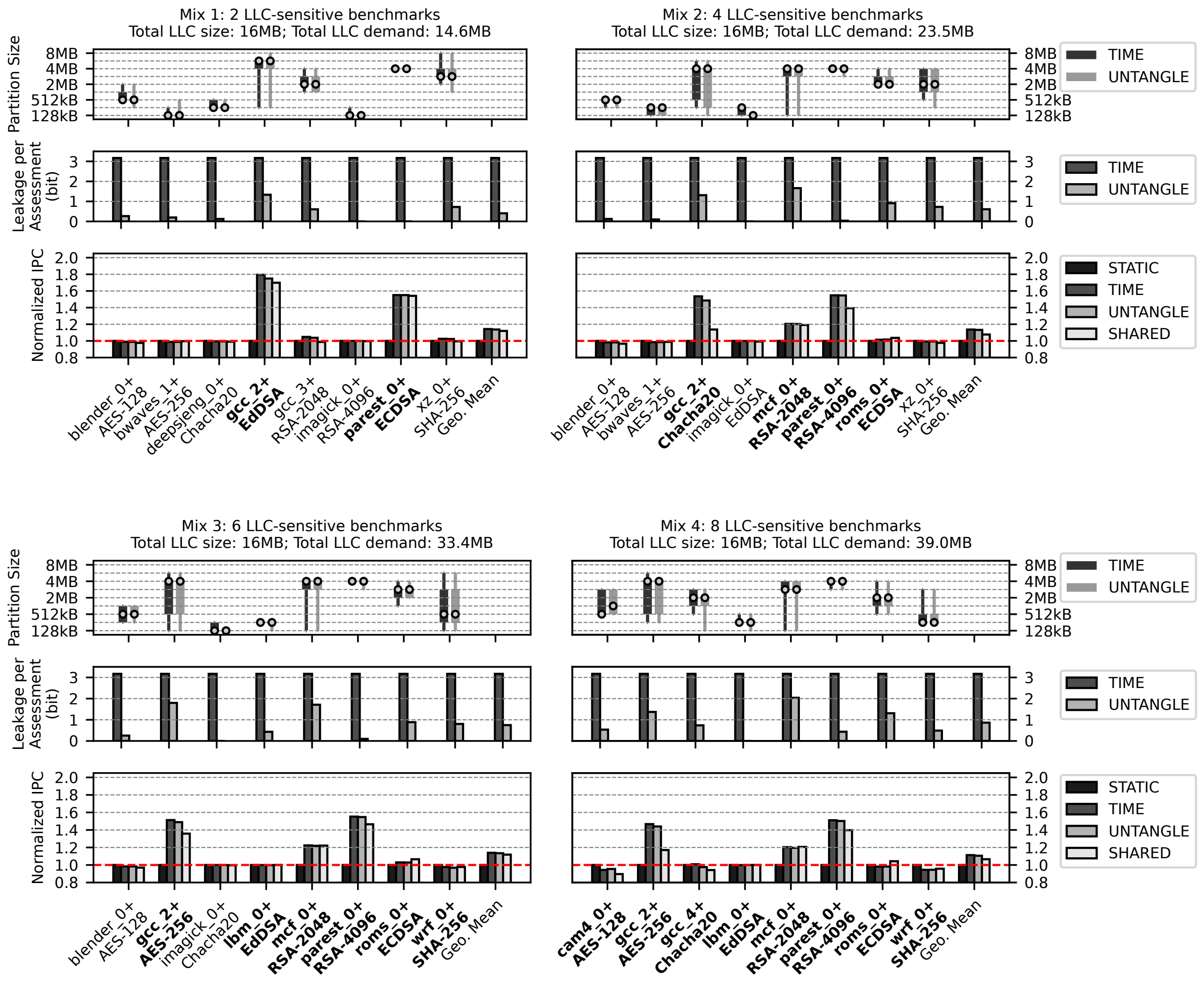


Figure 5.10: Comparing different partitioning schemes according to: IPC normalized to Static (bottom-most row of each group of three charts), leakage per assessment (middle row), and distribution of partition size (topmost row). The bars correspond to different workloads, where bolded workloads are LLC-sensitive.

`gcc_2`[3] and `parest_0`, attain high speedups over Static, and even slightly outperform Shared. The remaining LLC-insensitive workloads experience no slowdown, in spite of some of them using partitions smaller than the 2 MB of Static (see top chart). Overall, the system-wide speedup (i.e., the geometric mean of IPCs) of Time and Untangle over Static is 1.14. Shared (i.e., no partitioning) has a lower speedup of 1.12 because of cache conflicts between workloads.

Since the dynamic partitioning scheme in our evaluation supports 9 different actions (Section 5.8), Time leaks $\log 9 \approx 3.2$ bits per assessment for every workload (middle row chart). Based on our measurements, the leakage per assessment in Untangle is at most 1.3 bits per assessment, and on average 0.4 bits per assessment.

[3]To refer to a workload, we use the SPEC17 application name plus a number for the input.

Consider now the top-right group of charts in Figure 5.10. It shows Mix 2, which replaces `deepsjeng_0` and `gcc_3` from Mix 1 with `mcf_0` and `roms_0`. The Mix 2 workloads demand 23.5 MB of LLC in total. Due to the increased total LLC demand, `gcc_2` receives a smaller partition than in Mix 1, with a median size of 4 MB (top chart). As a result, its speedup is now lower than in Mix 1, namely 1.54 under TIME and 1.49 under UNTANGLE. `parest_0` and `mcf_0` attain good speedups, but `roms_0` does not translate its higher cache use into performance. Overall, TIME and UNTANGLE deliver a system-wide speedup of 1.14 and 1.13 respectively, while SHARED has a speedup of 1.08. Lastly, UNTANGLE has a leakage per assessment of 1.7 bits at most and 0.6 bits on average, while TIME leaks 3.2 bits per assessment.

The bottom-left group of charts in Figure 5.10 shows Mix 3, which includes two more LLC-sensitive workloads: `lbm_0` and `wrf_0`. These 8 workloads demand 33.4 MB of LLC, which is more than twice the available LLC. In this over-committed setting, three LLC-sensitive workloads have their demand fulfilled under TIME and UNTANGLE: `gcc_2`, `parest_0`, and `mcf_0`. Also, the rest of workloads do not suffer slowdown when compared to STATIC. Overall, TIME, UNTANGLE, and SHARED deliver a system-wide speedup of 1.14, 1.13, and 1.12 respectively.

For the Mix 2 and 3 workloads, we start to see an increase of leakage per assessment under UNTANGLE. The reason is that the workloads have more attacker-visible resizing actions that change the partition size, due to the high LLC pressure. In Mix 3, UNTANGLE has a maximum leakage per assessment of 1.8 bits and 0.7 bits on average.

In Mix 4 (bottom-right group of charts), all 8 workloads are LLC-sensitive. They demand a total LLC of 39.0 MB. Under this extreme LLC pressure, TIME and UNTANGLE can still fulfill three LLC-sensitive workloads. However, some of the remaining workloads start to suffer slowdown. Overall, both TIME and UNTANGLE have a system-wide speedup of 1.11, while SHARED has a lower speedup of 1.07. The leakage per assessment in UNTANGLE increases to 2.0 bits in the worst-case workload and 0.9 bits on average. As usual, TIME leaks 3.2 bits per assessment.

To summarize, TIME and UNTANGLE provide nearly the same speedups over STATIC, but UNTANGLE leaks information at a significantly lower rate. SHARED delivers slightly lower speedups due to cache conflicts between workloads. In UNTANGLE, it can be shown that, of all reassessments across mixes 1–4, 91% are MAINTAIN.

Total Leakage. Table 5.6 summarizes the leakage of the selected mixes under TIME and UNTANGLE. The table shows both the average leakage per assessment and the average total leakage per workload. Across the mixes, the leakage per assessment under UNTANGLE is 78% lower than under TIME.

Leakage of UNTANGLE under an active attacker. A powerful active attacker can put high pressure on the shared LLC, forcing the victim to make an attacker-visible resizing action at every single

Table 5.6: Leakage of Mixes 1-4 under Time and Untangle.

	Time		Untangle	
	Avg. leakage per assessment	Avg. total leakage	Avg. leakage per assessment	Avg. total leakage
Mix 1	3.2 bits	637.6 bits	0.4 bits	42.3 bits
Mix 2	3.2 bits	829.7 bits	0.6 bits	57.7 bits
Mix 3	3.2 bits	979.9 bits	0.7 bits	72.7 bits
Mix 4	3.2 bits	1,084.1 bits	0.9 bits	85.6 bits

assessment (Section 5.6.2). Then, the victim leaks at a higher rate. To study this environment, we measure the leakage under Untangle without the optimized covert channel model of Section 5.5.3. We find that the average leakage per assessment is 3.6 bits, averaged across all the workloads from all the mixes. This leakage is higher than with the optimization (0.7 bits). This worst-case leakage rate is very rare in a benign execution. However, even if it occurs, *Untangle still upholds the security guarantees*: at this increased leakage rate, the user-defined leakage threshold will be reached sooner, which will disable further resizings and, at worst, only hurt performance. This is not a limitation of *Untangle*, since an *active* attacker can always slow down the victim by forcing it to use the smallest partition.

5.9.1 Sensitivity Study

In this subsection, we conduct two additional sensitivity studies by varying the resizing interval and the leakage budget of dynamic LLC partitioning schemes, including Time and Untangle.

Resizing intervals. Recall that in our default configuration, Time resizes every 1 millisecond and Untangle resizes every 8 million retired instructions (Section 5.8). In this study, we examine the system's behaviors under a longer resizing interval. Specifically, we scale the resizing intervals of both Time and Untangle by a factor of s—i.e., Time resizes every s milliseconds and Untangle resizes every $8s$ million retired instructions. Additionally, we scale the monitoring window M_w to s million LLC references. Finally, for Untangle, we set the minimum wait time between resizes to s milliseconds and sample the random resizing action delay from a uniform distribution between $[0, s$ milliseconds). We use $s \in \{1, 2, 4, 7.5, 10\}$ in this study.

Figure 5.11 shows the system-wide speedup of Mixes 1–4 under Time and Untangle when varying the resizing interval scale s. Each cluster of bars corresponds to a specific mix and scheme. For bars within each cluster, the resizing interval scale s increases from the left to right. Overall, for the SPEC17 workloads that we used, both Time and Untangle have only small fluctuations in speedups when increasing s. This is because each SPEC17 benchmark runs for about ten minutes on native machines while the simulation can only run for hundreds of milliseconds due to the

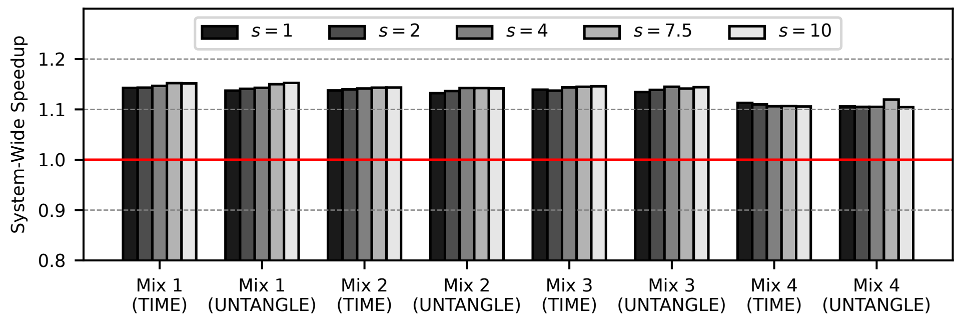


Figure 5.11: System-wide speedup over Static under different mixes and schemes when scaling the resizing intervals by s.

resource and simulator constraints. As a result, many SPEC17 benchmarks have relatively stable cache usage behaviors during the simulation period, making these benchmarks insensitive to the resizing interval. Note that some mixes show even slight increases in speedups under a larger s. Upon inspection, this is mainly due to the increased monitoring window M_w under a larger s. This increased M_w helps better capture LLC demand of LLC-sensitive workloads, such as `gcc_2`, thus granting them more LLC space.

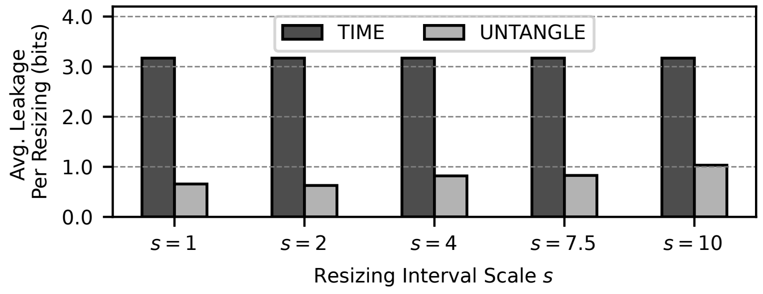


Figure 5.12: Leakage per assessment averaged across Mixes 1–4 under Time and Untangle when scaling the resizing intervals by s.

Figure 5.12 shows the leakage per resizing averaged across Mixes 1–4 under Time and Untangle when varying the resizing interval scale s. As usual, Time leaks 3.2 bits per assessment, regardless of s. For Untangle, the average leakage per assessment increases from 0.7 bits at $s = 1$ to 1.0 bits at $s = 10$. The reason for the increased leakage is that at a longer resizing interval, Untangle is less likely to choose to maintain the current partition size, so there are fewer chances to apply the optimized covert channel model discussed in Section 5.5.3. Indeed, the portion of assessments that result in a Maintain decision decreases from 91% at $s = 1$ to 78% at $s = 10$.

Figure 5.13 shows the total leakage averaged across Mixes 1–4 under Time and Untangle when varying the resizing interval scale s. The y-axis of the figure uses a logarithmic scale. As s increases from 1 to 10, the total leakage of both Time and Untangle is dramatically reduced from 880.1 bits

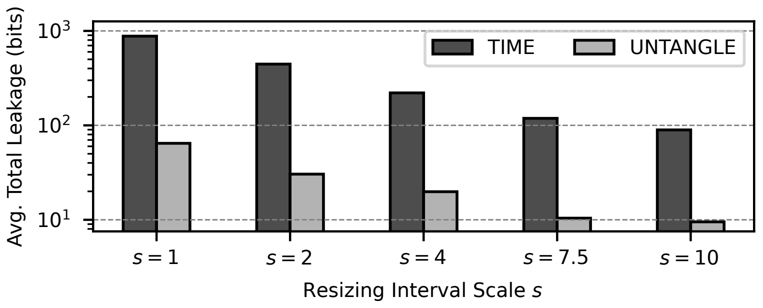


Figure 5.13: Total leakage averaged across Mixes 1–4 under TIME and UNTANGLE when scaling the resizing intervals by s. The y-axis uses a logarithmic scale.

to 89.6 bits and from 64.5 bits to 9.5 bits respectively. Overall, despite the increased leakage per assessment at a larger s, UNTANGLE still has much less leakage per assessment and total leakage than TIME at any evaluated s, while maintaining a speedup similar to TIME.

Note that, at all resizing interval scales, we observed unstable resizing behaviors that many assessments result in unnecessary resizings—e.g., an application repeatedly gains and then quickly loses a small amount of cache space. This behavior increases the scheduling leakage in UNTANGLE but provides minimal performance gain. In Section 5.9.2, we discuss a preliminary investigation into its main cause and a simple solution that effectively mitigates this behavior. As a result, the leakage reduction of UNTANGLE over TIME could be even bigger.

Leakage budget. The discussion so far assumes a scenario in which no leakage budget is set. As a result, we allow all applications to freely resize and report the leakage per assessment. In this study, we examine the speedup of TIME and UNTANGLE under a fixed budget. When an application exhausts its budget, it is forced to use its fair share of the LLC, which is 2 MB in our setup, and the application is prohibited from further resizing. An alternative policy to handle budget exhaustion would be to simply maintain its partition size at the time of exhaustion and stop further resizing. However, we prefer and use the policy of resetting the partition size to the fair share. This is because in the alternative policy, if an application A occupies a large LLC partition at the time of exhaustion, no other applications could ever claim any space from A until A terminates, even if A later might not need such a large partition, leading to resource starvation. To obtain the trade-off between speedup and leakage, we vary the leakage budget $b \in \{10\,\text{bits}, 25\,\text{bits}, 50\,\text{bits}, 100\,\text{bits}, \infty\}$. As using a short resizing interval can lead to rapid exhaustion of the leakage budget, especially under TIME, we scale the resizing interval by $s = 10$ in this study.

Figure 5.14 shows the system-wide speedup of Mixes 1–4 under TIME and UNTANGLE when varying the leakage budget b. In Figure 5.14, each cluster of bars corresponds to a specific mix and scheme. For bars within each cluster, the leakage budget b increases from left to right, with the rightmost bar representing the scenario where no leakage budget is set ($b = \infty$). Looking at the

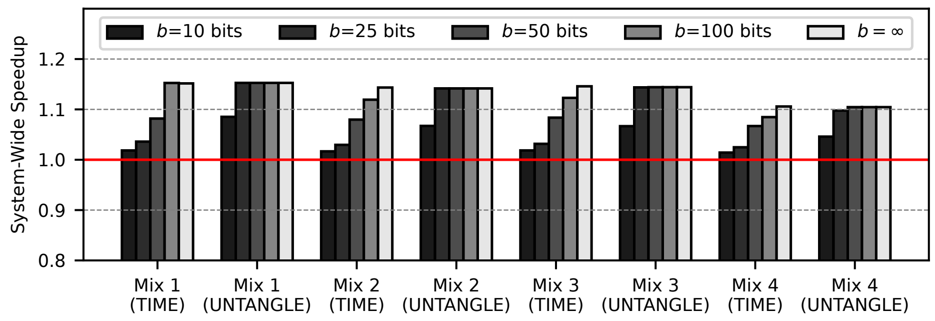


Figure 5.14: System-wide speedup over Static under different mixes and schemes when varying the leakage budget b and keeping $s = 10$.

bars corresponding to Time, we can see that the system-wide speedups are low when the leakage budget is small. Indeed, with $b = 10$ bits, Time achieves speedups of merely 1.02, 1.02, 1.02, and 1.01 in Mixes 1–4 respectively. This is because $b = 10$ bits permits Time only three resizing assessments in our configuration. Only when $b = 100$ bits, the speedups of Time can match or become close to the speedups under no leakage budget ($b = \infty$), depending on specific mixes.

Now looking at the bars corresponding to Untangle, even under a tight budget of $b = 10$ bits, Untangle achieves speedups of 1.09, 1.07, 1.07, and 1.05 in Mixes 1–4. With a budget of $b =$ 25 bits, Untangle practically matches the speedups under no leakage budget ($b = \infty$). Overall, for a small leakage budget, such as $b \leq 50$ bits, Untangle always significantly outperforms Time. Although the performance differences between Time and Untangle become smaller as b increases,

5.9.2 Discussion of Limitations & Future Work

Workloads with more dynamic behavior. As discussed in Section 5.9.1, many SPEC17 benchmarks have relatively stable cache usage behaviors during the span of simulation, which is only hundreds of milliseconds and is bounded by the simulator and computational resource constraints. In the future, it would be interesting to increase the simulation time and evaluate *Untangle* in a more dynamic environment, such as Function-as-a-Service [29], where containers constantly start and terminate, or in an environment that groups latency-sensitive workloads with batch workloads. One possible solution to overcome the limitation of simulation performance is to use FPGA-based simulation, such as FireSim [233]. Additionally, besides normalized IPC, it would be interesting to consider other performance metrics such as tail latency, which is likely more sensitive to infrequent resizings. As the main contribution of this chapter is to propose a generic model for tightly bounding and measuring information leakage in dynamic partitioning schemes, we leave this exploration for future work.

Limiting unnecessary resizings to further reduce the leakage under *Untangle*. Recall that we observed that applications often suffer from unnecessary resizings that offer minimal performance benefits but incur more scheduling leakage under *Untangle* (Section 5.9.1). This is mainly caused by the greedy action heuristic in our setup, which tries to maximize the global cache hit count across all applications (Section 5.8). As a result, the action heuristic will decide to resize even if the global cache hit count under the optimal partitions is only marginally higher than that of under the current partitions. Based on this insight, we now present a simple preliminary solution. The intuition of the solution is to resize only if the global cache hit count under the optimal partitions, denoted by H_{opt}, is sufficiently higher than that of under the current partitions, denoted by H_{cur}. For example, we can set a threshold τ such that we resize only if $(H_{opt} - H_{cur})/H_{cur} \geq \tau$.

We perform an evaluation of this preliminary solution with $\tau = 0.05$ under resizing intervals scales of $s = 1$ and $s = 10$ without setting a leakage budget. This solution is effective, as we observe that the leakage per assessment averaged across Mixes 1–4 is reduced from 0.7 bits to 0.1 bits for $s = 1$ and from 1.0 bits to 0.4 bits for $s = 10$. Additional, it can be shown that using $\tau = 0.05$ has no significant impact on system-wide speedups, while using $\tau = 0.1$ can cause performance losses in some mixes.

Generalizing from the unnecessary resizing problem, an interesting future direction to look into is the interplay between the resizing benefits and cost in a dynamic partitioning system. Here, the resizing cost can be information leakage, as in side-channel defenses, or simply the cost of reconfiguring partition sizes, such as the required data movement when resizing cache partitions. Based on the resizing cost, one might decide to avoid resizings that have low performance gains and save the resizing opportunity for a later phase of execution. This objective is non-trivial to achieve, especially in an environment with many applications and partitions. Potential solutions could be derived from Multiple Input Multiple Output (MIMO) system control or online learning-based approaches.

Leakage bounds under more resizing actions. Recall that in the LLC partitioning schemes evaluated, we support 9 possible resizing actions (Section 5.8). We expect future advanced dynamic partitioning schemes to support a larger number of possible resizing actions. For example, a scheme may offer many possible partition sizes to suit the diverse resource demands of cloud applications. In this case, each operation that resizes the partition to a different size is a different resizing action. Similarly, a scheme may partition n types of resources—e.g., LLC, L2, TLBs, interconnect—at the same time. If a resource type i supports a_i different resizing actions, the scheme will have up to $\prod_i a_i = a_1 \times a_2 \times \cdots \times a_n$ possible resizing actions.

The leakage bound of the conventional Time grows when the number of possible resizing actions increases. This is because we can only conservatively bound its action leakage to $\log |\mathcal{A}|$bits per resizing, where $|\mathcal{A}|$ is the number of possible actions. Considering again the example of partitioning n types of resources, the action leakage in conventional systems would be bounded by $\log \prod_i a_i = \sum_i \log a_i = \log a_1 + \cdots + \log a_n$ bits per resizing. The leakage grows quickly with respect to n.

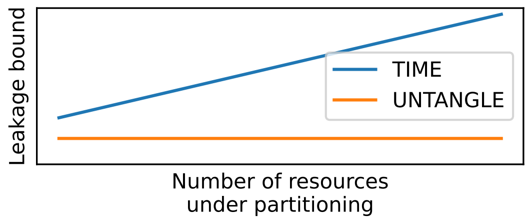


Figure 5.15: Leakage bounds as the number of resources under partitioning grows.

Using *Untangle*, the action leakage is independent of program timing. Based on that, *Untangle* uses program analysis and annotation to eliminate the action leakage all together. Then, the only remaining leakage is the scheduling leakage. Since the scheduling leakage rate is bounded by the maximum data rate of the covert channel, which only depends on when a resizing action occurs instead of what action is chosen, the scheduling leakage in *Untangle* has a fixed upper bound regardless of how many resizing actions are possible. Figure 5.15 illustrates the trend of leakage bound obtained by UNTANGLE and TIME as the number of resources under partitioning grows.

5.10 RELATED WORK

Types of Hardware Defenses. Hardware techniques to block microarchitectural side-channels fall into two categories. Randomization-based schemes [44, 45, 46, 47, 48, 49, 50, 51] attempt to obfuscate victim resource usage. These schemes offer high performance, but not comprehensive security guarantees. Partitioning-based schemes [53, 54, 55, 56] provide comprehensive security guarantees, but static partitioning incurs significant performance overhead [53].

Secure Dynamic Resource Partitioning. SecDCP [53] dynamically partitions cache resources based on a *tiered* security model: behaviors of sensitive programs cannot influence resizing decisions; only non-sensitive programs do. This model does not apply to cases where all programs are mutually distrusting and in the same security level (i.e., *peers*). In contrast, *Untangle* has a peer security model.

SecSMT [213] dynamically partitions pipeline resources. It supports both the tiered and peer security models. In the peer model, however, SecSMT only loosely bounds the leakage to 1 b per assessment (for 2 possible resizing actions). This is leakage overestimation. In contrast, *Untangle*'s leakage bounds are much tighter.

Quantifying the Leakage of Side Channels. Some works quantify the leakage of side channels in the absence of partitioning. Dynamic approaches examine specific victim executions and quantify their leakage trace [108, 234, 235, 236, 237, 238]. Thus, these approaches cannot produce a worst-case leakage bound. Other approaches use symbolic execution [110, 239] or abstract interpretation [111, 112, 113]. Although these approaches are sound, they cannot quantify leakage that depends on program timing.

5.11 CONCLUSION

This chapter presented *Untangle*, a framework for constructing low-leakage, high-performance dynamic partitioning schemes. *Untangle* formally splits the leakage into leakage from deciding what resizing action to perform and leakage from deciding when each resizing action occurs. Based on this breakdown, *Untangle* makes two contributions. First, it introduces a set of principles for constructing dynamic partitioning schemes that untangle program timing from the action leakage. Second, *Untangle* introduces a novel way to model the scheduling leakage without analyzing program timing. With these techniques, *Untangle* is able to quantify the leakage in a dynamic resizing scheme in a tighter way than prior work.

We applied *Untangle* to dynamic partitioning of the last-level cache. On average, workloads leak 78% less under *Untangle* than under a conventional dynamic partitioning approach, for approximately the same workload performance.

5.12 FIGURES: COMPLETE EVALUATION

The complete evaluation results are shown in Figures 5.16–5.22.

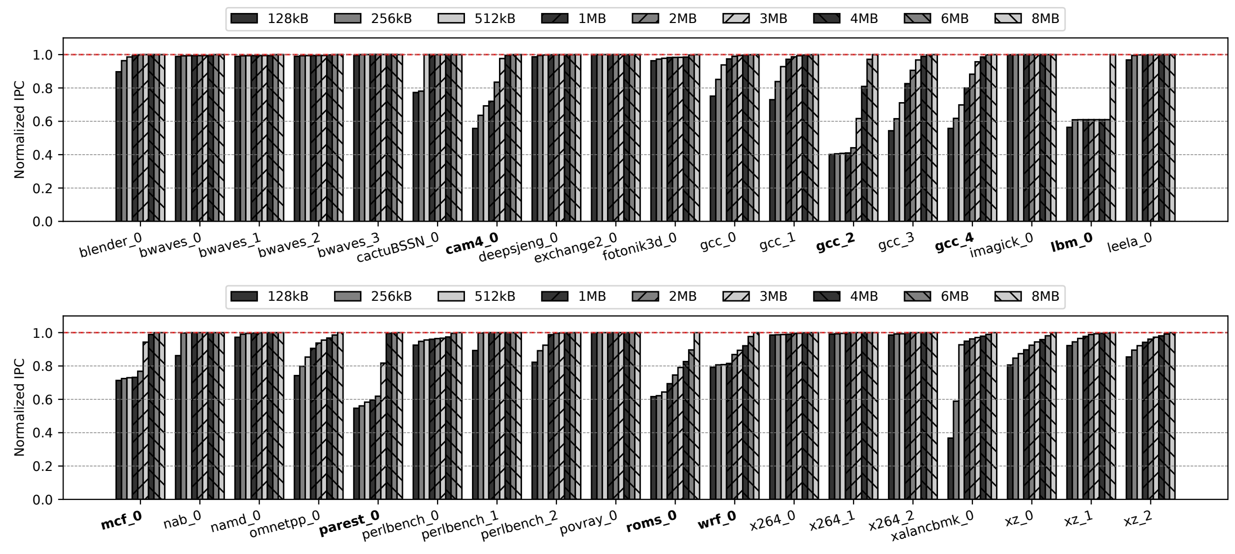


Figure 5.16: LLC sensitivity study of all 36 SPEC17 benchmarks. IPCs are normalized to the IPCs with an 8 MB partition. LLC-sensitive benchmarks are bolded.

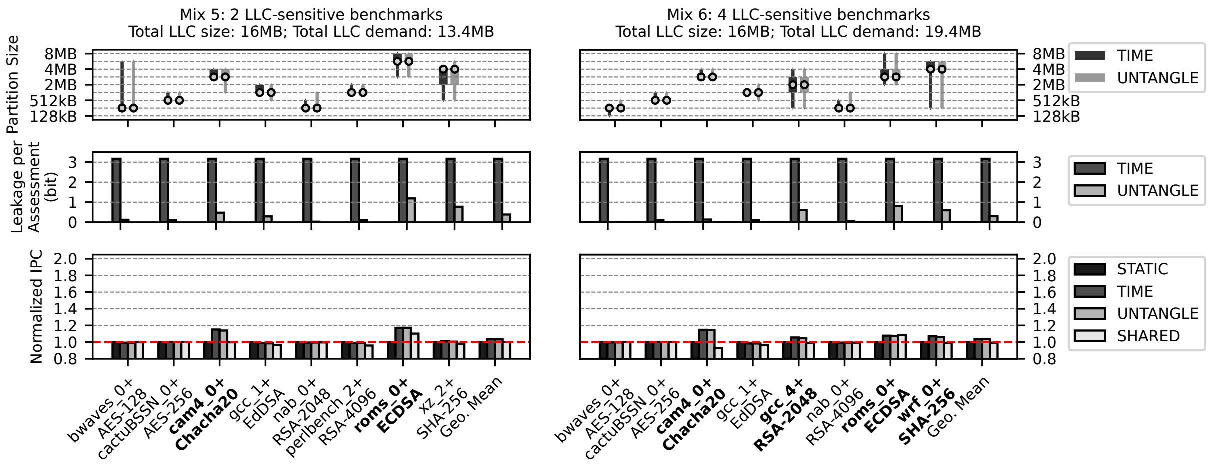


Figure 5.17: Comparing different partitioning schemes for workloads Mix 5 and Mix 6.

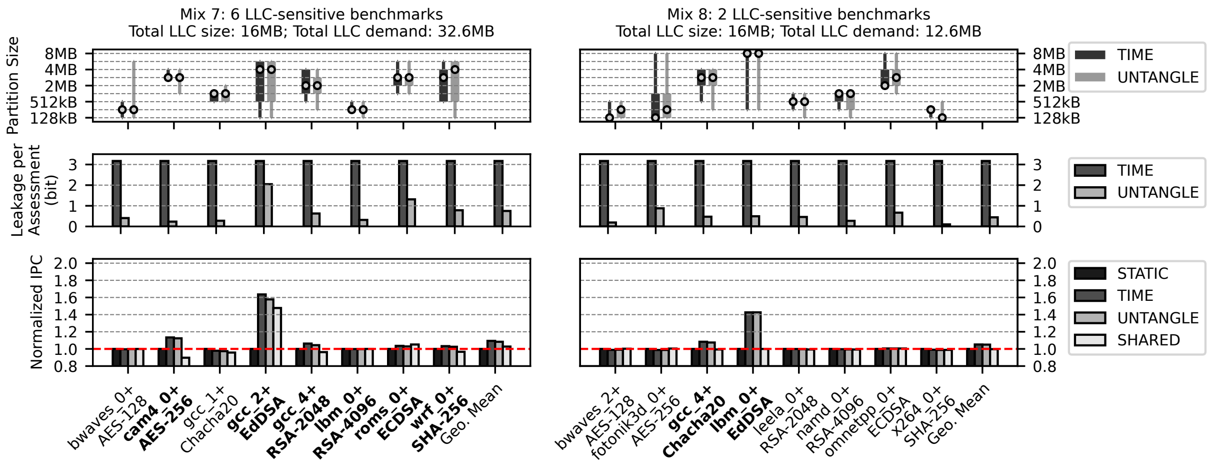


Figure 5.18: Comparing different partitioning schemes for workloads Mix 7 and Mix 8.

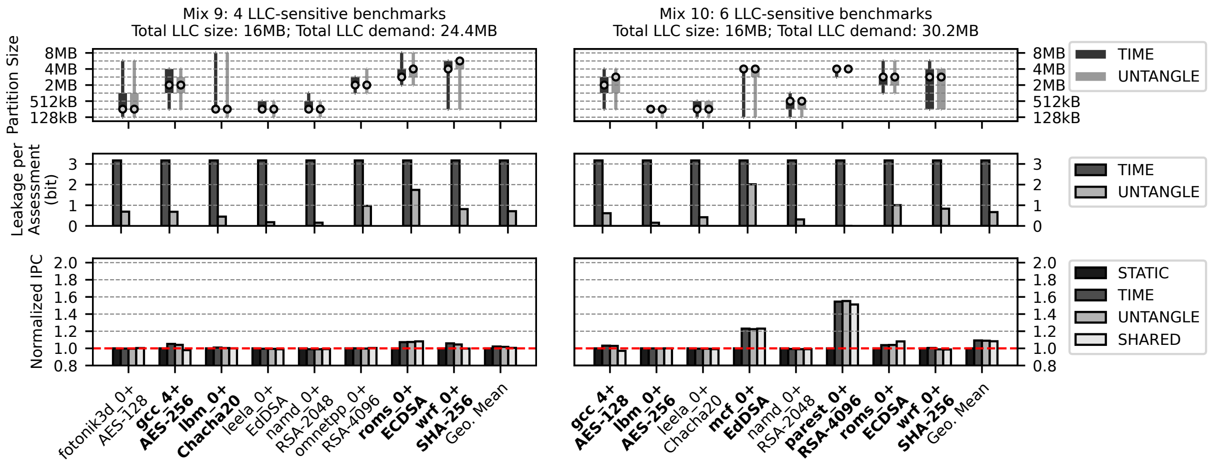


Figure 5.19: Comparing different partitioning schemes for workloads Mix 9 and Mix 10.

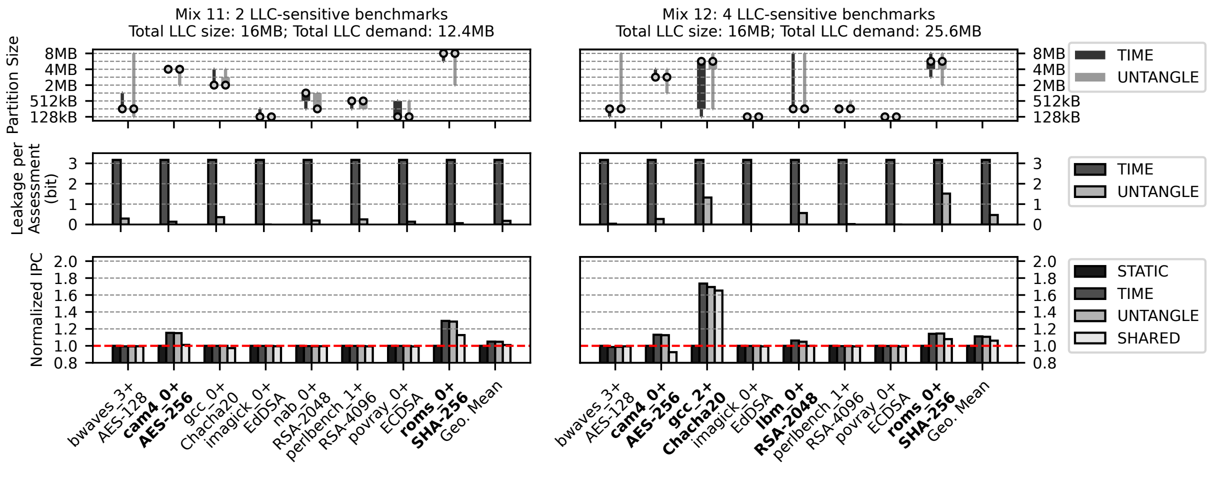


Figure 5.20: Comparing different partitioning schemes for workloads Mix 11 and Mix 12.

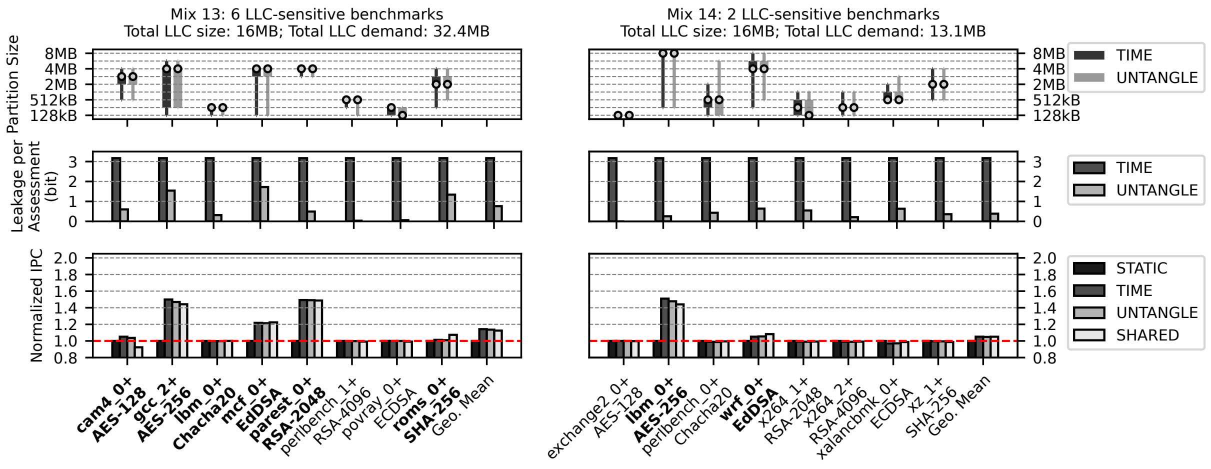


Figure 5.21: Comparing different partitioning schemes for workloads Mix 13 and Mix 14.

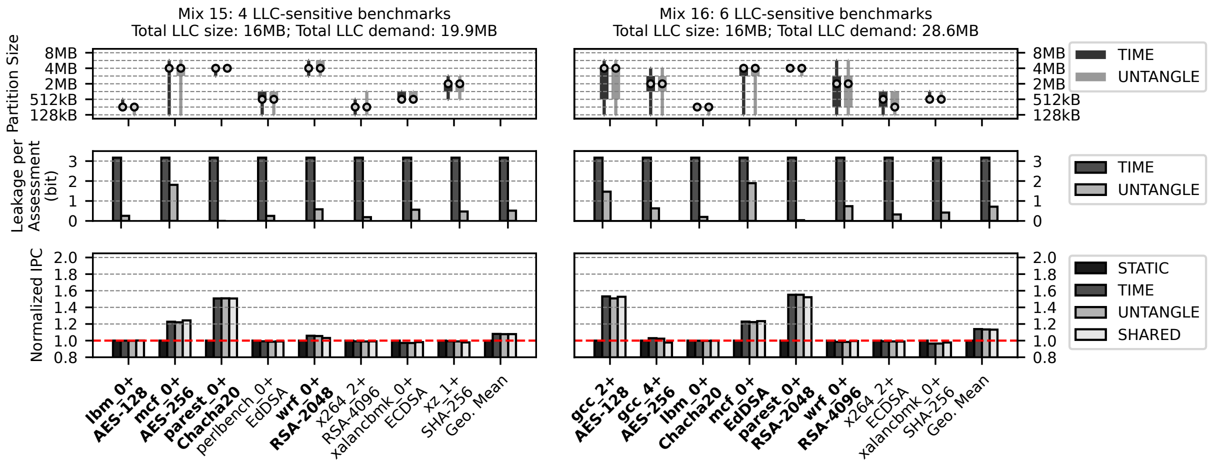


Figure 5.22: Comparing different partitioning schemes for workloads Mix 15 and Mix 16.

CHAPTER 6: Taming Speculative Loads in Secure Processors

6.1 INTRODUCTION

In this and the following chapter, we shift our focus to defending against speculative side-channel attacks (Section 2.3). These attacks are concerning to cloud vendors because an unprivileged, malicious cloud user can exploit them to access *arbitrary* memory of privileged software such as the OS kernel or hypervisor [240], which shares the physical hardware with the malicious user.

Given the danger of speculative side-channel attacks, many defense schemes have been proposed. Such schemes range from hardware-based (e.g., [54, 116, 118, 119, 120, 121, 122, 123, 124, 125, 241]) to software-only (e.g., [242, 243, 244, 245]) and hybrid (e.g. [246, 247, 248]). They prevent the early, unprotected execution of *transmitters*—i.e., instructions whose micro-architectural resource usage may reveal secret information [54, 116, 117]. While there are many types of transmitters, the most important one is loads, which, depending on the address they read, exercise different parts of the memory hierarchy.

A central idea in defense schemes against speculative execution attacks is an instruction's *Visibility Point* (VP) [118] (Section 2.3.3). An instruction reaches its VP when it is no longer vulnerable to pipeline squashes that are relevant to the threat model considered. For example, assume a threat model based on Spectre [18] and that transmitters are loads. A load reaches its VP when it can no longer be squashed by any branch misprediction—i.e., when all of its older branches are resolved.

Each instruction transitions from being pre-VP to reaching its VP, and then to becoming post-VP. A transmitter cannot safely execute before it reaches its VP. For example, we can prevent the load from executing by inserting a fence before it. Sometimes, a defense scheme provides special protection that allows a pre-VP transmitter to execute. For example, with the InvisiSpec scheme [118], pre-VP loads can be issued invisibly, but need to be followed by a second access later on.

When the transmitter reaches its VP, it can execute without protection. For example, once all the branches older than the load are resolved, we can remove the fence. As instructions reach their VPs and execute, they enable younger instructions to reach their own VPs. Hence, we intuitively say that "the older instructions pass the VP downstream."

The stall or protection of pre-VP transmitters slows down program execution over a conventional, unsafe processor. For example, pre-VP loads are either delayed by fences or have to be issued twice.

The more aggressive the threat model is, the more costly protecting pre-VP instructions becomes. Consider the Comprehensive threat model [246], where a load L reaches its VP only when it can no longer be squashed for any reason. In this model, reaching the VP requires that: (i) all branches older than *L* are resolved; (ii) neither *L* nor any older instruction can suffer exceptions; (iii) there is no unresolved older load or store that *L* or an older load could alias with; and (iv) neither *L* nor any older load can cause a memory consistency violation (MCV). Schemes that protect *L* until all of these conditions are true have substantial overhead.

Based on this discussion, a new approach to reduce the overhead of defense schemes against speculative execution attacks could be to try to *speed-up the advance of the VP* toward young instructions. If a technique could be found to do this, then potentially all the defense schemes could have lower overhead.

In this chapter, we propose such a technique, which we call *Pinned Loads*. To conceive it, we first examine, under the Comprehensive threat model, the delay induced to the VP advance by each of the four conditions listed above. We use the Comprehensive model because it is the most general one and covers recent attacks, including MCV-based attacks [20, 249]. In our analysis, we find that what delays VP progress the most is ensuring that no MCV is possible. This condition, therefore, adds the most overhead to safe program execution.

Based on this observation, we design *Pinned Loads* as a microarchitecture that tries to make loads invulnerable to MCVs as early as possible, therefore speeding-up VP progress. In our design, we assume the TSO memory consistency model [250, 251]. Recall that, under TSO, a load is conservatively flagged as causing an MCV and squashed when the core receives a coherence invalidation for the line accessed by the load or when the line is evicted from the cache. Hence, given a load *L* that has met all the conditions required to reach the VP except for the guarantee of no MCVs, *Pinned Loads* tries to ensure that no invalidation or eviction of *L*'s line is possible anymore. If *Pinned Loads* can ensure this, we say that it *pins L* in the reorder buffer—making *L* unsquashable and moving *L* to its VP. If we manage to do this for many loads, the VP makes fast progress and the execution speeds-up.

In this chapter, we also describe the hardware needed for *Pinned Loads*. Further, we propose two possible designs of *Pinned Loads*, which offer different tradeoffs between hardware requirements and delivered performance. Finally, we extend several popular defense schemes for speculative execution with *Pinned Loads*. As we run the SPEC17, SPLASH2, and PARSEC benchmark suites with them, we observe a substantial reduction of their execution overhead. Indeed, the average execution overhead of defense schemes that (i) either place fences before loads, (ii) or stall speculative loads that miss in the L1 (Delay-On-Miss [120, 121]), (iii) or stall speculative loads whose arguments are tainted (STT [116]) decreases by about 50%. Specifically, on SPEC17, *Pinned Loads* decreases the execution overhead of the fence-based defense from 112.6% to 51.3%, of Delay-On-Miss from 35.8% to 15.3%, and of STT from 24.8% to 13.2%; on SPLASH2/PARSEC, *Pinned Loads* decreases the execution overhead of the defense schemes from 113.1% to 46.4%, from 15.8% to 7.6%, and from 11.3% to 8.1%, respectively.

In summary, the chapter makes the following contributions:

• Introduces *Pinned Loads*, a novel technique to reduce the overhead of speculative-execution defense schemes by speeding-up VP progress.

• Presents the mechanisms behind *Pinned Loads*.

• Describes two different designs of *Pinned Loads*.

• Evaluates multiple *Pinned Loads*-extended popular defense schemes on an extensive application set.

6.2 BACKGROUND: MEMORY CONSISTENCY VIOLATIONS (MCVS)

In a multiprocessor system, the memory consistency model defines the order in which a processor's loads and stores are observed by other processors. When a store retires from the pipeline, its data is deposited into the write buffer. From there, when the memory consistency model allows, the data is merged into the cache, making it observable by all the other processors. In this chapter, we say that a store is performed when its data is merged into the cache; we say that a load is performed when it receives its data. In conventional, unsafe processors, loads can read from the memory hierarchy and be performed before they reach the ROB head, and even out of order—i.e., before older loads and stores in the ROB are performed. These out-of-order loads can lead to memory consistency violations (MCVs) if another processor observes an order not allowed by the memory consistency model.

A processor recovers from an MCV by using the instruction squash and rollback mechanism of speculative execution [252]. We discuss how this is done for the Total Store Order (TSO) memory consistency model [250, 251], which is the one used by the x86 architecture and assumed in this chapter. TSO forbids load to load reorderings (load→load), which is when a younger load is performed before an older load to a different address. Implementations of TSO prevent observable load→load reordering by ensuring that the value that a load reads when it is performed remains valid when the load retires. This guarantee is conservatively maintained by squashing a load that has performed, but not yet retired, if the processor receives a cache invalidation for the line read by the load. Moreover, the load is also squashed if the line read by the load is evicted from the cache before the load retires—since, on a subsequent external write, the cache may not receive an invalidation.

Strictly speaking, cache line invalidations and evictions do not need to squash the oldest load in the pipeline—since such load has not been reordered. A reorder can only occur when the line in question has been read by a load L that is not the oldest load in the pipeline, and then the hardware needs to squash *L* and all its successor instructions. This is the design that we use in our evaluation. However, for ease of explanation of our mechanism, we discuss the simpler implementation where any load that has read a line that is invalidated or evicted triggers a squash. This is the implementation used in Intel processors [20].

TSO also forbids load to store (load→store) and store to store (store→store) reorderings. Implementations of TSO prevent them by merging a store with the cache hierarchy only after the store instruction has retired, and by using a FIFO write buffer, ensuring that stores are drained in program order.

6.3 PINNED LOADS: MAIN IDEA & IMPACT

6.3.1 Advancing the VP is Crucial

The Visibility Point (VP) is an important concept in defense schemes against speculative execution attacks. When an instruction reaches its VP, it becomes safe to execute without any protection. Consider a load, which is the focus of this chapter. If the baseline defense is to place a fence before a load, then when the load reaches its VP, the fence can be removed. If the defense is to issue the load early invisibly, followed by a second access later [118], when a load reaches its VP, it is unnecessary to issue the load twice anymore.

The conditions that determine when a load reaches its VP depend on the threat model used. However, it is evident that any technique that can help a load reach its VP sooner will help speed-up execution under practically any defense scheme against speculative execution attacks: loads will execute sooner or with lower overhead, and will in turn enable subsequent loads to reach their VPs sooner. Intuitively, the hardware will be "moving the VP to younger loads" faster.

There are some defense schemes that, using certain assumptions, allow the unprotected issue of some loads that have otherwise not reached their VP—e.g., loads that have reached the Execution Safe Point in InvarSpec [246] or loads whose arguments are not tainted by transiently-read data in STT [116]. Even in these cases, enabling loads to reach their VP sooner is useful: the conditions that enable such unprotected early load execution depend on older loads actually reaching their VP. In this chapter, to keep the discussion simple, we will not discuss such "early safe" loads.

6.3.2 Focus on Memory Consistency Violations

For the Comprehensive threat model, Section 6.1 listed the four conditions necessary for a load to be free of potential squashes and, therefore, reach its VP. To design an effective "VP-advancing" technique, we need to understand how performance-limiting each of these conditions is in practice.

To this end, we take a processor that places a load-stalling fence before each load, and consider four possible times when to remove the fence—from typically earlier to later times. The times are when no squash is possible due to: (i) branches (*Ctrl Dep*), (ii) branches or aliasing (*Alias Dep*), (iii) branches, aliasing, or exceptions (*Exception*), and (iv) branches, aliasing, exceptions, or memory consistency violations (*MCV*). Figure 6.1 shows the resulting execution overhead of the environments (in a stacked manner) over a conventional unsafe processor. The processor is the one shown in Table 6.1, and the programs are those in the SPEC17 [231] suite (single-threaded), and in the SPLASH2 [253] and PARSEC [254] suites (with eight threads).

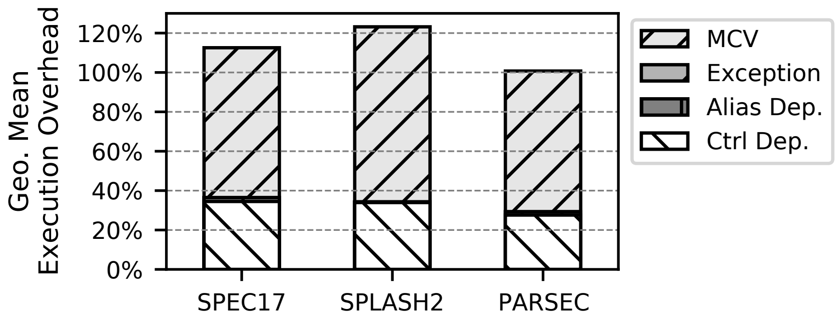


Figure 6.1: Effect of each reason that delays reaching the VP.

Of all the conditions, ensuring that no MCV is possible is, by far, the one that delays reaching the VP the most and, therefore, slows down execution the most. Waiting for branch resolution also has a substantial, yet smaller impact, while waiting for alias resolution and exception-free state is much less significant. Overall, as intuition suggests, while the absence of squashes due to *Ctrl Dep*, *Alias Dep*, or *Exception* is determined relatively soon, the absence of squashes due to *MCV* remains unresolved until the load is close to the head of the ROB. Hence, this condition substantially delays "moving the VP downstream" and, therefore, slows down program execution.

For this reason, in this chapter, we focus on making loads invulnerable to MCVs as early as possible. We pick a load *L* that has met all the conditions to reach its VP except for guaranteeing that *L* will not cause MCVs. Then, our goal is to *Pin L* in the ROB—i.e., to declare it unsquashable due to MCVs and hence declare that it has reached its VP—as early as possible.

Recall from Section 6.2 that a load is conservatively identified as causing an MCV and squashed when the core receives a coherence invalidation for the line accessed by the load or when the cache evicts this line. Hence, to declare load *L* as *Pinned*, the hardware needs to guarantee that none of these two events will occur for *L*.

Our proposed architecture, called *Pinned Loads*, guarantees it as follows. First, to guarantee that there will be no squash of *L* due to invalidations, *Pinned Loads* delays incoming invalidations to the line read by *L* until *L* retires. Since invalidations can only be delayed for a limited time period, the hardware also has to guarantee that L will eventually reach the ROB head and retire—at which point, no more invalidation delays will be needed. Consequently, *Pinned Loads* can only declare L as pinned if the core has enough resources to retire L and all the instructions older than L in the pipeline—in particular, the write buffer needs to have enough entries to fit all the yet-to-complete stores older than L.

Second, to guarantee that there will be no squash of L due to cache evictions, *Pinned Loads* pins only loads that access cache lines that it can guarantee are non-evictable. To obtain such a guarantee, *Pinned Loads* needs to reserve, for each core, a minimum number of lines W_d per set in the directory plus last-level cache (LLC). Furthermore, *Pinned Loads* knows the associativity W_{L1} of the L1 cache. With this information, *Pinned Loads* only declares L pinned if the lines accessed by L and by the set of already-pinned older loads: (i) do not overflow W_d for any directory plus LLC set, and (ii) do not overflow W_{L1} for any L1 cache set. In addition, *Pinned Loads* refuses to evict from its L1 cache and from the directory plus LLC any line that has been accessed by a currently-pinned load. Such eviction request may be a self eviction initiated by the local processor or a cross eviction initiated by another processor.

To keep the design simple, *Pinned Loads*: (i) pins all the loads that will eventually retire and (ii) does it in strict program order. Further, no load can be pinned before it has generated its address, since it can suffer an exception during address translation. After address translation, we assume the load cannot suffer exceptions.

6.3.3 Potential Performance Gains

To understand the performance gains enabled by *Pinned Loads*, consider a ROB with three independent loads. Recall that we assume a baseline processor implementation where even the oldest load in the ROB can suffer an MCV. Figure 6.2(a) shows the behavior of the conventional, unsafe processor. As denoted by the arrows, all three loads can be issued to memory in parallel. Figure 6.2(b) shows the behavior of a safe processor. In this case, a load can only be safely issued when it reaches its VP. Generally, this occurs when the load is close to the ROB head. The result is poor performance, as loads are issued late and only one load can be in progress at a time.

Consider now a safe processor augmented with *Pinned Loads*. We propose two designs, which will be detailed later. To understand them, consider a load L that has met all the conditions to reach the VP except for guaranteeing no MCVs. Our first design (*Early Pinning*) has special hardware that determines whether there is enough space in the cache hierarchy and directory to hold the line that L requests—given that there may already be other pinned loads. If the answer is yes, *Pinned Loads* declares L pinned even before issuing L to memory, and "passes the VP downstream." Our second design (*Late Pinning*) is simpler and has no such special hardware. In this design, L is first

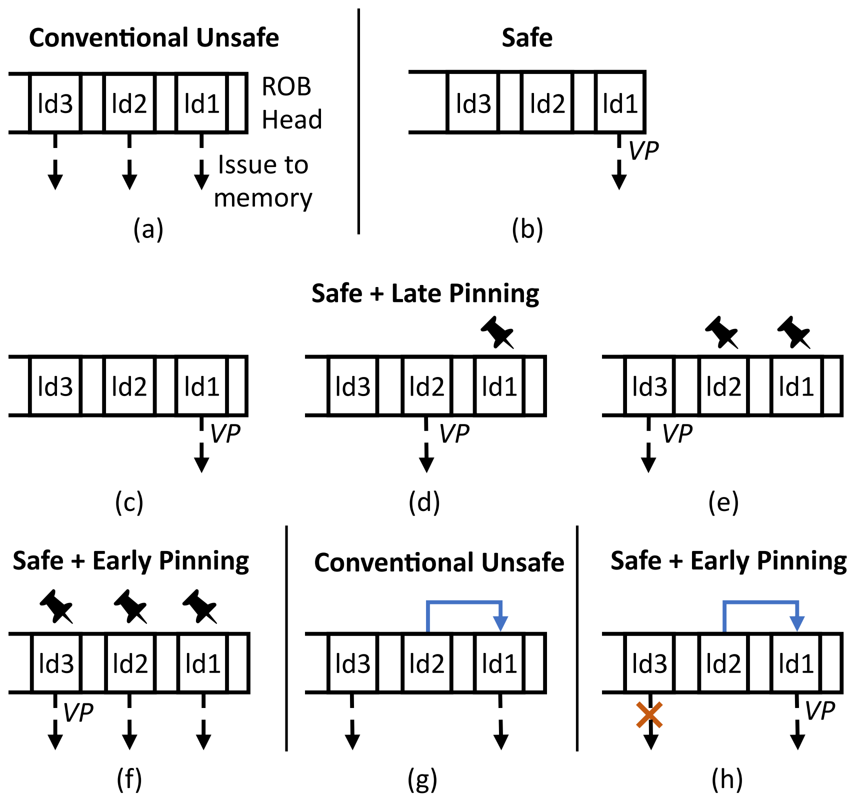


Figure 6.2: Overlapping of loads in the reorder buffer (ROB).

issued to memory. If *L* successfully brings the data to the L1 cache, hence proving that directory and caches have space for the line, *Pinned Loads* declares *L* pinned and "passes the VP downstream." These two designs offer different tradeoffs between hardware requirements and performance.

The behavior of the safe processor augmented with Late Pinning is shown in Figures 6.2(c)-(e). As shown in Figure 6.2(c), the oldest load reaches its VP (and issues to memory) *earlier* than in the safe processor: when only an MCV could squash it. However, in reality, no MCV-induced squash will occur: while the data has not returned, no MCV can occur by construction; and as soon as the data arrives, *Pinned Loads* will pin it and hence ensure no MCV can occur. Assume that, when the oldest load gets pinned, the second load reaches its VP. The second load then issues to memory (Figure 6.2(d)) and, on reception of the line, gets pinned. The process repeats for the third load in Figure 6.2(e). We see that, while loads are not issued in parallel as in the unsafe processor, they are issued much earlier than in the safe processor.

The behavior of the safe processor augmented with Early Pinning is shown in Figure 6.2(f). In this design, *Pinned Loads* can pin a load (and enable the next load to reach its own VP) even *before* the load issues to memory. Hence, as shown in the figure, the VP "is passed downstream" quickly and all the loads proceed in parallel. The result is safety and high performance.

For completeness, Figure 6.2(g) shows the case when the second load is dependent on the first one. The unsafe processor can issue the first and third loads in parallel. However, even the Early Pinning design cannot match the performance of the unsafe processor. Indeed, the second load's address depends on the return value *V* of the first load. Hence, the second load cannot be declared pinned until *V* is known and the load's address is translated—since there is a risk of an address translation exception. Since the second load is not pinned, the third one, although independent, cannot be claimed as pinned and issue (Figure 6.2(h)).

If *Pinned Loads* is able to remove most of the stall due to MCVs in Figure 6.1, the resulting performance may be close to that of a processor only stalling for branch resolution. In that case, the performance of a safe processor under the Comprehensive threat model would be close to that of a safe processor under the Spectre threat model.

If the processor supports the more aggressive implementation of TSO described in Section 6.2, where cache line invalidations and evictions do not squash the oldest load in the ROB, a more aggressive design of Late Pinning is possible. Specifically, as soon as the oldest load (i.e., *ld1* in Figure 6.2(c)) is free of all the other sources of squashes (i.e., branches, aliasing, and exceptions), it issues and "passes the VP downstream"—since it cannot be squashed anymore. Hence, *ld2* can be issued while *ld1* is still outstanding. Furthermore, when *ld2* receives the data, it gets pinned, and *ld3* can be issued even if *ld1* is still outstanding. Overall, while the safe baseline can have only one outstanding load, this more aggressive design of Late Pinning can support two outstanding loads, as long as one of them is the oldest one in the ROB—in addition to supporting the sequential issue of multiple loads much earlier, as indicated before. This is the design we use in the evaluation.

6.4 THREAT MODEL

We assume the Comprehensive threat model and various baseline hardware defense schemes that *Pinned Loads* augments for performance. The Comprehensive model is necessary to cover recent attacks, including attacks related to memory consistency [20, 249] (Section 6.10). Examples of baseline schemes that *Pinned Loads* can augment are those that protect pre-VP loads with blocked execution [116, 241, 255], execution only if they hit in the L1 [120, 121], or invisible execution that does not change the state of the cache hierarchy [118, 119, 122].

Pinned Loads does not modify the speculative execution security properties of the baseline defense schemes. The reason is because *Pinned Loads* does not modify the definition of VP; it simply enables loads to reach their VPs earlier.

Pinned Loads does not add new speculative side or covert channels. The reasons are: (i) a load is pinned only if it satisfies all the conditions for reaching its VP except for the possibility of causing an MCV, and (ii) a pinned load is guaranteed not to cause an MCV. These combined properties imply that, once a load is pinned, its retirement is guaranteed, and so any side-effects of its execution cannot result in *speculative leakage*. In particular, this argument applies to any new side-effects introduced by *Pinned Loads* itself (e.g., changes in the processing of coherence invalidations or evictions). These side-effects are only a function of the pinned load's operands, and thus do not leak *speculative* information.

Pinned Loads does not address *non-speculative* side channels. It is well known that, in a multi-threaded shared-memory environment, an attacker can exploit cache coherence states for timing-based non-speculative side channels [256].

6.5 DESIGN OF PINNED LOADS

In this section, we describe the *Pinned Loads* design and present the Late and Early Pinning variations. In Sections 6.6 and 6.7, we outline some implementation aspects, and compare to a related design. In the following, we refer to a line that is accessed by a currently-pinned load as a *pinned line*.

At its core, *Pinned Loads*: (i) delays incoming invalidations to pinned lines and (ii) prevents cache evictions of pinned lines. In addition, it has to ensure that the processor has enough resources to pin a load—i.e., enough write buffer entries for yet-to-complete stores, and enough cache and directory space for all the pinned lines. Finally, it has to ensure that delayed stores make progress.

Note that *Pinned Loads* never pins loads younger than in-ROB MFENCE or LOCK instructions because doing so would be incorrect. For example, pinning a load before an older lock is acquired would be equivalent to binding the value returned by the load before the lock is acquired.

6.5.1 Pinned Loads Mechanisms

Delaying Invalidations to Pinned Lines. Processors that support TSO [250, 251] conservatively avoid MCVs by squashing a yet-to-retire load issued by the processor when the L1 cache receives an invalidation for the line read by the load. When an invalidation is received in L1, the Load Queue (LQ) is snooped and, on finding a matching entry, the corresponding load and its successor instructions are squashed.

Pinned Loads keeps a record of the pinned lines. Such a record only requires one bit in each LQ entry, although other designs are possible (Section 6.6.1). When an invalidation arrives and the LQ snoop finds it is directed to a pinned line, the hardware denies the invalidation.

Supporting this functionality requires a modification to the write transaction of the cache coherence protocol. Figures 6.3(a) and (b) show the conventional and the *Pinned Loads* write transaction, respectively. In the figure, Core 1 has brought the line to its L1 cache in state shared (S) with a yet-to-retire load, and Core 2 issues a write to the line (arrow ⓪). In the conventional transaction (Figure 6.3(a)), Core 2 issues a GetX request to the directory (①). The directory returns the line plus the number of sharers to Core 2 (②), sends an invalidation to Core 1 (②), and enters a transient state that rejects other requests to the line. Core 1 invalidates its local copy of the line, snoops its LQ, squashes its load to the line, and sends an ack to Core 2 (③). Core 2 then sends an Unblock request to the directory (④), which exits the transient state and updates the sharers.

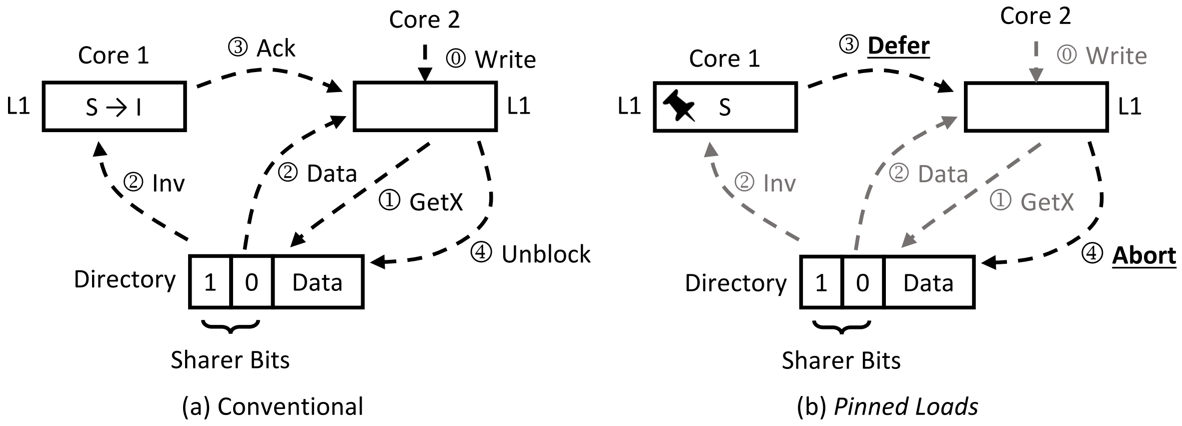


Figure 6.3: Conventional (a) and *Pinned Loads* (b) write transaction.

In the *Pinned Loads* transaction, when Core 1 receives the invalidation (②), the hardware snoops the LQ before invalidating the cache line. On finding a match with a pinned line, the cache is not invalidated, the load is not squashed, and a Defer message is sent to Core 2 (③). If Core 2 receives a Defer from any sharer of the line, it aborts the write and sends an Abort to the directory (④). The latter exits the transient state and does not change the sharer bits. Core 2 will now retry the write. To ensure that Core 2 is able to eventually write, additional support is added in Section 6.5.1.

Ensuring Enough Write Buffer Entries. Delaying invalidations is a temporary mechanism applied until the pinned load reaches retirement. Hence, before *Pinned Loads* marks a load L as pinned, it has to ensure that there are enough resources for L to reach retirement. One obvious resource required is related to stores: there need to be enough write buffer entries to be able to hold all the yet-to-complete stores that are older than L. This includes stores already in the write buffer and stores not yet in the write buffer. The reason is that, for L to retire, all of its older stores should be pushed into the write buffer.

If this condition is unmet, deadlock may ensue. To see why, consider the two cores in Figure 6.4. Core 1 has retired a store to line x to its write buffer. Its ROB contains another store and then a pinned load to line y. Core 2 has retired a store to line y to its write buffer. Its ROB contains another store and a pinned load to line x. Assume that ldx and ldy have loaded their data to the L1 caches, and that the write buffers can hold a single write. In the write buffer of Core 1, stx's attempt to write is denied by Core 2 because line x is pinned by ldx (①). Similarly, in the write buffer of Core 2, sty's attempt to write is denied because ldy is pinned. To make forward progress, either ldx or ldy have to retire. Load retirement would remove the pin, which would in turn allow the write in the other core to succeed, and execution to proceed. However, no load can retire (②): both ROBs have an older store that cannot leave the ROB because the write buffer is full.

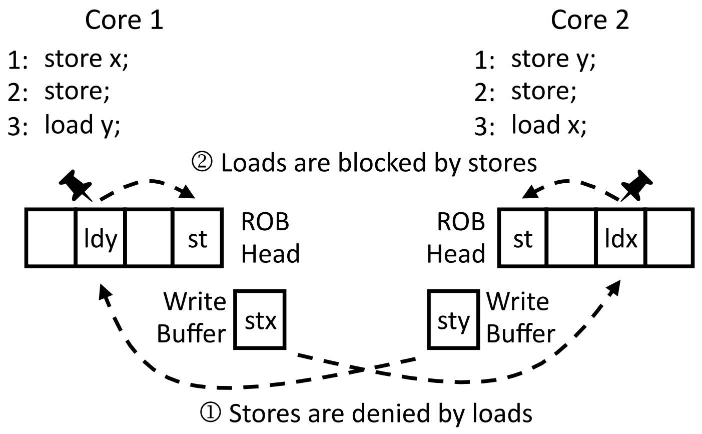


Figure 6.4: Deadlock due to insufficient write buffer entries.

To prevent this deadlock, before *Pinned Loads* declares a load pinned, it counts the number of yet-to-complete stores older than the load (already in the write buffer or not). The load is not pinned while such count is higher than the number of write buffer entries.

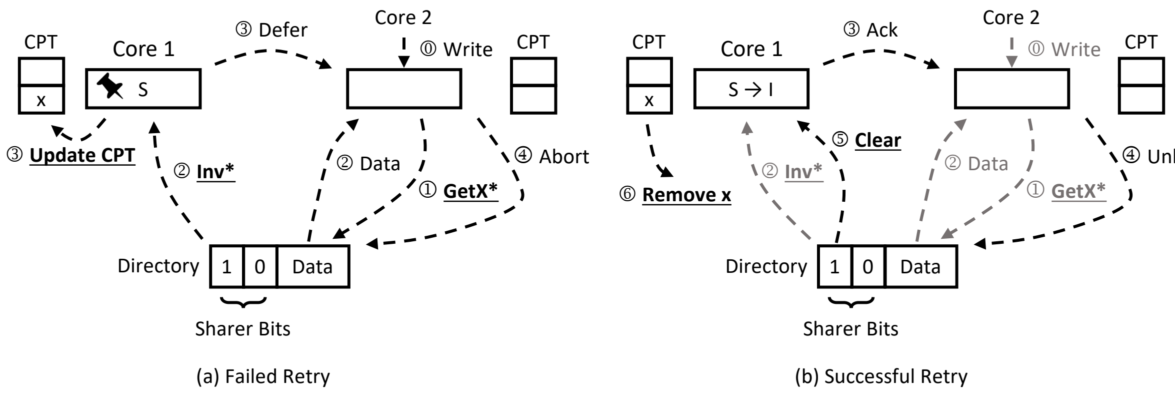


Figure 6.5: Mechanism to prevent store starvation. In the figure, x is the address of the line that the store is trying to update.

Preventing Evictions of Pinned Lines. Processors that support TSO also conservatively avoid

MCVs by squashing a yet-to-retire load when the L1 cache wants to evict the line read by the load. In *Pinned Loads*, the hardware prevents pinned lines from being evicted from L1.

The process is similar to how *Pinned Loads* denies invalidations in Section 6.5.1. Specifically, when the L1 wants to evict a line, the hardware checks whether the line is pinned. The record of what lines are pinned can be kept in the LQ, as described in Section 6.5.1, or in the L1 tags, as we will see in Section 6.6.1. If the line is found pinned, the eviction is denied. Then, the cache controller updates the replacement algorithm state as if the line had been accessed (to minimize future attempts to evict the line), and then selects a new victim from the same cache set.

The action of evicting a line from L1 may be initiated by a request from the local core, which may need to allocate space in any cache, or from another core, which may need to allocate space in the shared cache. Moreover, it may occur with inclusive, non-inclusive or exclusive cache hierarchies, and with different directory organizations.

Note that this mechanism is not unusual: conventional cache hierarchies sometimes need to deny cache line evictions, as is the case when the victim cache line is in a transient state. More details are given in Section 6.6.1.

Guaranteeing Space in Cache & Directory. A core cannot pin any number of cache lines. The number of pinned lines that map to a set in a private cache or to a set in a shared directory/LLC cannot be bigger than the associativity of these structures: all the pinned lines need to remain in the caches or directory/LLC, respectively. Consequently, before *Pinned Loads* declares a load *L* pinned, it has to ensure that the lines accessed by all the currently-pinned loads plus *L* can co-exist in the private caches and in the shared directory/LLC.

One approach to ensure that *L* can be pinned is to issue it first and observe whether it attains the cache and directory space needed; if so, it gets pinned. Another approach is to only issue and pin *L* if *Pinned Loads* can first guarantee that there will be space.

For this second approach, let us assume an inclusive cache hierarchy with private L1 caches and a shared L2 LLC with the directory. We discuss other cache hierarchy organizations in Section 6.6.2. In this case, *Pinned Loads* needs to know the associativity of L1 (W_{L1}) and, because the directory/LLC is shared by all the cores, the number of entries in each set of each directory/LLC slice that are reserved for each core (W_d). In addition, *Pinned Loads* needs to know the mapping of line addresses to sets in L1 and to slices and sets in the directory/LLC. Finally, *Pinned Loads* needs to have a small hardware-managed table that records, for each pinned load *L*, the L1 set and the directory/LLC slice and set where the line accessed by *L* maps. This table is called the *Cache Shadow Table* (CST) and is discussed in Section 6.6.2.

In this second approach, when *Pinned Loads* wants to pin load *L*, it first determines the L1 set and the directory/LLC set and slice where the line maps. Then, it accesses the CST and determines whether such sets and slice can hold one additional pinned line—i.e., whether with the addition of L, no more than W_{L1} and W_d pinned lines map to the same set in L1 and directory/LLC, respectively. If these conditions are all met, L is declared pinned; otherwise pinning needs to wait.

Preventing Store Starvation. Figure 6.3 showed that when a core pins a cache line, a write by another core is denied, and the hardware in the writer core has to keep retrying. Unfortunately, it is possible that the reader core (and other additional cores) keep reading and pinning the line. Since the decision to pin a line is made locally in each core, the readers may never know that there is a writer that is starving.

To avoid starvation, *Pinned Loads* uses the following idea: if a write request is denied, its retry works in a slightly different way, which prevents the indefinite repeated pinning of the line by other cores before the write succeeds. To support this idea, *Pinned Loads* adds a small hardware table in each core called the *Cannot-Pin Table (CPT)*. The CPT records the lines that the core cannot pin at the moment.

Figure 6.5 illustrates how the algorithm works. After the first write by Core 2 in Figure 6.3 was denied, Core 2 now retries with a new variant of GetX called GetX* (① in Figure 6.5(a)). After the directory receives GetX*, it sends a special invalidation, Inv*, to Core 1 and all the other current sharers (②). Upon receiving Inv*, Core 1 and all the other sharers add the address of the line (x) to their CPTs (③), meaning that they will not be able to pin the line again until the write succeeds. The sharers then reply to Core 2: if a sharer has the line pinned (as in Core 1), the sharer replies Defer to Core 2 (③); otherwise it replies Ack to Core 2 and invalidates its copy of the line. If Core 2 receives at least one Defer, it knows the line is pinned; hence it sends an Abort to the directory (④), which does not change its state. To minimize hardware modifications, the directory is not modified to record that a write is being denied.

From now on, none of the cores with x in their CPTs can pin the line—although they can read it. Other cores can still read the line and pin it. However, every single retry of the write will insert x in the CPTs of the sharers. In the worst case, all cores but the writer end up with x in their CPTs.

Eventually, all the reader cores will retire the pinned loads, and a retry by the writer will find that all the responses are Ack and there is no Defer (Figure 6.5(b)). Such Acks come from all the sharers recorded in the directory. The write has now succeeded. Hence, Core 2 sends the Unblock message to the directory (④). On reception of the Unblock message, the directory sends an extra Clear request to all the sharers (⑤) so they remove x from the CPT (⑥), and then updates the sharer information.

6.5.2 Late and Early Pinning Approaches

We propose two variations of *Pinned Loads* that offer different tradeoffs between hardware requirements and performance: *Early* and *Late Pinning*. In both designs, loads are pinned in program order, all loads that will eventually retire are pinned, and a load can only be pinned when it has met all the conditions to reach its VP except for guaranteeing no MCV.

Late Pinning (LP). This design does not include the CST of Section 6.5.1. A core does not know, at the point of issuing a load, whether the private caches and the shared directory/LLC will have space to hold the line—given all the older pinned loads. Hence, when a load meets all the conditions to reach the VP except for guaranteeing no MCVs, and *Pinned Loads* concludes that there are enough write buffer entries, *Pinned Loads* issues the load. If the core receives a response with the data, it means that private caches and shared directory/LLC have the space for the line; then, *Pinned Loads* declares the load pinned. Otherwise, *Pinned Loads* has to wait until the line can be loaded to declare the load pinned.

This design has two advantages. First, it is simpler because it has no CST. Second, cores can ignore the limitation of only pinning at most W_d lines per set and slice in the shared directory/LLC. A core can issue many loads that attempt to allocate lines in the directory/LLC; if they succeed, the loads are declared pinned. It is possible that a core ends up pinning more than its share of lines in a given directory/LLC set. Such a situation often improves performance and only infrequently ends up temporarily starving other cores.

This design's shortcoming is that the load's response from the memory system is in the critical path of declaring the load pinned and, hence, of "passing the VP to the next load". The result is that all loads access the memory system sequentially (Figures 6.2(c)-(e)), even if they are independent. Hence, performance is low in programs with bunched-up cache misses.

Early Pinning (EP). This design includes the CST. When a load L meets all the conditions to reach the VP except for guaranteeing no MCVs, and *Pinned Loads* ascertains that there are enough write buffer entries, the CST is checked. If the CST decides that the new line will find space in the private caches and the shared directory/LLC, L is declared pinned and the VP is "passed down"—potentially even before issuing L to memory.

The pluses and minuses of this design are the opposite of those of the previous one. The advantages are that independent loads are issued to memory with great parallelism (Figure 6.2(f)), even out of order, and that the VP "is passed to younger loads" faster. The result is high application performance. Recall, however, that if a load cannot be issued to memory due to a dependence, then subsequent, independent loads cannot be issued to memory either (Figure 6.2(h)).

The shortcomings of this design are the need for the CST hardware and the fact that a core will not attempt to pin more than its W_d share of lines per slice and set in the directory/LLC. This is because loads are pinned before being issued, and hence *Pinned Loads* has to guarantee space for their data in advance.

Note that the assignment of W_d maximum *pinned* lines per slice and set in the shared directory/LLC to each core is an agreement among cores; it does *not* require fixed set partitioning of the directory/LLC. Also, once the load that pinned a line retires, the line gets unpinned, and the line can remain in the directory/LLC for potential future use without counting toward the W_d maximum pinned lines allocated to the owner core.

6.6 KEY IMPLEMENTATION ASPECTS

In this section, we describe the implementation of key aspects of the *Pinned Loads* hardware.

6.6.1 Recording Pinned Lines

Pinned Loads provides hardware to record the currently-pinned lines. On any attempt to invalidate or evict a line, the hardware is checked to either allow or prevent the operation. Note that such hardware can be placed very close to the core. The reason is that, if a pinned load has already obtained its data, it has brought the line to L1, and any invalidation or eviction request for the line will reach L1. Alternatively, if the pinned load has not brought the line to L1 yet (which may happen in Early Pinning), since the load has not consumed the data yet, the consistency model does not squash the load on invalidation or eviction of the line from other cache levels. We present two possible designs to record pinned lines. Our chosen design is the first one.

Storing the Information in LQ. This design adds one Pinned bit to each LQ entry, indicating whether the load is pinned. When a load gets pinned, the core sets the bit in the load's LQ entry. When the L1 receives an invalidation or attempts to evict a line, the LQ is checked. If the hardware finds a matching entry with the Pinned bit set, the operation is denied. When a pinned load retires, it trivially becomes unpinned.

Most of the mechanisms in this design are already present in conventional processors. For example, in conventional processors, when a line is to be evicted from a cache level, the hardware informs higher levels of caches (i.e., smaller caches) so they also evict the line—with some variations depending on whether or not the cache hierarchy is inclusive. In some proposals, the higher levels may refuse to evict the line for performance or security reasons, prompting the initiating cache level to find another victim [257, 258]. *Pinned Loads* uses the same approach for pinned loads.

In conventional TSO cores, when the L1 wants to invalidate or evict a line, the hardware checks the LQ and, on a match, the corresponding load and its subsequent instructions are squashed. In *Pinned Loads*, the process is different in two ways. First, the invalidation or eviction may be denied. Second, the LQ check and the invalidation/eviction cannot happen in parallel: the check has to be done first in case the operation is denied.

Storing the Information in L1 Tags. This design adds a Pinned bit to each cache line in L1, to indicate whether or not the line is pinned. At runtime, when a load is to get pinned, *Pinned Loads* accesses the L1 and sets the Pinned bit of the line. When an L1 line receives an invalidation or is picked for eviction, if its Pinned bit is set, the operation is denied.

This design still keeps the Pinned bit in the LQ entries. Recall that a load can become pinned only after all of its older loads are pinned; hence the presence of the Pinned bit in the LQ enables the hardware to find this condition easily. In addition, LQ entries need one additional bit: the *Youngest Pinned Load* (YPL) bit. To understand its functionality, consider multiple pinned loads in the LQ that are accessing the same line. Only when the youngest of them retires can *Pinned Loads* clear the Pinned bit in the cache. Hence, for each pinned cache line, one of the LQ entries has the YPL bit set. When a new load is to be pinned, the hardware searches the LQ for an entry for the same line and the YPL bit set; if the entry is found, the hardware "passes the YPL bit" from the older to the newer entry and there is no need to set the Pinned bit in L1 cache again. When a pinned load with a set YPL bit retires, the L1 cache is accessed to clear the Pinned bit.

When using the Early Pinning of Section 6.5.2, a load may be declared pinned before the L1 receives the data. In this case, since the L1 does not have the line, we add a Pinned bit in the MSHR that the load uses. This is done as soon as a pinned load is issued. When the requested line is received and placed in the L1, the Pinned bit in the MSHR is copied to the L1.

The advantage of this design is that it decides whether to invalidate or evict an L1 line quickly, without waiting for an LQ access. This reduces the latency to respond to requests. However, a disadvantage is that this design requires extra requests from the pipeline to L1 to unpin lines. This fact puts extra pressure on L1 and increases the unpinning latency. Overall, because load pinning/unpinning operations are much more frequent than L1 invalidations or evictions, we do not use this design.

6.6.2 Optional Cache Shadow Table (CST)

The CST is a per-core hardware structure only used in Early Pinning. It records the mapping of each line pinned by the core—i.e., which set in L1 and which slice and set in the shared directory/LLC (Section 6.5.1). The hardware checks the CST before pinning a load to determine whether, with the addition of this load, all the pinned lines still have enough guaranteed space in the cache hierarchy.

A core has one CST for the directory/LLC and one for the L1 cache. Each CST is a hash table. Figure 6.6 shows the CST for the directory/LLC. Assume that *Pinned Loads* wants to pin a load L that accesses address A. First, *Pinned Loads* generates the set and slice numbers where A maps, hashes them, and uses the result to access a CST entry. An entry contains M records, each corresponding to a line. Each record has the hash of the line address, the LQ ID of the youngest pinned load that reads from the line, and a Valid bit. M is equal to or less than the maximum number of lines that can be pinned by the core in the same set and slice (i.e., W_d in Section 6.5.1).

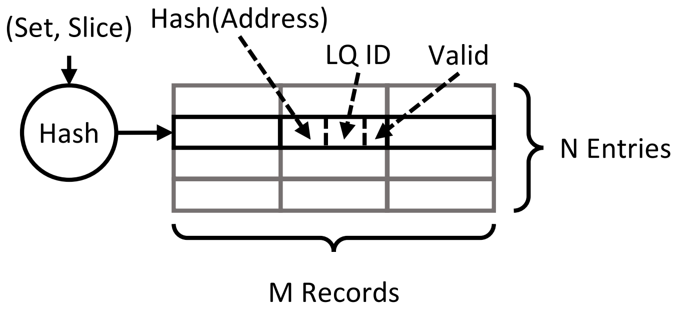


Figure 6.6: Cache Shadow Table (CST) for the directory/LLC.

At the indexed table entry, the hardware performs a CAM read to find if there is a valid entry for A. If there is, the line is already pinned by older loads, and hence the directory/LLC has enough resources to pin L. Then, the LQ ID field of the record is updated to L's LQ ID, and L is declared pinned.

If the CST entry does not contain a record for A, the hardware checks whether the entry has enough room for a new record. If so, a new record for A is created, its LQ ID field is updated to L's LQ ID, and L is declared pinned. Otherwise, the pinning is denied as there are not enough resources.

To reduce overhead, when a pinned load retires, we do not access the CST to potentially remove its entry. Instead, we let the potentially stale entry remain and remove it only when the hardware attempts to pin a new line. At that point, the hardware discovers if any of the records in the chosen entry has an LQ ID that is outside of the currently-used LQ entries. If so, the record is expunged.

One corner case that we handle is LQ ID wraparound, which could lead to using stale CST entries. We solve this problem by using a longer LQ ID tag in both the CST and LQ. For example, if the LQ has 64 entries, rather than using 6 bits for the LQ ID, we use 24 bits. Then, we use the modulo operation to map an LQ ID to a physical LQ entry. With this longer LQ ID tag, wraparound happens infrequently. When it happens, *Pinned Loads* stops pinning loads until all the pinned loads retire. During this time, loads reach their VPs and issue as they would on a safe scheme without *Pinned Loads*. Once all the pinned loads retire, the CST is cleared, and normal *Pinned Loads* execution resumes. Because wraparound is infrequent, the performance impact of this design is negligible. Other designs to handle LQ ID wraparound are possible.

The use of hashes in the CST may cause hash collisions. One class of collisions occurs when two different $\{set, slice\}$ pairs hash to the same CST entry. This is safe, as it only underestimates the capacity of a $\{set, slice\}$ combination. Another class of collisions occurs when the hashes of two different line addresses in a $\{set, slice\}$ match the same record. This collision needs to be detected. *Pinned Loads* detects it by always using the second field of the record (i.e., the LQ ID) to access the LQ entry and check whether the line address of the existing entry is indeed the same as the one we want to pin. If it is not, then we cannot pin the new load and *Pinned Loads* handles it as if there was not enough space in the $\{set, slice\}$.

The CST for the L1 cache operates similarly, except that there is no slice number, and that the number of records per entry M can be as high as the cache associativity.

Supporting Different Types of Cache Hierarchies Cache hierarchies can have different organizations, which may affect how *Pinned Loads* designs its CSTs. In particular, cache hierarchies typically have multiple levels of private caches (e.g., an L1 and an L2 level). In nearly all cases, there is no need to have a CST for a private L2 cache. This is true if the L2 is exclusive with respect to the L1: whether the L1 has enough space to hold a line does not depend on L2's organization. It is also almost always true if the L2 is inclusive with respect to the L1: L1 caches typically have a set count and an associativity that are lower than or equal to those of the L2 caches. Hence, it is almost always the case that, if a line has space in L1, it also has space in L2. If such statement is untrue, *Pinned Loads* would need a CST for a private L2. Finally, if the L2 is non-inclusive with respect to the L1, a more subtle analysis of the data flows allowed is required to determine the CST needs.

6.6.3 Cannot-Pin Table (CPT)

The CPT is a per-core hardware structure that records the addresses of lines that the core is not allowed to pin at the moment (Section 6.5.1). The CPT is placed near the LQ, which checks it before attempting to pin a line. A line's address is inserted in the CPT when the core receives an Inv*; the address is removed when the core receives a Clear. Our CPT can hold up to four addresses although, on average (Section 6.9.2), it only needs to hold one. If the CPT fills up and a request to insert an address cannot be serviced, the core stops pinning loads until the CPT is half empty.

We expect that a core that tries to write to a pinned line like Core 2 in Figure 6.5(a) will eventually succeed in inserting an entry in the CPT(s) of the reader core(s). However, there is a corner case when every time that Core 2 attempts to write, the CPT(s) in the reader core(s) are full and do not accept new entries. In this very unlikely case, Core 2 would never succeed in inserting its entry in the CPT(s).

To prevent this case, a more advanced design can add a small FIFO queue to the CPT with the IDs of writer cores that visited the node but found no space in the CPT. Then, when one CPT entry is released, it is reserved for a write from the core whose ID is at the head of the queue.

6.6.4 Effect of Limited-Sized Hardware Structures

Pinned Loads uses certain key hardware structures such as the CST, CPT, and extended LQ ID tag. Their limited size may sometimes cause *Pinned Loads* to operate with slightly lower performance, but never incorrectly. Specifically, when the CST cannot find space to pin a load, either because there is no space or because of a hash conflict, the core stops pinning loads until space can be found. Similarly, when the CPT fills up, the core stops pinning loads until the CPT is half empty. Finally, when LQ ID tag wraps around, the core stops pinning loads until all the pinned loads retire. In all cases, in the meantime, loads reach their VPs and issue as they would on a safe scheme without *Pinned Loads*. The execution is not as fast but it is correct and does not have deadlocks or livelocks.

6.7 COMPARISON TO A RELATED SCHEME

Our design to guarantee early that a load will not cause MCVs uses a mechanism to temporarily delay invalidations to a line. Ros et al. [259] proposed a mechanism with a similar goal in their WritersBlock protocol. Their purpose was to improve performance by allowing load-load reordering in TSO without squashes. Later, Tran et al. [248] applied the design to a speculative processor to allow loads to execute early without risking MCVs—the same goal as *Pinned Loads*.

We did not want to use Tran et al.'s aggressive design because its hardware is complex. In this section, we compare *Pinned Loads* to their design.

In the WritersBlock protocol, any load that has been issued speculatively causes its core to (i) reject an incoming write to the line and (ii) send a request to the directory, causing the directory to enter a new transient state for the line called WritersBlock. The rejected write is buffered and blocked in the directory. Other readers that arrive to the directory while in WritersBlock state can read the data. However, to prevent starvation, they get a ”tear-off” copy of the data: a copy that is uncacheable, does not get recorded in the directory, and can be used only once. Moreover, a directory entry in WritersBlock state cannot be evicted from the directory. Hence, if a read for a different line arrives to the directory/LLC and cannot allocate space because it would have to evict

WritersBlock-ed entries, the read gets a tear-off copy of the data from main memory and does not allocate a directory entry.

This is an aggressive design that allows any speculative load in any order to get the data—irrespective of how many older loads exist in the ROB, and without needing to guarantee that there is space to hold the line in caches or directory. However, the hardware changes required are major: a new transient directory state, which buffers the write and allows reads; writes that transition between three- and four-hop transactions; reads that dynamically turn uncacheable based on directory state; and new read transactions that directly grab data from memory and skip directory entry allocation. The result is challenging hardware. Perhaps more importantly, adding these no-directory-allocation and uncached paths creates *parallel paths* for transactions, namely cached and uncached paths, which are hard to verify for correctness.

In contrast, with *Pinned Loads*, we seek a simpler and safe design. A key source of complexity reduction is that *Pinned Loads* pins all the loads of a core *in strict program order*, and only when cache and directory *resources are guaranteed*. Further, we limit the complexity added to the coherence protocol as much as we can: we add no new directory states; we create no uncached or no-directory-allocation paths in the protocol; the directory buffers no new state; to attempt to write to pinned lines, we reuse processor retry mechanisms that have been used commercially to access busy directory lines; and, generally, we minimize the changes made to the directory and LLC, moving some functionality to structures that are local to cores.

6.8 EXPERIMENTAL METHODOLOGY

We model the architecture shown in Table 6.1 using cycle-level simulations with gem5 [230]. In the simulator, we model all the side effects of transient instructions. The baseline architecture is called UNSAFE, because it has no protection against speculative execution attacks. We use loads as transmitters and model the Comprehensive [246] and Spectre [18] threat models. In Comprehensive, squashes can be due to control-flow mispredictions, address aliasing, exceptions, and MCVs. In Spectre, the only relevant squashes are those due to control-flow mispredictions.

We augment the UNSAFE architecture with the hardware defense schemes in Table 6.2. These schemes protect loads until they reach their VP as follows: FENCE stalls loads with fences; Delay-On-Miss (DOM) [120, 121] stalls speculative loads that miss in the L1; STT [116] stalls loads whose arguments are tainted by transiently-read data.

We model each hardware defense scheme with the configurations of Table 6.3. They include COMP and SPECTRE, which are the defense schemes without extensions under the Comprehensive and Spectre model, respectively. They also include LP and EP, which are the defense schemes augmented with Late Pinning and Early Pinning, respectively, under the Comprehensive model.

Table 6.1 shows the area, dynamic read energy, and leakage power of the CST, which is the main *Pinned Loads* hardware structure. The data are obtained using Cacti [260] with 22nm technology.

Table 6.1: Parameters of the simulated architecture.

Parameter	**Value**
Architecture	1 (SPEC17) or 8 (SPLASH2 & PARSEC) out-of-order x86 cores at 2.0 GHz
Core	8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, LTAGE branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher
Private L1-D Cache	32 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher
Shared L2 Cache (LLC)	Slice: 2 MB, 64 B line, 16-way, 8 cycles RT latency
Coherence	Directory-based MESI protocol
Network	Ordered, 4×2 mesh, 128b link, 1 cycle/hop
DRAM	50 ns RT latency after L2
L1 CST	12 entries, 8 records/entry; Area: $0.0008mm^2$; Dynamic read energy: $0.6pJ$; Leakage power: $0.17mW$
Dir/LLC CST	40 entries, 2 records/entry; W_d: 2 per slice and set for each core; Area: $0.0005mm^2$; Dynamic read energy: $0.4pJ$; Leakage power: $0.17mW$
CPT	4 entries; Negligible area, energy, and power
LQ ID Tag	24 bits

Table 6.2: Hardware defense schemes modeled.

Scheme	**Description of the defense**
UNSAFE	No defense: unmodified x86 architecture
FENCE	Stall all speculative loads with fences
DOM	Stall speculative loads on L1 miss [120, 121]
STT	Stall loads that are tainted by transient data [116]

We run SPEC17 applications [231] on a single core, and SPLASH2 [253] and PARSEC [254] applications on 8 cores. For SPEC17, we use the *reference* input size. For each application, we use SimPoint [232] to generate up to 10 representative intervals that accurately characterize the end-to-end performance of the application. Each interval consists of 50M instructions. We run Gem5 on each interval with system-call emulation mode with 1M warm-up instructions. For SPLASH2 and PARSEC, we use the *simmedium* input size and run full-system simulation for the region of interest (ROI).

Table 6.3: Extensions added to the defense schemes.

Config.	**Description**
COMP	No extension: Unmodified scheme under Comprehensive model
LP	COMP + *Pinned Loads* with Late Pinning
EP	COMP + *Pinned Loads* with Early Pinning
SPECTRE	No extension: Unmodified scheme under Spectre model

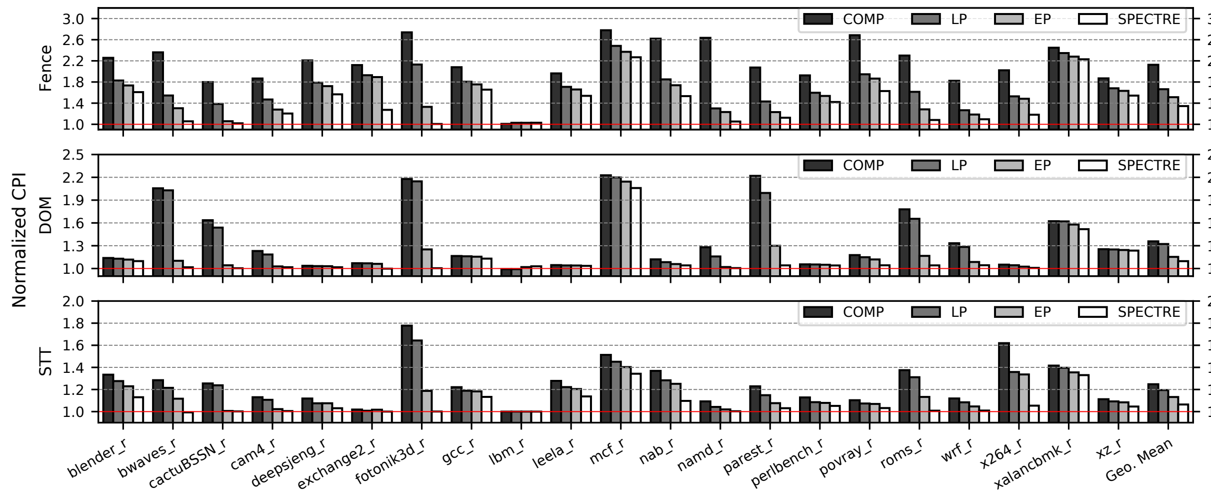


Figure 6.7: Normalized CPI of SPEC17 programs on different architecture configurations, all normalized to UNSAFE. The three plots correspond, from top to bottom, to configurations built on the FENCE, DOM, and STT defense schemes. Each plot has a different Y-axis range.

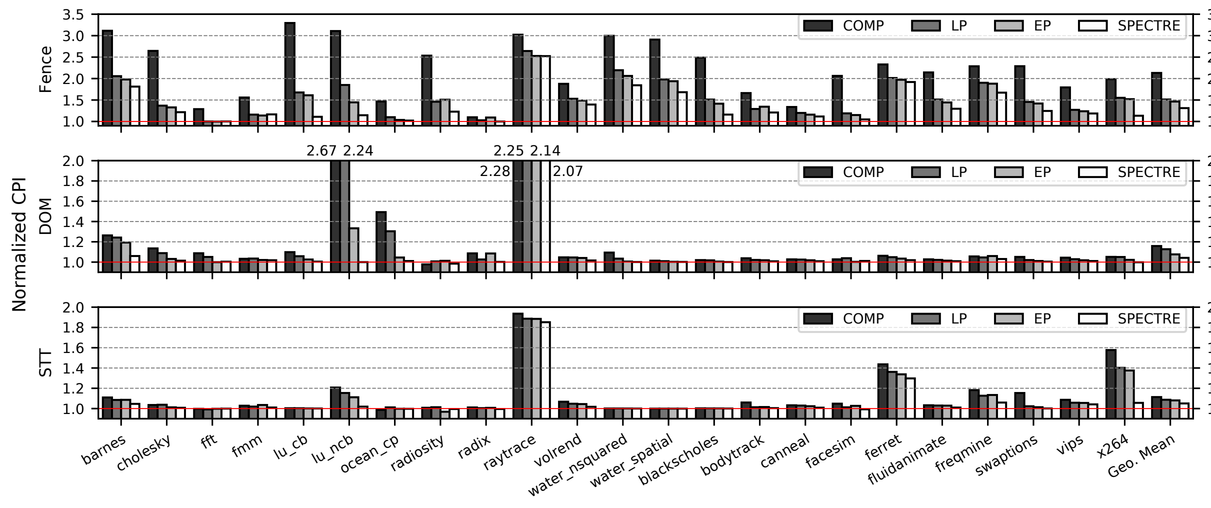


Figure 6.8: Normalized CPI of SPLASH2 and PARSEC programs on different architecture configurations, all normalized to UNSAFE. The three plots correspond, from top to bottom, to configurations built on the FENCE, DOM, and STT defense schemes. Each plot has a different Y-axis range.

6.9 EVALUATION

6.9.1 Overall Performance Results

Performance on SPEC17. Figure 6.7 shows the normalized CPI of SPEC17 programs on all the defense schemes with the extensions listed in Table 6.3 (Comp, LP, EP, and Spectre). The three plots correspond, from top to bottom, to the Fence, DOM, and STT defense schemes. Each plot has a different Y-axis range. All bars are normalized to Unsafe. Each plot shows each SPEC17 program and the geometric mean of all.

Going from top to bottom, we see that Fence has the highest execution overhead among all the schemes evaluated. On average, it has a geometric mean execution overhead of 112.6% with the Comprehensive model. With Late Pinning (LP), we reduce the execution overhead to 66.4%. By using Early Pinning (EP), we further reduce the execution overhead to 51.3%, which is close to the execution overhead with the Spectre threat model (34.5%).

DOM has a moderate execution overhead on SPEC17. On average, it has a geometric mean execution overhead of 35.8% under Comprehensive. Because DOM only delays speculative loads that miss in L1, it usually has high execution overhead on applications that have poor L1 hit rate, in which case LP cannot effectively pin the loads and "pass the VP" (Section 6.5.2). With LP, the average execution overhead is only reduced to 32.3%. EP, on the other hand, can better handle cache misses (Section 6.5.2), and reduces the average execution overhead to 15.3%. This is close to Spectre's (9.7%). EP provides huge speedups to benchmarks with high L1 miss rates, such as `bwaves` and `fotonik3d`.

STT's average execution overhead is 24.8% on SPEC17 under Comprehensive. It is the fastest scheme evaluated. LP reduces the average execution overhead to 19.5% and EP to 13.2%. The execution overhead under the Spectre model is 6.4%.

Overall, we see that augmenting existing defense schemes with EP substantially reduces the execution overhead of the schemes.

Performance on SPLASH2 and PARSEC. Figure 6.8 shows the normalized CPI of SPLASH2 and PARSEC applications. The figure is organized as in Figure 6.7. We see that Fence's geometric mean execution overhead is 113.1% under Comprehensive. Because of the relatively high L1 hit rate of SPLASH2 and PARSEC applications, both LP and EP offer good speedups: they reduce the execution overhead to 51.2% and 46.4%, respectively. The execution overhead under Spectre is 31.1%.

DOM has a moderate execution overhead on SPLASH2 and PARSEC under Comprehensive, mainly because of high L1 hit rate. On average, its execution overhead is 15.8%. LP reduces it to 12.7%, and EP further reduces it to 7.6%. The execution overhead under Spectre is 4.2%. The lu_ncb and raytrace applications have a high L1 miss rate, but lu_ncb's branches are resolved quickly (hence SPECTRE performs well), unlike raytrace's. EP reduces lu_ncb's execution overhead from 167.2% to 33.3%.

STT has small execution overheads on SPLASH2 and PARSEC. On average, it has a geometric mean execution overhead of 11.3% under Comprehensive. With LP, it is reduced to 8.7%. EP further reduces it to 8.1%. The execution overhead under Spectre is 5.1%. The x264 application still has much higher execution overhead under EP than under SPECTRE. The reason is that it has dependencies between loads, which is a pattern EP cannot efficiently handle.

Overall, for these programs, we observe that LP, and especially EP, substantially reduce the execution overheads of all schemes.

Network Traffic Overhead. While *Pinned Loads* does not change the network traffic of the SPEC17 applications, it could increase the traffic of the SPLASH2 and PARSEC applications. In practice, we find that enabling *Pinned Loads* on FENCE, DOM, and STT has no significant impact on network traffic. The reason is that very few writes and evictions have to retry due to pinning. Even in the worst-case applications, only 14.8 writes and 0.05 evictions are retried per *million* instructions.

Breakdown of the Execution Overhead. We now assess the big-picture impact of *Pinned Loads* on the execution overhead of the defense schemes. Figure 6.9 combines defense schemes (FENCE, DOM, and STT) and applications (SPEC17 and Parallel ones). For each combination, it shows, first, the execution overhead of COMP normalized to UNSAFE and broken down into the different sources of speculation. These bars are like those in Figure 6.1. The second and third bars of each combination are the execution overheads of the same defense scheme augmented with LP and EP, respectively. Note that two of the bars in the graph are cut off.

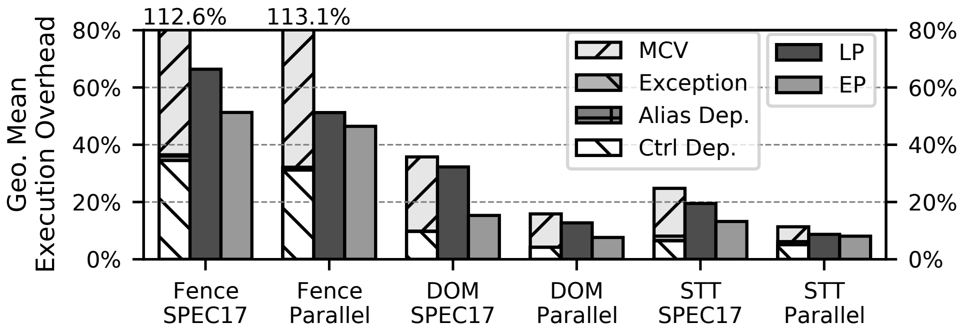


Figure 6.9: Breakdown of the execution overhead due to different sources and for different schemes, all relative to UNSAFE.

We see that, under the Comprehensive model, the execution overhead of every defense scheme is mainly caused by stalls to prevent potential MCVs and, to a lesser extent, control dependencies. LP and EP focus on removing the MCV overhead. We see that LP and, especially, EP eliminate most of the MCV overhead. The upper bound of EP's effectiveness is to eliminate all the MCV overhead. In that case, it would be nearly as if we only had control dependence overheads—which is the Spectre model overhead. From the figure, we see that, in the case of FENCE, EP has an absolute 15% higher overhead than Spectre.

6.9.2 Analysis of the Hardware Structures

Cache Shadow Table (CST) Configuration. Ideally, the CST should precisely track where each pinned line maps in the L1 cache and in the directory/LLC. However, in practice, to minimize area overhead, we reduce the CST's number of entries and number of records per entry. As a result, there are false positive conflicts: the CST claims the load cannot be pinned due to lack of space while, in reality, there is space.

To decide on the sizes of the CSTs, we perform a sensitivity analysis. Our chosen default sizes (Table 6.1) are 12 entries with 8 records per entry for the L1 CST and 40 entries with 2 records per entry for the Dir/LLC CST. With this design, we find that the average L1 CST false positive rates are smaller than 0.02% on SPEC17, and than 0.01% on SPLASH2 and PARSEC for all the defense schemes with EP. Further, the average Dir/LLC CST false positive rates are smaller than 0.4% on SPEC17, and than 0.02% on SPLASH2 and PARSEC for all the schemes with EP. Hence, false positives are rare.

We measured the execution overhead of the different defense schemes (with EP) and programs for different CST sizes. On average, the execution overhead with our chosen configuration is 3.6% higher than with an infinite CST.

Cannot-Pin Table (CPT) Size. A line is inserted into the CPT only when a write fails a retry after having been deferred. We collect the average and the maximum number of lines that are in the CPT at a time. We use an ideal CPT and run SPLASH2 and PARSEC. On average, the CPT only needs to hold one line, and the maximum number of lines is 4–7 for all the schemes. Thus, we use a default CPT with 4 entries (Table 6.1). With this size, we see less than 0.0001 CPT overflows per insertion attempt for a few applications, and no overflows for most.

Smaller Directory/LLC Partition Size. Our default EP design allows a core to pin up to 2 lines per set in the directory/LLC at a time (W_d is 2). We repeat our experiments with W_d equal to 1 while keeping the same CST size. We see that the overhead of the schemes with EP increases: for FENCE, it increases from 51.3% to 54.7% on SPEC17 and from 46.4% to 47.0% on parallel applications; for DOM, it changes from 15.3% to 18.5% on SPEC17 and from 7.6% to 8.0% on parallel applications; for STT, it increases from 13.2% to 14.7% on SPEC17 and remains the same on parallel applications. Consequently, keeping W_d equal to 2 is best.

Hardware Overhead. The main storage structure added by *Pinned Loads* is the CST used by EP. The CPT and the extended LQ ID tags are very small. With our default configuration and including the tags, the L1 CST is 444 bytes and the Directory/LLC CST is 370 bytes. We use CACTI 7.0 [260] to estimate the CST area, dynamic read energy, and leakage power at 22nm. As shown in Table 6.1, these numbers are very small.

6.10 OTHER RELATED WORK

Speculative execution attacks exfiltrate secret data by exploiting different types of speculation, such as control-flow speculation [18, 22, 27, 28, 261, 262, 263], memory dependence speculation [264], and memory consistency speculation [20, 249]. For memory consistency speculation, Ragab et al. [20] and Skarlatos et al. [249] demonstrate how an attacker from a core can repeatedly create squashes due to MCVs in another core. From here, many attacks are possible. For example, if the victim gets a random number, the attacker can force selective squashes and retries and bias the random number generator. Defending against this type of speculation attack is expensive. *Pinned Loads* substantially reduces the cost of such defense.

6.11 CONCLUSION

To reduce the overhead of defenses against speculative execution attacks, this chapter presented *Pinned Loads*, a general technique that helps instructions reach their VPs sooner. Under the Comprehensive threat model, we found that the progress of the VP is mostly impeded by waiting until no MCVs are possible. Hence, *Pinned Loads* tries to make loads invulnerable to MCVs as early as possible—a process we call *pinning* the loads in the ROB. In this chapter, we described the several hardware mechanisms needed by *Pinned Loads*, and two possible *Pinned Loads* designs with different tradeoffs. Our evaluation showed that *Pinned Loads* is very effective: extending the fence-insertion, Delay-On-Miss, and STT defense schemes with *Pinned Loads* reduces these schemes' average execution overhead on SPEC17 and on SPLASH2/PARSEC applications by about 50%. Specifically, on SPEC17, the execution overhead of the three defense schemes decreases from 112.6% to 51.3%, from 35.8% to 15.3%, and from 24.8% to 13.2%, respectively; on SPLASH2/PARSEC, the execution overhead decreases from 113.1% to 46.4%, from 15.8% to 7.6%, and from 11.3% to 8.1%, respectively.

CHAPTER 7: Faster Safe Execution Through Program Analysis

7.1 INTRODUCTION

In Chapter 6, we discussed a hardware-only mechanism that helps speculative instructions reach their VP early and thus reducing the performance overhead of various defenses to speculative side-channel attacks. In this chapter, we propose a software-hardware co-design scheme named *InvarSpec* that safely executes speculative instructions even before they reach their VP. InvarSpec is based on the observation that a speculative instruction can become *Speculation Invariant* at some point before turning non-speculative. By this, we mean that speculative instruction *i* reaches a point when (i) whether *i* will execute or not does not depend on speculative state, and (ii) the operands of *i* do not depend on speculative state. When a speculative instruction is speculation invariant and its operands are ready, we say it reaches its *Execution-Safe Point* (ESP).

Figure 7.1 shows two simple examples of speculation invariant loads. Consider the Comprehensive threat model [118], where all instructions remain speculative—and therefore squashable —until they reach the Reorder Buffer (ROB) head. Figure 7.1(a) shows a speculative load following an unresolved branch where the load address *x* is not dependent on any of the two branch paths. We say that *ld x* is speculation invariant and, as soon as *x* is ready, speculative *ld x* reaches its ESP. No matter which direction the branch finally takes, *ld x* will always execute and access the same address. Figure 7.1(b) shows the same speculative load following an earlier load whose return data *y* does not directly or indirectly affect the register that *ld x* uses to generate the *x* address. Once again, *ld x* is speculation invariant and, as soon as *x* is ready, *ld x* reaches its ESP.

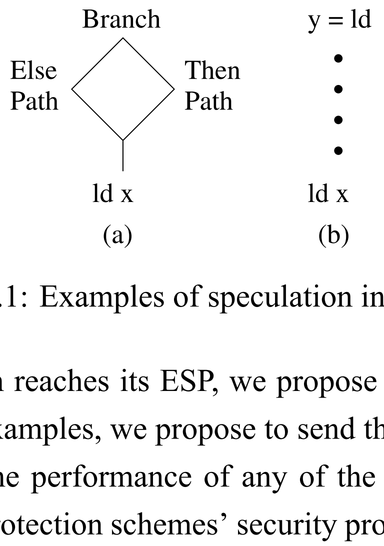


Figure 7.1: Examples of speculation invariance.

When a speculative instruction reaches its ESP, we propose to lift any protection and execute the instruction. In the previous examples, we propose to send the load request to memory without protection. With this strategy, the performance of any of the previous protection schemes will improve. At the same time, the protection schemes' security properties will not change: executing a speculation invariant instruction without protection will not reveal any more secrets than the underlying hardware protection scheme would reveal with the non-speculative execution of the instruction.

Unfortunately, hardware structures alone cannot exploit these insights because the hardware is only aware of the current speculative path being executed and does not reason about all possible paths. Instead, we need a program analysis infrastructure to analyze the program and inform the hardware of the speculation invariance of instructions.

We introduce a framework to exploit speculation invariance for higher performance without hurting security. InvarSpec includes a program analysis pass that identifies, for each instruction i under protection, the set of older instructions (e.g., the branch and the load $y = ld$ in Figure 7.1) that are *Safe* for i—i.e., those instructions that do not prevent i from becoming speculation invariant. At runtime, the InvarSpec micro-architecture loads this information and uses it to identify when speculative instructions can execute early, without protection.

InvarSpec is one of the first defense schemes against speculative execution attacks that combines cooperative compiler and hardware mechanisms. It consists of an analysis pass for binaries, currently implemented for x86 binaries, and pipeline micro-architecture that uses this information at runtime.

To evaluate InvarSpec, we apply its analysis pass on the SPEC17 and SPEC06 programs and model its micro-architecture in a cycle-level simulator. Our results show that InvarSpec is effective. On average, using InvarSpec reduces the execution overhead of fence-based protection from 195.3% to 108.2%, the execution overhead of DOM from 39.5% to 24.4%, and the execution overhead of InvisiSpec from 15.4% to 10.9%.

In summary, the chapter makes the following contributions:

- Presents *Speculation Invariance* to improve the performance of hardware security schemes against speculative execution attacks without hurting their security properties.
- Develops and evaluates the InvarSpec analysis pass.
- Develops the InvarSpec micro-architecture and uses it, together with the analysis pass, to improve the performance of three existing hardware security schemes.

7.2 SPECULATION INVARIANCE

7.2.1 Main Idea

As pointed out above, several defense schemes (at least [118, 119, 120, 121, 122]) use hardware mechanisms to block leakage while a transmitter is potentially transient and thus *unsafe*. If one could disable such mechanisms before the transmitter reaches its VP and, therefore, execute the transmitter speculatively without protection, one would reduce the overhead of these defense schemes.

In this chapter, we propose a combined compiler and hardware scheme called *InvarSpec* that allows the lifting of these protection mechanisms for speculative instructions. The key idea is to identify *Speculation Invariant* instructions and allow them to execute while speculative without protection.

A speculative instruction *i* becomes *Speculation Invariant* when (i) whether *i* will execute is not a function of speculative state, and (ii) the operands of *i* are not a function of speculative state. When an instruction is speculation invariant and its operands are ready, we say that the instruction reaches its *Execution-Safe Point* (ESP).

Intuitively, ESP is the earliest point when speculative instruction *i* can execute and is guaranteed to eventually commit using the exact same operands—no matter how many times it is squashed by older instructions due to incorrect speculation. At an instruction's ESP, InvarSpec permits its speculative execution without protection.

Since the definition of speculative instruction depends on the threat model (e.g., Spectre or Comprehensive as defined in Section 2.3.3), speculation invariance and ESP for an instruction depend on the threat model. For example, assume that the branch in Figure 7.1(a) is unresolved and that there is no unresolved branch between the two loads in Figure 7.1(b). In Figure 7.1(a), *ld x* is speculation invariant under both Spectre and Comprehensive; in Figure 7.1(b), *ld x* is only speculative (and speculation invariant) under Comprehensive.

Figure 7.2(a) shows four points in the lifetime of a load instruction—which we use as a representative transmitter. Time increases to the right. The Ready point is when the load operands become available and the load is ready to be sent to memory speculatively. Current defense schemes place restrictions on what the load can do at this point. Sometime later, the load becomes speculation invariant and reaches its ESP. At this point, with InvarSpec, the load can be sent to memory without protection. Later, the load reaches its VP, where it becomes non-speculative and can be safely sent to memory without protection. Finally, the load retires. Effectively, InvarSpec moves the safe point of sending the load to memory from VP to ESP, reducing the overhead of the defense mechanism.

According to the threat model that we use (Section 7.3), it is safe to expose the side effects of a speculation invariant instruction. Its execution does not reveal any more secrets than the underlying hardware defense scheme would reveal with the non-speculative execution of the instruction.

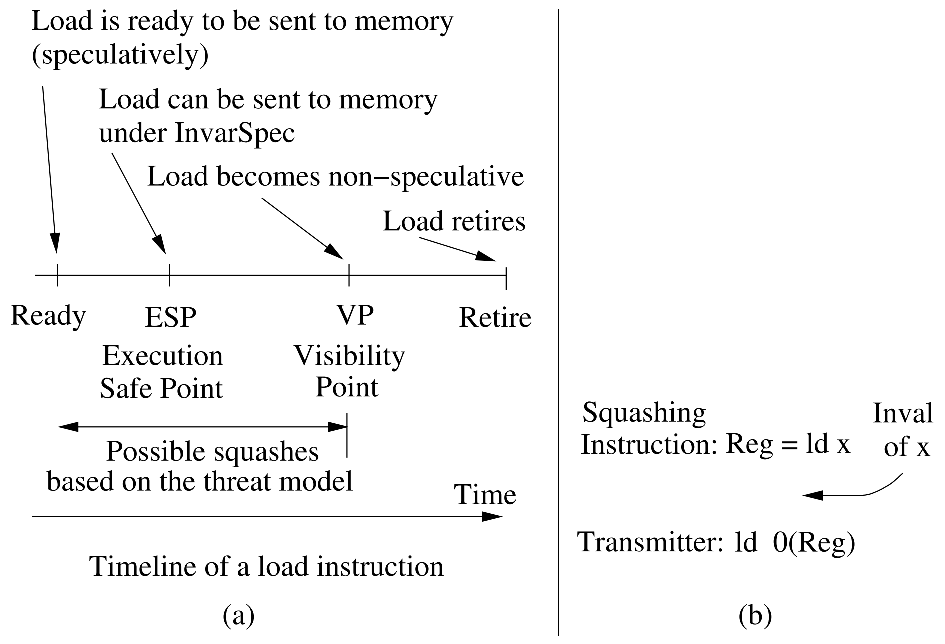

Load is ready to be sent to memory (speculatively)
Load can be sent to memory under InvarSpec
Load becomes non−speculative
Load retires
Ready
ESP
Execution Safe Point
VP
Visibility Point
Retire
Possible squashes based on the threat model
Time
Timeline of a load instruction
(a)
Squashing Instruction: Reg = ld x
Inval of x
Transmitter: ld 0(Reg)
(b)

Figure 7.2: Supporting speculation invariance.

7.2.2 Definitions

The InvarSpec framework has two important types of instructions: *Transmitters* and *Squashing* ones. Transmitters (e.g., loads) are inherited from the defense scheme that InvarSpec augments. Squashing instructions are those that can cause squashes that may lead to security violations. Squashing instructions are defined by the threat model. For the Spectre model, they are branches; for the Comprehensive model, they are branches, loads, and any instructions capable of causing exceptions. For example, a load may be squashed on reception of an invalidation for the address that it loaded, due to the processor's memory consistency mechanisms.

In this chapter, we use loads as the transmitters and apply the Comprehensive model. In addition, we focus our analysis on the most challenging squashing instructions: branches (which can be mispredicted) and loads (which may be involved in exceptions or consistency violations and, on re-execution, can read a new value). Instructions other than loads may also be involved in squashes due to exceptions, but they are much easier to handle. We discuss exceptions in Section 7.2.6.

Any instruction *i* that follows a squashing one and that has executed speculatively, may have to be squashed. Only if *i* had reached its ESP when it executed, it is guaranteed that, even after the squash, *i* will be re-executed and will use the same operands. For this reason, in InvarSpec, it is key to identify when a transmitter reaches its ESP. Only then can the transmitter execute without protection.

If InvarSpec only uses hardware support to identify when a transmitter reaches its ESP, it produces conservative results. The hardware uses the following algorithm.

A transmit instruction *i* reaches its ESP when its operands are ready and each of its older squashing instructions in the ROB has: (i) executed and (ii) produced its final result. As a shorthand for conditions (i) and (ii), we will say that the older squashing instruction has reached its *Outcome Safe Point* (OSP)—i.e., the point where its result will not change irrespective of any future squashes.

When has an executed squashing instruction "produced its final result"? If we do not consider loads, we can say that non-load squashing instructions have produced their final result when all older branches have resolved. However, loads work differently. A load may reach its ESP (because it is also a transmitter) and execute, then get squashed, and then, as it re-executes with the same operand (i.e., the memory address), it may read a different value from memory—if another thread has written to the same location in between. An example is shown in Figure 7.2(b), where *ld x* reads a value into *Reg*, then gets squashed by an invalidation of *x* and then, as it re-executes, reads a different value from location x. Consequently, for a load to reach its OSP, it has to reach its ESP, execute, and then reach a point where it cannot be squashed anymore—typically, the ROB head.

Therefore, for the TSO-based x86 architecture and squashing instructions that we consider, the condition for an executed squashing instruction *i* to reach its OSP is as follows. First, if *i* is not a load, *i* reaches its OSP when (i) all the older branches in the ROB are resolved and (ii) there is at most one older load in the ROB, which is at a point where it cannot be squashed anymore. Second, if *i* is a load, *i* reaches its OSP when is at a point in the ROB where it cannot be squashed anymore. As indicated before, under the Comprehensive threat model, loads cannot be squashed anymore only when they are at the ROB head.

7.2.3 Using Program Analysis Information

A program analysis pass can help the hardware algorithm just described to be more aggressive. It can identify, for each transmitter, the set of older squashing instructions that are *Safe* for the transmitter.

Safe instructions for an instruction *i* are older squashing instructions that, even if they have not executed and generated their final result (i.e., they have not reached their OSP), they cannot prevent *i* from becoming speculation invariant.

Intuitively, the transmitter can become speculation invariant despite the fact that these older squashing instructions have not yet completed. Consequently, the hardware *does not need to consider them* when determining whether the transmitter is speculation invariant.

What are safe branches and safe loads for the x86 architecture? For a given load *i*, safe branches are those whose outcome cannot affect whether *i* will execute and what operands *i* will use. An example was shown in Figure 7.1(a). For a given load *i*, safe loads are those whose return data cannot affect directly or indirectly the address that *i* loads from. An example was shown in Figure 7.1(b). If, instead, load *i* is control dependent on a branch or data dependent on a load, then the branch or load is not safe for *i*.

The InvarSpec framework includes an analysis pass that takes a source or executable program and determines, for each transmitter, the set of safe squashing instructions. It then places these instructions' program counters (PCs) in a *Safe Set* (SS) for the transmitter.

At runtime, when a transmitter is about to execute and the InvarSpec hardware wants to determine whether the transmitter has reached its ESP, the hardware computes the ESP condition described in the box of Section 7.2.2. However, the hardware also reads the transmitter's SS and prunes from the computation all of the squashing instructions in the ROB that are in the SS of the transmitter. Specifically, older branches and loads that are in the SS do not need to have reached their OSP for the hardware to conclude that the transmitter has reached its ESP. As a result, the transmitter reaches its ESP sooner and can execute sooner.

Finally, from this discussion, it is clear that we want the squashing instructions j that are not in the SS of the transmitter to reach their OSP as soon as possible. Sadly, each of them needs to fulfill the conditions listed on Section 7.2.2, which require that even older squashing instructions execute and reach their OSP. Fortunately, we can speed-up this process if we also generate the SS for each squashing instruction j. Any instruction in j's SS can be disregarded as we compute the conditions for j to reach its OSP. With this insight, we help j reach its OSP sooner. Hence, InvarSpec also builds the SS for squashing instructions.

7.2.4 The Complete InvarSpec Framework

The InvarSpec framework has two parts: (i) an analysis pass that generates the SS for transmit and squashing instructions, and (ii) hardware that, at runtime, loads the SSs and computes the ESP conditions. The analysis has two levels of support. The first one, called *Baseline*, populates the SS of instruction *i* with only those squashing instructions that are safe for *i* no matter what execution path the program takes.

The second level, called *Enhanced*, is more aggressive. It additionally places in the SS of *i* some squashing instructions that are *not* safe for some execution paths—as long as the hardware can detect when these paths are executed and prevent *i* from being executed until *i* is indeed speculation invariant. With this support, when the other paths are followed, *i* can be executed earlier. *Enhanced* improves the analysis by exploiting dynamic path execution behavior.

7.2.5 Handling Store-to-Load Forwarding

When a load reaches its ESP and there is an older store, we need to ensure that whether the store and the load alias is invisible to an attacker. Otherwise, the attacker could deduce the address of the load. Specifically, if store and load alias and the load gets the data from the store, the attacker can deduce the alias by not observing a load access to the cache hierarchy. To solve this problem, we change the microarchitecture slightly as follows. The load is always issued to the cache hierarchy and if, at this point or later, the store address is found to alias, the load gets the data from the store and ignores the data returned from the cache hierarchy.

Relevant to InvarSpec is to understand when is the point where a load reaches its OSP. Such point requires not only that the load not be squashable anymore. In also requires that all of its older stores have been resolved—and hence that the load has been able to read the correct data, either from memory or from a store. InvarSpec implements this algorithm.

7.2.6 Handling Exceptions

Branches and loads are challenging instructions because, when they cause or are involved in a squash, they may change (i) what subsequent program instructions execute, and (ii) what operand values such subsequent instructions take.

Consider now exceptions. We assume an environment with no self-modifying code and no attacker-tampered executable. Here, there are two cases to consider. One is when the OS is able to service the exception and resume the program execution. The second case is when the exception causes program termination.

For the first case, InvarSpec's analysis only considers exceptions that involve the re-execution of loads, since loads may read a new value on re-execution. When only non-load instructions are involved, the re-execution after the exception is the same as the execution before. Hence, non-loads involved in exceptions do not need to be considered by InvarSpec.

The second case is when the exception causes program termination. In this case, we argue that no harm occurs from executing any speculation invariant transmitters that appear after the excepting instruction in program order. The reason is that such instructions are, by definition, control- and data-flow independent of the excepting instruction. As a result, unless the programmer or compiler explicitly places a fence in the code, the programmer can have no expectation about their execution order with respect to the excepting instruction—e.g., a different compiler could have hoisted these speculation invariant transmitters above the excepting instruction. Overall, program-termination exceptions do not affect InvarSpec's analysis either.

In summary, InvarSpec's analysis only needs to be concerned with exceptions that involve the re-execution of loads and are non terminating. Hence, the analysis of squashing instructions is limited to branches (which mispredict) and loads (which are involved in non-terminating exceptions and consistency violations).

7.3 THREAT MODEL

InvarSpec inherits the transmitters and the threat model from the hardware defense scheme that it augments. In this chapter, we augment defense schemes that use loads as the transmitters and Comprehensive as the threat model (Section 2.3.3). As indicated above, in this threat model, the analysis only needs to focus on two types of squashing instructions: branches (which can mispredict) and loads (which can re-load a new value after a non-terminating exception or a memory consistency violation—i.e., when certain speculatively loaded data receives an invalidation or suffers a cache eviction). The other type of instructions involved in exceptions are handled by existing hardware and the OS. In our model, victim and attacker can run on different cores or on different SMT contexts of a core.

InvarSpec allows speculative transmitters that are speculation invariant to execute speculatively without protection. InvarSpec is secure because it does not change the security properties of the defense scheme that it augments. Indeed, the execution of these speculative instructions does not reveal any more information than the underlying defense scheme would reveal with the non-speculative execution of the instructions.

We are expressly not considering attacks where the exact timing of when these speculative instructions execute would create a side channel. The defense schemes discussed [118, 119, 120, 121, 122] use the same assumption.

We assume that the SS information generated by the analysis pass for a program and attached to its executable is correct (e.g., signed and checked for trusted binaries). This is the case in the cross- and in-domain settings (Section 2.3). In these settings, the victim is compiled by a benign compiler that generates a correct SS. In contrast, in the domain-bypass setting, the program itself is malicious. However, domain-bypass attacks [19, 21, 23, 25, 265] exploit an implementation issue—deferred handling of exceptions—which is fixed in upcoming processors [266], and so are not the focus of forward-looking defenses. We consider them out of scope.

We further assume that a program's SS information is not tampered with. In the cross-domain setting, the victim has no motivation to tamper with its own SS. In the in-domain setting, the sandbox prevents any attacker-controlled code from tampering with the SS (which would be computed or verified by the sandbox's trusted runtime system). Moreover, the integrity of the SS in distributed software packages can be verified together with the integrity of the entire package, using well-established integrity verification techniques such as digital signatures [267, 268]. In all of these cases, if an attacker is able to tamper with the victim's SS, then she is able to modify the binary, which means that she can mount much more harmful attacks than speculative execution attacks.

7.4 THE INVARSPEC ANALYSIS PASS

InvarSpec includes an intra-procedural program analysis pass that accepts as input a program in source code or binary. Source code is preferred, since it allows a better analysis because it contains more information. InvarSpec is also told what kind of instructions are transmitters and squashing ones, and the threat model. InvarSpec can support multiple threat models and augment multiple hardware defense schemes.

The analysis pass generates, for each transmit and squashing instruction i, the set of squashing instructions that are safe for i. The program counters (PCs) of these safe squashing instructions form the Safe Set (SS) for i. The InvarSpec pass has two levels: the *Baseline* analysis and the more aggressive *Enhanced* analysis. We consider each in turn.

7.4.1 Baseline Analysis

Basic Algorithm. InvarSpec starts by generating the Program Dependence Graph (PDG) [269] of each procedure in the program. The PDG represents the dependence relationships among the instructions in the procedure. Each instruction is a node, and a directed edge from node i to node j means that i is directly control or data dependent on j. The edge is labeled "CD" if it is a control dependence, or "DD" if it is a data dependence.

The algorithm to generate the PDG of a procedure takes as inputs the procedure's control-flow graph (CFG) and data-dependence graph (DDG). The DDG includes dependencies through both registers and memory. For each instruction i, the algorithm adds an outgoing edge to all the instructions d that i directly depends on.

InvarSpec then computes the SS for each transmit and squashing instruction in the procedure. Algorithm 7.1 shows the pseudo-code for *getSS*, which computes the SS for instruction i. *getSS* takes as inputs i and the CFG, DDG, and PDG of the procedure. It first computes *ancSI*, which is the set of all the squashing instructions that are ancestors of i in the CFG. These are potential candidates for the SS. Then, Line 7.1 calls *getIDG*, which computes the Instruction Dependence Graph (IDG) of i.

The IDG of i is a subgraph of the PDG that includes i plus all the instructions that may affect

Algorithm 7.1: Computing the SS for an instruction.

```
Function getSS(i, CFG, DDG, PDG) is
    $ancSI \leftarrow \{a \in \text{getAnces}(CFG, i) \mid \text{isSquashInsn}(a)\}$
    $IDG \leftarrow \text{getIDG}(i, CFG, DDG, PDG)$
    $deps \leftarrow \{d \in \text{getDesc}(IDG, i) \mid \text{isSquashInsn}(d)\}$
    $SS \leftarrow ancSI \setminus deps$
    return SS
end
Function getIDG(i, CFG, DDG, PDG) is
    $IDG \leftarrow \text{DirectedGraph}()$
    for $d$ in getCtrlDeps($CFG, i$) do
        addNode($IDG, d$)
        addEdge($IDG, i, d,$ "$CD$")
        addDescGraph($d, IDG, PDG$)
    end
    for $d$ in getDataDeps($DDG, i$) do
        if $\neg(\text{isLoad}(i) \wedge \text{isStore}(d))$ then
            addNode($IDG, d$)
            addEdge($IDG, i, d,$ "$DD$")
            addDescGraph($d, IDG, PDG$)
        end
    end
end
```

whether i executes or the values of i's source operands. Intuitively, the instructions in the IDG should not be placed in the SS for i. If i is a load, the IDG does *not* contain stores that may update the memory *location* that i loads. Such stores are in the DDG because the DDG captures all the data dependencies, including those that affect the load's result; such stores are *not* in the IDG because they cannot affect whether i executes or the values of i's operands.

getIDG first creates an empty IDG graph (Line 7.1). It then adds to the graph all the instructions that i has direct control dependence on or that i's source operands have direct data dependence on. Finally, for each such instruction, *getIDG* calls *addDescGraph*, which adds to the IDG all the descendants of the instruction in the PDG.

Back to *getSS*, Line 7.1 collects all the squashing instructions from the IDG into *deps*; i itself is not in *deps* unless it depends on itself (due to a program loop). Finally, Line 7.1 subtracts *deps* from *ancSI*. The result is the SS of i.

Procedure Calls. The InvarSpec analysis pass is intra-procedural and, therefore, only considers dependencies inside a procedure. Interactions between procedures are handled as follows. First, consider a caller procedure. InvarSpec conservatively assumes that the callee may modify any memory location. Hence, InvarSpec treats a procedure call instruction as a store that may alias with any subsequent loads. For registers, InvarSpec uses calling conventions, which preserve some register values.

Second, consider a callee procedure. The SS of an instruction does not contain PCs of squashing instructions outside of the procedure. This design conservatively assumes that all squashing instructions outside of the procedure are unsafe. While this design is conservative, it is sound.

In a recursive procedure, the caller is the same as the callee. In this case, more dependencies may exist between instructions in the procedure than captured by our intra-procedural analysis. To see why, consider Figure 7.3. In the example, instruction *ld x* is a transmitter, and *br* is a squashing instruction that we would prefer to be in the SS of *ld x*. However, because the call is recursive, and the branch decides whether the call is executed, the *ld x* in the callee depends on the *br* in the caller. More generally, if a recursive procedure call (Line 3) has a control dependence or a data dependence (e.g., due to call arguments) on a squashing instruction, that squashing instruction should not be placed in the SS of any other instruction in the procedure.

```
1  foo() {
2    if (...) { // br
3      foo(); // call
4    }
5    ld x; // ld
6  }
```

Figure 7.3: Code snippet with a recursive call.

Unfortunately, we cannot simply solve the problem via program analysis: because of procedure pointers and indirect recursive calls, it is typically hard to identify recursive functions. Hence, we use hardware as follows. We still place the above squashing instruction in the SS of *ld x*, but the hardware places a fence at the beginning of each procedure. Such fence only prevents the execution of subsequent transmitters until the call instruction reaches the ROB head. With this support, the callee is not affected by squashing instructions from the caller. In practice, this support causes only a minor slowdown to the code run with InvarSpec, since compilers typically inline short functions in the caller. Our fence support handles not only direct but also indirect recursion.

Soundness & Completeness of Analysis. Our analysis labels squashing instructions as safe or unsafe. Soundness considers whether an unsafe squashing instruction may be labeled as safe, and completeness whether a safe squashing instruction may not be labeled as such.

The InvarSpec analysis is sound because it closely follows the definition of speculation invariance within procedures: no execution path from a safe squashing instruction to a transmitter can affect whether the transmitter executes or what source operands it uses. When our analysis cannot determine all execution paths, e.g., due to indirect jumps, it conservatively does not place the squashing instruction in the SS.

The InvarSpec analysis is not complete, due to at least two reasons. The first one is that it is not inter-procedural, which would be expensive and potentially unsound. The second one is the limitations of pointer-aliasing analysis. Incompleteness hurts performance but not correctness.

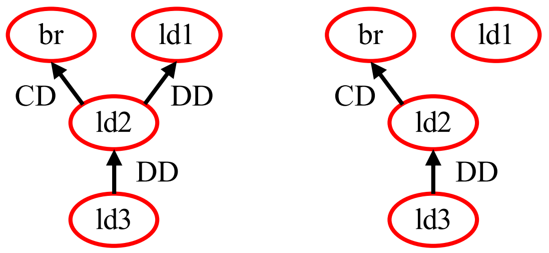
7.4.2 Enhanced Analysis

Key Insight. The Baseline analysis considers all possible dependencies when generating an SS. However, some of the dependencies may not occur on all execution paths. If we could neglect such dependencies, unless they really do occur, we could make the SS bigger, which could lead to speed up.

To illustrate the problem, consider the code in Figure 7.4(a), where $ld3$ is a transmitter. Assume that $ld1$ takes a long time to execute (e.g., because z misses in the cache), while br resolves quickly and, typically, is not taken. Figure 7.4(b) shows the IDG of $ld3$. We see that $ld3$ has a data dependency on $ld2$; $ld2$ has a control dependency on br and a data dependency on $ld1$.

```
y = ld z; // ld1
if (a) { // br
  x = ld y; // ld2
}
ld x; // ld3
```

(a)



(b) (c)

Figure 7.4: Code pattern that can be sped-up: (a) source code, (b) IDG of transmitter $ld3$, and (c) pruned IDG of $ld3$ computed by the Enhanced analysis.

Given this IDG, using InvarSpec's Baseline analysis, $ld3$'s SS will not contain $ld2$, br, or $ld1$ because they are in $ld3$'s IDG. They can affect the execution of $ld3$. Hence, InvarSpec will not send $ld3$ to memory until all three instructions have reached their OSP.

However, consider the case when br quickly resolves as *not taken*, and $ld1$ takes a long time to complete. In this case, $ld3$ is stalled by $ld1$, although $ld3$ has *no* runtime dependency on $ld1$.

The root of this stall is that InvarSpec's Baseline analysis does not consider the true runtime dependencies (i.e., the Baseline analysis is not flow/path-sensitive [270]). If, instead, we consider the path taken, we can show that, since $ld3$ depends on $ld1$ only if $ld2$ appears in ROB (i.e., br is taken), putting $ld1$ in $ld3$'s SS is actually safe.

Specifically, if br resolves and $ld2$ appears in the ROB, $ld2$ effectively *shields* $ld3$ from $ld1$: $ld3$ will not be sent to memory until $ld2$ reaches its OSP. By that time, $ld1$ has reached its OSP. Hence, the scheme does not need to directly check for $ld1$. Placing $ld1$ in $ld3$'s SS still results in a correct execution.

If, instead, br resolves and $ld2$ does not appear in the ROB, $ld3$ can safely execute without waiting for $ld1$. Hence, putting $ld1$ in $ld3$'s SS keeps correctness and makes the execution faster than with the Baseline analysis. Overall, in $ld3$'s IDG, we can effectively remove the edge to $ld1$ (Figure 7.4(c)).

Understanding the Enhanced Analysis. Based on the previous discussion, InvarSpec's Enhanced algorithm involves taking the IDG of an instruction i and removing some of the squashing instructions. The squashing instructions that are removed can be placed in the SS of i; the ones that remain cannot.

To understand when a squashing instruction can be removed, we need to understand when a squashing instruction shields another. Specifically, given an instruction i that depends on a squashing instruction j, which in turn depends on a squashing instruction k, when does j shield i from k?

We have seen in Figure 7.4(b) that if the edge from j to k is a data dependence, j shields i, and we can remove the edge from j to k (i.e., the edge from $ld2$ to $ld1$). Instruction i cannot reach its ESP until j reaches its OSP, and in turn j cannot reach its ESP (let alone its OSP) until k reaches its OSP. By the time j reaches its OSP, k cannot affect i anymore.

However, if the edge from j to k is a control dependence, the behavior is different. An example is the edge from $ld2$ to br in Figure 7.4(b). Branch br controls the value of x that $ld3$ uses: either the value returned by $ld2$ or not. $ld3$ cannot be sent to memory until br has reached its OSP. If we removed the edge from $ld2$ to br, $ld3$ would not wait for br's OSP, which could cause an incorrect execution. Indeed, suppose we remove it. Then, suppose that br mispredicts as not taken, and hence $ld2$ is not in the ROB to shield $ld3$. In this case, $ld3$ would be incorrectly sent to memory before br reached its OSP. Overall, the edge from $ld2$ to br cannot be removed and br cannot be in $ld3$'s SS.

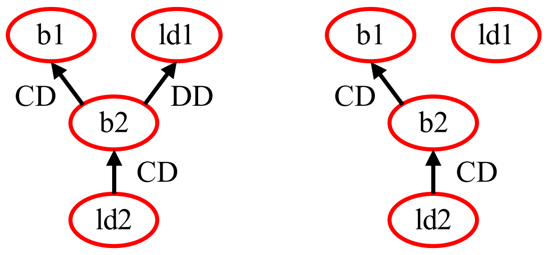
Consider now when the instruction i is control dependent on a squashing instruction j, to find out what instructions can j shield. Figure 7.5(a) shows an example code where $ld2$ is the transmitter. $ld2$ is control dependent on $b2$ which, in turn, is control dependent on $b1$ and data dependent on $ld1$. Figure 7.5(b) shows the corresponding IDG.

In this example, $b2$ shields $ld2$ from $ld1$: $b2$ will not reach its OSP until $ld1$ reaches its OSP, by which time $ld2$ does not need to consider $ld1$. Hence, InvarSpec can remove the edge from $b2$ to $ld1$ and put $ld1$ in $ld2$'s SS. The code will now run faster if $ld1$ takes long to execute, $b1$ reaches its OSP quickly and is not taken.

On the other hand, $b2$ does not shield $ld2$ from $b1$. If we removed the edge from $b2$ to $b1$,

```
y = ld z; // ld1
if (a) { // b1
  if (y) { // b2
    return;
  }
}
ld x; // ld2
```

(a)



(b) (c)

Figure 7.5: Code pattern to show when edges can be removed.

$ld2$ would not wait for $b1$'s OSP, which could cause an incorrect execution. Indeed, suppose we remove the edge. Then, suppose that $b1$ mispredicts as not taken, and hence $b2$ is not in the ROB to shield $ld2$. In this case, $ld2$ would be incorrectly sent to memory before $b1$ reached its OSP. Hence, the $b2$ to $b1$ edge needs to remain in the IDG, and $b1$ cannot be in $ld2$'s SS. Figure 7.5(c) shows the resulting IDG.

Overall, outgoing DD edges from squashing instructions can be removed, while CD edges cannot. The fundamental reason is that runtime data dependencies are path-sensitive—i.e., they are a function of the execution path followed. Control dependencies are path-insensitive, in that they exist irrespective of which of the two paths is taken by the execution.

If a DD edge starts from a non-squashing instruction, the edge cannot be removed. This is because a non-squashing instruction does not prevent a younger instruction from executing and, therefore, cannot shield it.

Enhanced Algorithm. Based on the previous discussion, we now outline InvarSpec's Enhanced analysis. Algorithm 7.2 shows the pseudo-code of function *pruneIDG*, which takes the IDG of an instruction *i* and generates a *pruned* IDG for *i*. The function traverses all the nodes in the IDG except *i* (the IDG root). If an instruction in the IDG is squashing, we check its outgoing edges. All the edges that are DD are removed.

The pruned IDG is then passed to function *getSS* of Algorithm 7.1 to compute the SS of the instruction. Because some squashing instructions are now unreachable in the pruned IDG, the Enhanced algorithm places more instructions in the SS of the instruction than the Baseline one. The result is a faster execution of the program.

7.4.3 Truncating the Safe Set

The SS of an instruction can contain the PCs of many instructions. To keep the hardware simpler, we propose to truncate the SS to a fixed size. For performance, we would like to keep only "the most useful" SS PCs. These are the PCs of the safe squashing instructions that are the most likely

Algorithm 7.2: Pruning an IDG.

```
Function pruneIDG(IDG) is
    for i in getNodes(IDG) \ {getRoot(IDG)} do
        if isSquashInsn(i) then
            for e in getOutEdge(IDG, i) do
                if isDataDep(e) then
                    removeEdge(IDG, e)
                end
            end
        end
    end
    return IDG
end
```

to be in the ROB when the transmitter enters the ROB. The PCs of safe instructions that are far in dynamic execution and thus already likely out of the ROB are less important to keep.

To find the most useful SS PCs for instruction *i*, the analysis pass statically finds the shortest distance, measured in the number of instructions in the function's CFG, between each safe squashing instruction and i. Then, it keeps in the SS the *N* safe squashing instructions with the smallest distances. It further removes those instructions whose distance is larger than the size of the ROB. We call the scheme $Trunc_N$.

In the SS of an instruction *i*, each safe instruction is encoded as the signed difference between the PC of the instruction and the PC of i. We call them *Offsets*.

7.5 THE INVARSPEC HARDWARE

To use the SS information, InvarSpec adds two micro-architecture modules. One compares the SS of an instruction to the older squashing instructions in the ROB; the other holds the SS and brings it to the pipeline on demand. For the second module, we present two possible designs.

7.5.1 Comparing the SS in the ROB

InvarSpec adds a hardware buffer in the pipeline that contains an entry for each dynamic instruction *i* in the ROB that is a transmitter (i.e., a load) or a squashing one (i.e., a load or a branch). We call it the *Inflight Buffer* (IFB) (Figure 7.6). Each IFB entry contains the following information for *i*: (i) its PC, (ii) a bit $\overline{T}$ that tells that *i* is not a transmitter, (iii) a *Ready* bitmask used to periodically check if *i* has become speculation invariant (SI), (iv) a bit set when *i* becomes SI, and (v) a bit set when *i* reaches its OSP. The Ready bitmask has as many bits as IFB entries.

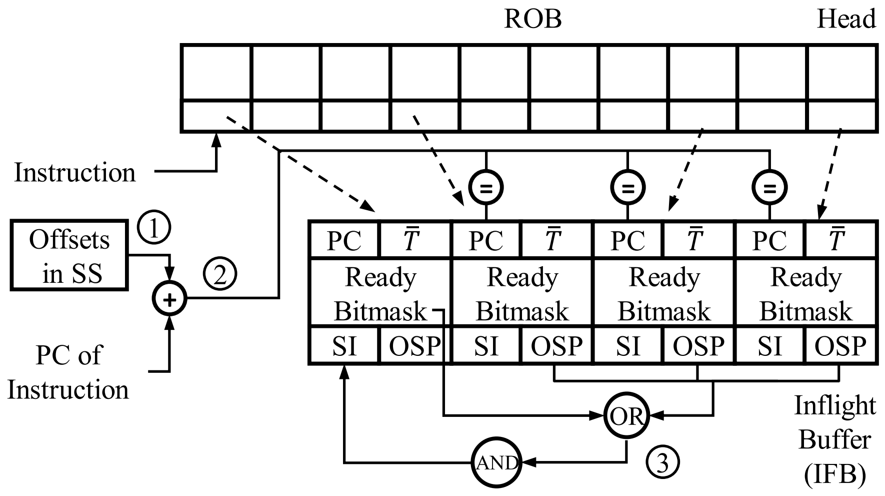


Figure 7.6: Hardware to use the SS in the ROB.

ROB entries have pointers to their corresponding IFB entries. IFB entries are allocated and deallocated in program order when the corresponding instruction is inserted in and removed from the ROB, respectively. Both IFB and ROB are circular buffers.

When a transmit or squashing instruction *i* is inserted in the ROB, its SS is requested (①), as we will see in Section 7.5.2. The offsets in the SS are summed-up to *i*'s PC, creating a set of safe squashing instruction PCs (②). The resulting PCs are compared to the PCs in the IFB entries that are both older than *i* and belong to squashing instructions. Note that, in the Comprehensive threat model that we use, transmit instructions are also squashing ones. Hence, the PCs from the SS are compared to the PCs in all the older IFB entries.

Based on these comparisons, the Ready bitmask in the IFB entry for instruction *i* is set as follows. If IFB entry *k* has a PC that matches one of the PCs obtained from the SS or has the OSP bit set, we know that the entry cannot prevent *i* from becoming speculation invariant (SI): either entry *k* belongs to a safe squashing instruction or to an instruction that has already reached its OSP. In either case, bit *k* in the Ready bitmask of instruction *i* is set. Further, the bits for those IFB entries not yet owned by any instructions, and for the entry owned by *i*, are set. The only Ready bitmask bits of *i* that remain clear are those for older squashing instructions that are not safe for *i* and have not reached their OSP.

If *i* is not a transmitter ($\overline{T}$=1), *i* can execute as soon as its operands are ready. In our configuration, this is the case for branches. Otherwise, *i* can only execute when it becomes SI and its operands are ready. In either case, the hardware tries to find when *i* becomes SI by checking, at every cycle, if the IFB entries that caused Ready bitmask bits to remain clear do set their OSP bit. As seen in Figure 7.6, this is done by simply taking the OSP bits from all the IFB entries and bit-ORing them with the Ready bitmask (③). When all the resulting bits are set, it means that all the squashing instructions older than *i* are either safe or have reached their OSP. At this point, *i* has become SI and sets its SI bit. If *i* is a transmitter, it can now execute as soon as its operands are ready.

After an instruction has satisfied the condition for its SI bit to be set and has executed, the logic to set its OSP bit depends on what type of instruction it is. Specifically, if it is a branch, the hardware sets its OSP bit right away. If it is a load, setting the OSP bit has to wait until the load reaches the point where it cannot be squashed anymore based on the threat model. For the Comprehensive model that we use, this is when the load reaches the ROB head.

There are two corner cases that are easily solved. First, if the IFB runs out of space, the ROB stops taking in new instructions. Second, if the SS for instruction *i* is not yet in the pipeline when *i* in inserted in the ROB, and there are older entries in the IFB that have their OSP bit clear, the hardware assumes that such entries are all unsafe. Hence, the corresponding Ready bitmask bits remain clear.

7.5.2 Storing and Bringing the SS to the Pipeline

The InvarSpec pass generates the SSs for the Squashing and Transmit Instructions (STIs) in the program. However, a sizable fraction of the STIs have empty SSs. Hence, we envision the InvarSpec pass to mark in the executable those STIs that have a non-empty SS.

Logically, such a mark can be a set bit in the opcode of the STI. In practice, in the x86 ISA, there is no such bit available. Hence, we can use an approach that has been used by Intel for lock elision: re-purpose a previously-ignored instruction prefix to mark instructions [271]. Specifically, we can reuse the *XRELEASE* prefix—which today is meaningful only for stores—to mark that the prefixed STI (a load or a branch in our case) does have an SS. This means that the encoding of STIs with an SS grows by the 1-byte prefix.

This approach changes the executable, but maintains backward compatibility. Because current processors ignore this prefix for STIs, the new executable runs on any x86 machine.

With this support in place, we now focus on how to store the SS and bring it to the pipeline on demand. We propose two alternatives: a software-based solution that is simple but makes the executable backward incompatible, and a hardware-based solution that is more complex but keeps backward compatibility. We outline each in turn, but we will only evaluate the one that keeps backward compatibility.

Software-Based Solution. In this solution, the InvarSpec analysis pass embeds the SS of an STI in the code of the program, right after the STI. For example, the pass could add an SS with 12 PC offsets of 10-bits each, for a total of 15 bytes. As an STI with prefix is decoded, the decoding hardware extracts the adjacent SS from the code stream. When the STI is inserted in the ROB, its

SS is readily available for the operation ① in Figure 7.6. This solution is simple but not backward compatible.

Hardware-Based Solution. In this solution, the InvarSpec analysis pass stores the SSs in data pages, and the core has a small *SS Cache* that keeps the recently-used SSs close to the pipeline for easy access in the future. Since the most frequently-executed STIs are in loops, a small SS cache typically captures the great majority of dynamic SSs needed.

We propose a simple design where, for each page of code, there is a data page at a fixed Virtual Address (VA) offset that holds the SSs of the STIs in that page of code. Further, the VA offset between *each* STI and its SS is fixed. This design does increase the memory consumed by a program by potentially the size of its instruction page working set (Section 7.7.2). However, it enables fast SS access.

Figure 7.7(a) shows a page of code and its SS page at a fixed VA offset (Δ). When the former is brought into physical memory, the latter is also brought in. The figure shows a prefixed STI and its SS. If the distance between the VAs of two consecutive prefixed STIs is less than the size of an SS, one of the STIs loses the prefix.

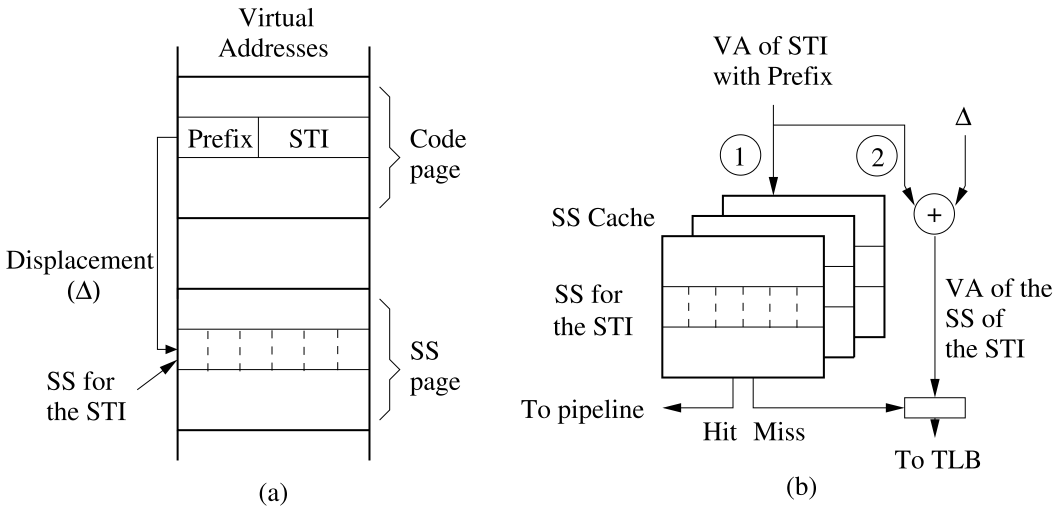


Figure 7.7: Hardware solution to store and access the SS. In the figure, STI means Squashing or Transmit Instruction.

Figure 7.7(b) shows the action taken when a prefixed STI is decoded. The VA of the STI is sent to the SS cache (①). The SS cache is a small, set-associative cache that contains the most recently-used SSs. Due to the good locality of STIs in loops, most of the time, the SS cache hits. In this case, it provides the SS to the pipeline on time to be used when the STI is inserted in the ROB.

On an SS miss, the STI's VA is added to the Δ offset (②) to obtain the VA of the SS. This address is sent to the TLB to obtain the Physical Address (PA). After that, *but only when the STI reaches* *its Visibility Point (VP)*, a request is sent to the cache hierarchy to obtain the SS, and bring it to the SS cache. As a result, this STI is unable to use its SS; it will be used in a future invocation of the same STI when the SS request hits in the SS cache.

The SS cache does not introduce any side channel because no side effect occurs until the STI reaches its VP. Specifically, on an SS cache miss, we saw that the SS request is not sent to the cache hierarchy until the STI's VP, providing no information to the attacker. On an SS cache hit, the SS cache's LRU bits are not updated until the STI reaches its VP.

7.6 EXPERIMENTAL METHODOLOGY

Architectures Modeled. We model the architecture shown in Table 7.1 using cycle-level simulations with Gem5 [230]. All the side effects of transient instructions are modeled. Our baseline architecture is a conventional processor with no protection against speculative-execution attacks. We call it UNSAFE.

Parameter	**Value**
Architecture	2.0 GHz out-of-order x86 core
Core	8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, TAGE branch predictor, 4096 BTB entries, 16 RAS entries
L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher
L1-D Cache	64 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher
L2 Cache	2 MB, 64 B line, 16-way, 8 cycles RT latency
DRAM	50 ns RT latency after L2
SS Cache	64 sets, 4-way, 2 cycle RT latency, each entry has 12 10-bit PC offsets ($Trunc_{12}$). For 22nm: area is 0.0088mm^2, dyn. rd. energy is 2.95pJ, leakage power is 2.31mW
IFB	76 entries. For 22nm: area is 0.0022mm^2, dyn. rd. energy is 0.99pJ, leakage power is 0.58mW

Table 7.1: Parameters of the simulated architecture.

We augment this architecture with several hardware defense schemes that use loads as the transmitters. We use the Comprehensive threat model, with both branches and loads as squashing instructions. The defense schemes are: (i) delaying with fences all speculative loads until they reach their Visibility Point (VP) [118] (FENCE); (ii) Delay-On-Miss, which delays speculative loads that miss in L1 until their VP [120, 121] (DOM); and (iii) InvisiSpec, which executes speculative loads

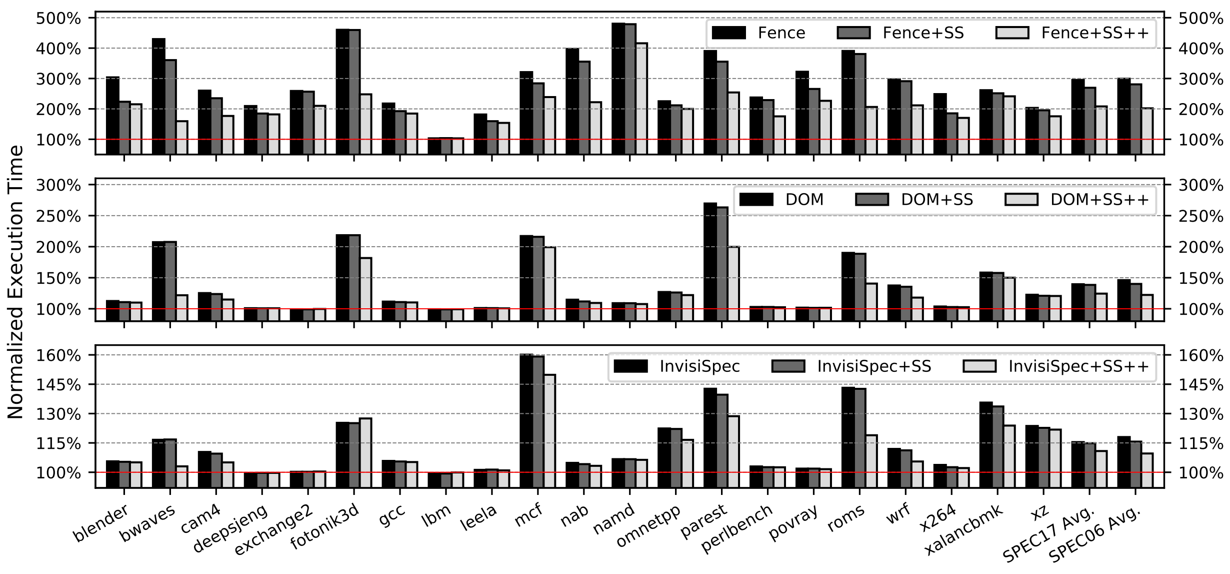


Figure 7.8: Execution time of the applications on different architecture configurations, all normalized to UNSAFE. The three plots correspond, from top to bottom, to configurations related to the FENCE, DOM, and INVISISPEC defense schemes. Each plot has a different Y-axis range.

invisibly before their VP [118] (INVISISPEC). We model these defense schemes as they are (D), augmented with the Baseline InvarSpec analysis (D+SS), and augmented with the Enhanced InvarSpec analysis (D+SS++). The resulting configurations are shown in Table 7.2.

Configuration	**Description**
UNSAFE	Unmodified x86 architecture
FENCE	Delay all speculative loads with fences [118]
FENCE+SS	FENCE augmented with Baseline InvarSpec
FENCE+SS++	FENCE augmented with Enhanced InvarSpec
DOM	Delay speculative loads on L1 miss [120, 121]
DOM+SS	DOM augmented with Baseline InvarSpec
DOM+SS++	DOM augmented with Enhanced InvarSpec
INVISISPEC	Execute speculative loads invisibly [118]
INVISISPEC+SS	INVISISPEC augmented with Baseline InvarSpec
INVISISPEC+SS++	INVISISPEC augmented with Enhanced InvarSpec

Table 7.2: Defense configurations modeled.

Applications and Analysis Pass. We run SPEC17 [231] and SPEC06 [272] applications with the reference input size. Because of simulation issues and binary analysis tool malfunction, we do not report on 2 applications out of 23 from SPEC17 and 4 out of 29 from SPEC06. For each application, we use SimPoint [232] to generate up to 10 representative intervals that accurately characterize the end-to-end performance of the application. Each interval contains 50 million instructions. We run Gem5 on each interval with system-call emulation mode with 1 million warm-up instructions.

Our InvarSpec analysis pass implementation is based on Radare2 [273], a state-of-the-art open-source binary analysis tool. It is performed on x86 binaries. However, it can also be implemented as a compiler pass and be performed on source code during compilation.

7.7 EVALUATION

7.7.1 Overall Performance Results

Figure 7.8 shows the execution time of SPEC17 and SPEC06 applications on all the configurations of Table 7.2. The three plots correspond, from top to bottom, to configurations related to the Fence, DOM, and InvisiSpec defense schemes. Each plot has a different Y-axis range. All bars are normalized to Unsafe. Each plot shows each SPEC17 application, the average of SPEC17 applications, and the average of SPEC06 applications.

Going from top to bottom, we see that Fence is the slowest scheme among all schemes evaluated. On average, it has an overhead of 195.3% on SPEC17 and 199.3% on SPEC06. Fence+SS++ reduces the average overhead significantly, from 195.3% to 108.2% on SPEC17, and from 199.3% to 101.9% on SPEC06.

DOM exhibits a bimodal behavior on SPEC17 applications. While it has low overhead on about half of the applications, its overhead is very high on the rest. For example, the overhead is 169.6% on `parest` and 107.3% on `bwaves`. On average across all applications, DOM's overhead is 39.5% on SPEC17 and 46.1% on SPEC06. Adding support for Enhanced SS on top of DOM (DOM+SS++) substantially reduces this overhead. Enhanced SS is typically effective in the cases when DOM has high overhead. Specifically, it brings down `parest`'s overhead to 99.7% and `bwaves`'s to 21.8%. It does so by allowing cache-missing loads that are speculation invariant to proceed—rather than stalling them. On average, DOM+SS++ reduces the execution overhead from 39.5% to 24.4% on SPEC17, and from 46.1% to 22.3% on SPEC06.

InvisiSpec's average overhead is 15.4% on SPEC17 and 18.0% on SPEC06. This overhead is lower than the corresponding DOM overhead. InvisiSpec+SS++ speeds-up the execution over InvisiSpec. On average, the overhead of InvisiSpec+SS++ is only 10.9% on SPEC17 and 9.6% on SPEC06. In InvisiSpec+SS++, when a speculative load is ready to issue to memory, if it is speculation invariant, it is issued to memory normally; in InvisiSpec, the load is issued as an invisible load and hence requires two loads.

7.7.2 SS Analysis

We evaluate the performance impact of InvarSpec's design choices by conducting sensitivity studies for FENCE+SS++, DOM+SS++, and INVISISPEC+SS++ on SPEC17.

SS coverage. One design decision is how many bits to use to encode an *SS offset*, i.e., the distance between the PCs of a safe instruction and a transmitter. This number affects how many offsets an SS entry can encode.

Figure 7.9 shows the average normalized execution time of the schemes on SPEC17 when varying the number of bits per SS offset. The size of SS is fixed to 12 offsets. All data are normalized to the corresponding base hardware scheme (FENCE, DOM, and INVISISPEC). We see that, as the number of bits decreases, the execution time increases with different degrees. When the number of bits is smaller than 10, the performance degradation becomes non-negligible. Thus, our design uses 10 bits to encode an SS offset, which provides a performance similar to the unlimited number of bits.

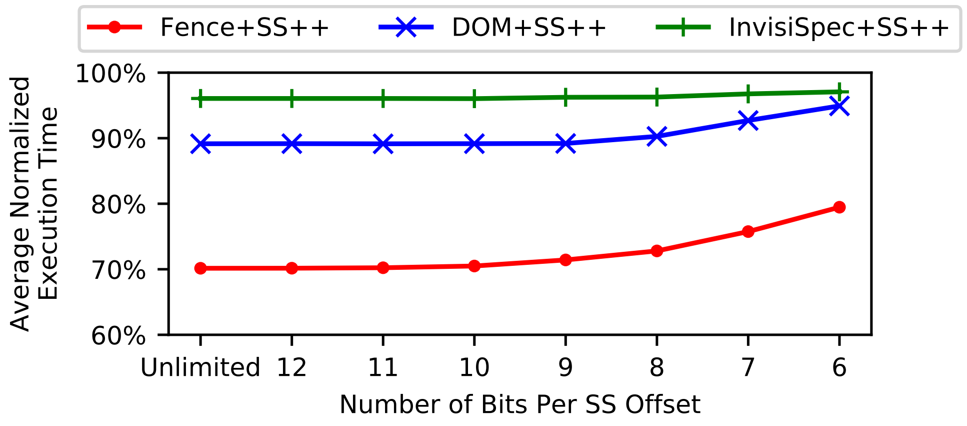


Figure 7.9: Normalized execution time when varying the number of bits per SS offset. All execution times are normalized to their corresponding base hardware schemes without InvarSpec.

Truncation. Another design decision is the SS size, namely the maximum number of SS offsets to keep in an SS entry. Figure 7.10 shows the average normalized execution time of the schemes with various SS sizes. Each SS offset is 10 bits. All data are normalized as in Figure 7.9. We see that, as the SS size increases, the execution time decreases. Compared to an unlimited SS size, all truncation configurations have a performance degradation. An SS size equal to 12 offsets is a good design point and, therefore, is our default design.

SS Cache. Figure 7.11 characterizes the SS cache. It shows the SS cache hit rate (right Y axis) and average normalized execution time of the applications (left Y axis) for different SS cache geometries. We compare our default configuration (4-way set-associative with 64 sets) to geometries

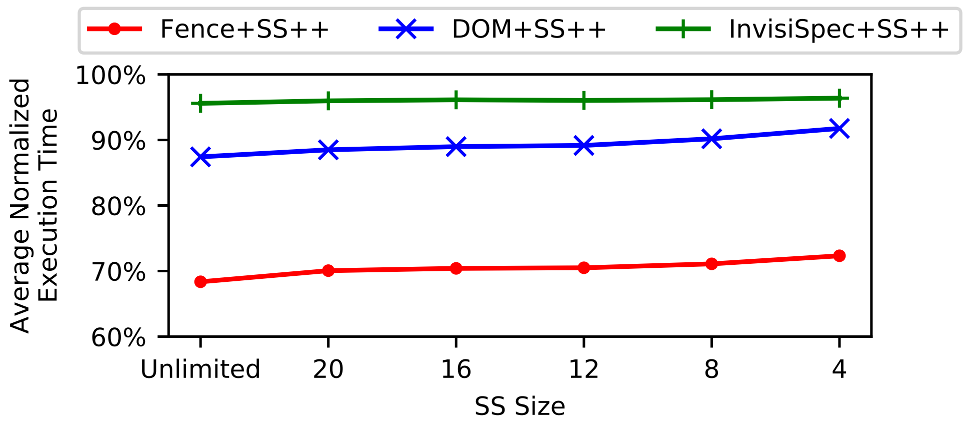


Figure 7.10: Normalized execution time when varying the SS size. All execution times are normalized to their corresponding base hardware schemes without InvarSpec.

with the same associativity but fewer or more sets. We also compare to a fully-associative cache of the same size (256 lines). All execution times are normalized as in Figure 7.9.

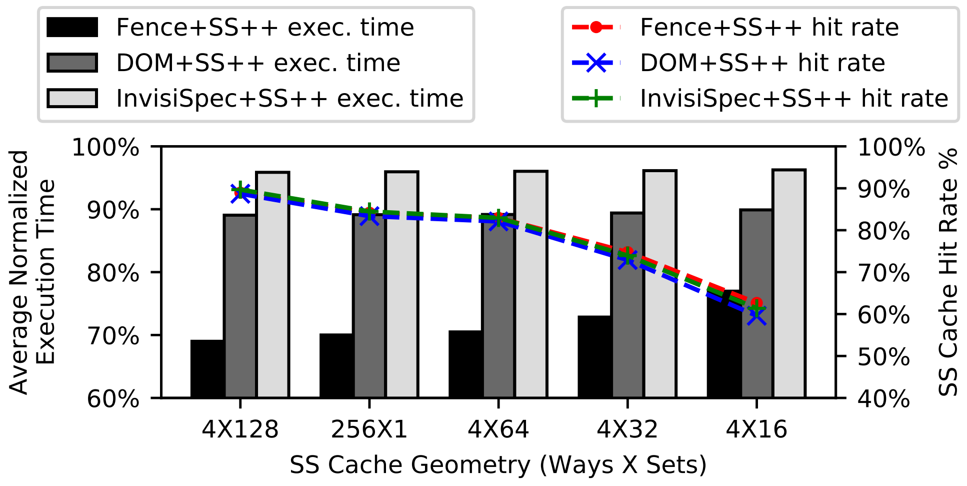


Figure 7.11: Normalized execution time and SS cache hit rate when changing the SS cache. Execution times are normalized to the corresponding base hardware schemes without InvarSpec.

Increasing the SS cache size from our default configuration only slightly decreases the execution time of DOM+SS++ and InvisiSpec+SS++, but Fence+SS++'s execution time keeps decreasing as the SS cache size grows. Decreasing the SS cache size from our default configuration increases the execution time of every scheme.

The average hit rate shows that the cache size is more important than the associativity. Reducing the cache size for the same associativity decreases the hit rate. However, for the same size, increasing the associativity from 4 to full causes a minimal change. Overall, our default design strikes a good trade-off between performance and hardware complexity.

Memory Footprint. We measure that, on average, about half of the code pages in an application have at least one non-empty SS. To estimate an upper bound on the amount of memory required to store this SS state at run time, we add-up all the code pages in an application that have at least one non-empty SS. In reality, not all of the SS pages may be in memory at the same time. We call the resulting size the *Conservative SS Footprint*. We also measure the peak memory usage of each application with the reference input at any point in the program execution. We call the resulting size the *Peak Memory during Execution*.

Table 7.3 shows the two metrics for the 5 applications with the largest conservative SS footprint, and the average metrics across all SPEC17 applications. We can see that the memory overhead of storing the SS state is negligible compared to the peak memory that an application uses. The SS only causes a 0.55% memory overhead on average. For `blender`, which has the largest SS footprint, the overhead is only 1.32%.

SPEC17 App.	**Conservative SS Footprint (MB)**	**Peak Memory during Execution (MB)**
blender	8.24	626.31
perlbench	8.00	413.09
wrf	7.70	172.15
gcc	5.87	1277.55
cam4	5.27	853.91
SPEC17 Avg.	2.55	462.05

Table 7.3: Assessing the memory footprint of the SS state.

7.7.3 Hardware Overhead

The main InvarSpec hardware is the SS cache and the IFB. The SS cache is relatively simple because it stores only read-only data. We used CACTI 7.0 [260] to estimate the area and power of the storage component of these structures for 22nm technology. As shown in Table 7.1, the area, dynamic read energy, and leakage power of the storage structures is small.

7.7.4 Discussion

Interaction with a JIT Compiler. Our scheme is compatible with a JIT compilation environment. In this case, the dynamic generation of a binary is augmented with a step that runs the InvarSpec analysis pass and generates the SSs. In practice, this step does not take long because it substantially reuses information that the compiler has just generated. We cannot provide an accurate estimate of this extra execution time because our implementation of the InvarSpec analysis pass is not optimized.

Reducing Execution Overhead Further. There are several approaches that could further reduce the execution overhead with InvarSpec. Three that come to mind are to increase the SS size, increase the SS cache size, and improve the Enhanced compiler analysis. The first approach can only decrease the execution overhead by a few percentage points. Indeed, Figure 7.10 showed the overhead with unlimited-sized SS entries, which is an upper bound. The second approach also gives modest gains. We have evaluated a configuration with an infinite SS cache with unlimited-sized SS entries. The result is that FENCE+SS++, DOM+SS++, and INVISISPEC+SS++ further reduce the average execution overhead from 108.2% to 90.4%, from 24.4% to 21.8%, and from 10.9% to 10.2%, respectively. The third approach, namely improving the Enhanced compiler analysis, may deliver more significant gains, especially if it involves adding inter-procedural analysis. Such approach likely involves non-trivial effort, and is our future work.

7.8 RELATED WORK

As indicated in Sections 7.1 and 2.3.3, there are many defense schemes against speculation attacks. Some are software schemes, based on stopping speculation either with fences [242, 243, 244] or by injecting data-dependencies into the code [243, 245]. There are many hardware schemes (e.g., [54, 116, 118, 119, 120, 121, 122, 124, 125, 241]). Of these hardware schemes, many of those that do not consider timing attacks can be extended to support InvarSpec (e.g., [118, 119, 120, 121, 122]). InvarSpec enhances hardware techniques with software information.

The STT [116], SpecShield [241], and NDA [255] hardware schemes have a different threat model than those that InvarSpec extends in this chapter. Indeed, the schemes in this chapter protect all data from being leaked by speculative execution; STT, SpecShield, and NDA protect only data that is read by mis-speculated execution, and consider data in retired register file state not to be a secret.

To see the difference, consider the example in Figure 7.12. In the example code, although `secret` would not be leaked in a non-speculative execution, STT, SpecShield, and NDA do not apply protection to the mis-speculated `load(secret)` instruction, because `secret` was read into a register by a bound-to-commit instruction. In contrast, the schemes that InvarSpec extends in this chapter do not allow performing the `load(secret)` without protection before the branch resolves.

Despite this difference in protection scope, the main principle of InvarSpec to statically analyze code and dynamically disable defense protection earlier could also be adapted to extend schemes

```
secret = load(secret_ptr); // soon to commit
if (...) { // mispredicted branch
  load(secret);
}
```

Figure 7.12: Example that exposes the difference between protecting all data versus only speculatively-read data.

such as STT, SpecShield, and NDA.

Finally, there are many designs that aim to block cache-based covert channels, using randomization [49, 199], encryption [46, 47], cache partitioning [49, 53, 274, 275], or other mechanisms [61, 276]. They do not address speculative execution attacks.

7.9 CONCLUSION

This chapter introduced *Speculation Invariance*, and showed that it can be used to reduce the overhead of speculative execution defenses without changing security properties. It also presented the *InvarSpec* framework, which includes a program analysis pass to identify *Safe* instructions, and micro-architecture that uses this information to find and issue speculation invariant instructions earlier. InvarSpec is one of the first defense schemes for speculative execution that combines co-operative compiler and hardware mechanisms. It effectively enhances hardware defense schemes: it reduces the average execution overhead of fence protection from 195.3% to 108.2%, of DOM from 39.5% to 24.4%, and of InvisiSpec from 15.4% to 10.9%.

CHAPTER 8: Conclusion and Future Work

This thesis systematically studies practical side-channel attacks and defenses in public clouds. First, the thesis presented a comprehensive study on risks of and techniques for co-location between two mutually distrusting users in modern public cloud FaaS environments. We introduced novel physical host fingerprinting techniques based on the timestamp counter. These fingerprinting techniques are highly accurate. We used these techniques to reverse engineer the instance placement policy of Google Cloud Run, a serverless platform, and found exploitable placement behavior. Exploiting the placement behavior, the attacker can reliably achieve co-location with a target victim with minimal financial cost.

Next, the thesis demonstrated a series of LLC side-channel attack techniques that help extract information in a noisy, dynamic production cloud environment. We showed that the state-of-the-art eviction set construction algorithms are ineffective on Cloud Run due to noise in the production environment. To address the challenge, we then introduced L2-driven candidate address filtering and a binary search-based algorithm for address pruning to speed up eviction set construction. Subsequently, parallel probing and leveraged power spectral density were introduced to identify the victim's target LLC set and extract information. In the end, we demonstrated, for the first time, end-to-end cross-tenant information leakage in the production Google Cloud environment. Following our report, Google filed a critical-level bug report to their product team and AWS revised their security whitepaper.

To defend against side-channel attacks with low execution overhead and security guarantees, the thesis focused on schemes that use dynamic partitioning of hardware resources. These schemes can be performant and efficient as they can dynamically adjust partition sizes to the demand of applications. However, the resizing of partitions can reintroduce information leakage. This thesis proposed a framework named Untangle that tightly quantifies information leakage in dynamic partitioning schemes. Relying on the leakage quantification enabled by Untangle, one can define a security policy such that if the runtime leakage goes beyond a user-defined threshold, no further resizings are allowed. Using this design, the user can make an informed security-performance trade-off.

Finally, the thesis developed two techniques to improve the performance of transient execution defenses without undermining their security guarantees. The first technique is based on the insight that the execution of speculative instructions is mostly impeded by waiting until memory consistency violations (MCVs) are impossible. Based on this insight, the thesis proposed Pinned Loads, a hardware design that tries to make loads invulnerable to MCVs as early as possible, improving performance. The second technique is based on the insight that there exist "safe instructions" that cannot influence the execution and operand values of vulnerable instructions. This insight leads to a static program analysis that finds such safe instructions and communicates these instructions to the hardware defense. As a result, the hardware defense can permit earlier execution of vulnerable instructions without waiting for the completion of safe instructions, reducing execution overhead.

8.1 FUTURE WORK

Our overarching goal is to build secure and efficient public clouds resistant to side-channel attacks. This thesis serves as the first step towards this goal. In the upcoming years, we will comprehensively examine and redesign the full cloud-computing stack to secure the public cloud from side-channel attacks. This endeavor will involve interdisciplinary collaborations with experts in systems and networking, security, machine learning, programming languages, and software engineering. Here is an outline of our future research directions.

Discovering new side channels across the cloud-computing stack. While our current work [101, 277] has demonstrated end-to-end side-channel attacks in a production public cloud, we believe this is just the beginning. The evolving heterogeneity of cloud hardware, including the rise of FPGAs, SmartNICs, and other accelerators, can introduce new classes of hardware side channels. Similarly, extensive sharing in cloud software services—like schedulers, remote storage, and load balancers—can lead to software side-channel vulnerabilities. We aim to systematically uncover these vulnerabilities across the cloud stack. We also envision using software testing techniques as a potential strategy to efficiently discover these channels.

Assessing side-channel attacks on diverse applications. Side-channel attacks traditionally target cryptographic libraries. However, modern clouds serve a variety of applications. Our research will study what information do side channels leak in various non-cryptographic cloud applications. Questions that we will explore include: Can attackers eavesdrop on a VoIP call? Can attackers monitor user activities in an e-commerce website run by their competitors? Can attackers steal documents in collaborative editing applications? Can medical records be exfiltrated from a health-care application that runs inside a trusted-execution environment (TEE)? We are keen to explore if such leakages might, under regulations like GDPR, classify as data breaches.

Designing secure, cloud-native hardware-software interfaces. Designing secure hardware for public clouds is challenging, as cloud applications have large variations in their resource needs and security goals. A promising approach to tackle this challenge is hardware-software co-design. Drawing on our prior experience in this area [246, 249], we will design new interfaces that secure public clouds while maintaining high flexibility. This involves developing efficient hardware resource-isolation primitives and adapting the cloud's software infrastructure to these primitives.

Notably, technologies developed along this direction not only mitigate side-channel vulnerabilities, but also help minimize performance interference among tenants, improving the quality of service.

Managing the risks of side-channel attacks in clouds. Although resource isolation offers the strongest security guarantees, it might not be applicable to certain resources due to the prohibitive costs of completely isolating these resources between tenants. Building on the approach of Untangle [197], we will explore probabilistic defenses and assess information leakage using information theory. Furthermore, we will study: (1) leveraging the noise inherent in the cloud environment to reduce the leakage of probabilistic defenses, (2) determining the acceptable amount of information leakage for various application types, and (3) balancing information leakage and performance.

8.1.1 Other Directions

Resource sharing, a fundamental optimization, is also prevalent in emerging hardware designs like machine-learning accelerators and computing paradigms like edge computing. In the future, we will uncover and mitigate side-channel vulnerabilities in these new designs and paradigms.

Secure machine-learning (ML) hardware acceleration. As ML becomes ubiquitous, numerous hardware accelerators are being proposed to improve the performance and energy efficiency of ML applications. Much like general-purpose processors, these accelerators are expected to be used in multi-tenant environments. This necessitates a solution to secure computing in accelerators, as ML applications often process sensitive user information such as medical records and personal photos. We will extend trusted-execution environment (TEE) designs and side-channel defenses to harden ML accelerators. Specifically, given the heterogeneous nature of accelerators, We will focus on developing general frameworks that efficiently generate secure accelerators, tailored to threat models and application needs, while minimizing manual involvement.

Side channels in edge computing. Unlike cloud computing, edge computing operates closer to data sources such as Internet-of-Things (IoT) devices. This proximity means that resource sharing in edge computing inherently has physical locality. For instance, a set of IoT devices in a household might share the same edge gateway, while services monitoring air pollution and traffic in a neighborhood might share the same local networking and computing infrastructure. A significant consequence of this paradigm is that any side-channel leakage is associated with a particular physical area, allowing attackers to exfiltrate information from victims that are in close physical proximity. In the long term, we will collaborate with domain experts to discover and mitigate side channels in edge computing.

REFERENCES

[1] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *The Cryptographers' Track at the RSA Conference 2006 (CT-RSA)*, ser. Lecture Notes in Computer Science, D. Pointcheval, Ed., vol. 3860. Springer, 2006. [Online]. Available: https://doi.org/10.1007/11605805_1 pp. 1–20.

[2] C. Percival, "Cache missing for fun and profit," https://www.daemonology.net/papers/cachemissing.pdf, 2005.

[3] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2015. [Online]. Available: https://doi.org/10.1109/SP.2015.43 pp. 605–622.

[4] G. Irazoqui Apecechea, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES," in *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2015. [Online]. Available: https://doi.org/10.1109/SP.2015.42 pp. 591–604.

[5] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. H. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/SP.2019.00004 pp. 888–904.

[6] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium*. USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom pp. 719–732.

[7] 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 (DIMVA)*, ser. Lecture Notes in Computer Science, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14 pp. 279–299.

[8] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium*. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras pp. 955–972.

[9] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, 2019.

[10] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[11] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS*, 2018.

[12] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019. [Online]. Available: https://doi.org/10.1145/3319535.3363194 pp. 785–800.

[13] A. Cabrera 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 (S&P)*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/SP.2019.00066 pp. 870–887.

[14] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *25th USENIX Security Symposium*. USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl pp. 565–581.

[15] M. S. İnci, B. Gülmezoğlu, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, vol. 9813. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_18 pp. 368–388.

[16] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Binoculars: Contention-based side-channel attacks exploiting the page walker," in *31st USENIX Security Symposium*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-zirui pp. 699–716.

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014. [Online]. Available: https://doi.org/10.1145/2660267.2660356 pp. 990–1003.

[18] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher et al., "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019, pp. 1–19.

[19] 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*, 2018, pp. 973–990.

[20] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks," in *USENIX SEC*, Aug. 2021. [Online]. Available: Paper=https://download.vusec.net/papers/fpvi-scsb_sec21.pdfWeb=https://www.vusec.net/projects/fpvi-scsbCode=https://github.com/vusec/fpvi-scsb

[21] 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[22] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *ESORICS*, 2019, pp. 279–299.

[23] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar et al., "Fallout: Leaking data on meltdown-resistant cpus," in *CCS*, 2019, pp. 769–784.

[24] 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.

[25] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *CCS*, 2019, pp. 753–768.

[26] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *IEEE Symposium on Security & Privacy*, 2021.

[27] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv e-prints*, p. arXiv:1807.03757, Jul 2018.

[28] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018, pp. 2109–2122.

[29] A. AWS, "Serverless Computing - AWS Lambda - Amazon Web Services," https://aws.amazon.com/lambda/, 2023.

[30] G. Cloud, "Cloud Run: Container to production in seconds | Google Cloud," https://cloud.google.com/run/, 2023.

[31] M. Azure, "Azure Functions - Serverless Functions in Computing | Microsoft Azure," https://azure.microsoft.com/en-us/products/functions, 2023.

[32] P. Ambati, I. Goiri, F. V. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini, "Providing SLOs for resource-harvesting VMs in cloud platforms," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ambati pp. 735–751.

[33] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini, "SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3447786.3456225 pp. 1–16.

[34] Y. Zhang, I. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3477132.3483580 pp. 724–739.

[35] A. Fuerst, S. Novakovic, I. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini, "Memory-harvesting VMs in cloud platforms," in *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507725 pp. 583–594.

[36] A. AWS, "The Security Design of the AWS Nitro System - AWS Whitepaper," https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html, 2022.

[37] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *2009 ACM Conference on Computer and Communications Security (CCS)*, E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds. ACM, 2009. [Online]. Available: https://doi.org/10.1145/1653662.1653687 pp. 199–212.

[38] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud," in *24th USENIX Security Symposium*, J. Jung and T. Holz, Eds. USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu pp. 929–944.

[39] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *24th USENIX Security Symposium*, J. Jung and T. Holz, Eds. USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan pp. 913–928.

[40] M. S. İnci, B. Gülmezoğlu, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! cross-VM RSA key recovery in a public cloud," *IACR Cryptology ePrint Archive*, p. 898, 2015. [Online]. Available: http://eprint.iacr.org/2015/898

[41] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security*, 2016.

[42] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS*, 2019.

[43] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *IEEE S&P'09*, 2009.

[44] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, 2019.

[45] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, “A Lightweight Isolation Mechanism for Secure Branch Predictors,” in *58th ACM/IEEE Design Automation Conference (DAC’21)*, 2021.

[46] 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 Association, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/werner pp. 675–692.

[47] M. K. Qureshi, “New attacks and defense for encrypted-address cache,” in *46th International Symposium on Computer Architecture (ISCA)*, S. B. Manne, H. C. Hunter, and E. R. Altman, Eds. ACM, 2019. [Online]. Available: https://doi.org/10.1145/3307650.3322246 pp. 360–371.

[48] Q. Tan, Z. Zeng, K. Bu, and K. Ren, “PhantomCache: Obfuscating cache conflicts with localized randomization,” in *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *34th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2007, pp. 494–505.

[50] F. Liu and R. B. Lee, “Random fill cache architecture,” in *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014. [Online]. Available: https://doi.org/10.1109/MICRO.2014.28 pp. 203–215.

[51] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, “Systematic analysis of randomization-based protected cache architectures,” in *42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00011 pp. 987–1002.

[52] M. K. Qureshi, “CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping,” in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2018. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00068 pp. 775–787.

[53] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, “SecDCP: Secure dynamic cache partitioning for efficient timing channel protection,” in *53rd Annual Design Automation Conference (DAC)*. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2897937.2898086 pp. 74:1–74:6.

[54] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, “DAWG: A defense against cache timing attacks in speculative execution processors,” in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2018. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00083 pp. 974–987.

[55] G. Dessouky, T. Frassetto, and A. Sadeghi, “HybCache: Hybrid side-channel-resilient caches for trusted execution environments,” in *29th USENIX Security Symposium*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky pp. 451–468.

[56] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, “BRB: Mitigating Branch Predictor Side-Channels,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA’19)*, 2019.

[57] G. Saileshwar, S. Kariyappa, and M. K. Qureshi, “Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning,” in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021. [Online]. Available: https://doi.org/10.1109/SEED51797.2021.00015 pp. 37–49.

[58] I. Corparation, “Introduction to cache allocation technology in the intel xeon processor e5 v4 family,” https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, Feb, 2016.

[59] D. Townley, K. Arikan, Y. D. Liu, D. Ponomarev, and O. Ergin, “Composable Cachelets: Protecting enclaves from cache side-channel attacks,” in *31st USENIX Security Symposium*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/townley pp. 2839–2856.

[60] G. Dessouky, E. Stapf, P. Mahmoody, A. Gruler, and A. Sadeghi, “Chunked-Cache: On-demand and scalable cache isolation for security architectures,” in *29th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-225/

[61] F. Liu, Q. Ge, Y. Yarom, F. McKeen, C. V. Rozas, G. Heiser, and R. B. Lee, “CATalyst: Defeating last-level cache side channel attacks in cloud computing,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2016. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446082 pp. 406–418.

[62] Z. N. Zhao, H. Ji, A. Morrison, D. Marinov, and J. Torrellas, “Pinned loads: Taming speculative loads in secure processors,” in *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022), Lausanne, Switzerland, February 28–March 4, 2022*. ACM, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507724

[63] G. Cloud, “Invoking with an HTTPS Request | Cloud Run Documentation | Google Cloud,” https://cloud.google.com/run/docs/triggering/https-request, 2023.

[64] G. Cloud, “Container runtime contract | Cloud Run Documentation | Google Cloud,” https://cloud.google.com/run/docs/container-contract, 2023.

[65] A. AWS, “AWS Lambda - FAQs,” https://aws.amazon.com/lambda/faqs/, 2023.

[66] M. Azure, “Azure functions scale and hosting | microsoft learn,” https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#timeout, 2023.

[67] G. Cloud, “About container instance autoscaling | Cloud Run Documentation | Google Cloud,” https://cloud.google.com/run/docs/about-instance-autoscaling, 2023.

[68] K. Contributors, “Kubernetes Scheduler | Kubernetes,” https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2023.

[69] G. Cloud, “About execution environments | Cloud Run Documentation | Google Cloud,” https://cloud.google.com/run/docs/about-execution-environments, 2023.

[70] gVisor Contributors, “The Container Security Platform | gVisor,” https://gvisor.dev/, 2023.

[71] G. Cloud, “Cloud Run release notes | Cloud Run Documentation | Google Cloud,” https://cloud.google.com/run/docs/release-notes, 2023.

[72] H. Kasture and D. Sanchez, “Tailbench: a benchmark suite and evaluation methodology for latency-critical applications,” in *2016 IEEE International Workshop/Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[73] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: a study of emerging scale-out workloads on modern hardware,” in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, T. Harris and M. L. Scott, Eds. ACM, 2012, pp. 37–48.

[74] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 301–312.

[75] Google, “Google Docs,” https://docs.google.com/, 2023.

[76] M. Contributors, “memcached - a distributed memory object caching system,” https://memcached.org/, 2018.

[77] R. Labs, “Redis In-Memory Data Structure,” https://redis.io, 2022.

[78] E. Schurman and J. Brutlag, “The user and business impact of server delays, additional bytes, and http chunking in web search,” in *Velocity Web Performance and Operations Conference*. O’Reilly Media, 2009.

[79] G. Cloud, “Cloud Functions | Google Cloud,” https://cloud.google.com/functions, 2023.

[80] O. Aciiçmez, “Yet another microarchitectural attack: exploiting i-cache,” in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[81] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *2012 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012. [Online]. Available: https://doi.org/10.1145/2382196.2382230 pp. 305–316.

[82] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *ct-rsa*, 2008.

[83] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *MICRO*, 2016.

[84] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES," in *oakland*, 2015.

[85] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium*, J. Jung and T. Holz, Eds. USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss pp. 897–912.

[86] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *26th USENIX Security Symposium*. USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen pp. 51–67.

[87] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *ACSAC*, 2006, pp. 473–482.

[88] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *NDSS*, 2020.

[89] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX Security*, 2012, pp. 159–173.

[90] D. Evtyushkin and D. V. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS*, 2016, pp. 843–857.

[91] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE S&P*, May 2015.

[92] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *MICRO*. IEEE, 2020, pp. 28–41.

[93] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *CCS*, 2018, p. 131−145.

[94] P. Cronin and C. Yang, "A fetching tale: Covert communication with the hardware prefetcher," in *HOST*, 2019, pp. 101–110.

[95] Y. Chen, L. Pei, and T. E. Carlson, “Leaking control flow information via the hardware prefetcher,” *arXiv preprint arXiv:2109.00474*, 2021.

[96] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “ASLR on the Line: Practical Cache Attacks on the MMU,” in *NDSS*, Feb. 2017.

[97] S. van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, “RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches,” in *EuroSec*, Apr. 2017.

[98] A. Arcangeli, I. Eidus, and C. Wright, “Increasing memory density by using KSM,” in *Proceedings of the Linux Symposium*, 2009, pp. 19–28.

[99] M. Schwarzl, E. Kraft, M. Lipp, and D. Gruss, “Remote memory-deduplication attacks,” in *29th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2022.

[100] Linux, “Core scheduling; the Linux kernel documentation,” https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html, 2022.

[101] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, “Everywhere all at once: Co-location attacks on public cloud FaaS,” in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2024. [Online]. Available: https://doi.org/10.1145/3617232.3624867

[102] 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 *2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015. [Online]. Available: https://doi.org/10.1145/2810103.2813708 pp. 1406–1418.

[103] P. W. Deutsch, Y. Yang, T. Bourgeat, J. Drean, J. S. Emer, and M. Yan, “Dagguise: mitigating memory timing side channels,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 329–343.

[104] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)*. IEEE, 2006, pp. 423–432.

[105] B. C. Schwedock and N. Beckmann, “Jumanji: The case for dynamic nuca in the datacenter,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 665–680.

[106] M. Sun, T. Wei, and J. C. Lui, “Taintart: A practical multi-level information-flow tracking system for android runtime,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.

[107] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *2012 Third World Congress on Software Engineering*. IEEE, 2012, pp. 101–104.

[108] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 235–252.

[109] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[110] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 797–809.

[111] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the 22th USENIX Security Symposium*, 2013, pp. 431–446.

[112] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 406–421.

[113] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 657–674.

[114] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Res. and Dev.*, pp. 25–33, 1967.

[115] M. Johnson, *Superscalar microprocessor design*, ser. Prentice Hall series in innovative technology. Prentice Hall, 1991.

[116] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *MICRO*, 2019, pp. 954–968.

[117] Intel, "Refined Speculative Execution Terminology," https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology, 2020.

[118] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *MICRO*. IEEE, 2018, pp. 428–441.

[119] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[120] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 264–276.

[121] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 723–735.

[122] S. Ainsworth and T. Jones, "MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State," in *International Symposium on Computer Architecture (ISCA)*, May 2020.

[123] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 707–720.

[124] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An Undo Approach to Safe Speculation," in *MICRO*, October 2019.

[125] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019, pp. 395–410.

[126] G. Cloud, "Virtual Private Cloud (VPC) | Google Cloud," "https://cloud.google.com/vpc", 2023.

[127] D. Inc, "Docker: Accelerated, Containerized Application Development," https://www.docker.com/, 2023.

[128] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling MEC services in fast and secure way," in *Sixth International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019, Granada, Spain, October 22-25, 2019*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/IOTSMS48152.2019.8939164 pp. 209–214.

[129] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020.

[130] I. Corparation, "Intel 64 and IA-32 architectures software developer's manual," Combined Volumes, Dec, 2021.

[131] L. Contributors, "Linux Source Code," https://github.com/torvalds/linux/blob/e62252bc55b6d4eddc6c2bdbf95a448180d6a08d/arch/x86/kernel/tsc.c, 2023.

[132] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer... in the cloud," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3037697.3037703 pp. 599–613.

[133] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/SP.2019.00042 pp. 39–54.

[134] W. Contributors, "Pentium III - Wikipedia," https://en.wikipedia.org/wiki/Pentium_III, 2023.

[135] G. Cloud, "Pricing | Cloud Run | Google Cloud," https://cloud.google.com/run/pricing, 2023.

[136] E. B. Fowlkes and C. L. Mallows, "A method for comparing two hierarchical clusterings," *Journal of the American Statistical Association*, vol. 78, no. 383, pp. 553–569, 1983.

[137] S. M. Ross, *Introductory Statistics*. Academic Press, 2017.

[138] A. AWS, "Secure and resizable cloud compute - Amazon EC2 - Amazon Web Services," https://aws.amazon.com/ec2/, 2023.

[139] C. Fang, H. Wang, N. Nazari, B. Omidi, A. Sasan, K. N. Khasawneh, S. Rafatirad, and H. Homayoun, "Repttack: Exploiting cloud schedulers to guide co-location attacks," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-237/

[140] A. S. Foundation, "Apache Cassandra | Apache Cassandra Documentation," https://cassandra.apache.org/_/index.html, 2023.

[141] B. Gregg, "The Speed of Time," https://www.brendangregg.com/blog/2021-09-26/the-speed-of-time.html, 2021.

[142] A. M. D. Inc, "AMD64 architecture programmer's manual," Volumes 1-5, June, 2023.

[143] Y. Azar, S. Kamara, I. Menache, M. Raykova, and F. B. Shepherd, "Co-location-resistant clouds," in *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*. ACM, 2014. [Online]. Available: https://doi.org/10.1145/2664168.2664179 pp. 9–20.

[144] Y. Han, T. Alpcan, J. Chan, C. Leckie, and B. I. P. Rubinstein, "A game theoretical approach to defend against co-resident attacks in cloud computing: Preventing co-residence using semi-supervised learning," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 3, pp. 556–570, 2016. [Online]. Available: https://doi.org/10.1109/TIFS.2015.2505680

[145] V. Varadarajan, T. Ristenpart, and M. M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan pp. 687–702.

[146] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*. Springer, 2016, pp. 118–140.

[147] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011. [Online]. Available: https://doi.org/10.1109/SP.2011.31 pp. 313–328.

[148] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 639–650.

[149] Cloudflare, "Security Model - Cloudflare Workers docs," https://developers.cloudflare.com/workers/learning/security-model/, 2023.

[150] H. M. Makrani, H. Sayadi, N. Nazari, K. N. Khasawneh, A. Sasan, S. Rafatirad, and H. Homayoun, "Cloak & co-locate: Adversarial railroading of resource sharing-based attacks on the cloud," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*. IEEE, 2021, pp. 1–13.

[151] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 2017, pp. 469–482.

[152] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. USENIX Association, 2016, pp. 363–378.

[153] T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," in *2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA*. IEEE Computer Society, 2005. [Online]. Available: https://doi.org/10.1109/SP.2005.18 pp. 211–225.

[154] D. L. Mills, J. Martin, J. L. Burbank, and W. T. Kasch, "Network time protocol version 4: Protocol and algorithms specification," *RFC*, vol. 5905, pp. 1–110, 2010. [Online]. Available: https://doi.org/10.17487/RFC5905

[155] L. Polčák, J. Jirásek, and P. Matoušek, "Comment on ″remote physical device fingerprinting″ ," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 5, pp. 494–496, 2013.

[156] D. A. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "TCP extensions for high performance," *RFC*, vol. 7323, pp. 1–49, 2014. [Online]. Available: https://doi.org/10.17487/RFC7323

[157] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium*, T. Holz and S. Savage, Eds. USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp pp. 549–564.

[158] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *16th International Conference on Applied Cryptography and Network Security (ACNS)*, ser. Lecture Notes in Computer Science, B. Preneel and F. Vercauteren, Eds., vol. 10892. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-93387-0_5 pp. 83–102.

[159] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks," in *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3460120.3484816 pp. 2906–2920.

[160] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Fifteenth EuroSys Conference 2020*. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387517 pp. 30:1–30:14.

[161] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (ATC)*, H. S. Gunawi and B. C. Reed, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang pp. 133–146.

[162] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *9th Biennial Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[163] OpenSSL, "Montgomery ladder implementation of OpenSSL 1.0.1e," https://github.com/openssl/openssl/blob/46ebd9e3/crypto/ec/ec2_mult.c#L268, 2011.

[164] G. Cloud, "Cloud Computing Services | Google Cloud," https://cloud.google.com/, 2023.

[165] A. AWS, "Cloud Computing Services - Amazon Web Services (AWS)," https://aws.amazon.com/, 2023.

[166] M. Azure, “Cloud Computing Services | Microsoft Azure,” https://azure.microsoft.com/, 2023.

[167] W. Song and P. Liu, “Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/song pp. 427–442.

[168] P. Welch, “The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms,” *IEEE Transactions on audio and electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967.

[169] J. D. McCalpin, “Mapping addresses to L3/CHA slices in Intel processors,” Texas Advanced Computing Center, University of Texas at Austin, Tech. Rep., 2021.

[170] Z. Xue, J. Han, and W. Song, “CTPP: A fast and stealth algorithm for searching eviction sets on Intel processors,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 2023. [Online]. Available: https://doi.org/10.1145/3607199.3607202 pp. 151–163.

[171] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang, “SkyLake-SP: A 14nm 28-core Xeon® processor,” in *2018 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2018. [Online]. Available: https://doi.org/10.1109/ISSCC.2018.8310170 pp. 34–36.

[172] Intel, “PerfMon event - Skylake-X server events,” https://perfmon-events.intel.com/skylake_server.html, 2023.

[173] SPEC, “SPEC CPU2006 and CPU2017 flag description - platform settings for new H3C systems,” https://www.spec.org/cpu2017/flags/New_H3C-Platform-Settings-V1.3-SKL-RevE.html, 2023.

[174] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer.js: A remote software-induced fault attack in JavaScript,” in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, ser. Lecture Notes in Computer Science, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_15 pp. 300–321.

[175] M. Schwarz, M. Lipp, and D. Gruss, “JavaScript zero: Real JavaScript and zero side-channel attacks,” in *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[176] P. Vila, B. Köpf, and J. F. Morales, “cgvwzq/evsets: Tool for testing and finding minimal eviction sets,” https://github.com/cgvwzq/evsets, 2020.

[177] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, “Github - zzrcxb/llcfeasible,” https://github.com/zzrcxb/LLCFeasible/blob/9bd94240/libs/cache/evset.c#L459, 2023.

[178] A. Beloglazov and R. Buyya, “Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers,” in *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science (MGC)*. ACM, 2010, p. 4.

[179] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong, “EnaCloud: An energy-saving application live placement approach for cloud computing environments,” in *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, 2009, pp. 17–24.

[180] A. Beloglazov and R. Buyya, “Energy efficient resource management in virtualized cloud data centers,” in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, 2010, pp. 826–831.

[181] G. Cloud, “Setting request timeout (services) | Cloud Run Documentation | Google Cloud,” https://cloud.google.com/run/docs/configuring/request-timeout#setting, 2023.

[182] P. Stoica and R. Moses, *Spectral Analysis of Signals*. Pearson Prentice Hall Upper Saddle River, NJ, 2005, vol. 452.

[183] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer, “Adaptive insertion policies for high performance caching,” in *34th International Symposium on Computer Architecture (ISCA)*. ACM, 2007. [Online]. Available: https://doi.org/10.1145/1250662.1250709 pp. 381–391.

[184] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, “High performance cache replacement using re-reference interval prediction (RRIP),” in *37th International Symposium on Computer Architecture (ISCA)*. ACM, 2010, pp. 60–71.

[185] H. Wong, “Intel Ivy Bridge cache replacement policy,” https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, 2013.

[186] D. Johnson, A. Menezes, and S. A. Vanstone, “The elliptic curve digital signature algorithm (ECDSA),” *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001. [Online]. Available: https://doi.org/10.1007/s102070100002

[187] M. Joye and S. Yen, “The Montgomery powering ladder,” in *Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523. Springer, 2002. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_22 pp. 291–302.

[188] P. Q. Nguyen and I. E. Shparlinski, “The insecurity of the elliptic curve digital signature algorithm with partially known nonces,” *Designs, Codes and Cryptography*, vol. 30, no. 2, pp. 201–217, 2003.

[189] Y. Yarom and N. Benger, “Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack.” *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.

[190] S. Fan, W. Wang, and Q. Cheng, “Attacking OpenSSL implementation of ECDSA with a few signatures,” in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2976749.2978400 pp. 1505–1515.

[191] G. D. Micheli, R. Piau, and C. Pierrot, “A tale of three signatures: Practical attack of ECDSA with wNAF,” in *12th International Conference on Cryptology in Africa (AFRICACRYPT)*, ser. Lecture Notes in Computer Science, A. Nitaj and A. M. Youssef, Eds., vol. 12174. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-51938-4_18 pp. 361–381.

[192] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, “LadderLeak: Breaking ECDSA with less than one bit of nonce leakage,” in *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3372297.3417268 pp. 225–242.

[193] J. Cao, J. Weng, Y. Pan, and Q. Cheng, “Generalized attack on ECDSA: Known bits in arbitrary positions,” *Designs, Codes and Cryptography*, vol. 91, no. 11, pp. 3803–3823, 2023. [Online]. Available: https://doi.org/10.1007/s10623-023-01269-7

[194] N. A. Howgrave-Graham and N. P. Smart, “Lattice attacks on digital signature schemes,” *Designs, Codes and Cryptography*, vol. 23, pp. 283–290, 2001.

[195] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/1953048.2078195

[196] M. Pal, “Random forest classifier for remote sensing classification,” *International Journal of Remote Sensing*, vol. 26, no. 1, pp. 217–222, 2005.

[197] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, “Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes,” in *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023. [Online]. Available: https://doi.org/10.1145/3582016.3582033 pp. 771–788.

[198] G. Saileshwar and M. K. Qureshi, “MIRAGE: mitigating conflict-based cache attacks with a practical fully-associative design,” in *30th USENIX Security Symposium*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar pp. 1379–1396.

[199] F. Liu, H. Wu, K. Mai, and R. B. Lee, “Newcache: Secure cache architecture thwarting cache side-channel attacks,” *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016. [Online]. Available: https://doi.org/10.1109/MM.2016.85

[200] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00050 pp. 955–969.

[201] Y. Guo, X. Xin, Y. Zhang, and J. Yang, "Leaky Way: A conflict-based cache covert channel bypassing set associativity," in *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00053 pp. 646–661.

[202] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[203] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid prototyping for microarchitectural attacks," in *31st USENIX Security Symposium*. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/easdon pp. 3861–3877.

[204] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Github - libtea/frameworks," https://github.com/libtea/frameworks/blob/7af59e01/libtea/src/libtea_cache_improved.c#L503, 2022.

[205] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Github - iaik/armageddon," https://github.com/IAIK/armageddon/blob/96ebc2d8/libflush/libflush/eviction/eviction.c#L266, 2016.

[206] A. Abel and J. Reineke, "nanobench: A low-overhead tool for running microbenchmarks on x86 systems," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 2020. [Online]. Available: https://doi.org/10.1109/ISPASS48437.2020.00014 pp. 34–46.

[207] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache," *CoRR*, vol. abs/1908.03383, 2019. [Online]. Available: http://arxiv.org/abs/1908.03383

[208] Z. Xue, J. Han, and W. Song, "Github - comparch-security/ctpp," https://github.com/comparch-security/ctpp, 2023.

[209] S. Jahagirdar, G. Varghese, I. Sodhi, and R. Wells, "Power management of the third generation Intel core micro architecture formerly codenamed Ivy Bridge," in *2012 IEEE Hot Chips 24 Symposium (HCS), Cupertino, CA, USA, August 27-29, 2012*. IEEE, 2012. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2012.7476478 pp. 1–49.

[210] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos pp. 1967–1984.

[211] A. Abel and J. Reineke, "Github - andreas-abel/nanobench," https://github.com/andreas-abel/nanoBench, 2020.

[212] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *2020 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00082 pp. 20–38.

[213] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "Secsmt: Securing smt processors against contention-based covert channels," in *USENIX Security Symposium*, 2022.

[214] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 213–224.

[215] T. M. Cover and J. A. Thomas, *Elements of information theory (2nd edition)*. Wiley, 2006.

[216] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," in *ASPLOS*, 2009.

[217] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif 3.0: Java information flow," July 2006. [Online]. Available: http://www.cs.cornell.edu/jif

[218] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 213–224.

[219] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 297–307.

[220] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 563–574.

[221] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. V. Rozas, A. Morrison, F. McKeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. R. Alameldeen, "Speculative interference attacks: breaking invisible speculation schemes," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 1046–1060.

[222] K. Shen and W. Yu, “Fractional programming for communication systems—part i: Power control and beamforming,” *IEEE Transactions on Signal Processing*, vol. 66, no. 10, pp. 2616–2630, 2018.

[223] W. Dinkelbach, “On nonlinear fractional programming,” *Management science*, vol. 13, no. 7, pp. 492–498, 1967.

[224] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[225] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[226] D. Townley and D. Ponomarev, “SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors,” in *28th International Conference on Parallel Architectures and Compilation Techniques (PACT’19)*, 2019, pp. 43–54.

[227] T. P. G. D. Group, “PostgreSQL: The world’s most advanced open source database,” https://www.postgresql.org/, 2023.

[228] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, “Context: A generic approach for mitigating spectre,” in *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[229] O. Contributors, “OpenSSL 3.0.5,” https://github.com/openssl/openssl/releases/tag/openssl-3.0.5, 2022.

[230] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator,” *ACM SIGARCH Computer Architecture News*, 2011.

[231] J. Bucek, K.-D. Lange, and J. v. Kistowski, “SPEC CPU2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[232] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “Simpoint 3.0: Faster and more flexible program phase analysis,” *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[233] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, “FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA ’18. Piscataway, NJ, USA: IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00014 pp. 29–42.

[234] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "Data–differential address trace analysis: Finding address-based side-channels in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 603–620.

[235] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 161–173.

[236] Y. Yuan, Z. Liu, and S. Wang, "Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[237] Y. Yuan, Q. Pang, and S. Wang, "Automated side channel analysis of media software with manifold learning," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4419–4436.

[238] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 859–874.

[239] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leakage in cache attacks via symbolic execution," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 1, pp. 1–27, 2019.

[240] J. Wikner and K. Razavi, "RETBLEED: arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner pp. 3825–3842.

[241] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2019.

[242] Intel, "Speculative Execution Side Channel Mitigations," https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[243] P. Turner, "Retpoline: a Software Construct for Preventing Branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[244] ARM, "Cache Speculation Side-channels," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper, Oct. 2018.

[245] C. Carruth, "Speculative Load Hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[246] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas, "Speculation invariance (invarspec): Faster safe execution through program analysis," in *MICRO*. IEEE, 2020, pp. 1138–1152.

[247] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spec-cfi: Mitigating spectre attacks using cfi informed speculation," in *IEEE S&P*, 2020, pp. 39–53.

[248] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, "Clearing the shadows: Recovering lost performance for invisible speculative execution through hw/sw co-design," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 241–254.

[249] D. Skarlatos, Z. N. Zhao, R. Paccagnella, C. W. Fletcher, and J. Torrellas, "Jamais vu: thwarting microarchitectural replay attacks," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3445814.3446716 pp. 1061–1076.

[250] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors," *CACM*, no. 7, pp. 89–97, July 2010.

[251] S. I. Inc and D. L. Weaver, *The SPARC architecture manual*. Prentice-Hall, 1994.

[252] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *ICPP'91*, 1991.

[253] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.

[254] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[255] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *MICRO*, 2019.

[256] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 168–179.

[257] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 347–360.

[258] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr, and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies," in *MICRO*, 2010, pp. 151–162.

[259] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-speculative load-load reordering in TSO," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080220 pp. 187–200.

[260] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 14:1–14:25, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3085572

[261] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution," *arXiv e-prints*, p. arXiv:1802.09085, Feb 2018.

[262] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[263] C. Canella, J. V. 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*, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[264] J. Horn, "Speculative Store Bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=15282018.

[265] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *IEEE S&P*, 2019, pp. 88–105.

[266] I. Cutress, "The Intel Second Generation Xeon Scalable: Cascade Lake, Now with Up To 56-Cores and Optane!" *AnandTech*, Apr. 2019. [Online]. Available: https://www.anandtech.com/show/14146/intel-xeon-scalable-cascade-lake-deep-dive-now-with-optane

[267] L. B. Michael, M. J. Mihaljevic, S. Haruyama, and R. Kohno, "A framework for secure download for software-defined radio," *IEEE Communications Magazine*, vol. 40, no. 7, pp. 88–96, 2002.

[268] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *International Workshop on Information Hiding*. Springer, 2002, pp. 400–414.

[269] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[270] B. Hardekopf and C. Lin, “Flow-sensitive pointer analysis for millions of lines of code,” in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 289–298.

[271] Intel, “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, Feb. 2019.

[272] J. L. Henning, “SPEC CPU2006 Benchmark Descriptions,” *ACM SIGARCH Computer Architecture News*, 2006.

[273] Radare2, “UNIX-like reverse engineering framework and command-line toolset,” https://github.com/radareorg/radare2.

[274] J. Liedtke, N. Islam, and T. Jaeger, “Preventing denial-of-service attacks on a /spl mu/-kernel for WebOSes,” in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 73–79.

[275] J. Shi, X. Song, H. Chen, and B. Zang, “Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2011, pp. 194–199.

[276] T. Kim, M. Peinado, and G. Mainar-Ruiz, “STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud,” in *USENIX Security*, 2012.

[277] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, “Last-level cache side-channel attacks are feasible in the modern public cloud,” in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024), Volume 2, La Jolla, CA, USA, April 27–May 1, 2024*. ACM, 2024. [Online]. Available: https://doi.org/10.1145/3620665.3640403

[278] O. Aciicmez and J.-P. Seifert, “Cheap hardware parallelism implies cheap security,” in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. IEEE, 2007, pp. 80–91.

[279] QEMU, “QEMU version 4.2.0 released,” https://www.qemu.org/2019/12/13/qemu-4-2-0/, 2019.

[280] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, “Microscope: Enabling microarchitectural replay attacks,” in *ISCA*. IEEE, 2019, pp. 318–331.

[281] A. Glew, G. Hinton, and H. Akkary, “Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions,” 1997, uS Patent 5,680,565.

[282] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, “Casa: End-to-end quantitative security analysis of randomly mapped caches,” in *MICRO*. IEEE, 2020, pp. 1110–1123.

[283] D. J. MacKay and D. J. Mac Kay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[284] H. Okhravi, S. Bak, and S. T. King, "Design, implementation and evaluation of covert channel attacks," in *HST*, 2010, pp. 481–487.

[285] 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 *30th USENIX Security Symposium*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella pp. 645–662.

[286] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *ASPLOS*, 2021, pp. 1077–1090.

[287] M. Lipp, V. Hažić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *AsiaCCS*, 2020, pp. 813–825.

[288] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *31st USENIX Security Symposium*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/tatar pp. 989–1007.

[289] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *WCAE*, 2004, pp. 22–es.

[290] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," in *AsiaCCS*, 2020, pp. 481–493.

[291] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *CCS*, 2016, pp. 380–392.

[292] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE S&P*, 2013, pp. 191–205.

[293] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-leak forwarding: Leaking data on meltdown-resistant cpus," *arXiv preprint arXiv:1905.05725*, pp. 15–20, 2019.

[294] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *CCS*, 2016, pp. 368–379.

[295] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[296] S. Casey, "How to determine the effectiveness of hyper-threading technology with an application," *Intel Technology Journal*, vol. 6, no. 1, p. 11, 2011.

[297] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *ESORICS*. Springer, 2011, pp. 355–371.

[298] 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*, 2018, pp. 937–954.

[299] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *USENIX Security*, 2017, pp. 1041–1056.

[300] J. V. Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3152701.3152706 pp. 4:1–4:6.

[301] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *32nd IEEE Symposium on Security and Privacy, SP 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011. [Online]. Available: https://doi.org/10.1109/SP.2011.22 pp. 490–505.

[302] E. L. Lehmann and J. P. Romano, *Testing statistical hypotheses*. Springer Science & Business Media, 2006.

[303] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, July 1970.

[304] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, 2000.

[305] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 5, pp. 651–664, 2010. [Online]. Available: https://doi.org/10.1109/TKDE.2009.209

[306] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[307] A. Partow, "C++ Bloom Filter Library," https://github.com/ArashPartow/bloom, 2020.

[308] M. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: eradicating controlled-channel attacks against enclave programs," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/

[309] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: closing controlled channels with self-paging enclaves," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387541 pp. 7:1–7:16.

[310] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, “Varys: Protecting SGX enclaves from practical side-channel attacks,” in *Proc. of the USENIX Annual Technical Conference (ATC)*, 2018.

[311] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, “Detecting privileged side-channel attacks in shielded execution with Déjá Vu,” in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[312] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[313] 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 *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[314] Z. Zhou, M. K. Reiter, and Y. Zhang, “A software approach to defeating side channels in last-level caches,” in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[315] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, “Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races,” in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[316] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, “Strong and efficient cache side-channel protection using hardware transactional memory,” in *Proc. of the USENIX Security Symposium (USENIX)*, 2017.

APPENDIX A: Contention-Based Side-Channel Attacks Exploiting the Page Walker

A.1 INTRODUCTION

As discussed in Section 2.2, side channels can be classified along two axes [278]: stateful vs. stateless and direct vs. indirect. In this appendix, we perform the first investigation of *stateless-indirect* channels by exploiting interactions between in-flight operations stemming from the hardware page walker and other sources (e.g., program memory operations). We show how stateless-indirect channels are possible and enable powerful new attacks. We call our channel and attack framework *Binoculars*.

We find that because indirect memory operations are issued outside the purview of normal processor structures (e.g., the reorder buffer) they "live by different rules" and exhibit novel interactions with other memory operations. Based on these interactions, we construct novel attack primitives.

First, we show that shared resource contention between page walker (indirect) loads and regular (direct) memory operations can cause *significant* delays in thread execution time (e.g., up to 20,000 cycles) stemming from a single dynamic instruction. This magnitude of delay dwarfs the one created by any other microarchitectural side channel by at least two orders of magnitude. It enables Binoculars to create new low-noise attacks that are relatively easy to perform and observe despite our channel being stateless.

Second, we show how the contention depends on the addresses of the memory operations involved. We show that this address dependence applies not only to high-order address bits (e.g., the page number) or lower-order address bits (e.g., the bits that map the address to a cache set) but also to *intra cache line address bits* and *across address spaces*. In fact, we show that Binoculars can leak more bits of a victim's (virtual) memory address than any prior channel across address spaces.

Using the above attack primitives, we perform end-to-end attacks on security-critical programs. To start, we design and optimize a covert channel using Binoculars' underlying stateless-indirect channel that can achieve a high capacity of 1116 KB/s on a Cascade Lake-X machine. We then design a side-channel attack that steals keys from OpenSSL's side-channel resistant ECDSA by learning the ECDSA nonce k. Here, the nonce is computed by an implementation of the Montgomery ladder algorithm that is hardened against timing side channels. Binoculars is able to amplify subtle nonce-dependent behaviors occurring during execution into large timing delays that can be measured with low noise. This is critical for the attack to succeed, since each run of ECDSA uses a different nonce k. Finally, we fully break kernel ASLR (KASLR).

This appendix makes the following contributions:

• We investigate and demonstrate the first stateless-indirect channel. It is based on implicit loads issued by the page walker. The resulting attack framework, *Binoculars*, has a high signal-to-noise ratio and leaks a wide range of virtual address bits.
• We design and implement two Binoculars attack primitives. One leaks the byte offset of a store within the page. The other leaks the full virtual page number of a TLB-missing request.
• We demonstrate end-to-end attacks on real hardware, which include extracting the nonce k in ECDSA with a single victim run and fully breaking KASLR.

A.2 ADDITIONAL BACKGROUND ON X86 MEMORY SYSTEM

A.2.1 Page Tables in x86

The hardware performs virtual to physical address translation by first partitioning the virtual address into a page number and an offset, and then mapping the virtual page number to a physical page number using a *page table* data structure created by the operating system (OS). In x86-64, the page table is a 4-level radix tree that supports multiple page sizes. We focus on the basic case of 4 KB pages. A page table search is called a *page walk* and is done by a hardware unit called the *page walker* on a TLB miss.

Figure A.1a shows the page table structure and the page walk process. Address translation uses four levels of page tables, which we refer to as PL_4, PL_3, PL_2, and PL_1. The root level, PL_4, is pointed to by the CR3 register. Each page in the page tables contains an array of 512 8-byte *page table entries* (PTEs). The virtual page number is decomposed into four 9-bit *PL indexes*, each of which selects a PTE from its corresponding level of the tree. Each PTE holds the physical page number of the next level of the tree or, at the lowest level, the final translation. Overall, to perform a page walk, the page walker issues four loads in total.

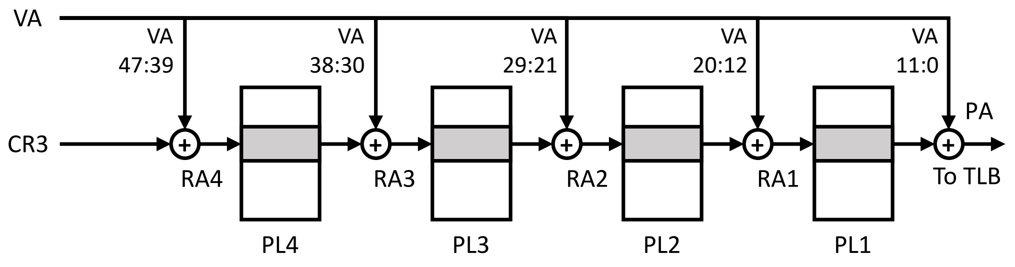
Because the 4-level page table supports only a 48-bit virtual address space, the 64-bit virtual addresses in x86-64 must be *canonical*—meaning that bits 64–48 are equal to bit 47. The address space is divided into two equal halves [130]. The lower canonical half is user space, while the upper canonical half is used by the OS kernel. An unprivileged user can only allocate pages in the lower canonical half.

To speed up the virtual address translation process, x86 processors cache address translations in two levels of translation lookaside buffers (TLBs). The first level TLBs (iTLB and dTLB) cache instruction and data translations, respectively. The second level TLB (sTLB) is larger and caches both instruction and data translations. A page walk is triggered if a translation request misses in all levels of TLBs. To minimize the TLB-miss penalty, the page walker loads check the cache hierarchy and so they can benefit from cached PTEs.

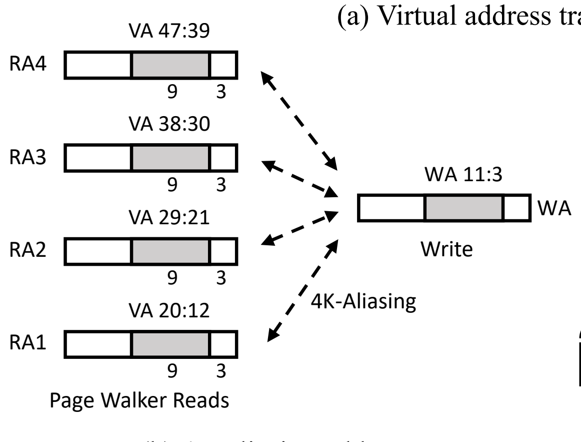
A.2.2 False Dependences in the L1D Cache

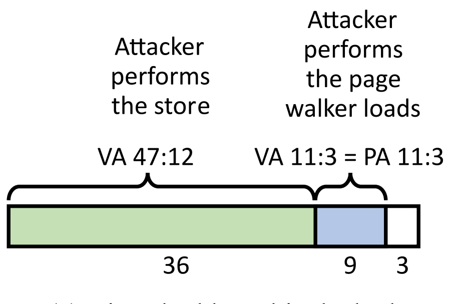
On Intel processors, the L1D cache is virtually indexed and physically tagged. It uses part of the virtual address (VA) bits (e.g., bits 11-6) as the index to find the cache set and uses the physical address (PA) tag to select the cache line within the set. This design enables the L1D cache to be accessed in parallel with the TLB translation.

The L1D cache uses the 12 least significant bits (i.e., the offset part) of the VA to detect potential dependences between multiple reads and writes that are issued to it, before their translations finish. If a read and a write target addresses with the same offsets (i.e., their 12 least significant bits are the same), then a dependence is possible. When a potential dependence is detected, one of the requests is squashed and will retry. The L1D cache thus conservatively prevents simultaneously reading and writing of addresses that have the same 12 least significant bits (i.e., they are *4K-aliasing*) even though there might be no dependence between the requests (i.e., the dependence may be a *false dependence*) [9, 10, 130]. Depending on the implementation, the dependence check can be done at a word granularity [9]—i.e., the read and the write addresses only need to share bits 11–2 to be counted as potentially dependent. In this appendix, we say that two addresses have *4K-aliasing* if they have the same bits 11–2. These addresses are subject to false dependences.



(a) Virtual address translation.




Attacker performs the store
Attacker performs the page walker loads
VA 47:12
VA 11:3 = PA 11:3
36
9
3

(b) 4K-aliasing addresses. (c) Virtual address bits leaked.

Figure A.1: Overview of the Binoculars attack.

A.3 THREAT MODEL

We consider an attacker who is an unprivileged user on a hypertheaded multi-core x86 machine. The attacker's goal is to learn some of the bits of the address operand of specific memory load/store instructions in some victim, through local hardware resource utilization changes modulated by the victim. The victim may be another process (belonging to a different user) or the OS kernel. Either way, the victim does not share virtual or physical memory with the attacker processes. We assume that the attacker knows the contents of the victim's executable.

We assume a system configuration similar to that in prior cross-hyperthread side-channel attacks [8, 9, 10, 12, 13]. The system has Hyper-Threading enabled, and the attacker can interact with the OS scheduler to run its attack process on a hyperthread that shares the same physical core with the victim hyperthread. For attacks that rely on observing delays in the victim's execution, we do not assume a cooperative victim that times and reports its own execution.

A.4 THE BINOCULARS ATTACK

The CacheBleed [10] and MemJam [9] attacks have shown that existing processors are vulnerable to false dependences between writes issued by a thread and reads issued by another thread (Section A.2.2). In this appendix, we show, for the first time, an attack that exploits false dependences between writes issued by a thread and reads issued by the hardware during a page walk triggered by a second thread. We call the new attack *Binoculars*.

Compared to the prior false dependence attacks, we will see in this section that Binoculars is both easier to setup and leaks new bits. Specifically, if the attacker is the writer, Binoculars can leak the virtual page number of the victim access that triggers the page walk. On the other hand, if the attacker is the thread that triggers the page walk, Binoculars can leak page offset bits 11-3 of the address written by the victim. Note, the victim and attacker do not share an address space.

To demonstrate Binoculars, we run experiments on Intel Xeon W-2245 (Cascade Lake-X), Intel i7-7820X (Skylake-X), and Intel Xeon E3-1246 v3 (Haswell-EP) platforms. One hyperthread reads from an address that causes a miss in both TLB levels and hence triggers a page walk. Recall that, during the page walk, the hardware issues up to four loads to the data cache hierarchy, corresponding to the requested entries in the four page table levels. In Figure A.1a, the four page levels are called PL_4, PL_3, PL_2, and PL_1, and the actual addresses read are RA_4, RA_3, RA_2, and RA_1.

The other sibling hyperthread keeps writing to an address WA. We perform two experiments: one where the WA has a false dependence with one of the RA_i, and one where it does not. Following past work [9, 10], a false dependence is obtained with 4K-aliasing — in our case, when bits 11-3 of the two addresses are the same because we issue 8-byte loads. We repeat each experiment 100 times, measuring the time taken by the reader hyperthread to complete its TLB-missing access.

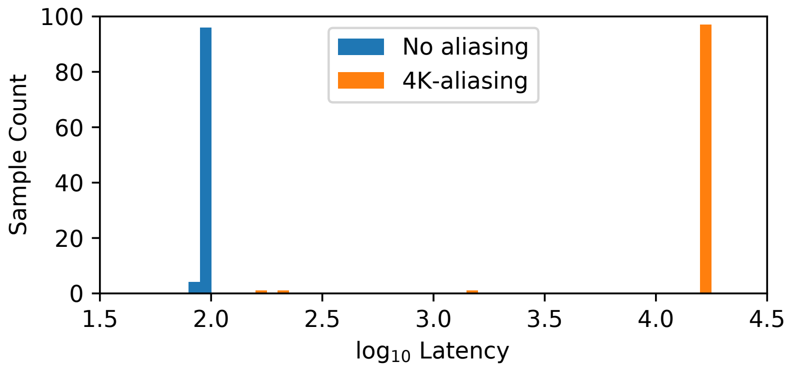


Figure A.2: Distribution of the latency of a TLB-missing access while the sibling hyperthread keeps issuing writes whose address may or may not alias with the page walker loads. The data is measured on an Intel Skylake-X.

Figure A.2 shows the histogram of measured read latencies in the two experiments running on a Skylake-X. To make the histogram readable, we plot the X axis in logarithmic scale. From the figure, we see that when the page walker loads and the store are not 4K-aliasing, the page read takes about 100 cycles (including the time for reading the timestamp). However, when there is 4K-aliasing, the latency goes up to ≈20,000 cycles (or $10^{4.3}$ in the figure). The page walker load is stalled and delayed for a long time. This very obvious difference in latency is exploited by Binoculars to leak address bits.

In this section, we discuss the two directions of the Binoculars attack: when the attacker triggers the page walk (Section A.4.1) and when it performs the repeated writes (Section A.4.2).

A.4.1 Leaking the Page Offset of a Store Address

In this attack, the attacker triggers the page walk and the victim performs repeated writes to the same address. The attacker keeps changing the address of the page that triggers the page walk and measures the latency of an access to the page. When the attacker observes a high access latency, it can deduce that the page offset of a page walk read and of the write have a false dependence. Since, in this attack, the information flows from victim stores to attacker page walker loads, we call this primitive the *store→load channel*.

To understand the attack, consider Figure A.1b, which shows the addresses of the four loads issued during a page walk. Given a TLB-missing access to a virtual address VA, the hardware first reads address RA_4, whose address bits 11-3 are equal to VA bits 47-39. After that, the page walker reads address RA_3, whose 11-3 address bits are equal to VA bits 38-30. Then, the page walker reads RA_2 and RA_1. Bits 2-0 of RA_4, RA_3, RA_2, and RA_1 are 000 because these loads read 8-byte page table entries, which are aligned to 8-byte boundaries. If any of these four addresses has a false dependence with the address WA written by the victim, a long latency access ensues. The false dependence occurs when two addresses have the same bits 11-3 because we issue 8-byte loads—i.e., the two addresses have 4K aliasing (Section A.2). Hence, this attack can learn bits 11-3 of the victim store address, which are the page offset bits with sub-cacheline granularity (Figure A.1c).

```
const u32 secret_offset = 0x528;
char *page = mmap(NULL, PAGE_SIZE, ...);
while (true) {
  page[secret_offset] = 0xff;
}
```

Figure A.3: Victim program that demonstrates the store→load channel..

Figures A.3 and A.4 show simplified programs that demonstrate the store→load channel. The demonstration extracts bits 11-3 of a store address in a victim program. As shown in Figure A.3, the victim program allocates a page and keeps writing to it at a fixed page offset (i.e., `0x528`), which is a secret. As shown in Figure A.4, the attacker program first allocates 512 continuous pages (Line 4). The page table entries (PTEs) of these 512 pages fill a 4KB PL_1 page, since each PTE is 8 bytes. Then, for each page, the attacker flushes the page's translation from TLBs and issues an access to the page, which triggers a page walk. The page walk of one of the pages will issue a load to a RA_1 address whose bits 11-3 match bits 11-3 in the victim's WA address. Because of this 4K aliasing, the latency of the access to this particular page will be higher.

We run both programs on two different hyperthreads of the same physical core. For each page of the attacker, we measure the page access latency 100 times and use the average value. The page walker read of each of the 512 pages tests a different 8-byte aligned address offset within a 4KB page. Figure A.5 shows the average latency measured at each PL_1 offset on a Skylake-X. As the plot shows, the page access latencies are very low at most PL_1 offsets. When the PL_1 offset gets close to the victim's secret offset, $0x528$, the access latency starts to increase, and it reaches its peak value when the PL_1 offset exactly matches the secret offset. We obtain similar results on a Haswell-EP and a Cascade Lake-X.

```
const u32 npages = 512;
u32 latencies[npages];
const u64 size = PAGE_SIZE * npages;
char *base_page = mmap(NULL, size, ...);
for (u32 i = 0; i < npages; i++) {
  char *page = base_page + i * PAGE_SIZE;
  invalidate_tlb(page);
  u64 t_start = read_timestamp();
  maccess(page);
  u64 t_end = read_timestamp();
  u32 PL1_index = ((u64)page & 0x1ff000) >> 12;
  latencies[PL1_index] = t_end - t_start;
}
```

Figure A.4: Attacker program that demonstrates the store→load channel..

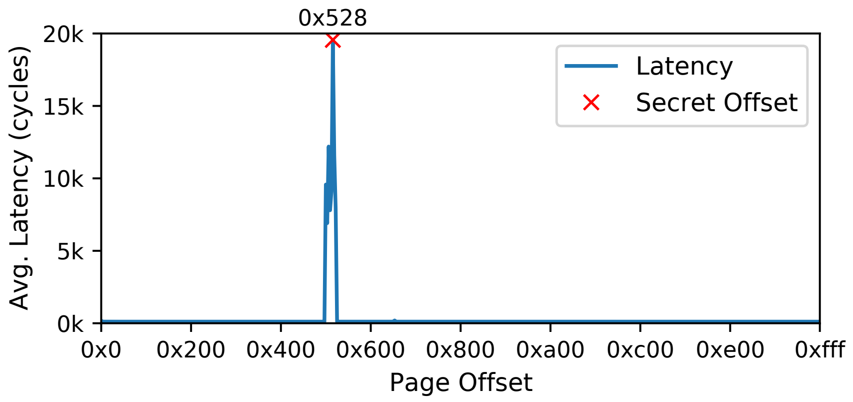


Figure A.5: Demonstration of the store→load channel on a Skylake-X.

The reason why the peak is not sharper is that page walker loads can also be stalled by stores that access the same L1 cache set. In this case, the two addresses only need to share bits 11-6. As we will see in Section A.5, this second type of false dependence is harder to induce.

To maximize the number of different offsets monitored by a single TLB-missing access, the attacker can carefully allocate a page at an address that has a different PL offset at each level. In this case, as shown in Figure A.1b, the attacker can theoretically monitor up to four different offsets, using the page walker loads from PL_4, PL_3, PL_2, and PL_1. However, in the current implementation of x86-64, an unprivileged user can only allocate pages in the lower half of the 64-bit VA space (Section A.2.1), which means that the attacker does not have full control of the PL_4 index (i.e., bits 47-39 of the VA) and cannot use it to monitor arbitrary store offsets.

In addition, since we assume an unprivileged attacker (Section A.3), the attacker cannot use privileged instructions to flush the TLB. Instead, to evict a target translation from the TLB, she has to build an eviction set of pages. Note that the hash function used in Skylake-X to map a page to a set in the TLB uses the PL_1 index and part of the PL_2 index (i.e., bits 26-12 of the VA) [8]. As a result, to build an eviction set, the attacker uses pages with different PL_3 indexes but the same PL_2 and PL_1 indexes. Consequently, the attacker cannot typically use PL_3 indexes to monitor store offsets, and has to limit herself to monitoring two offsets (using PL_2 and PL_1 indexes) with each TLB-missing access.

```
const u64 addr = 0x5d21ca821000ull;
char *page = mmap(addr, PAGE_SIZE, ...);
while (true) {
  wait_for_attacker();
  invalidate_tlb(page);
  u64 t_start = read_timestamp();
  maccess(page);
  u64 t_end = read_timestamp();
  u64 t_diff = t_end - t_start;
  // signal the attacker process; pass t_diff
  signal_attacker(t_diff);
}
```

Figure A.6: Victim program that demonstrates the load→store channel.

```
const u32 nindexes = 512;
u32 latencies[nindexes];
char *page = mmap(NULL, PAGE_SIZE, ...);
for (u32 idx = 0; idx < nindexes; idx++) {
  u32 offset = idx << 3;
  signal_victim(); // signal the victim process
  while (wait_for_victim()) {
    page[offset] = 0xff;
  }
  latencies[idx] = get_victim_latency();
}
```

Figure A.7: Attacker program that demonstrates the load→store channel.

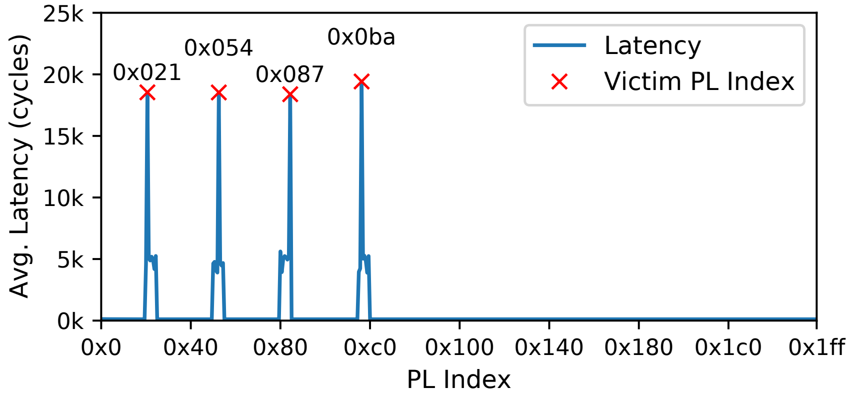


Figure A.8: Demonstration of the load→store channel on a Skylake-X.

A.4.2 Leaking the Virtual Page Number of the Address of an Access

In this attack, the attacker repeatedly stores to a given offset in a page and the victim suffers a TLB miss that triggers a page walk. The attacker keeps changing the page offset of the store address and observes the victim's performance. When the victim's access latency is high, one of the page walk loads is 4K-aliasing with the attacker's store. The attacker can then learn the *PL* index of a level of the page table entry. By continuing to change the page offset of the store address, the attacker can recover the *PL* indexes of all the different page levels. As a result, as we will see later, the attacker will be able to learn the full virtual page number (VPN) of the victim's TLB-missing memory accesses (i.e., bits 47-12). Because the information in this attack flows from victim page walker loads to attacker stores, we call this primitive the *load→store channel*.

The process of the attack is shown in Figure A.1b. A page walk issues, in the worst case, loads to addresses RA_4, RA_3, RA_2, and RA_1. Each of these addresses includes, in bits 11-3, a portion of the VA of the page accessed. When one of these addresses has the same bits 11-3 bits as the attacker's store address (WA)—i.e., it 4K-alias with WA—the victim's access suffers a long latency. Based on the observed latency, the attacker can deduce the four sets of 11-3 bits in some order. With some additional experiments that will be detailed later, the attacker can put together the whole VPN of the victim access (Figure A.1c).

Figures A.6 and A.7 show simplified programs that demonstrate the load→store channel. To measure the latency of the victim's access, the code unrealistically assumes that attacker and victim processes can communicate via shared memory to synchronize, and that the victim measures its own latency and reports it to the attacker process. This setting is for demonstration only. A realistic setting will be shown in Section A.8, where the attacker only relies on the end-to-end execution time of the victim.

The victim program (Figure A.6) first allocates a page at virtual address `0x5d21ca821000`, which corresponds to indexes to PL_4, PL_3, PL_2, and PL_1 equal to `0x0ba`, `0x087`, `0x054`, and `0x021`, respectively. Then, the victim program enters a loop where, in each iteration, the victim: (i) waits for the attacker to signal it, (ii) invalidates the translation of the page from the TLBs, (iii) accesses the page and measures the access latency, and (iv) signals the attacker, passing the access latency. The attacker program (Figure A.7) first allocates buffers for latency results and a page to write to. Then, it enters a loop that iterates over all the possible 512 indexes of page table entries (PTEs) in a page. For each of the resulting PL address offsets, the attacker: (i) signals the victim process, (ii) keeps writing to an address at the PL offset in the page until the victim sends it a signal, and (iv) receives the latency of the victim access and saves it.

We run both programs on two hyperthreads of a physical core. For each PL index, we measure the latency 100 times and save the average value. Figure A.8 shows the resulting average latency for each PL index on a Skylake-X processor. Looking at the figure, we see there are four clear latency spikes. They are at indexes `0x021`, `0x054`, `0x087`, and `0x0ba`. These four spikes correspond to the four 9-bit PL indexes of the victim page. We obtain similar results on a Haswell-EP and a Cascade Lake-X.

From these latency results alone, we cannot determine which spike corresponds to which page table level. The full VPN is one of the permutations of these four indexes. There are multiple strategies to identify the correct permutation. For example, if we know which memory region the victim accesses (e.g., heap or stack), we can identify the possible PL_4 or even PL_3 indexes, since these memory regions usually have unique ranges of high-order VA bits. If the victim also happens to access neighboring pages (i.e., pages that differ in PL_1 indexes), the attacker should observe nearby spikes, and these spikes correspond to PL_1 indexes. After determining the PL_4, PL_3, and PL_1 indexes, we know which one is the PL_2 index. Last, as will be shown in Section A.8, if the memory access is to a global variable, the PL_1 index can be derived from the variable's offset in the segment.

We can easily redesign the attack so that attacker and victim do not need to synchronize, and the victim does not need to measure the latency of its own accesses. Instead, the attacker measures the latency of its stores. The idea is that, when the victim's page walker load is stalled for a long time due to 4K aliasing, the victim's pipeline is blocked, and shared resources are freed-up for the attacker. As a result, the attacker sees lower latency for its own stores because of less port contention. Consequently, in Figure A.9, we change the code from Figure A.7 so that, in each iteration, the attacker measures the latency of issuing $10,000$ stores—and neither synchronizes nor receives any latency measurement from the victim. This is a more realistic design. Figure A.10 shows the average latency of those $10,000$ stores at different indexes on a Skylake-X. It is clear that latencies drop at the victim's PL indexes.

```
// Attacker program, port contention version
const u32 nindexes = 512;
u32 latencies[nindexes];
char *page = mmap(NULL, PAGE_SIZE, ...);
for (u32 idx = 0; idx < nindexes; idx++) {
  u32 offset = idx << 3;
  u64 t_start = read_timestamp();
  for (u32 i = 0; i < 10000; i++) {
    page[offset] = 0xff;
  }
  u64 t_end = read_timestamp();
  latencies[idx] = t_end - t_start;
}
```

Figure A.9: Port-contention version of the load→store channel attack program.

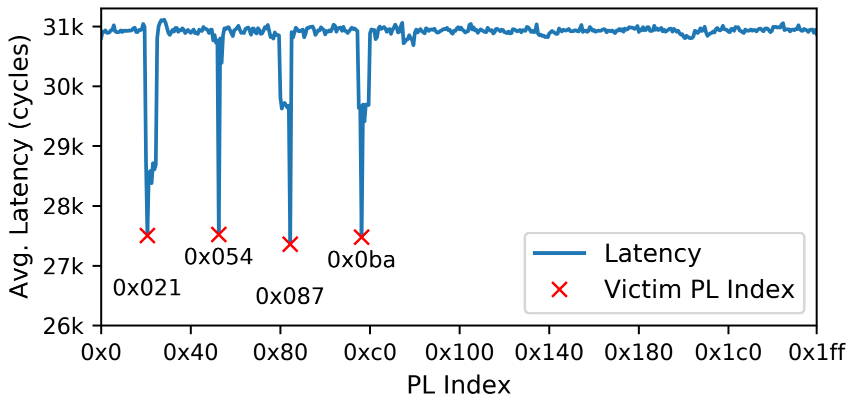


Figure A.10: Demonstration of the load→store channel with the port-contention version on a Skylake-X.

A.4.3 Extensions

Cross Virtual Machine Attack. Binoculars also works if attacker and victim are in two different virtual machines that share the same physical core. Because in a virtualized environment, a TLB-missing access also triggers page walker loads, which are subject to contention with stores from the sibling thread. To verify this, we repeat our experiments in a virtualized environment with QEMU-KVM (4.2.1) [279] on Skylake-X. The attacker and victim programs run in two different virtual machines that share the same physical core and run Ubuntu 20.04 LTS (5.4.0-105-generic). Our experiments show results that are similar to the ones in a non-virtualized environment, and thus demonstrate Binoculars can be used for cross virtual machine attacks.

Table A.1: Comparing the characteristics of different side-channel attacks.

Attack	**Timing Difference**[*]	**Virtual Address (VA) Bits Leaked**	**Leakage Granularity**	**Cross VA Space?**	**Cross Core?**
Port Contention [12, 13]	10^{-1} cycles	n/a	μOps	Yes	No
TLBleed [8]	10^{1} cycles	Bits 26-12[†]	Memory page	Yes	No
CacheBleed [10], MemJam [9]	10^{1} cycles	Bits 11-2	Sub-cacheline	Yes	No
Prime+Probe [1]	10^{2} cycles	n/a	Cache set	Yes	Yes
Flush+Reload [6]	10^{2} cycles	Offset in segment	Cacheline	No	Yes
AVX2-Based [22]	10^{2} cycles	n/a	AVX2 instruction	Yes	No[‡]
Binoculars store→load channel	10^{4} cycles	Bits 11-3	Sub-cacheline	Yes	No
Binoculars load→store channel	10^{4} cycles	Bits 47-12	Memory page	Yes	No

[*] Magnitude of the maximum timing difference that can be caused by a *single dynamic instruction or operation* that the attack uses.

[†] Applicable to an Intel Skylake-X platform [8]. Actual bits may vary on different microarchitectures.

[‡] Our threat model (Section A.3) only considers local hardware resource utilization changes caused by the victim. Since AVX2-based attacks exploit power management mechanisms that only affect each individual physical core, we do not consider it as a cross-core channel under our threat model.

Other Paging Schemes. The previous discussion is mainly focused on a 4-level paging design, which issues four page walker loads to translate a VPN. If other paging schemes (e.g., huge page, 5-level paging) are used, Binoculars will still work with different attacker capabilities.

For the store→load channel, the attacker has no incentive to use huge pages: doing so would reduce the number of page walker loads and correspondingly the number of page offsets that can be observed in a single page walk. Using 5-level paging, on the other hand, can boost the attack as one page walk can monitor five store offsets. For the load→store channel, using huge pages reduces the number of low-order VPN bits that an attacker can extract. But it is still possible to attack (kernel) ASLR, as its entropy usually resides in high-order VPN bits (see Section A.8). Using 5-level paging, the attacker will observe five latency spikes associated with each level of translation, instead of four spikes shown in Figure A.8 and A.10, which makes finding the correct permutation of spikes and recover the full VPN harder.

Other CPUs. We believe the root cause of the Binoculars attack is related to Intel's optimization of page walker loads (see Section A.5). Therefore, Binoculars is likely exclusive to Intel processors. For example, the same experiments on an AMD-EPYC-7502 processor show that it does not exhibit the Binoculars channel.

A.4.4 Discussion

Table A.1 compares the characteristics of Binoculars and existing side-channel attacks. From left to right, we compare: (i) the maximum timing difference that can be induced by a *single dynamic instruction or operation* that the attack uses, (ii) the virtual address bits leaked, (iii) the granularity of the leakage, and whether the attack is effective (iv) across address spaces or (v) across cores.

The second column shows the first advantage of *Binoculars*: its contention effect is very strong. *A single dynamic instruction* can create timing differences that are several orders of magnitude higher than in any other conventional side-channel attack. For example, with port contention, a dynamic instruction can cause a latency increase equal to a fraction of a cycle on average [12, 13]. Therefore, an attacker requires thousands of dynamic instructions to magnify the effects, or tens of thousands of replays to denoise the channel [280]. In CacheBleed [10] and MemJam [9], a dynamic instruction exploiting a false dependence causes a 10-cycle average latency increase. In side-channel attacks that rely on timing differences between cached and uncached accesses such as TLBleed, Prime+Probe, and Flush+Reload, the differences range from tens of cycles [8] to a few hundred cycles [1, 6]. AVX2-based side channel that is exploited in NetSpectre [22] can also cause a timing difference of a few hundred cycles. Instead, a Binoculars access can trigger a stall of up to 20,000 cycles. This property makes Binoculars more resilient to noise, which means that it can recover secrets with fewer runs and with a higher confidence. Also, because of the long duration of the stall, it is possible to observe the contention even if the attacker cannot measure the time very precisely—e.g., due to lacking a high-resolution timer.

The third column of Table A.1 shows a second advantage of *Binoculars*: it leaks a wide range of virtual address (VA) bits. The column lists the VA bits leaked by each attack. For example, TLBleed [8] can recover the bits that are used by TLB hash functions (i.e., bits 26-12 on a Skylake-X for the sTLB). Hence, TLBleed can observe only a victim's memory accesses at a page granularity, and cannot extract the full VPN. CacheBleed [10] and MemJam [9] recover low-order intra-cacheline bits (i.e., bits 11-2), but miss out on high-order bits. In Flush+Reload [6], because of ASLR, a shared memory segment can be allocated at different VA in different processes. As a result, Flush+Reload can only recover VA bits that are not subject to ASLR, namely the offset to the base of the segment—at a cache line granularity. With Binoculars, the attacker can learn VA bits 11-3 with the store→load channel and bits 47-12 (i.e., the full VPN) with the load→store channel.

The fourth column of Table A.1 shows the leakage granularity of each attack. Binoculars provides sub-cacheline resolution with the store→load channel and page-level granularity with the load→store channel. The fifth and sixth columns show whether the attack works across address spaces and across physical cores. Since Binoculars requires no shared memory, it can attack a victim running in a different address space. However, it cannot attack a victim on a different core.

Finally, Binoculars has a third, although not unique, advantage: it does not require complex state preparation. For example, most cache-based side-channel attacks require the victim data to be present at a given level of the memory hierarchy before the attack. For that, the attacker has to carefully manipulate cache state, which is slow, sometimes complex, and requires fine-grained synchronization with the victim process. Binoculars only needs page walkers to trigger loads. Contention occurs regardless of what level in the cache hierarchy the page walker loads read from.

A.5 ROOT CAUSE ANALYSIS

In this section, we explore the root cause of the strong resource contention triggered by Binoculars (up to $20,000$ cycles on a Skylake-X processor, Section A.4).

Source of the Contention. We first confirm that the contention originates from the page walk triggered by TLB misses. To this end, we monitor various TLB- and page-walker-related performance counter sub-events (Table A.2) during an experiment similar to the one in Figure A.2. In the experiment, one hyperthread performs a TLB-missing page access while the sibling hyperthread keeps writing to an address that is, (or is not), 4K-aliasing with one of the page walker loads. Before the measurement, we first warm up the page by accessing it multiple times so that its data is cached; then, we invalidate its translation from all the TLB levels.

Table A.2: List of performance counter events.

Parent Event	**Sub-event**	**Description**
DTLB_LOAD_MISSES	MISS_CAUSES_A_WALK	Number of page walks (including incomplete walks)
DTLB_LOAD_MISSES	WALK_COMPLETED	Number of completed page walks
DTLB_LOAD_MISSES	WALK_DURATION[†]	Count of *core clock cycles* when the page walker is servicing page walks
PAGE_WALKER_LOADS	DTLB_L1[*]	Number of page walker loads that hit in L1D+Fill Buffer
PAGE_WALKER_LOADS	DTLB_L2[*]	Number of page walker loads that hit in L2
PAGE_WALKER_LOADS	DTLB_L3[*]	Number of page walker loads that hit in L3
PAGE_WALKER_LOADS	DTLB_MEMORY[*]	Number of page walker loads that read from main memory
MEM_LOAD_RETIRED	L1_HIT	Number of load instructions that hit in L1D (excluding page walker loads)
n/a	Unhalted Core Cycles[†]	Count of *core clock cycles* when the core is running

[*] Although these sub-events are only documented for Haswell-EP and Broadwell-EP, we find that they still exist and are functional on newer microarchitectures like Skylake-X and Cascade Lake-X.

[†] These sub-events count *core clock cycles*, which are subject to turbo-boost. The rest of the appendix uses *reference clock cycles*, which are not.

Table A.3 shows performance counter values collected on an Intel Skylake-X for both the 4K-aliasing and no-aliasing cases. If the store is not aliasing with the page walker load, the page walker starts and completes one page walk to handle the TLB miss (`MISS_CAUSES_A_WALK` and `WALK_COMPLETED` events, respectively), which takes 42 core clock cycles (`WALK_DURATION`), and all its page walker loads plus the data access hit in the L1D cache. It takes 180 core clock cycles in total to complete the page walk, the data access, and then stop the performance counters.

In the 4K-aliasing case, however, the page walker starts two page walks but only finishes one. Also, the sub-event `WALK_DURATION` has a value that is close to `Unhalted Core Cycles`. These results indicate that the core spends most its cycles servicing the page walk, which takes a long time to complete. Also, the page walk seems to be aborted and restarted. The counters for page walker loads all indicate there are no L1D misses, which means that the slow page walk is not caused by cache-missing loads. These observations confirm that the contention indeed comes from the page walker. We hypothesize that the contention is so strong that it leads to resource starvation of the page walker, which triggers a "watchdog" to abort the page walk and restart it with a higher priority over shared resources.

Table A.3: Performance counter values on a Skylake-X.

Sub-event	**No Aliasing**	**4K-Aliasing**
MISS_CAUSES_A_WALK	1	2
WALK_COMPLETED	1	1
WALK_DURATION (core clock cycles)	42	16452*
DTLB_L1	4	4
DTLB_L2	0	0
DTLB_L3	0	0
DTLB_MEMORY	0	0
L1_HIT	1	1
Unhalted Core Cycles (core clock cycles)	180	16584*

* These core clock cycles correspond to $\approx 20,000$ reference clock cycles.

Cause of Starvation. To validate our starvation hypothesis, we rely on Intel's patents on virtual memory translation. According to one of Intel's patents, the page walker issues "stuffed" loads that bypass the RS and the ROB [281]. This mechanism is presented as an optimization to avoid any scheduling latency that the RS or the ROB may cause.

After the stuffed load is dispatched by the page walker, it is handled by the memory-order buffer (MOB). The MOB checks for potential conflicts with pending stores—-i.e., whether a store may be writing to the address read by the stuffed load. If a potential conflict is found, the page walker aborts the walk and retries when the conflict is resolved. Although this might sound like the root cause of the contention, our further experimentation finds that only stores from *the same thread* can cause conflicts, as the MOB is not shared by the two hyperthreads, which disproves this explanation.

If the MOB finds no conflicts, the stuffed load is issued to the L1D cache. In this step, the L1D cache may "squash" the stuffed load under certain circumstances. If the squash happens, the page walker will re-dispatch the stuffed load as soon as possible, and the re-dispatched stuffed load may get squashed by the L1D cache again. *This behavior can starve the stuffed load indefinitely*. As will be discussed later, we indeed find a performance counter sub-event that suggests that the L1D cache receives thousands of read requests from the stuffed load during a stalled page walk.

Magnitude of Starvation. Given that the L1D cache rejects data accesses for various reasons (Section A.2.2), why can Binoculars stall a page walk for up to $20,000$ cycles while attacks like CacheBleed and MemJam only delay a data access for a few cycles? We hypothesize the answer is related to instruction scheduling differences.

In CacheBleed and MemJam, the conflicts are between explicit data loads and stores. Data loads and stores are processed by the ROB and the RS, and are scheduled by the same Out-of-Order (OoO) engine of the physical core. The OoO engine can therefore detect and mediate between the conflicts after a few failed L1D accesses. In Binoculars, however, the conflicts are between implicit stuffed page walker loads and explicit data stores. Because stuffed loads are managed outside of the RS and the ROB, we hypothesize that the OoO engine cannot detect such conflicts. Consequently, the OoO engine simply allows the explicit data stores from the other hyperthread to run "at full speed", without realizing that one hyperthread is trying to perform a page walk and failing, as its stuffed loads are getting squashed. The page walker thus suffers from resource starvation and eventually triggers a mechanism that aborts and restarts the page walk (presumably with a higher priority).

Cause of L1D Squashes. We find that both set conflicts and false dependences can cause stuffed loads to be squashed by the L1D cache, depending on the writing thread's behavior. Our analysis here is based on identifying *undocumented* performance counters for these events.

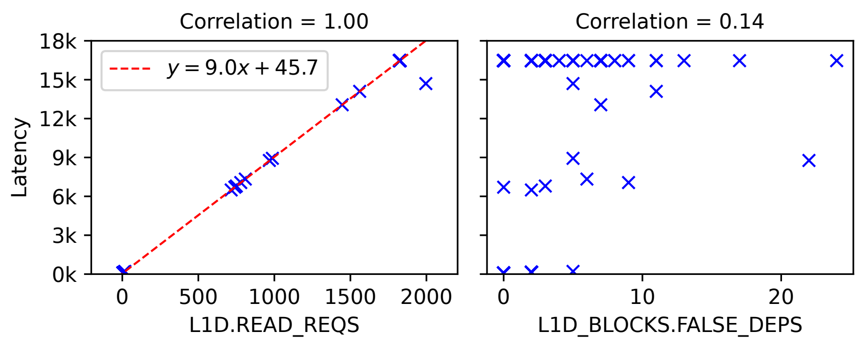
To identify relevant counters, we perform a brute force search over all possible counter sub-events, searching for the ones that are highly correlated with the access latency of TLB-missing loads. We perform the search by trying every combination of the two 1-byte-long fields, `EventSel` and `UMask`, which determine the sub-event in the performance counter configuration model-specific registers [130]. Our search finds two interesting undocumented sub-events: (1) `EventSel=0x51, UMask=0x20` and (2) `EventSel=0xbf, UMask=0x01`. Based on the `EventSels` of these two sub-events and our reverse engineering, the first sub-event likely counts the number of L1D read requests, including both successful and squashed requests. The second sub-event likely counts the number of failed L1D read requests due to false dependences (Section A.2.2). In the rest of our discussion, we will refer to these two sub-events as `L1D.READ_REQS` and `L1D_BLOCKS.FALSE_DEPS` respectively. We also find that the `L1D.READ_REQS` sub-event is present only on Haswell-EP but not on newer generations. Therefore, we will focus on results on Haswell-EP for the rest of the discussion.

We perform three experiments to understand whether Binoculars L1D squashes are due to 4K-aliasing or L1D set conflicts. The experiments monitor these undocumented sub-events as one hyperthread performs a TLB-missing memory access, which triggers a page walk that reads from RA_4, RA_3, RA_2, and RA_1, while its sibling hyperthread keeps writing to an address WA. The experiments differ in the bits shared by RA and WA, and in the frequency of writing to WA: (1) BINOCULARS-4K: RA_1 and WA share bits 11-3 (i.e., 4K-aliasing); (2) BINOCULARS-SAMESET: RA_1 and WA share bits 11-6 but differ in bits 5-3 (i.e., they are mapped to the same L1D cache set); (3) BINOCULARS-4K-LOWFREQ: RA_1 and WA share bits 11-3, but the writer thread has a reduced write frequency, as it executes arithmetic instructions between writes. We ensure that the page walker loads, the data load, and the stores only access up to two unique cache lines in an L1D set, i.e., fewer than the associativity of the L1D cache. We repeat each experiment 1000 times.

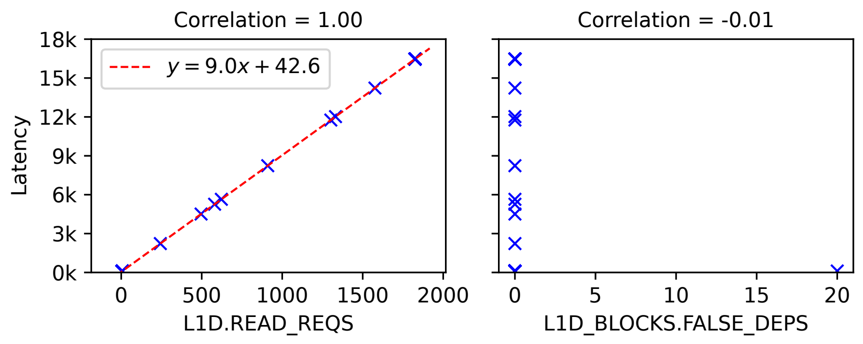
Figure A.11 shows the results on a Haswell-EP in reference clock cycles (the maximum stall on a Haswell-EP is around $16,000$ cycles). The red dashed lines are fitted linear regression lines. In the BINOCULARS-4K experiment (Figure A.11a), the access latency is strongly correlated to the number of L1D read requests, which confirms that the stuffed page walker loads are repeatedly squashed by the L1D cache and re-dispatched by the page walker. From the fitted line, on average, it takes 9 cycles to squash and retry a stuffed load. However, looking at the right plot of Figure A.11a, the correlation between the access latency and `L1D_BLOCKS.FALSE_DEPS` is very low, which suggests false dependences are not the main cause of L1D squashes in this experiment.

The BINOCULARS-SAMESET experiment (Figure A.11b) still shows many high-latency events that are correlated to `L1D.READ_REQS`. Compared to BINOCULARS-4K, however, it has significantly fewer events that reach the maximum latency (131/1000 events in BINOCULARS-SAMESET versus 847/1000 events in BINOCULARS-4K). Also as expected, `L1D_BLOCKS.FALSE_DEPS` is always 0 because the page walker load is not 4K-aliasing with stores. This experiment shows that without false dependences, contention and even starvation can still occur as long as the RA_1 and the WA are mapped to the same L1D cache set (i.e., they suffer *set conflicts*). However, they occur less frequently than they would in BINOCULARS-4K. Recall that we see a similar behavior in Figure A.5.

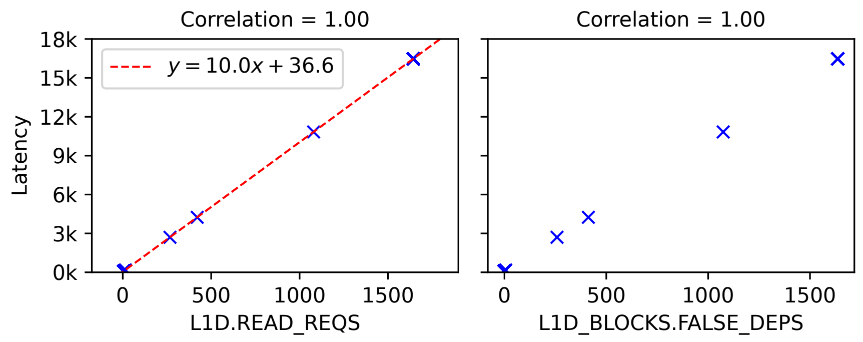
Finally, in the BINOCULARS-4K-LOWFREQ experiment (Figure A.11c), the latency is still strongly correlated to `L1D.READ_REQS` and it can reach the maximum $16,000$-cycle latency. But now it takes 10 cycles to squash and retry. Also, the latency is strongly correlated to `L1D_BLOCKS.FALSE_DEPS`, which means that false dependences in the L1D cache become the main reason of L1D squashes when the writer thread has a reduced frequency.



(a) BINOCULARS-4K.



(b) BINOCULARS-SAMESET.



(c) BINOCULARS-4K-LOWFREQ.

Figure A.11: Scatter plot between the access latency and the two undocumented sub-events. All plots share the same Y axes.

The results of the three experiments lead us to conclude that both set conflicts and false dependences can cause stuffed loads to be squashed by the L1D cache, depending on the writer thread's behavior. We believe that set conflicts only happen in an early stage of a read access, while false dependences occur in a later stage. This explains the one-cycle difference in the squash-and-retry latency. We believe that set conflicts require stricter timing requirements to trigger (e.g., that read and write requests arrive at the same cycle) compared to false dependences. Finally, we believe that set conflicts (when they occur) dominate false dependences—i.e., when a set conflict occurs, a false dependence will not happen.

The above explain the results we see. When stores are frequent (Figures A.11a and A.11b), set conflicts are more likely to occur. This explains Figure A.11a (set conflicts occur frequently and dominate false dependences when they do) and Figure A.11b (in which set conflicts occur frequently and false dependences are impossible). This also explains that in Figure A.11c set conflicts are less likely and thus false dependences dominate. Finally, this explains why starvation occurs less frequently in BINOCULARS-SAMESET than in BINOCULARS-4K: since a read request can "slip through" set conflicts due to the strict timing requirements, high latency is hard to build up in BINOCULARS-SAMESET. In BINOCULARS-4K, the "slipped-through" request will likely be squashed due to a false dependence in the next cycle, which makes the starvation more likely to happen.

A.6 BINOCULARS COVERT CHANNEL

A covert channel is a communication channel that allows two cooperating parties to bypass system policies to communicate with each other. Most covert channels are synchronous, where the transmission process is divided into time epochs for synchronization. In every epoch, the sender encodes one or several bits of information by changing microarchitectural states, while the receiver decodes the information by observing the changes. Depending on the mechanism used by the covert channel, in every epoch, after the receiver decodes the transmission, it may need to precondition the channel for the next epoch [282]. To keep the sender and the receiver well-synchronized, an epoch has to be long enough to cover the encoding and decoding operations, and the potential preconditioning operations.

A robust metric to measure a covert channel's transmission capability (i.e., its raw throughput and bit-error rate) is the *channel capacity* [283]. This metric measures the highest rate of reliable information transmission that a communication channel supports. It is computed by $r \times (1 - H(p))$, where r is the raw throughput of the channel, p is the probability of a bit error, and H is the binary entropy function. Using this formula, we can see that a high-capacity covert channel requires a high raw throughput (which is determined by the length of an epoch and the number of bits it can transmit per epoch) and a low bit-error rate (which is determined by the noise in the channel). This metric is also used in some prior work [14, 284, 285].

A straw man Binoculars covert channel works as follows. Before the transmission, sender and receiver agree on a page offset. To send a bit 1, the sender keeps writing to the agreed offset until the end of the epoch. To send a bit 0, the sender does nothing and waits for the next epoch. To decode the information, the receiver issues and times a TLB-missing memory access to a target page, whose page walk includes a load that 4K-aliases with the sender write.

If the receiver measures a high access latency, the sender is sending a bit 1; otherwise, it is sending a bit 0. After that, to precondition the channel, the receiver accesses a TLB eviction set to evict the target page from the TLB.

Table A.4: Comparison of covert channels with average capacity higher than 100 KB/s.

Attack	**Channel Capacity**	**Cross Address Space?**	**Cross Core?**
Streamline [286]	1733 KB/s	No	Yes
Lord of the Ring(s) [285]	518 KB/s	Yes	Yes
Take-a-Way [287]	505 KB/s	Yes	No
Flush+Flush [7]	463 KB/s	No	Yes
L1 Prime+Probe [2]	400 KB/s	Yes	No
Flush+Reload [6, 7]	298 KB/s	No	Yes
Binoculars (Cascade Lake-X)	1116 KB/s	Yes	No
Binoculars (Skylake-X)	622 KB/s	Yes	No
Binoculars (Haswell-EP)	177 KB/s	Yes	No

Unfortunately, this straw man scheme does not achieve a high channel capacity. Since the page walker loads can be stalled for up to $20,000$ cycles (Section A.4), an epoch has to be longer than that, which drastically limits the channel capacity. Fortunately, in practice, such large stall times are unnecessary to build a low error-rate covert channel. Therefore, we carefully tune the number of stores that are executed by the sender. We want to make sure that these stores can create reliably-high timing differences on the receiver side while keeping the epochs short.

To further improve the channel capacity, on the receiver side, we build a large TLB eviction set using methods similar to ones in [8, 288]. At every epoch, the receiver chooses the target page from that set. Moreover, the chosen target page is different from the target page used in the previous epoch. With this design, we can ensure that the read access to the target page not only decodes the information, but also evicts the translation of the page that will be used as the target in the next epoch. This design eliminates the need of preconditioning the channel through explicit eviction of TLB entries. Hence, we can support an even shorter epoch.

Finally, we also make sure that all the pages in the TLB eviction set are mapped to the same physical page. As a result, accessing them one after another does not evict their data from the caches, which removes any noise due to cache misses.

We evaluate the average capacity of the Binoculars covert channel on Intel Haswell-EP, Skylake-X, and Cascade Lake-X platforms. In each platform, we run the sender and the receiver for 100 times to transmit a 1MB-long randomly-generated message.

Table A.4 lists the channel capacity of Binoculars on each platform, as well as the capacities of prior covert channels. From the table, we see that on a Haswell-EP, Binoculars attains a moderate channel capacity of 177 KB/s with a 1.2% bit error rate. On a Skylake-X, the channel capacity increases to 622 KB/s and the bit error rate decreases to 0.9%. Finally, on a Cascade Lake-X, which is one of the latest Intel server microarchitectures, the capacity reaches 1116 KB/s with a low bit error rate of 0.6%.

The main reason for the large channel capacity variations across different platforms is the processor performance. In newer generations, processors can execute the same number of stores in a shorter period of time, and these stores can also cause more contention effects. As a result, a newer processor requires fewer stores to cause enough contention and a shorter epoch to execute them. For example, on a Haswell-EP, we need to execute 380 stores on the sender side every epoch. On a Cascade Lake-X, the number is reduced to only 80 stores, which only require a 420-cycle epoch.

Table A.4 also lists the characteristics of existing covert channels with a channel capacity greater than 100 KB/s. Among all these channels, Binoculars has the second highest channel capacity[1] (on Skylake-X and Cascade Lake-X), and is only behind Streamline [286]. Although Streamline has a higher channel capacity and supports cross-core communication, Binoculars does not require shared memory thus works across address spaces.

A.7 ATTACKING MONTGOMERY LADDER AND ECDSA

We use Binoculars to obtain the private key used by OpenSSL's ECDSA implementation. Our attack targets the Montgomery ladder, a widely used optimization for computing scalar multiplication on elliptic curves [187]. OpenSSL's ECDSA implementation uses the Montgomery ladder to calculate the point $k \times G$ during signing, where the scalar k is a nonce (i.e., an ephemeral key). Our goal is to learn the nonce k, which together with the signature can be used to derive the private key used for signing [186, 188].

Figure A.12 shows the Montgomery ladder implementation used in OpenSSL 1.0.1e. The code iterates over the bits of k. In each iteration, it performs an elliptic curve point addition and doubling by calling the functions `gf2m_Madd` and `gf2m_Mdouble`, respectively. The current bit, k_i, determines the big number variables written to in each step. If $k_i = 1$, (x_1, z_1) is added to and (x_2, z_2) is doubled; if $k_i = 0$, the order is reversed. In the following, we denote function calls to `gf2m_Madd` and `gf2m_Mdouble` under the $k_i = 1$ direction as `Madd1` (Line 6) and `Mdouble1` (Line 8), and the calls under the $k_i = 0$ direction as `Madd0` (Line 10) and `Mdouble0` (Line 11).

[1]Parallel to our work, TLB;DR [288] built a more performant covert channel with an average channel capacity of 1375 KB/s.

[2]The code is from function `ec_GF2m_montgomery_point_multiply` at `crypto/ec/ec2_mult.c:268` [163].

```
for (; i >= 0; i--) {
  word = scalar->d[i];
  while (mask) {
    if (word & mask) { // checks ki
      // compute (x1,z1)=(x1/z1)+(x2/z2)
      if (!gf2m_Madd(group,&point->X,x1,z1,x2,z2,ctx))  goto err;
      // compute (x2,z2)=2*(x2/z2)
      if (!gf2m_Mdouble(group, x2, z2, ctx))  goto err;
    } else {
      if (!gf2m_Madd(group,&point->X,x2,z2,x1,z1,ctx))  goto err;
      if (!gf2m_Mdouble(group, x1, z1, ctx))  goto err;
    }
    mask >>= 1;
  }
  mask = BN_TBIT;
}
```

Figure A.12: Montgomery ladder implementation used in OpenSSL 1.0.1e[2].

While this implementation is data-oblivious to the sequence of operations and end-to-end timing, it nevertheless has a secret-dependent order of stores to (x_1, z_1) and (x_2, z_2). Since stores to these two pairs of variables have different page offsets, *Binoculars can identify the store order by monitoring stores to these offsets, and thereby recover the k_i values.*

Challenge. The nonce k in ECDSA changes at every run and never repeats. An attacker only has *one chance* to capture a k and cannot rely on repeated runs to de-noise the channel. As a result, the attacker needs a side channel with an extremely high signal-to-noise ratio to exfiltrate k with a single victim run. There exist partial key recovery techniques for ECDSA [188, 190, 191, 192] that allow an attacker to reconstruct the private key from multiple signatures and part of corresponding ks. However, most of them assume that the known parts of ks are error free [190], or at least that they have a low bit-error rate (e.g., less than 2% [191, 192]).

A.7.1 Attack Method

We assume the attacker can obtain a signature from the victim (e.g., by making a network request) and use Binoculars to monitor the victim's signing execution. The attacker's process does not need to share any physical memory with the victim.

Our attack infers the value of k_i using the store→load channel to monitor the order of stores to (x_1, z_1) and (x_2, z_2). To avoid monitoring four variables, we only monitor stores to a single variable (e.g., x_2). We infer the value of k_i based on whether the stores happen in the first half of a Montgomery ladder iteration (in which `Madd0` executes) or in the second half of an iteration (in which `Mdouble1` executes).

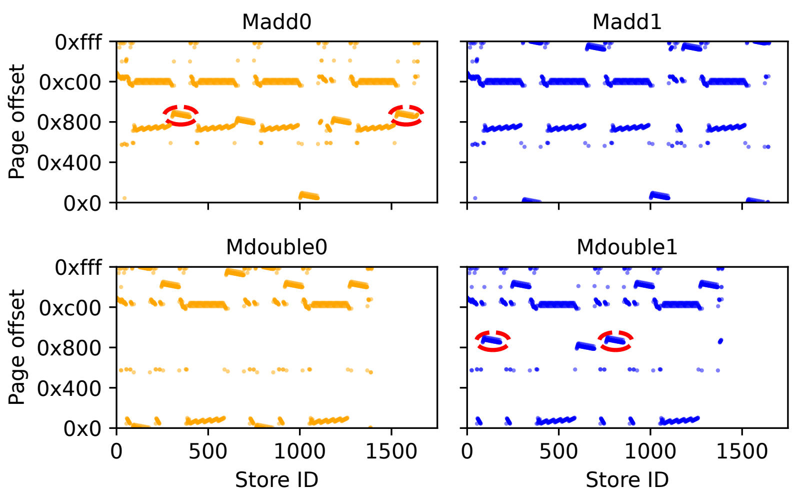


Figure A.13: Page offsets of stores in `Madd0`, `Mdouble0`, `Madd1`, and `Mdouble1`. Red circles highlight stores to variable x_2. The traces are collected with Intel Pin [289] on a Skylake-X.

Crucially, we design a realistic end-to-end attack that does not assume synchronization between victim and attacker, such as knowing when Montgomery ladder iterations begin and end. Our attack method contains the following three steps:

Step 1: Identify Target Store Page Offsets. Finding store page offsets to monitor can be done offline. Page offsets depend only on memory allocation details, which are fixed for a given environment. (In particular, they are independent of ASLR, which only randomizes high-order bits of addresses.) This means that running the same OpenSSL as the victim in the same environment is sufficient for the attacker to find suitable offsets for using in a later online attack.

To minimize noise, we prefer offsets that are exclusively used by one of the four variables (e.g., x_2). Figure A.13 shows traces of page offset stored to by `Madd0`, `Mdouble0`, `Madd1`, and `Mdouble1`, obtained using Intel's Pin tool [289]. The X axis is the order of stores inside the function and the Y axis is the page offset of each store. Stores to x_2 are highlighted with red circles. Since x_2 is a big number that occupies multiple double words, offsets of stores to x_2 form a continuous descent "slope" with a step of 8 bytes in the figure, instead of a single point. We find that x_2's offsets are good candidates for low-noise monitoring, as only `Madd0` and `Mdouble1` store to these offsets, and only when writing to x_2.

Step 2: Monitor Victim Stores. The attacker process is co-located on the same physical core as the victim, but on a sibling hyperthread. While the victim is signing, the attacker process keeps recording the latency of TLB-missing loads that monitor stores to the chosen page offsets, as well as the timestamp of each measurement. The latency trace is then saved for the next step.

Figure A.13 shows that for a given offset of x_2, only a few stores are executed in a short period during signing. To obtain a detectable signal in the store→load channel from these stores, we time four dependent TLB-missing loads instead of one. Since a single TLB-missing load can monitor two unique offsets (Section A.4.1), these four loads can monitor eight neighboring offsets of x_2 to increase the chance of contention. Also, because these four loads are dependent, their contention effects are built up and observable.

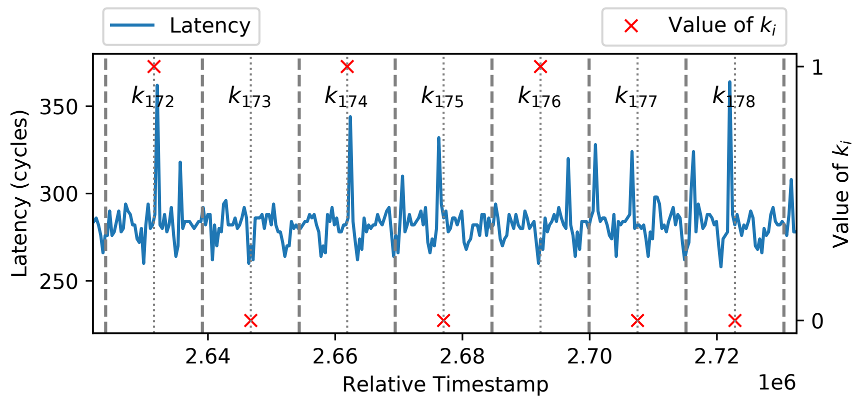


Figure A.14: A snippet of measured raw latencies. Grey vertical dashed lines indicate start of Montgomery ladder iterations. Grey vertical doted lines are the halves of iterations. Red crosses are the ground truth of k. Timestamps are relative to the first Montgomery ladder iteration. Measured on a Skylake-X.

Step 3: Process Signal. To recover the k_i values from the latency trace collected in step 2, we need to (1) identify when the victim stores to x_2; and (2) find boundaries of Montgomery ladder iterations, so that we can know whether the stores are performed in the first or second half of an iteration.

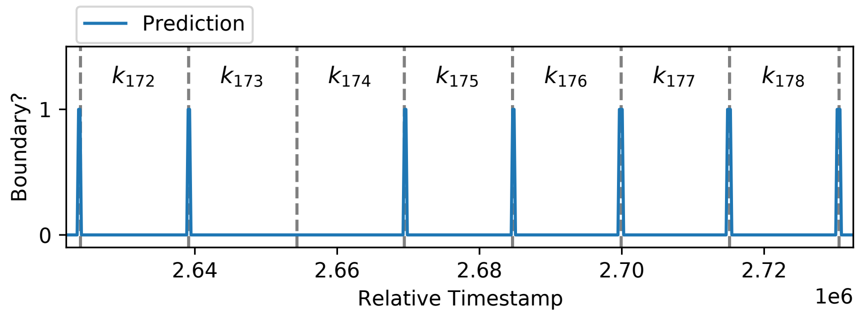


Figure A.15: Montgomery ladder iteration boundaries predicted by the classifier on the trace in Figure A.14. Timestamps are relative to the first Montgomery ladder iteration.

Figure A.14 shows a snippet of a latency trace collected on a Skylake-X, while the attacker monitors stores to x_2. We thus expect to see latency increases when the victim executes `Madd0` or `Mdouble1`. Red crosses show the k_i values. Iteration boundaries and halves are marked by vertical grey dashed lines and thin dotted lines, respectively. The lines are plotted by instrumenting the victim, *but in a real attack, they must be recovered by the attacker from the trace.* On average, the probing latency is around 300 cycles on the test machine.

Using the vertical lines as references, we see that when k_i is 0, there are two high latencies events in the first half of the Montgomery ladder iteration (bits k_{175}, k_{177}, and k_{178}). And when k_i is 1, high latency events occur in the second half of an iteration (bits k_{172}, k_{174}, and k_{176}).

Not all contention effects are obvious, however (e.g., bit k_{173}). We therefore use a supervised machine learning model, random forest classifier [195, 196], to predict iteration boundaries and k_i values from the latency trace. The following details how the model is used.

Preprocess. Because the attacker process keeps measuring latencies without any synchronization, the trace forms an unevenly spaced time series. We transfer the data into an equally spaced time series by resampling the raw data at a fixed period with linear interpolation. Since most machine learning models work best when the data under classification roughly follows a standard normal distribution, we normalize the resampled latencies by subtracting the mean and then dividing by the standard deviation.

Predict iteration boundaries. To recover Montgomery ladder iteration boundaries from the latency trace, we use a binary random forest classifier. Our classifier takes as input a vector of 160 normalized latencies and predicts whether the center of the vector is an iteration boundary.

Figure A.15 shows the classifier's outputs for the trace snippet in Figure A.14. The blue line is the classifier output on each timestamp and the grey vertical dashed lines are the ground truth of iteration boundaries. While the classifier manages to recover most boundaries, it sometimes misses a boundary (e.g., between k_{173} and k_{174}). We overcome this problem by, ironically, exploiting the Montgomery ladder's constant-time property. Because it executes the same sequence of operations regardless of k_i, the iteration length is relatively constant. We can therefore estimate the average iteration period from the predictions and use it to fix missing boundaries and remove false positives.

Predict k_i. Finally, we train another random forest classifier that takes as input a vector of normalized latencies from an iteration i, and predicts the value of k_i. For each prediction, the classifier also outputs a confidence score.

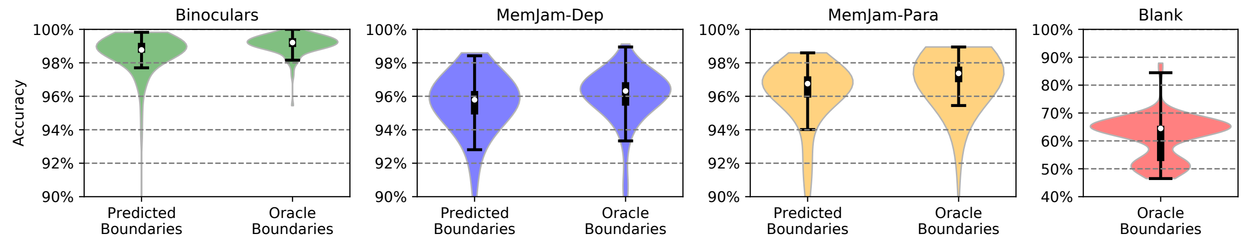


Figure A.16: Violin plots of each approach's accuracy distribution on a Skylake-X. Each distribution contains 100 predictions. Blank with predicted boundaries are omitted since the boundary predictor fails to output any meaningful data for it. All plots except Blank share the same Y axis range starting from 90%.

A.7.2 Results

Setup. We evaluate our attack method on a Skylake-X and a Cascade Lake-X with OpenSSL 1.0.1e on Ubuntu 20.04 LTS (5.4.0-105-generic). We use OpenSSL 1.0.1e strictly as a demonstration benchmark; after version 1.0.1e, OpenSSL switched to an invulnerable branchless Montgomery ladder implementation. We configure cores to run in performance mode without fixing their frequency. Cores for experiments are isolated to minimize context switches. We use the default compilation flags to compile OpenSSL. The curve that we are targeting is `sect571r1`, which uses a 571-bit nonce. We use the binary random forest classifier machine learning model from scikit-learn 1.0.2 [195] for signal processing.

We evaluate the attack's end-to-end accuracy with three other approaches for monitoring victim stores (Step 2) besides Binoculars, while keeping the rest of the steps same: (1) Blank: the attack process only measures time. This approach serves as a sanity check to show that the signal we observe is not caused by any resource contention on reading the timestamp. (2) MemJam-Para: this approach relies on false dependence in the L1D cache (Section A.2.2) that is described in MemJam [9]. Similar to the setup in MemJam, we measure the latency of eight parallel loads that are 4K-aliasing with a target store offset. (3) MemJam-Dep: this approach replaces eight parallel loads in MemJam-Para with four dependent loads to enhance the contention. (4) Binoculars (*this work*): this approach relies on the store→load channel.

For each approach, we collect 100 latency traces to train the first random forest classifier that predicts Montgomery ladder iteration boundaries. This step takes about 1 minute to collect and process the traces (about 500k training samples), and 10 minutes to train on a 16-core machine. Then, we use another 30 traces to train the second random forest classifier that predicts k_i. This step takes about 2 minutes to collect, process, and train on the same machine (about 17k training samples).

Finally, we evaluate the end-to-end accuracy of each method on another 100 traces and discuss the security implications (i.e., nonce recovery). Note that we do not include results on a Haswell-EP, because none of these four approaches can achieve good accuracy with a single victim run on it, mainly due to its low performance and thus weak contention effects.

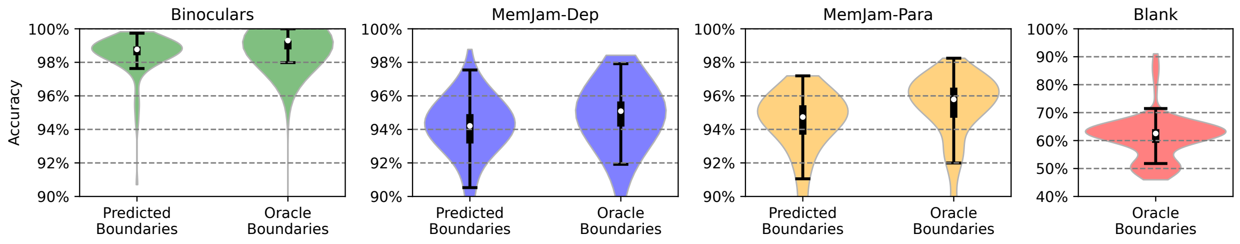


Figure A.17: Violin plots of each approach's accuracy distribution on a Cascade Lake-X.

Accuracy. Figures A.16 and A.17 show the accuracy distribution of each approach on different CPU platforms. For each approach, we show the results with iteration boundaries predicted by the boundary classifier and with oracle boundaries. The exception is BLANK, in which we only show results with oracle boundaries since the boundary classifier cannot output any meaningful prediction for its traces. For each distribution, the area represents the density of samples at a given accuracy. A wider area means the corresponding accuracy is more likely to occur. The thick short black bar represents the first to third quartile range. The thin long black bar represents the lower and upper bounds after filtering outliers with the quartile range. The white dot represents the median of the distribution.

BINOCULARS has the highest accuracies on both CPU platforms. On average, its accuracies are 98.5±0.3% on a Skylake-X and 98.4±0.3% on a Cascade Lake-X ($N = 100, P = 0.95$, same N and P value below), or with oracle boundaries, 99.1±0.1% on a Skylake-X and 98.7±0.8% on a Cascade Lake-X. Using oracle boundaries, sometimes BINOCULARS can even recover the full nonce without any error. Compared to BINOCULARS, other approaches have lower accuracy and higher deviation. MEMJAM-PARA, on average, can achieve 96.0 ± 0.6% on a Skylake-X and 94.1 ± 0.5% on a Cascade Lake-X. On average, MEMJAM-DEP's accuracies are 94.9±0.7% on a Skylake-X and 93.8±0.6% on a Cascade Lake-X. BLANK achieves accuracies that are only slightly better than random guessing (i.e., 50%), indicating—as expected—that just reading the timestamp cannot reveal k_i.

Nonce Recovery. To completely recover the full 571-bit-long nonce k, we need to find and correct erroneous bits in the predictions through brute forcing. Since we do not know how many bits are incorrect and where those bits are, we have to first guess the number of erroneous bits n_e, starting from 1, then try to flip all $C_{571}^{n_e}$ combinations. If no correct solutions are found, we will increment n_e and repeat the process.

To speed up the brute force search, we improve the method based on an observation that most erroneous bits have low confidence scores. In the improved method, we first sort all the predicted bits by their confidence scores in an ascending order. Then, we pick the top N_L low-confident bits and try to flip all $C_{N_L}^{n_e}$ combinations for a given n_e. Since N_L can be much smaller than the bit-length of k (i.e., 571), the search space is significantly reduced. Note that if the N_L low-confident bits fail to cover some erroneous bits, the brute force will fail. In that case, we will increase N_L and retry.

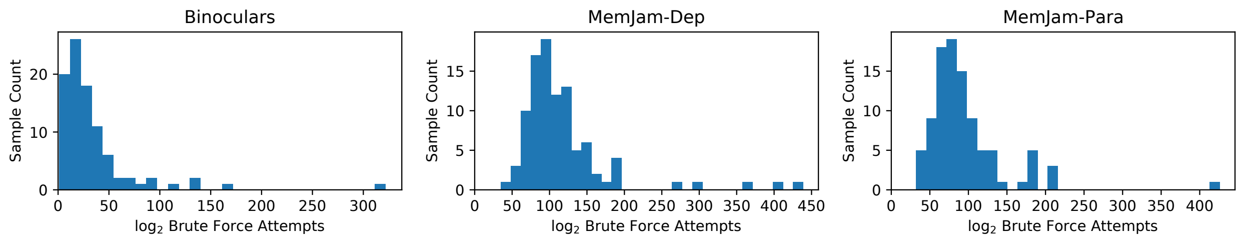


Figure A.18: Required $\log_2$ brute force attempts to recover the full nonce k with the improved method on Skylake-X (Based on k_i predictions with *predicted boundaries*).

Figures A.18 and A.19 show histograms of $\log_2$ brute force attempts with the improved method. These histograms are based on k_i predictions with predicted Montgomery ladder iteration boundaries. On a Skylake-X (Figure A.18), BINOCULARS requires a median of $2^{23.4}$ brute force attempts to recover k, which is feasible. If we assume an acceptable brute force attempts threshold at 2^{40}, BINOCULARS can succeed on 78.5% of traces. However, MEMJAM-DEP and MEMJAM-PARA require many more brute force attempts. Their median attempts are $2^{101.6}$ and $2^{82.0}$ respectively. With the same brute force threshold, they can only recover 1.0% and 3.1% of traces.

On a Cascade Lake-X (Figure A.19), every approach requires slightly more brute force attempts. From left to right, BINOCULARS requires $2^{24.7}$ median brute force attempts, and recovers k on 77.9% of traces with a brute force threshold of 2^{40}. While MEMJAM-DEP and MEMJAM-PARA require $2^{130.8}$ and $2^{132.5}$ median brute force attempts. Under the same brute force threshold, they can only recover 1.0% and 0.0% of traces.

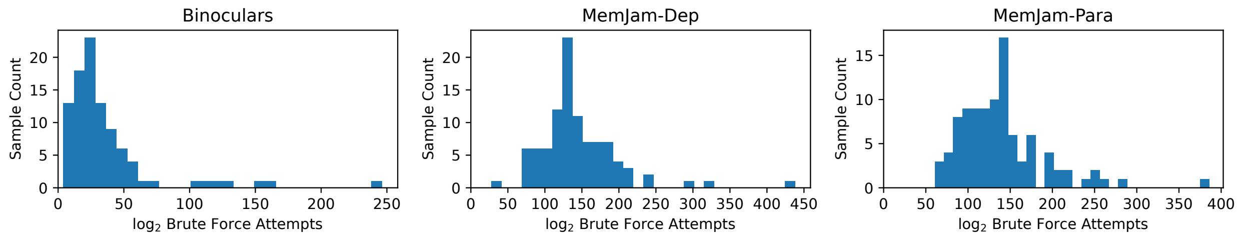


Figure A.19: Required $\log_2$ brute force attempts to recover the full nonce k with the improved method on Cascade Lake-X (Based on k_i predictions with *predicted boundaries*).

A.8 COMPROMISING KASLR

Linux uses kernel address space layout randomization (KASLR) to increase the difficulty of exploiting memory safety vulnerabilities. Linux randomizes the base addresses of several kernel memory regions at boot time. Since the possible kernel's address range is 1 GB (30 bits) and base addresses are aligned to 2 MB boundaries (21 bits), Linux's KASLR has 9 bits of entropy (512 choices) [290]. The address bits randomized are bits 29-21, i.e., the PL_2 index.

Attack Method. Using the load→store channel, an attacker can recover the full virtual page number (VPN) of a TLB-missing victim (kernel) memory load. Assuming the offset of the accessed page within its kernel segment is known, the base address of the segment can be derived, breaking KASLR.

We implement this idea by attacking a system call that accesses a global variable, whose offset within the kernel image is known for a given kernel build. We choose the `SYS_time` system call (similar to prior work [287]) which accesses the global variable `tk_core`. The attack is similar to the setup in Figure A.8, with the difference that the attacker measures the end-to-end execution time of the victim. The attacker runs two hyperthreads on the same physical core. The first hyperthread flushes the TLBs and measures the latency of calling `SYS_time`, while the second hyperthread keeps writing to each possible *PL* offset. Because the TLBs are flushed, the system call's read of `tk_core` will miss in TLBs and trigger a page walk. Consequently, the attacker will observe system call latency spikes at offsets that are 4K-aliasing with any PL index of `tk_core`. To minimize noise caused by irrelevant TLB-missing loads, the first thread makes an invalid system call to warm up the system call handler before calling `SYS_time`.

Binoculars is fundamentally different from most prior KASLR attacks [23, 290, 291, 292, 293, 294] which rely on monitoring microarchitectural side effects of accessing a mapped or unmapped address. Such attacks can be defeated by software mitigations like FLARE [290], which creates fake mappings for all possible kernel addresses, or kernel page-table isolation (KPTI), which unmaps most kernel pages in user space [295]. In contrast, Binoculars directly observes the kernel's TLB-missed memory loads, which allows the attacker to break KASLR even if KPTI or FLARE is deployed. Compared to prior work that also monitors victim's memory accesses [287], Binoculars can completely break KASLR thanks to its wide address bits coverage, in contrast to reducing entropy in [287].

Results. We use the same hardware set as in Section A.4, running Ubuntu 20.04 LTS (5.4.0-105-generic). On Haswell-EP and Skylake-X platforms, which are vulnerable to Meltdown, KPTI is enabled. We rely on `/proc/kallsyms` to collect the ground truth of the global variable `tk_core`'s address.

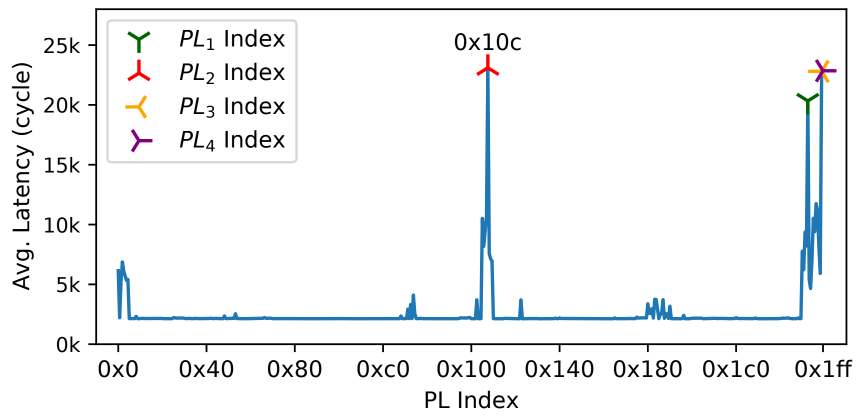


Figure A.20: Average latency of calling `SYS_time` when varying store offsets on a Skylake-X.

Figure A.20 shows the average latency of calling `SYS_time`, measured at each offset on a Skylake-X. The global variable `tk_core` is located at `0xffffffffa19f4f40` in this boot, which corresponds to indexes to PL_4, PL_3, PL_2, and PL_1 equal to `0x1ff`, `0x1fe`, `0x10c`, and `0x1f4`, respectively, which are marked in the figure. While the latencies are huge at these indexes, we do not see peaks as sharp as the ones in Figure A.8 due to TLB-missing accesses that are not to `tk_core`. We therefore run a peak detection algorithm.

To identify which peak corresponds to `tk_core`'s PL_2 index, we need to discard the PL_4, PL_3, and PL_1 indexes. We use the fact that the PL_4 and PL_3 indexes are the same constants in any Linux kernel image base address. Moreover, we can learn `tk_core`'s PL_1 index from its offset in the kernel image, which is known for a given kernel build (and readable in the image's symbol table).

To measure the accuracy of identifying the PL_2 index, we reboot the system 10 times, and recover the index 100 times per boot (1000 recoveries in total). We achieve accuracies of 100.0%, 98.7%, and 92.6% on the Skylake-X, Haswell-EP, and Cascade Lake-X, respectively. These accuracies, however, are of a single attacker run. They can be trivially improved to 100% on all three platforms by repeating the runs and picking the most frequent PL_2 index guess.

A.9 POTENTIAL MITIGATIONS

The root cause of the Binoculars attack is the starvation of hardware page walker loads by concurrent stores due to false dependence and/or set conflicts in the L1D cache. A complete fix of Binoculars thus requires hardware-level changes.

Software-wise, two mitigations can be applied. A system can disable hyperthreading, or only allow mutally trusted programs to share a physical core (e.g., core scheduling [100]). However, this mitigation can under-utilize hardware resources and lead to system performance degradation [296]. Alternatively, potential victims can be rewritten with data-oblivious programming practices [41, 42, 43], so that they do not make secret-dependent memory accesses. But while data-oblivious code is invulnerable to Binoculars and many other side channels, it usually requires a non-trivial amount of effort to rewrite and verify the program, and incurs significant execution overhead.

A.10 RELATED WORK

Attacks on Montgomery ladder. There have been several attacks on Montgomery ladder. Yarom et al. [189] extend Flush+Reload [6] to attack the same vulnerable implementation in Figure A.12. Compared to Binoculars, their attack requires the attacker to share memory with the victim, and has lower average accuracy (95.7%).

Brumley et al. [297] attack a different part of the OpenSSL implementation to recover the logarithm of k, which is then used in a lattice attack to recover the private key. Unlike Binoculars, this attack requires thousands of signatures and its success rate is low.

Exploiting page walker loads. Page walker loads go through the cache hierarchy and so can be observed with cache-based side channels. There have been side-channel attacks exploiting page walker loads to build stateful-indirect channels [96, 97, 298, 299]. Using these channels, the attack's granularity is limited to a cache line (64 bytes), which means that it cannot distinguish accesses to neighboring pages, whose 8-byte PTEs share the cache line with the target page. Binoculars is also capable of performing such monitoring with the load→store channel and has a finer granularity.

A.11 CONCLUSION

In this appendix, we investigated and demonstrated the first stateless-indirect channel by exploiting interactions between in-flight page walk loads on behalf of one thread and stores by another thread. We introduced a new side-channel attack called *Binoculars*. Unlike conventional stateless channels, Binoculars creates significant timing perturbations (e.g., up to $20,000$ cycles)—making it easy to monitor. We showed that the perturbations are address dependent, and designed two Binoculars attack primitives to leak a wide range of virtual address bits in victim memory operations. Using these primitives, we demonstrated end-to-end attacks on real hardware, which include extracting the nonce k in ECDSA with a single victim run, and fully breaking kernel ASLR.

APPENDIX B: Thwarting Microarchitectural Replay Attacks

B.1 INTRODUCTION

As shown in Chapter 4, a limitation of microarchitecural side channels is that they are often very noisy. To extract information, the execution of the attacker and the victim processes has to be carefully orchestrated [1, 2, 6], and often does not work as planned. Hence, the attacker needs to rely on many executions of the victim code section to obtain valuable information. Further, secrets in code sections that are executed only once or only a few times are hard to exfiltrate.

Unfortunately, a new type of attack called Microarchitectural Replay Attack (MRA) [280] is able to eliminate the measurement variation in (i.e., to denoise) most microarchitecural side channels. This is the case even if the victim runs only once. Such capability makes the plethora of existing side-channel attacks look formidable and suggests the potential for a new wave of powerful side-channel attacks.

MRAs use the fact that, in out-of-order cores, pipeline squashes due to events such as exceptions, branch mispredictions, and memory consistency model violations trigger the re-execution of dynamic instructions. Hence, in an MRA, the attacker repeatedly squashes one or more instructions to force the squash and re-execution of a younger victim instruction *V* many times. This ability enables the attacker to cleanly observe the side-effects of *V*.

MRAs are powerful because they exploit a central mechanism in modern processors: out-of-order execution with in-order retirement. Moreover, MRAs are not limited to transient execution attacks: the instruction *V* that is replayed can be a correct instruction that will eventually retire. Finally, MRAs come in many forms. While the first MRA [280] exposed the side effects of *V* by repeatedly causing a page fault on an older instruction, similar results can be attained with other events that trigger pipeline flushes.

To thwart MRAs, one has to eliminate instruction replay or at least bound the number of replays that a victim instruction *V* may suffer. The goal is to deny the attacker the opportunity to see many executions of *V*.

This appendix presents the first mechanism to thwart MRAs. We call it *Jamais Vu*. The simple idea is to record the instructions that are squashed. Then, when any of these instructions is re-inserted into the Reorder Buffer (ROB), *Jamais Vu* automatically places a fence before it to prevent the attacker from squashing the instruction execution again. In reality, pipeline refill after a squash may not bring in the same instructions that were squashed, or not in the same order. Consequently, *Jamais Vu* has to be carefully designed.

At a high level, there are two main design questions to answer: how to record the squashed instructions and for how long to keep the record of them. *Jamais Vu* presents several designs that give different answers to these questions, effectively providing different trade-offs between security, execution overhead, and implementation complexity.

Architectural simulations using SPEC17 applications show the effectiveness of the *Jamais Vu* designs. One design, called *Epoch-Loop-Rem*, effectively mitigates MRAs, has an average execution time overhead of 13.8% in benign executions, and only needs counting Bloom filters associated with the ROB. An even simpler design, called *Clear-on-Retire*, has an average execution time overhead of only 2.9%, although it is less secure.

The contributions of this appendix are as follows:

• *Jamais Vu*, the first defense mechanism to thwart MRAs. It selectively fences instructions to prevent replays.

• Several designs of *Jamais Vu* that provide different tradeoffs between security, execution overhead, and complexity.

• An evaluation of these designs using simulations.

```
inst_1
inst_2
...
transmit(x)
```

(a) Straight-line code where the attacker can cause exceptions.

```
if (cond_1) {...}
else {...}
if (cond_2) {...}
else {...}
...
transmit(x)
```

(b) Sequence of branches where the attacker can cause mispredictions.

```
//always false
if (i == expr)
  x = secret;
else
  x = 0;
transmit(x);
```

(c) Condition-dependent transmitter.

```
//always false
if (i == expr)
  transmit(x);
```

(d) Transient transmitter.

```
for i in 1..N
  //always false
  if (i == expr)
    x = secret;
  else
    x = 0;
  transmit(x);
```

(e) Condition-dependent transmitter in a loop with an iteration-independent secret.

```
for i in 1..N
  //always false
  if (i == expr)
    transmit(x);
```

(f) Transient transmitter in a loop with an iteration-independent secret.

```
for i in 1..N
  //always false
  if (i == expr)
    transmit(x[i]);
```

(g) Transient transmitter in a loop with an iteration-dependent secret.

Figure B.1: Code snippets where an attacker can use an MRA to denoise the address accessed by the *transmit* load.

B.2 BACKGROUND: MICROARCHITECTURAL REPLAY ATTACKS

A *Microarchitectural Replay Attack* (MRA) [280] uses one or more instructions to repeatedly trigger pipeline flushes, therefore forcing the re-execution of a younger instruction *I* multiple times. This capability enables the attacker to observe any side-effects of *I* multiple times, eliminating the measurement noise.

Skarlatos et al. [280] introduced MRAs by using a malicious Operating System (OS) to repeatedly trigger page faults on a memory access instruction in an SGX environment. Specifically, the OS picks a memory access instruction called *Replay Handle* that occurs shortly before a security-sensitive instruction *I*. The OS sets up the attack by flushing the TLB entry that stores the virtual-to-physical translation of the replay handle access, and clearing the Present bit of the corresponding page table entry. The OS allows the program to resume execution and execute the replay handle. A TLB miss occurs, followed by a page walk. The instructions following the replay handle, including *I*, execute in the shadow of the page walk, creating side effects: they leave some state in the cache subsystem or create contention for hardware structures in the core. This allows an attacker thread running in the system to perform a measurement of the secret data. At the end of the page walk, the hardware raises a page fault exception and squashes the instructions in the pipeline. The OS is then invoked to handle the page fault, but chooses to keep the Present bit cleared. The program then resumes and re-executes the replay handle, creating a TLB miss and page walk again. The instructions following the replay handle, including *I*, execute again until a pipeline flush occurs. This process is repeated as many times as desired until the attacker extracts the secret information.

MRAs are more general than the specific instantiation prototyped by Skarlatos et al. [280]. For example, there are multiple events that cause a pipeline flush, such as various exceptions, branch mispredictions, memory consistency model violations, and interrupts. Moreover, to trigger the repeated pipeline flushes, one does not need a privileged process. For example, it can be shown that memory consistency model violations triggered by a non-privileged process can also create MRAs [249].

In this appendix, we refer to the instruction that causes the pipeline flush as the *Squashing* (S) instruction; we refer to the younger instructions in the ROB that the Squashing one squashes as the *Victims* (V). The type of Victim instruction that the attacker wants to replay is one whose usage of and/or contention for a shared resource depends on a secret. We call such an instruction a *transmitter*. Loads are obvious transmitters, as they use the shared cache hierarchy. However, many instructions can be transmitters, including those that use functional units.

B.3 THWARTING MRAS

B.3.1 Understanding the Types of MRAs

MRAs come in many forms. Table B.1 shows three orthogonal characteristics that can help us understand these threats. The first one is the source of the squash. Recall that there are many sources, namely various exceptions (e.g., page faults [280]), branch mispredictions, memory consistency model violations, and interrupts [300]. With some sources, a single Squashing instruction can trigger pipeline flushes repeatedly, while with others, a Squashing instruction can only flush the pipeline a very limited number of times. Examples of the former are attacker-controlled page faults and memory consistency model violations; examples of the latter are branch mispredictions. The former can create more leakage.

Table B.1: Characteristics of microarchitectural replay attacks.

Characteristic	**Why It Matters**
Source of squash?	Determines: (i) the number of pipeline flushes and (ii) where in the ROB the flush occurs
Victim is transient?	If yes, it can leak a wider variety of secrets
Victim is in a loop accessing the same secret every iteration?	If yes, it is harder to defend: (i) leaks from multiple iterations add up (ii) *multi-instance* squashes

Moreover, some sources trigger the flush when the Squashing instruction is at the ROB head, while others can do it at any position in the ROB. The former include exceptions, while the latter include branch mispredictions and memory consistency violations. The former create more leakage because they typically squash and replay more Victims.

Figure B.1(a) shows an example where repeated exceptions on one or more instructions *inst_i* can squash and replay a transmitter many times. This is one of the examples used in [280]. Figure B.1(b) shows an example where attacker-instigated mispredictions in multiple branches can result in the repeated squash and replay of a transmitter. Different branch structures and different orders of branch resolution result in different replay counts.

The second characteristic in Table B.1 is whether the Victim is *transient*. Transient instructions are speculatively-executed dynamic instructions that will not commit. MRAs can target both transient and non-transient instructions. Transient Victims are more concerning: since the programmer and compiler do not expect their execution, they can leak a wider variety of secrets.

Figure B.1(d) shows an example where an MRA can attack a transient instruction through branch misprediction. The transmitter should never execute, but the attacker trains the branch predictor so that it does. Figure B.1(c) shows a related example. The transmitter should not execute using the secret, but the attacker trains the branch predictor so that it does.

The third characteristic in Table B.1 is whether the Victim is in a loop accessing the same secret in every iteration. If it is, MRAs are more effective for two reasons. First, the attacker has more opportunities to force the re-execution of the transmitter and leak the secret. Second, since the loop is dynamically unrolled in the ROB during execution, the ROB may contain multiple instances of the transmitter, already leaking the secret multiple times. Only when a squash occurs will any MRA defense engage. We call a squash that squashes multiple transmitter instances leaking the same secret in an unrolled loop a *multi-instance* squash.

Figures B.1(e) and (f) are like (c) and (d), but with the transmitter in a loop. In these cases, the attacker can create more leaks of the transmitter by training the branch predictor so these branches mispredict *in every iteration*. In the worst case, the branch in the first iteration resolves after K loop iterations are loaded into the ROB and have executed. By the time the multi-instance squash occurs, x has been leaked as many as K times. Only then is the MRA defense engaged.

Figure B.1(g) is like (f) except that the transmitter leaks a different secret every iteration. In this case, it is easier to minimize the leakage.

B.3.2 Our Approach to Thwarting MRAs

To see how to thwart MRAs, consider Figure B.2(a), where a Squashing instruction S causes the squash of all the younger instructions in the ROB (Victims V_0 ... V_n). The idea is to detect this event and record all the Victim instructions. Then, as the Victim instructions are re-inserted into the ROB, precede each of them with a fence. We want to prevent the re-execution of each V_i until V_i cannot be squashed anymore. In this way, the attacker cannot observe the side effects of V_i more than once. The point when V_i cannot be squashed anymore is (i) when V_i is at the head of the ROB, or (ii) when no older instruction than V_i in the ROB or any other event (e.g., a memory consistency violation) can squash V_i. This point has been called the Visibility Point (VP) of V_i [118].

For highest performance, the type of fence used should be one that only prevents the execution of the V_i instruction, where V_i can be any type of transmitter instruction. Further, when V_i reaches its VP, the fence should be automatically disabled by the hardware, so that V_i can execute.

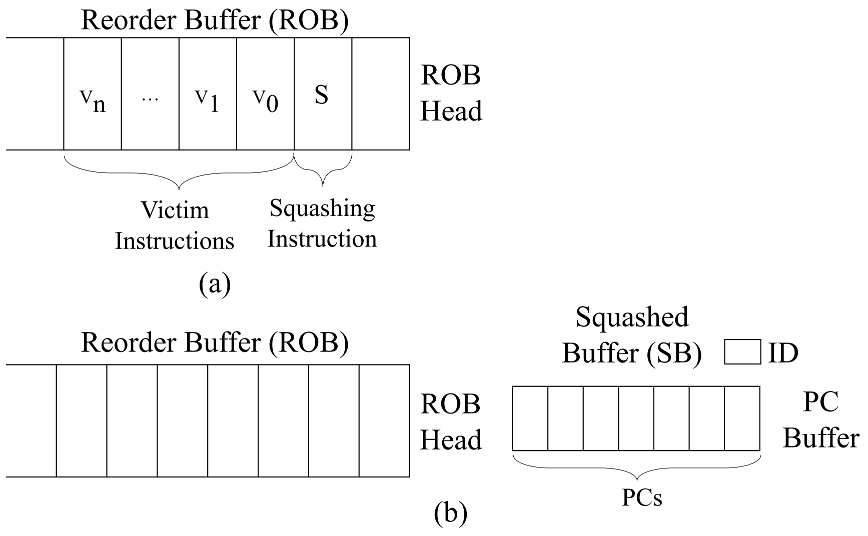


Figure B.2: Reorder Buffer (ROB) and Squashed Buffer (SB).

In this approach, there are two main decisions to make: (i) on a squash, how to record the Victim instructions? and (ii) for how long to keep this information? As a straw-man, consider a squash triggered by an exception in *inst_1* of Figure B.1(a). Before the squash, several dynamic instructions *V* younger than *inst_1* may have already partially executed speculatively. As the program re-starts after the squash, the processor will re-execute these same *V* instructions, in the exact same order. Hence, our defense can be as follows. When the squash occurs, we record the *V* dynamic instructions in a list, in program order; then, as each V_i in *V* is about to re-execute, we precede it with a fence. When V_i reaches its VP, we remove V_i from the list. Once the list becomes empty, we can resume normal, fence-free execution.

In reality, most squashes are more complicated, especially when we have branches (Figure B.1(b)) or execute transient instructions (Figure B.1(d)). In these cases, program re-start may not result in the re-execution of all the recorded Victims, or perhaps not in the same order as the first time. Moreover, when we have loops such as in Figure B.1(e), the list of Victims of a squash may include multiple dynamic instances of the same static instruction—each from a different loop iteration—possibly leaking the same secret. Consequently, we will need more elaborate designs.

Finally, no amount of fencing can prevent the repeated re-execution of the Squashing instruction when such instruction is squashed during its execution. An example is the replay handle in Skarlatos *et al.* [280], which is forced to suffer repeated page faults. Hence, we suggest handling an attack on these Squashing instructions themselves differently. Specifically, the hardware should not allow a dynamic instruction to trigger more than a very small number of repeated pipeline flushes before raising an attack alarm.

B.4 THREAT MODEL

We consider supervisor- and user-level attackers. In both cases, we assume the attacker can monitor a microarchitectural side channel (e.g., those in Section 2.2). This is easily realized when the attacker has supervisor-level privileges, as in the original MRA paper for the SGX setting [280]. It is also possible, subject to OS scheduler assumptions, when the attacker runs unprivileged code [301]. In addition, we assume that the attacker can trigger squashes in the victim program to perform MRAs. Which squashes are possible depends on the attacker. In the supervisor-level setting, the attacker can trigger squashes due to exceptions such as page faults, or due to branch mispredictions by priming the branch predictor state. In the user-level setting, the attacker has more limited capabilities. For example, it may be able to trigger branch mispredictions by priming the branch predictor state [18] but cannot cause exceptions.

B.5 PROPOSED DEFENSE SCHEMES

B.5.1 Outline of the Schemes

A highly secure defense against MRAs would keep a fine-grain record of all the dynamic instructions that were squashed. When one of these instructions would later attempt to re-execute, the hardware would fence it and, when it reached the VP, remove it from the record. In reality, such a scheme is not practical due to the potentially large storage requirements and the difficulty of identifying the same dynamic instruction. Hence, *Jamais Vu* proposes three classes of schemes that discard this state early. The schemes differ on when and how they discard the state.

A scheme called *Clear-on-Retire* discards any Victim information as soon as the program makes forward progress—i.e., when the Squashing instruction reaches its VP (and hence will retire). A scheme called *Epoch* discards the state when the current "execution locality" or *epoch* terminates, and execution moves to another one. Finally, a scheme called *Counter* keeps the state forever, but it compresses it so that all dynamic instances of the same static instruction keep their state merged. Each of these policies to discard or compress state creates a different attack surface.

B.5.2 Clear-on-Retire Scheme

The rationale for the simple *Clear-on-Retire* scheme is that an MRA leaks information by stalling a program's forward progress and repeatedly re-executing the same set of instructions.

Hence, when an MRA defense manages to force forward progress, it is appropriate to discard the record of Victim instructions. Therefore, *Clear-on-Retire* clears the Victim state when the Squashing instruction reaches its VP.

Clear-on-Retire stores information about the Victim instructions in a buffer associated with the ROB called the *Squashed Buffer (SB)*. Figure B.2(b) shows a conceptual view of the SB. It is composed of a *PC Buffer* and an identifier register (*ID*). The PC Buffer contains the set of program counters (PCs) of the Victim instructions. Since a squash may discard multiple iterations of a loop in the ROB, the SB may contain the same PC multiple times. The ID register contains information that identifies the Squashing instruction—i.e., the one that caused the squash. Such information includes the PC of the instruction and its position in the ROB.

Multiple instructions in the ROB may cause squashes, in any order. For example, in Figure B.1(b), the branch in Line 3 may cause a squash first, and then the branch in Line 1 may cause a squash. At every squash, the Victims' PCs are added to the PC Buffer. However, ID is only updated if the Squashing instruction is *older* than the one currently in ID. This is because the older instruction will retire first and hence its retirement is needed to make forward progress.

The *Clear-on-Retire* algorithm works as follows. On a squash, the PCs of the Victims are added to the PC Buffer, and ID is updated if necessary. When trying to insert an instruction *I* in the ROB, if *I* is in the PC Buffer, a fence is placed before *I*. When the instruction in ID reaches its VP, since the program is making forward progress, the SB is cleared and all the fences introduced by *Clear-on-Retire* are nullified.

To understand why ID needs to store both the Squashing instruction's PC and its ROB index, note that there are two types of Squashing instructions. One type, such as mispredicted branches, remain in the ROB after they trigger a squash; the other type, such as instructions suffering an exception or loads suffering a memory consistency violation, are removed from the ROB after they trigger a squash. For the first type, *Clear-on-Retire* does not use the PC field in ID; it only uses the ROB index in ID to determine the relative age of any two Squashing instructions. For the second type, since the instruction is removed from the ROB, the ROB index in ID is meaningless. Hence, *Clear-on-Retire* uses the PC in ID to identify the Squashing instruction when it is re-inserted into the ROB. At that point, *Clear-on-Retire* saves into ID the instruction's new ROB index.

The first row of Table B.2 describes *Clear-on-Retire*. The scheme is simple and has the most inexpensive hardware. The SB can be implemented as a simple Bloom filter (Section B.6.1).

One shortcoming of *Clear-on-Retire* is that it has some unfavorable security scenarios. Specifically, the attacker could choose to make *slow* forward progress toward the transmitter *I*, forcing every single instruction encountered to be a Squashing one.

In practice, this scenario may be hard to set up since, for maximum effectiveness, the squashes have to occur in strict order, from older to younger predecessor of I. Indeed, if a Squashing instruction S_1 squashes I, and I is then re-inserted into the ROB with a fence, a second Squashing instruction S_2 older than S_1 will not squash I's execution again. The reason is that I is fenced and has not yet executed.

Table B.2: Proposed defense schemes against microarchitectural replay attacks.

Scheme	**Removal Policy**	**Rationale**	**Pros/Cons**
Clear-on-Retire	When the Squashing instruction reaches its visibility point (VP)	The program makes forward progress when the Squashing instruction reaches its VP	+ Simple scheme + Most inexpensive hardware - Some unfavorable security scenarios
Epoch	When an epoch completes	An epoch captures an execution locality	+ Inexpensive hardware + High security if epoch chosen well - Need compiler support
Counter	No removal, but information is compacted	Keeping the difference between squashes and retirements low minimizes leakage beyond natural program leakage	+ Conceptually simple - Intrusive hardware - May require OS changes - Some pathological patterns

B.5.3 Epoch Scheme

The rationale for the *Epoch* scheme is that an MRA attacks an "execution locality" of a program, which has a certain combination of Victim instructions. Once program execution moves to another locality, the re-execution (and squash) of some of the original Victims is not seen as dangerous. Hence, it is appropriate to discard the record of Victim instructions from a locality when moving to another locality. We refer to an execution locality as an *Epoch*. Possible epochs are a loop iteration, a whole loop, or a subroutine.

Like *Clear-on-Retire*, *Epoch* uses an SB to store information about the Victim instructions. However, the design is a bit different. First, *Epoch* requires the hardware to find *start-of-epoch* markers as it inserts instructions into the ROB. We envision that such markers are added by the compiler. Second, the SB needs one {ID, PC-Buffer} pair for each in-progress epoch. The ID now stores a small-sized, monotonically-increasing epoch identifier; the PC Buffer stores the PCs of the Victims squashed in that particular epoch.

The *Epoch* algorithm works as follows. As instructions are inserted into the ROB, the hardware records every start-of-epoch marker. On a squash, the Victim PCs are stored in different PC Buffers depending on the epoch they belong to. The IDs of the PC Buffers are set to the corresponding epoch IDs. Note that a given PC may be in multiple PC Buffers and even multiple times in the same PC Buffer. Then, when trying to insert an instruction *I* in the ROB, if *I* is in the PC Buffer of the current epoch, *I* is fenced. Finally, when the first instruction of an epoch reaches its VP, the hardware clears the {ID, PC-Buffer} of any *older* epoch.

When a program re-starts after a squash, the first instruction re-enters the ROB with the same epoch ID as that of the oldest squashed instruction. For example, suppose that instruction *I* of epoch *i* suffers a page fault while younger instructions from epochs *i+1* and *i+2* are also in the ROB. The hardware flushes *I* and all subsequent instructions. After the page fault is repaired, *I* re-enters the pipeline as belonging to epoch *i*, not epoch *i+3*. Effectively, *Epoch* resets the epoch ID to the point of the squash.

Epoch protects the scenario where, after the squash, the re-execution exercises the same set of epochs that were executed speculatively before the squash and left Victim instructions in the PC Buffers—although, perhaps, the re-execution executes different instructions than before in such epochs. However, *Epoch* does not target the case when, after the squash, the re-execution exercises a different set of epochs: e.g., when, because of a branch misprediction, a subroutine is now called that was not called before, or a loop that was initially executed is now not executed anymore. In these cases, we consider that the re-execution has moved to different localities and, therefore, *Epoch* does not need to match the new instructions with the older Victims.

The second row of Table B.2 describes *Epoch*. The scheme is also simple and has inexpensive hardware. It can also implement the PC Buffers as Bloom filters. *Epoch* has high security if epochs are chosen appropriately, as the Victim information remains for the whole duration of the epoch. A drawback of *Epoch* is that it needs compiler support.

An epoch can be long, in which case its PC Buffer may contain too many PCs to operate efficiently. Hence, our preferred implementation of this scheme is a variation of *Epoch* called *Epoch-Rem* that admits PC removal. Specifically, when a Victim from an epoch reaches its VP, the hardware removes its PC from the corresponding PC Buffer. This support reduces the pressure on the PC Buffer. This functionality is supported by implementing the PC Buffers as *counting* Bloom filters (Section B.6.2).

B.5.4 Counter Scheme

The *Counter* scheme never discards information about Victim squashes. However, to be implementable, the scheme merges the squash information from all the dynamic instances of the same static instruction into a single variable. Specifically, *Counter* records, for any given static instruction, the difference between the number of times it has been squashed and the number of times it has retired. *Counter*'s goal is to keep such difference small. The rationale is that, if both counts are similar, an MRA is unlikely to exfiltrate much more information than what the program naturally leaks.

While *Counter* can be implemented like the two previous schemes, a more intuitive implementation associates Victim information with each static instruction. A simple design adds a Squashed bit to each static instruction *I*. When *I* gets squashed, its Squashed bit is set. From then on, an attempt to insert *I* in the ROB causes a fence to be placed before *I*. When *I* reaches its VP, the bit is reset. After that, a future invocation of *I* is allowed to execute with no fence.

In reality, multiple dynamic instances of the same static instruction may be in the ROB at the same time and get squashed together. Hence, we use a *Squashed Counter* per static instruction rather than a bit. The algorithm works as follows. Every time that dynamic instances of the instruction get squashed, the counter increases by the number of squashed instances; every time that an instance reaches its VP, the counter is decremented by one. The counter does not go below zero. Finally, when an instruction is inserted in the ROB, if its counter is not zero, the hardware fences it. This is the *Counter* scheme that we propose.

To reduce the number of stalls, a variation of this scheme allows a Victim to execute without a fence as long as its counter is lower than a threshold.

The third row of Table B.2 describes *Counter*. The scheme is conceptually simple. However, it requires somewhat intrusive hardware. One possible design requires counters that are stored in memory and get cached on demand into a special cache next to the L1 (Section B.6.3). This counter cache or the memory needs to be updated every time a counter changes. In addition, the OS needs changes to allocate and manage pages of counters for the instructions.

Counter has some pathological patterns. Specifically, an attacker may be able to repeatedly squash an instruction by interleaving the squashes with retirements of the same static instruction. In this case, one access leaks a secret before being squashed, while the other access is benign, retires, and decreases the counter. This pattern is shown in Figure B.1(e). In every iteration, the branch predictor incorrectly predicts the condition to be true, x is set to *secret*, and the transmitter leaks x. The execution is immediately squashed, the *else* code executes, and the transmitter retires. This process is repeated in every iteration, causing the counter to toggle between one and zero.

B.5.5 Analysis of the Security of the Schemes

To assess the relative security of the schemes, we compare their worst-case leakage for each of the code snippets in Figure B.1. While the snippets in Figure B.1 only show some of the possible patterns, they cover a broad spectrum of cases. Indeed, they show examples of transmitters in straight-line code and in loops; replays due to exceptions (Figure B.1(a)) and branch mispredictions; transmitters executed transiently (e.g., Figure B.1(d)) and non-transiently; and transmitters with iteration-independent and iteration-dependent secrets.

A summary of the analysis is shown in Table B.3. We measure leakage as the number of executions of the transmitter for a given secret. We report Transient Leakage (TL) when the transmitter is a transient instruction and Non-Transient Leakage (NTL) when it is not. For the *Epoch* scheme, we show the leakage for one design that uses iterations as epochs (*Iter*) and for one that uses loops as epochs (*Loop*). For each of these two designs, we consider an implementation without removal of Victim PCs from the PC Buffers when they reach their VP (*NR*) and with removal of them (*R*).

In Figure B.1(a), since the transmitter should commit, the NTL is one. The TL is found as follows. In *Clear-on-Retire*, the attacker could make each instruction older than the transmitter a Squashing one. In the very worst case, the squashes occur in program order, and the timing is such that the transmitter is squashed as many times as the ROB size minus one. Hence TL is ROB size minus 1. While this is a large number, it is smaller than the leakage in the original MicroScope attack [280], where TL is infinite because one instruction can cause any number of squashes. In all *Epoch* designs, the transmitter is squashed only once. Hence, TL is 1. *Counter* sets the transmitter's counter to 1 on the first squash; no other speculative re-execution is allowed. Hence, TL is 1.

Figure B.1(b) is conceptually like (a). The NTL in all schemes is 1. The TL of *Counter* and *Epoch* is 1. In *Clear-on-Retire*, in the worst-case where all the branches are mispredicted and resolve in program order, the TL is equal to the number of branches that fit in the ROB minus one slot (for the transmitter).

Figures B.1(c) and (d) are very simple examples. NTL is 0 (since in Figure B.1(c) x is never set to the secret in a non-speculative execution) and TL is 1 for all schemes.

In Figure B.1(e), NTL is zero. However, the attacker may cause the branch to be mispredicted in every iteration. To assess the worst-case TL in *Clear-on-Retire*, assume that, as the N-iteration loop dynamically unrolls in the ROB, K iterations fit in the ROB. In this case, the worst-case is that each iteration (beyond the first $K-1$ ones) is squashed K times. Hence, TL in *Clear-on-Retire* is $K * N$. In *Epoch* with iteration, since each epoch allows one squash, the TL is N (with and without PC removal). In *Epoch* with loop without removal, in the worst case, the initial K iterations are in the ROB when the squash occurs, and we have a *multi-instance* squash (Section B.3.1).

Hence, the TL is K. In *Epoch* with loop with removal, since every retirement of the transmitter removes the transmitter PC from the SB, TL is N. Finally, in *Counter*, since every iteration triggers a squash and then a retirement, TL is N.

Figure B.1(f) is like Figure B.1(e), except that the transmit instruction never retires for any value of x. As a consequence, *Epoch* with loop with removal does not remove it from the SB, and *Counter* does not decrease the counter. Hence, their TL is K.

Finally, Figure B.1(g) is like B.1(f) except that each iteration accesses a different secret. The NTL is zero. The TL for *Clear-on-Retire* is K because of the dynamic unrolling of iterations in the ROB. For the other schemes, TL is 1 in the worst case.

Overall, for the examples shown in Table B.3, *Epoch* at the right granularity (i.e., loop level) without removal has the lowest leakage. With removal, the scheme is similar to *Counter*, and better than *Epoch* with iteration. *Clear-on-Retire* has the highest worse-case leakage. Further analysis with more code patterns is part of our future work, and will provide more insights.

Table B.3: Worst-case leakage count in the proposed defense schemes for some of the examples in Figure B.1. For a loop, N is the number of iterations and, as the loop dynamically unrolls in the ROB, K is the number of iterations that fit in the ROB.

Case	**Non-Transient Leakage (NTL)**	**Transient Leakage (TL)**					
		Clear-on -Retire	**Epoch**				**Cntr**
			Iter		**Loop**		
			NR	R	NR	R	
(a)	1	ROB-1	1				1
(b)	1	BR_{ROB}-1	1				1
(c),(d)	0	1	1	1	1	1	1
(e)	0	K*N	N	N	K	N	N
(f)	0	K*N	N	N	K	K	K
(g)	0	K	1	1	1	1	1

B.5.6 A Statistical Model for Security Analysis

This subsection analyzes the implications of the leakage bounds in Table B.3 on the security of a system. We consider the MRA prototyped by MicroScope [280], where a victim program performs two multiplications or two divisions based on a test on a secret value. The attacker forces the victim to continuously replay the operations, while a monitor thread keeps performing division operations, and recording what fraction of the divisions take longer than a certain threshold latency. The authors found that, if the victim is performing divisions, the monitor sees 64 divisions with over-the-threshold latency in 10000 samples; if the victim is performing multiplications, the monitor sees 4 divisions with over-the-threshold latency in 10000 samples.

Based on this prototype, we model an MRA environment as follows. The attacker observes X operations with over-the-threshold latency in N samples. X follows a binomial distribution. When the secret is 0, the probability of observing an over-the-threshold operation is P_0, thus $X \sim Bin(N, P_0)$. When the secret is 1, the probability is P_1, thus $X \sim Bin(N, P_1)$. Based on the MicroScope prototype, we use $P_0 = 4/10000$ and $P_1 = 64/10000$.

During an attack, the attacker can have two hypotheses:

1. H_0: the secret is 0, i.e., $X \sim Bin(N, P_0)$.

2. H_1: the secret is 1, i.e., $X \sim Bin(N, P_1)$.

To test which one of H_0 and H_1 to accept, the attacker runs the Uniformly Most Powerful (UMP) test [302] with a single cut-off C. If the attacker measures $X/N < C$, she accepts H_0 and predicts that the secret is 0; if the attacker measures $X/N > C$, she accepts H_1 and predicts that the secret is 1. There are four possible outcomes:

Table B.4: The probability of each test outcome.

Prediction / Truth	$secret = 0$
$secret = 0$	$P(correct \vert s = 0) = \sum_{x/N<C} \binom{N}{x} P_0^x (1 - P_0)^{N-x}$
$secret = 1$	$P(incorrect \vert s = 1) = \sum_{x/N<C} \binom{N}{x} P_1^x (1 - P_1)^{N-x}$

Prediction / Truth	$secret = 1$
$secret = 0$	$P(incorrect \vert s = 0) = \sum_{x/N>C} \binom{N}{x} P_0^x (1 - P_0)^{N-x}$
$secret = 1$	$P(correct \vert s = 1) = \sum_{x/N>C} \binom{N}{x} P_1^x (1 - P_1)^{N-x}$

- True secret s is 0:

  1. The attacker correctly predicts 0 with a probability
     $P(correct|s = 0)$.

  2. The attacker incorrectly predicts 1 with a probability
     $P(incorrect|s = 0)$.

- True secret s is 1:

  3. The attacker correctly predicts 1 with a probability
     $P(correct|s = 1)$.

4. The attacker incorrectly predicts 0 with a probability $P(incorrect|s = 1)$.

Among the four possible outcomes, the first and third cases result in a correct prediction, while the second and fourth cases result in an incorrect prediction. Table B.4 shows the probability of each outcome.

To determine an optimal cut-off C, we calculate the likelihood ratio and require it to be 1:

$$\text{Likelihood ratio} = \frac{L(H_0)}{L(H_1)} = \frac{\binom{N}{C} P_0^C (1-P_0)^{N-C}}{\binom{N}{C} P_1^C (1-P_1)^{N-C}} = 1$$

After canceling the common parts of the numerator and denominator:

$$\left[\frac{P_0(1-P_1)}{P_1(1-P_0)}\right]^C \left[\frac{(1-P_0)}{(1-P_1)}\right]^N = 1$$

then applying ln to both sides:

$$C\ ln\left[\frac{P_0(1-P_1)}{P_1(1-P_0)}\right] + N\ ln\left[\frac{(1-P_0)}{(1-P_1)}\right] = 0$$

finally:

$$C = -\frac{ln\left[\frac{(1-P_0)}{(1-P_1)}\right]}{ln\left[\frac{P_0(1-P_1)}{P_1(1-P_0)}\right]} N$$

Using the values of $P_0 = 4/10000$ and $P_1 = 64/10000$ from the MicroScope experiment, we obtain $C = 21.67N/10000$. This is an optimal choice for the cut-off.

If the attacker wants to exfiltrate the secret bit with more than 80% success rate, each of the probabilities of correct outcomes, namely $P(correct|s = 0)$ and $P(correct|s = 1)$, need to be greater than 80%. By solving the equations of $P(correct|s = 0) > 0.8$ and $P(correct|s = 1) > 0.8$ in Table B.4 for $C = 21.67N/10000$, we find that N needs to be $N >= 251$. This means that the attacker *needs at least 251 replays* to extract a single bit with 80% success rate. If the attacker wants to exfiltrate an entire byte with 80% success rate, then she needs $\sqrt[8]{80\%} = 97.2\%$ success rate on extracting every single bit. In our case, this means that she *requires at least 1107 replays for each bit* extraction and 8856 replays in total. The longer the secret is, the more the replays required are.

These replay counts are higher than the *very worst* leakage counts of the *Jamais Vu* schemes in Table B.3. It is true that, in the cases of loops (Rows (e) and (f) in the table), the number of iterations N of the loop may be large. However, these leakage counts require that all the loop iterations read from *the same location*, which is very rare given loop-invariant code-motion compiler optimizations. Furthermore, the values of aforementioned probabilities P_0 and P_1 from MicroScope [280] were obtained by re-executing the same set of instructions with *the same replay handle*. *Jamais Vu*, instead, forces the attacker to continuously change replay handle. Hence, the attack's success rate will be even smaller.

Overall, from this estimation, we conclude that the leakage bounds provided by our proposed *Jamais Vu* schemes make the schemes reasonably secure. Without *Jamais Vu*, the attacker can extract a secret that has an arbitrary length with 100% success rate [280].

B.6 MICROARCHITECTURAL DESIGN

B.6.1 Implementing *Clear-on-Retire*

The PC Buffer in the SB needs to support three operations. First, on a squash, the PCs of all the Victims are inserted in the PC Buffer. Second, before an instruction is inserted in the ROB, the PC Buffer is queried to see if it contains the instruction's PC. Third, when the instruction in the ID reaches its VP, the PC Buffer is cleared.

These operations are easily supported with a hardware Bloom filter [303]. Figure B.3 shows the filter's structure. It is an array of M entries, each with a single bit. To insert an item in the filter (BF), the instruction's PC is hashed with n hash functions (H_i) and n bits get set: $BF[H_1]$, $BF[H_2]$, ... $BF[H_n]$. The filter can be implemented as an n-port direct-mapped cache of M 1-bit entries.

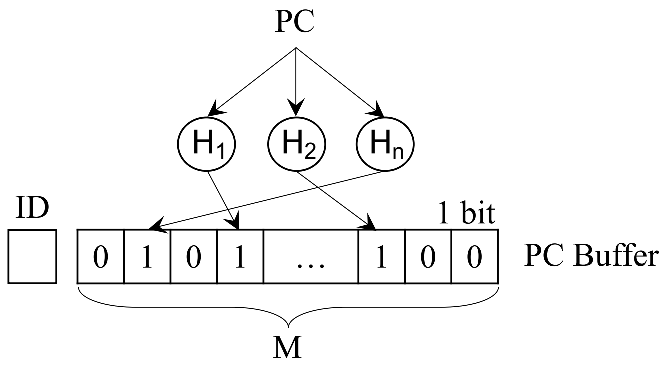


Figure B.3: SB with a PC Buffer organized as a Bloom filter.

A Bloom filter can have false positives but no false negatives. A false positive occurs when a PC is not in the PC Buffer but it is deemed to be there due to a conflict. This situation is safe, as it means that *Clear-on-Retire* will end up putting a fence before an instruction that does not need it.

In practice, if we size the filter appropriately, we do not see many false positives when running benign programs. Specifically, as we will see in Section B.9.3, for a 192-entry ROB, a filter with 1232 bits and 7 hash functions has less than 0.5% false positives.

B.6.2 Implementing *Epoch*

The SB for *Epoch* is like the one for *Clear-on-Retire* with two differences. First, there are multiple {ID, PC-Buffer} pairs—one for each in-progress epoch. Second, in *Epoch-Rem*, which supports the removal of individual PCs from a PC Buffer, each PC Buffer is a counting Bloom filter [304].

Figure B.4 shows the SB with multiple counting Bloom filters. The latter are like plain filters except that each entry has k bits. To insert an item in a filter, the n entries selected by the hashes are incremented by one—i.e., $BF[H_1]$++, $BF[H_2]$++, ... $BF[H_n]$++. To remove the item, the same entries are decremented by one. An n-port direct-mapped cache of M k-bit entries is used.

A counting Bloom filter can suffer false positives which, in our case, are harmless. In addition, it can also suffer false negatives. A false negative occurs when a Victim should be in the PC Buffer but it is deemed not to. In *Jamais Vu*, they are caused in two ways. The first one is when a non-Victim instruction NV to be inserted in the ROB is incorrectly believed to be in the filter because it conflicts with existing entries in the filter. Later, when NV reaches its VP, it causes the removal of state from Victim instructions from the filter. After that, when Victims are checked for membership, they are not found, triggering a false negative.

The second case is when the filter does not have enough bits per entry and, as a new Victim is inserted, an entry saturates. In this case, information is lost. Later, Victim state that should be in the filter will not be found in the filter.

False negatives reduce security because no fence is inserted where there should be one. However, by appropriately sizing the Bloom filter relative to the ROB size, we can reduce the upper bound of false negatives [305]. In practice, as we will see in Section B.9.3, because each counting Bloom filter only contains Victims from one epoch, we find that only 0.02% and 0.006% of the accesses are false negatives in *Epoch* with loops and iterations, respectively.

Note that an attacker cannot explicitly cause hashed addresses to bunch-up into a few Bloom-filter entries and saturate them. The reason is that the attacker does not control how the Victim instructions following a squash scatter into the Bloom filter.

Handling Epoch Overflow. The SB has a limited number of {ID, PC-Buffer} pairs. Therefore, it is possible that, on a squash, the Victim instructions belong to more epochs than PC Buffers exist in the SB. For this reason, *Epoch* augments the SB with one extra ID not associated with any PC Buffer called *OverflowID*. To understand how it works, recall that epoch IDs are monotonically increasing. Hence, we may find that Victims from a set of high-numbered epochs have no PC Buffer to go. In this case, we store the ID of the highest-numbered epoch of any Victim in *OverflowID*. From now on, when a new instruction is inserted in the ROB, if it belongs to an epoch whose ID: (i) owns no PC Buffer and (ii) is no higher than the one in *OverflowID*, we place a fence before the instruction. The reason is that, since we have lost information about Victims in that epoch, we do not know whether the instruction is a Victim. When the epoch whose ID is in *OverflowID* is fully retired, *OverflowID* is cleared.

As an example, consider Figure B.5(a), which shows a ROB full of instructions. The figure groups the instructions according to their epoch and labels the group with the epoch ID. Assume that all of these instructions are squashed and that the SB only has four {ID, PC-Buffer} pairs. Figure B.5(b) shows the resulting assignment of epochs to {ID, PC-Buffer} pairs. Epochs 14 and 15 overflow and, therefore, *OverflowID* is set to 15. Any future insertion in the ROB of an instruction from epochs 14 and 15 will be preceded by a fence. Eventually, some {ID, PC-Buffer} pairs will free-up and may be used by newer epochs such as Epoch 16. However, all instructions from Epochs 14 and 15 will always be fenced.

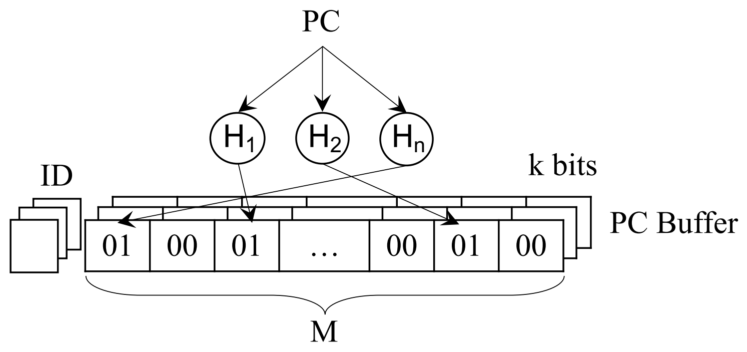


Figure B.4: SB with multiple PC Buffers organized as counting Bloom filters.

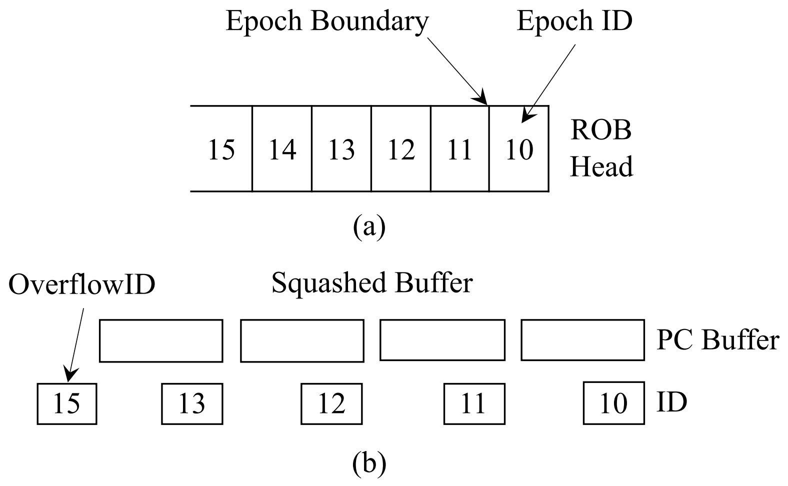


Figure B.5: Handling epoch overflow.

B.6.3 Implementing *Counter*

To implement *Counter*, *Jamais Vu* stores the counters for all the instructions in data pages, and the core has a small *Counter Cache* (CC) that keeps the recently-used counters close to the pipeline for easy access. Since the most frequently-executed instructions are in loops, a small CC typically captures the majority of the counters needed.

We propose a simple design where, for each page of code, there is an associated data page at a fixed Virtual Address (VA) *Offset* that holds the counters of the instructions in the page of code. Further, the VA offset between each instruction and its counter is fixed, to ease access. In effect, this design increases the memory consumed by a program by the size of its instruction page working set.

Figure B.6(a) shows a page of code and its page of counters at a fixed VA offset. When the former is brought into physical memory, the latter is also brought in. The figure shows a memory line with several instructions and the associated line with their counters. We envision each counter to be 4 bits.

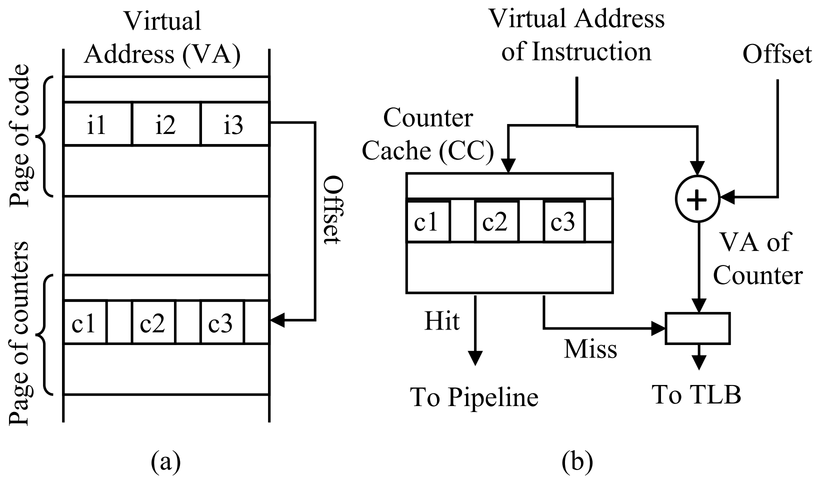


Figure B.6: Allocating and caching instruction counters.

Figure B.6(b) shows the action taken when an instruction is about to be inserted in the ROB. The VA of the instruction is sent to the CC, which is a small, set-associative cache that contains the most recently-used lines of counters. Due to good instruction locality, the CC hits most of the time. On a hit, the corresponding counter is sent to the pipeline to be examined.

If, instead, the CC misses, a *CounterPending* signal is sent to the pipeline. To avoid adding new side channels, no other action is taken until the corresponding instruction reaches its Visibility Point (VP). At that point, the *Offset* will be added to the instruction's VA to obtain the VA of the counter, and this address will be sent to the TLB to obtain the Physical Address (PA) of the counter.

After that, a request will be sent to the cache hierarchy to obtain the line of counters, store it in the CC, and pass the counter to the pipeline.

The operations in the pipeline are as follows. If the counter is not zero or a *CounterPending* signal is received, two actions are taken. First, a fence is inserted in the ROB before the instruction. Second, when the instruction reaches its VP, the counter is decremented and stored back in the CC or, if a CounterPending signal was received, the request mentioned above is sent to the cache hierarchy to obtain the counter. When the counter is returned, if it is not zero, the counter is decremented. The counter is stored in the CC.

In our design, we want the CC to add no side channels. Hence, on a CC hit, the CC's LRU bits are not updated until the instruction reaches its VP. Further, on a CC miss, we delay the request to the cache hierarchy for the counter until the instruction reaches its VP.

B.6.4 Handling Context Switches

To operate correctly, *Jamais Vu* performs the following actions at context switches. In *Clear-on-Retire* and *Epoch*, the SB state is saved to and restored from memory as part of the context. This enables the defense to remember the state when execution resumes. In *Counter*, the CC is flushed to memory to leave no traces behind that could potentially lead to a side-channel exploitable by the newly scheduled process. The new process loads the CC on demand. These operations can be done safely by the trusted environment.

B.7 COMPILER PASS

Epoch includes a program analysis pass that places "start-of-epoch" markers in the program. The pass accepts as input a program in source code or binary. Source code is preferred, since it contains more information and allows a better analysis.

We consider two designs: one that uses loops as epochs and one that uses loop iterations as epochs. In the former, an epoch includes the instructions between the beginning and the end of a loop, or between the end of a loop and the beginning of the next loop; in the latter, an epoch includes the instructions between the beginning and the end of an iteration, or between the end of the last iteration in a loop and the beginning of the first iteration in the next loop. In both *Epoch* designs, procedure calls and returns are also epoch boundaries.

The analysis is intra-procedural and uses conventional control flow compiler techniques [306]. It searches for back edges in the control flow of each function, and from there identifies the natural loops. Once back edges and loops are identified, the *Epoch* compiler inserts the epoch boundary markers.

To mark an x86 program, our analysis pass places a previously-ignored instruction prefix [271] in front of every first instruction of an epoch. The processor ignores this prefix, and our simulator recognizes that a new epoch starts. This approach changes the executable, but because current processors ignore this prefix, the new executable runs on any x86 machine. The size of the executable increases by only 1 byte for every static epoch. For epoch boundaries formed by procedure calls and returns, the compiler does not need to mark anything. The simulator recognizes the x86 procedure call and return instructions and starts a new epoch.

B.8 EXPERIMENTAL SETUP

Architectures Modeled. We model the architecture shown in Table B.5 using cycle-level simulations with gem5 [230]. The baseline architecture is called UNSAFE, because it has no protection against MRAs. The defense schemes are: (i) *Clear-on-Retire* (CoR), (ii) *Epoch* with iteration (EPOCH-ITER), (iii) *Epoch-Rem* with iteration (EPOCH-ITER-REM), (iv) *Epoch* with loop (EPOCH-LOOP), (v) *Epoch-Rem* with loop (EPOCH-LOOP-REM), and (vi) *Counter* (COUNTER).

From Table B.5, we can compute the sizes of the *Jamais Vu* hardware structures. *Clear-on-Retire* uses 1 non-counting Bloom filter. The size is 1232 bits. *Epoch* uses 12 Bloom filters. For *Epoch-Rem*, since the counting Bloom filters use 4 bits per entry, the total size is 12 times 4,928 bits, or slightly above 7KB. A Bloom filter has 14 read and 7 write ports. The Counter Cache (CC) in *Counter* contains 128 entries, each with the counters of one I-cache line. Since the shortest x86 instruction is 1 byte and a counter is 4 bits, each line in the CC is shifted 4 bits every byte, compacting the line into 32B. Hence, the CC size is 4KB.

Application and Analysis Pass. We run SPEC17 [231] with the reference input size. Because of simulation issues with gem5, we exclude 2 applications out of 23 from SPEC17. For each application, we use SimPoint [232] methodology to generate up to 10 representative intervals that accurately characterize the end-to-end performance. Each interval consists of 50 million instructions. We run gem5 on each interval with syscall emulation with 1M warm-up instructions.

Our program analysis pass is implemented on top of Radare2 [273], a state-of-the-art open-source binary analysis tool. We extend Radare2 to perform epoch analysis on x86 binaries.

B.9 EVALUATION

B.9.1 Thwarting Proof-of-Concept (PoC) MRA

To demonstrate *Jamais Vu*'s ability to thwart MRAs, we implement a PoC MRA on gem5 similar to the port contention attack in [280]. After testing a secret, the victim thread performs a division operation. The attacker picks 10 Squashing instructions that precede the test and the division. The code is similar to Figure B.1(a). In UNSAFE, the attacker causes 5 squashes on each of the 10 Squashing instructions, for a total of 50 replays of the division operation. With *Clear-on-Retire*, the number of replays decreases to 10, since each Squashing instruction can only cause a single replay. With *Epoch*, there is a single replay because all the code belongs to a single epoch. With *Counter*, there is a single replay because the division only commits once.

Table B.5: Parameters of the simulated architecture.

Parameter	**Value**
Architecture	2.0 GHz out-of-order x86 core
Core	8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, L-TAGE branch predictor, 4096 BTB entries, 16 RAS entries
L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher
L1-D Cache	64 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher
L2 Cache	2 MB, 64 B line, 16-way, 8 cycles RT latency
DRAM	50 ns RT latency after L2
Counter Cache	32 sets, 4-way, 2 cycle RT latency, 4b/counter
Bloom Filter	1232 entries, 7 hash functions. Non-counting: 1b/entry. Counting: 4b/entry
{ID, PC-Buffer}	12 pairs in *Epoch*; 1 pair in *Clear-on-Retire*

B.9.2 Execution Time

Jamais Vu proposes several schemes that offer different performance, security, and implementation complexity trade-offs. Figure B.7 shows the normalized execution time of SPEC17 applications on all schemes but *Epoch* without removals, which we consider later. Time is normalized to UNSAFE.

Among all the schemes, COR has the lowest execution time overhead. It incurs only a geometric mean overhead of 2.9% over UNSAFE. It is also the simplest but least secure design (Table B.3). EPOCH-ITER-REM has the next lowest average execution overhead, namely 11.0%. This design is also very simple and is more secure, especially as we will see that false negatives are very rare. The next design, EPOCH-LOOP-REM, has higher average execution time overhead, namely 13.8%. However, it has simple hardware and is one of the two most secure designs (Table B.3)—again, given that, as we will see, false negatives are very rare. Finally, COUNTER has the highest average execution overhead, namely 23.1%. It is one of the two most secure schemes, but the implementation proposed is not as simple as the other schemes. From all these schemes, EPOCH-LOOP-REM and perhaps CoR appear to be the most appealing.

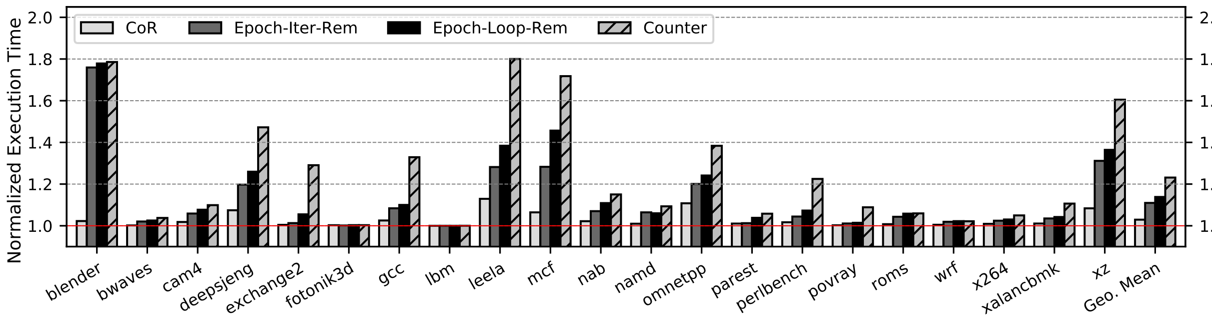


Figure B.7: Execution time for all the schemes except *Epoch* without removals. Time is normalized to UNSAFE.

The schemes not shown in the figure, namely EPOCH-ITER and EPOCH-LOOP are not competitive. They have an average execution overhead of 22.6% and 63.8%, respectively. These are substantial increases over the schemes with removals, with modest gains in simplicity and security.

B.9.3 Sensitivity Study

Each *Jamais Vu* scheme has several architectural parameters that set its hardware requirements and efficiency. Recall that CoR uses a Bloom filter, while EPOCH-ITER-REM and EPOCH-LOOP-REM use counting Bloom filters. To better understand the different Bloom filters, we first perform a sensitivity study of their parameters. Then, we evaluate several Counter Cache organizations for COUNTER.

Number of Bloom Filter Entries. Figure B.8 shows the geometric mean of the normalized execution time and the false positive rates (FP) on SPEC17, when varying the size of the Bloom filter. We consider several sizes, which we measure in number of entries. Recall that each entry is 1 bit for CoR and 4 bits for the other schemes. We pick each of these number of entries by first selecting a *projected element count* (i.e., the number of items that we expect to be inserted in the Bloom filter, as shown in parenthesis in the figure) and running an optimization pass [307] for a target false positive probability of 0.01. From the figure, we see that a Bloom filter of 1232 entries strikes a good balance between execution and area overhead, with a false positive rate of less than 0.5% for all the schemes.

Number of {ID, PC-Buffer} Pairs. Another design decision for Epoch-Iter-Rem and Epoch-Loop-Rem is how many {ID, PC-Buffer} pairs to have. If they are too few, overflow will be common. Figure B.9 shows the average normalized execution time and the overflow rates on SPEC17, when varying the number of {ID, PC-Buffer} pairs. The overflow rate is the fraction of insertions into PC Buffers that overflow. From the figure, we see that, as the number of {ID, PC-Buffer} pairs decreases, the execution time and overflow rates increase. Supporting 12 {ID, PC-Buffer} pairs is a good design point.

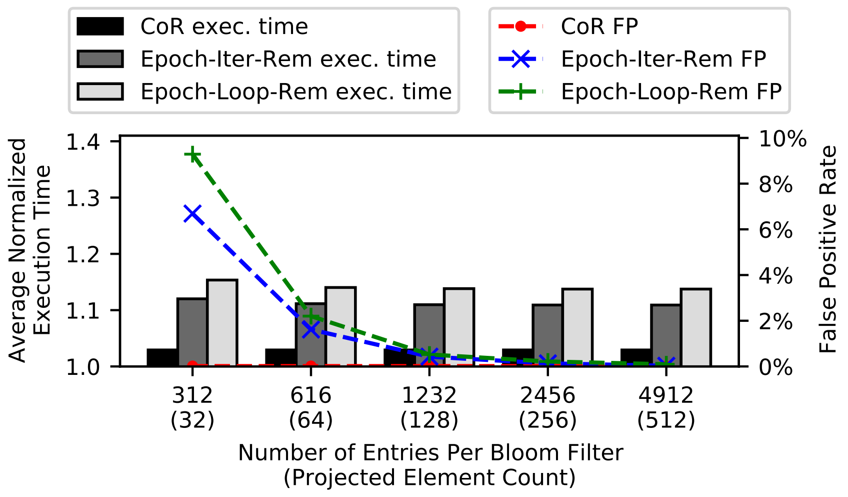


Figure B.8: Average normalized execution time and false positive rate (FP) when varying the number of entries per Bloom filter. The numbers in parenthesis are the maximum number of items to be inserted in the Bloom filter to attain a target false positive probability of 0.01.

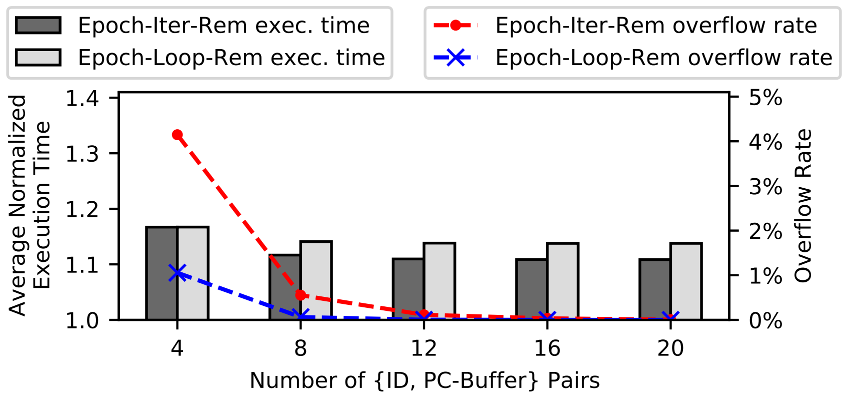


Figure B.9: Average normalized execution time and overflow rate when varying the number of {ID, PC-Buffer} pairs.

Number of Bits Per Counting Bloom Filter Entry. The counting Bloom filters in Epoch-Iter-Rem and Epoch-Loop-Rem use a few bits per entry to keep the count.

Figure B.10 shows the average normalized execution time and the false negative rates (FN) on SPEC17, when varying the number of bits per entry. We see from the figure that the number of bits per entry has little impact on the performance. However, as the number of bits per entry decreases beyond four, the false negative rate increases rapidly. For four bits per entry, the false negative rate is an acceptable 0.02% for Epoch-Loop-Rem and 0.006% for Epoch-Iter-Rem.

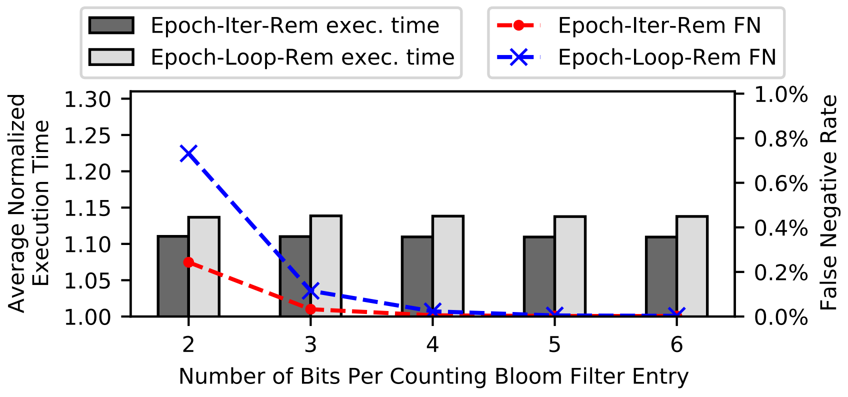


Figure B.10: Average normalized execution time and false negative rate (FN) when varying the number of bits per counting Bloom filter entry.

False negatives can be caused either by conflicts in the filter or by not having enough bits in an entry. In the latter case, when the counter in the entry saturates, it cannot record further squashes and information is lost. To estimate the relative impact of these two sources of false negatives, we took our default Bloom filter of 1232 entries and four bits per entry, and artificially eliminated conflicts. We did this by recording the inserted items in an ideal hash table that has no conflicts. We found that the resulting false negative rates are 0.004% and 0.002% for Epoch-Loop-Rem and Epoch-Iter-Rem, respectively. These numbers are comparable to the false negative rates obtained by taking the default Bloom filter and simply adding one extra bit per entry.

Counter Cache (CC) Geometry. Figure B.11 shows the CC hit rate as we vary the ways and sets of the CC. We see that the CC hit rate increases with the number of entries, but that changing the associativity of the CC from 4 to full does not help. Overall, our default configuration of 32 sets and 4 ways performs well. It attains an average hit rate of 93.7%, while a larger cache or a fully-associative one improves the hit rate only a little. A smaller cache hurts the hit rate substantially.

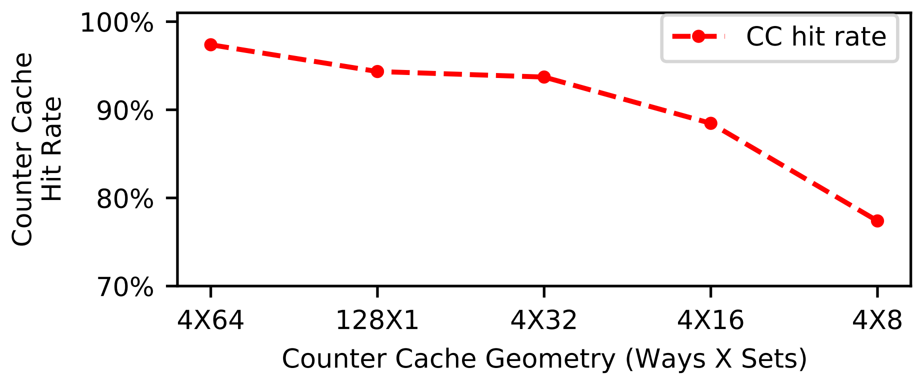


Figure B.11: CC hit rate when varying the cache geometry.

B.10 RELATED WORK

There are some works related to mitigating MRAs.

Preventing Pipeline Squashes. The literature includes several solutions that can mitigate specific instances of MRAs. For example, page fault protection schemes [308, 309, 310, 311] can be used to mitigate MRAs that rely on page faults to cause pipeline squashes. The goal of these countermeasures is to block controlled-channel attacks [312, 313] by terminating victim execution when an OS-induced page fault is detected. The most recent of these defenses, Autarky [309], achieves this through a hardware/software co-design that delegates paging decisions to the enclave. However, attacks that rely on events other than page faults to trigger pipeline squashes (Section B.3.1) would still overcome these point-mitigation strategies. In contrast, *Jamais Vu* is the first comprehensive defense that addresses the root cause of MRAs, namely that instructions can be forced to execute more than once.

Preventing Side Channel Leakage. Another strategy to mitigate MRAs is to prevent speculative instructions from leaking data through side channels. For example, several works have proposed to mitigate side channels by isolating or partitioning microarchitectural resources [44, 55, 61, 226, 276, 310, 314, 315, 316], thus preventing the attacker from accessing them during the victim process' execution. These defenses prevent adversaries from leaking data through specific side channels, which ultimately makes MRAs's ability to denoise these channels less useful. In practice, however, no holistic solution exists that can block all side channels. Further, new adversarial applications of MRAs may be discovered that go beyond denoising side-channel attacks.

B.11 CONCLUSION

This appendix presented *Jamais Vu*, the first technique to thwart MRAs. *Jamais Vu* detects when an instruction is squashed and, as it is re-inserted into the pipeline, places a fence before it. The three main *Jamais Vu* designs are *Clear-on-Retire*, *Epoch*, and *Counter*, which offer different trade-offs between security, execution overhead, and implementation complexity. One design, called *Epoch-Loop-Rem*, effectively mitigates MRAs, has an average execution time overhead of 13.8% in benign executions, and only needs counting Bloom filters. An even simpler design, called *Clear-on-Retire*, has an average execution time overhead of only 2.9%, although it is less secure.