PRINCIPLED APPROACHES FOR MITIGATING MICRO-ARCHITECTURAL SIDE-CHANNEL ATTACKS

BY

JIYONG YU

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, 2023

Urbana, Illinois

Doctoral Committee:

Assistant Professor Christopher W. Fletcher, Chair
Professor Josep Torrellas
Professor Darko Marinov
Professor Trent Jaeger, Pennsylvania State University
Associate Professor Adam Morrison, Tel Aviv University
Dr. Fangfei Liu, Intel Labs

ABSTRACT

Micro-architectural side-channel attacks are a critical security threat that arises as a result of modern processors' pursuit of performance and efficiency. In those attacks, malicious actors exploit the micro-architectural implementation of processors to attack victim software, by monitoring how data-dependent micro-architectural resource usage varies in response to the victim's secret information. By focusing on hardware, this intricate security attack can exfiltrate sensitive information in a software-invisible manner. As future processors continue to increase in complexity, the risk posed by micro-architectural side-channel attacks is expected to escalate.

This thesis represents a significant advancement in developing secure, comprehensive, and high-performance micro-architectural side-channel mitigation solutions. While various mitigations have been proposed to address these attacks, existing approaches either target specific attack types, leaving vulnerabilities against other or future side-channel attacks open, or induce substantial performance degradation. The ideal mitigation solution, therefore, should both offer strong and comprehensive security guarantees while maintaining modest performance overhead.

To overcome these challenges, the key idea underpinning our solutions is enforcing information-flow properties at the hardware level: once all side-channel vulnerabilities are identified, and all secret information is correctly tracked and annotated, blocking micro-architectural side-channel leakage is simply preventing side channels from consuming the secret. Based on this key idea, we developed the data-oblivious ISA (OISA), which for the first time, incorporates side-channel-specific security specification at the ISA level and enforces the desired information-flow properties in commodity hardware. To address the recent surge of speculative side-channel attacks, we further designed Speculative Taint Tracking (STT), which employs the same principle for achieving provable security against speculative side channels in general. We further improve the performance of STT with Speculative Data-Oblivious Execution (SDO) without sacrificing its security properties.

In addition to the proposed mitigation frameworks, we also examined several existing point-mitigation strategies and developed new attacks circumventing those mitigations. We demonstrated how common control-flow leakage attack mitigations fail with a new attack variant capable of extracting the byte-granular PC information of arbitrary victim's dynamic instruction. We also showcased why eliminating timers is insufficient in blocking cache side-channel attacks by identifying new primitives for monitoring cache state. Although

these attacks may be further mitigated with new point defenses, our claim is that defending against micro-architectural side-channel attacks should not become a cat-and-mouse game. Instead, comprehensive mitigations, such as the solutions proposed in this thesis, should be adopted to effectively combat current and future attacks.

To My Families.

ACKNOWLEDGMENTS

Completing a Ph.D. in computer science from one of the top schools in the world is a challenging and demanding journey, filled with uncertainties and immense pressure. However, it is also a path that brings moments of joy, encouragement, and personal growth. There is no way I can be where I am today without the support and motivation of numerous individuals who have played pivotal roles in my life.

First and foremost, I would like to express my utmost gratitude and respect to my advisor, Chris Fletcher. He accepted me as one of his first students, when I barely knew anything about doing research, and guided me throughout this six-year Ph.D. journey. In my honest opinion, Chris is not only an exceptional advisor, but also a perfect human being ("perfect" is usually not in the dictionary of a researcher/scientist; but I feel like it's the right word for Chris). He is kind, smart, humorous, diligent, and consistently displays unwavering enthusiasm and positivity towards research. He is patient and empathetic to every student and always maintains great relationships with every one of his colleagues. I can easily write another thesis solely focusing on his outstanding qualities. The most invaluable part of my graduate school career was learning from Chris, from developing skills for conducting research projects from start to finish, to how to treat people around me with respect and kindness. I will definitely miss the days when I knocked on your door and asked whether you have time for a project update so that I can soak up more knowledge and wisdom during our meetings. I wish you all the best in your future career as a tenured professor.

Second, I would extend my gratitude to some of my close collaborators, who also graciously served as members of my Ph.D. thesis committee: Josep Torrellas, Darko Marinov, Adam Morrison, Trent Jaeger, and Fangfei Liu. An important piece of my Ph.D. career is the regular project meetings we had, where we would discuss progress and problems. It was during these meetings that I gained a wealth of knowledge from each and every one of you. Josep, Darko, and Trent, all being distinguished and highly respected senior professors in computer architecture and system security fields, taught me valuable lessons on becoming a true researcher with their remarkable abilities to identify those problems and communicate complex concepts. Adam and Fangfei, with their sharp intellects, consistently raised thought-provoking questions that pinpointed critical aspects we might have overlooked. To me, you all are truly the best professors and researchers in this field, and it has been my greatest privilege to work with all of you.

Next, I would express my appreciation to my incredible lab mates: Kartik Hedge, Hadi

Asgharimoghaddam, Jose Rodrigo Sanchez Vicarte, Rohit Agrawal, Mohamad El-Hajj, Riccardo Paccagnella, Sushant Dinesh, Rutvik Choudhary, Nandeeka Nayak, Hannah Liang, and Boru Chen. It is a great experience working alongside all of you in one lab, engaging in discussions about research ideas during the weekly meetings, and chatting about everything during our leisure time. When I look back, the most memorable part about Ph.D. was working late nights with you guys in the lab fighting for the paper deadlines—it's certainly not fun and usually filled with distress and pain, but only through these hardships can we become better. There is an old Chinese saying by Confucius that states: "When three people walk together, there must be one who can be my teacher". However, I must say that all of you have been my teacher. Working with all of you and developing close friendships with you guys has been an invaluable asset throughout my years in graduate school. To those of you who have graduated, I wish you all the best in your post-Ph.D. career. And to those of you who are still students, I wish you a successful and fruitful Ph.D. journey.

Next, I have been fortunate to work with several industry researchers during my internships at Intel Labs and Microsoft Research. These experiences provided me with profound insight to propel my own research studies. I would like to thank Carlos Rozas, Frank Mckeen, Fangfei Liu (once again), Thomas Unterluggauer, Ron Gabor at Intel, and Weidong Cui, Xinyang Ge at Microsoft Research for sharing your profound knowledge and providing me guidance during my summer internships. I hope one day in the future I would have to opportunity to work with you again.

I would also like to thank many other people who helped me during my years at UIUC. First, I want to thank Mengjia Yan for her guidance in my early years; her work has shown me why hardware security is a great field for research. Second, I want to thank my advisor and mentors in my undergraduate years at the University of Michigan. I am so grateful to Prof. Trevor Mudge and Prof. Jason Mars who opened up the gate of computer architecture research for me. Third, I would like to thank my friends, including but not limited to: Jinyang Li, Zirui Zhao, Houxiang Ji, Jing Ying, Liming Yang, Ruijie Wang, Tianshi Wang, Yunan Zhang, Dongxing Liu, Yihan Pang, Boyuan Tian, Yifan Yuan, Jingcheng Ma. "True friends are like stars, you may not always see them, but you know they are always there." Although I would not be in Champaign anymore with you all, I firmly believe that distance will not diminish our friendship, and I look forward to reuniting with you all in the future.

Last but definitely not least, the true heroes are all my family members, including my dear wife Ru Huang. I have not seen my parents and grandparents for more than three years until this moment. However, every time I see them through video chat I can feel the love and support they have for me. For instance, my grandma always ensures that I am healthy and not getting covid, and my father patiently acts as an amplifier, trying to repeat every

word I said (my grandma's hearing is impaired). Words cannot describe how much I miss you and I will be back to see you all as soon as I can. My wife, Ru Huang, is a true blessing in my life. Since we met each other during the start of the pandemic, he has been by my side, giving me all the love and support I can ask for, and also encouraging me to pursue what I want. I promise that I will not let any of you down and will continue to strive for excellence throughout my career.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Over the past several decades, the third industrial revolution—also referred to as the digital revolution—has brought sweeping changes to every corner of the world, influencing every aspect of human life, including education, communication, work, and entertainment. Central to this revolution lies the development of microprocessors, which have unleashed immense computing power and catalyzed innovations across various disciplines. And critical to the advancement of microprocessors is computer architecture design, which has continuously pushed the boundaries of computer performance and efficiency with innovations in underlying hardware structures and organizations.

However, in the late 1990s, researchers began to realize that the growing sophistication of those processor design optimizations may lead to serious security issues, known as *micro-architectural side channels*, leaking sensitive data stealthily across mutually distrusted parties [1]. Since then, many processor structures and optimizations have been exploited as side-channel attacks, ranging from the initial cache attack [2], to more recent speculative execution attacks [3, 4]. Those attacks have proven to cause a wide range of security and privacy threats, from breaking cryptographic implementations [5, 6, 7], stealing personal information [8, 9], to monitoring user activities [10, 11, 12], and in the worst case scenario, leaking the entirety of the program memory [4]. With modern processors persisting in seeking performance and efficiency by incorporating more sophisticated, but potentially more vulnerable designs, mitigating micro-architectural side-channel attacks has become a matter of utmost urgency.

1.1 MICRO-ARCHITECTURAL SIDE-CHANNEL ATTACKS

Computation is a process of generating output from a given input based on a protocol that defines the input-output relationship. The protocol can take various forms depending on the level of abstraction. At the software level, a protocol can be a program written to process some user input; while at the hardware level, a protocol can be a combinational circuit describing the transitions of the register state. Cyber-attackers develop techniques to attack these protocols in different computing systems to undermine different security properties, such as confidentiality and integrity.

Side-channel attacks are a common type of attack that targets confidentiality. The fundamental difference between side-channel attacks and other common attacks, such as memory corruption attacks [13], lies in how they exploit the protocol. Common attacks target

the design errors within the protocol itself, such as program bugs, coercing the protocols into behaving in unintended ways, ultimately generating output directly revealing the secret input. In contrast, side-channel attacks do not rely on any program bugs to misuse the protocol. The key insight of side-channel attacks is that, protocols implemented in the real world inherently produce various forms of computational side effects that depend on the input. By measuring those computational side effects, side-channel attacks may obtain information about the secret input through the correlation between the observed side effects and the input, as illustrated in Figure 1.1a [14].

(a) Side-channel attacks

(b) Micro-architectural side-channel attacks

Figure 1.1: Relationship between side-channel attacks and micro-architectural side-channel attack

Micro-architectural side-channel attacks constitute a significant class of side-channel attacks. In these attacks, as shown in Figure 1.1b, the attacker focuses on the practical implementation of processors, with the objective of learning the secret input through its measurement of the micro-architectural resource usage resulting from processing the input on the targeted processor. This approach is driven by the fact that many micro-architectural structures incur data-dependent state changes: as an example, the location of data within a processor cache directly depends on the data address. Therefore, an attacker aware of the data's residency inside the cache can easily deduce information about the data address.

Since the discovery of micro-architectural side-channel attacks, numerous hardware structures have been exploited as side channels, such as cache [5, 15], Translation Lookaside Buffers (TLBs) [16], predictors [17, 18, 19, 20], micro-op caches [21], execution units [22], and data prefetchers [23, 24]. As the complexity of modern microprocessors continues to evolve, more and more side-channel vulnerabilities will likely be discovered in the future.

1.2 PROBLEMS OF EXISTING MICRO-ARCHITECTURAL SIDE-CHANNEL MITIGATIONS

With the emergence of micro-architectural side-channel attacks as a critical security threat, it is imperative for researchers and practitioners to develop effective strategies for mitigating those attacks. Unfortunately, existing mitigations either deliver weak and limited security guarantees, or induce substantial performance degradation.

For instance, researchers have proposed secure hardware designs to fix processor components prone to side-channel attacks, including secure cache [25], secure TLB [26], and others. However, given a processor is a synergy of numerous different hardware components, fixing vulnerabilities individually without a comprehensive study of the entire processor is bound to be incomplete.

There have also been protection schemes aimed at comprehensively mitigating micro-architectural side channels. One straightforward solution is to eliminate resource sharing by running the victim program on dedicated hardware exclusively. While this approach can be considered comprehensive as it prevents attackers from observing the hardware usage (since most micro-architectural side-channel attacks require co-residency with the victim program), it is impractical for general applications. Modern processors are designed to handle multiple tasks simultaneously [27], so restricting a processor to a single victim software would lead to significant resource under-utilization, which may further exacerbate as future processors pursue more aggressive parallel processing.

Another popular approach that claims to comprehensively mitigate micro-architectural side channels without relying on physical isolation is data-oblivious programming. Data-oblivious programming encapsulates a series of secure coding practices with the goal of eliminating any secret-dependent hardware usage. Yet, achieving data obliviousness usually entails introducing redundant operations to conceal the actual secret-dependent execution behavior, resulting in significant performance overhead in practice.

1.3 THESIS CONTRIBUTIONS AND ORGANIZATION

Given the limitations of existing micro-architectural side-channel mitigations, the main objective of this thesis is to design hardware-based mitigations that deliver *strong and comprehensive security guarantees* that cover micro-architectural side-channel attacks broadly, and at the same time *minimizing the performance overhead* caused by the defense mechanism. Central to our solution is the key idea of consolidating different individual side-channel attacks into general and clear micro-architectural side-channel abstractions. These abstractions serve as micro-architectural specifications to help enforce security policies in hardware designs, and offer a unified protection mechanism for different, and even future attacks.

Our abstractions are directly inspired by the existing information flow literature [28]. At a high level, our abstractions define *secrets* as data inside micro-architecture that requires protection. In addition, data-dependent hardware execution behaviors are classified as *channels*. Thus, mitigating micro-architectural side-channel attacks can be formulated as blocking the information flow from any secrets to the identified channels. The benefit of this

methodology is two-fold. In terms of security, preventing the channels from ever consuming secrets comprehensively blocks different types of side-channel leakages. Performance-wise, the protection mechanism is relaxed whenever possible, for example when processing public data (*non-secrets*), or when any data is processed by data-independent hardware execution (*non-channels*).

This thesis consists of two parts. First, this thesis presents several side-channel protection frameworks which demonstrate how specifying proper abstractions helps to mitigate side-channel attacks comprehensively and efficiently. Chapter 3 describes OISA, which applies this methodology at the hardware-software interface level, by specifying the concept of secrets and channels through new instruction-set architecture (ISA) extensions. The new side-channel-centric security definition introduced by OISA can be used to guide the development of side-channel-resistant hardware, and simultaneously enable provable side-channel-free programming. While OISA concentrates on micro-architectural side-channel attacks in general, Chapter 4 and 5 introduce new abstractions for the more recent speculative execution attacks. The two proposed mitigations, named STT and SDO, apply the security abstractions originated from OISA to construct software-invisible defenses that overcome the additional security threats introduced by speculative execution attacks with tolerable performance costs.

Second, we pinpoint the weaknesses of existing point mitigations in concrete side-channel attack scenarios, thereby advocating for the adoption of robust and comprehensive mitigations such as our proposed schemes, as such schemes have not yet been implemented in real processors. In Chapter 6, we investigate the idea of fixing side-channel vulnerabilities by introducing code transformations in response to new side-channel attacks, specifically in the context of control-flow inference attacks. Our study presents a new instance of control-flow leakage attacks that exposes new attack opportunities and surpasses state-of-the-art defenses. Chapter 7 explains in practice how constraining the attacker's observability over the micro-architectural state fails, as modern complex hardware contains hidden primitives that enable the attacker to retrieve micro-architectural state information.

We now present the organization of the thesis with the overview of each chapter.

Chapter 2 – Background This chapter presents a framework for understanding micro-architectural side-channel attacks, which can be divided into classic side-channel attacks and more recent speculative side-channel attacks. We explain how these two types of attack work and the important part of the attack process, using existing attacks as examples. We also discuss mitigations for these attacks.

Chapter 3 – Data-Oblivious ISA (OISA) This chapter presents OISA—a hardware-software interface that enables formally-provable, portable, and performant side-channel-free programming. OISA is built upon data-oblivious programming, a technique that theoretically eliminates micro-architectural side-channels, but in practice remains vulnerable due to the lack of security guarantees from the hardware. OISA provides a security specification that establishes the security foundation for data-oblivious programming while maintaining the flexibility required for designing hardware optimizations.

The idea behind OISA is a combination of new ISA-level security specifications with hardware information-flow tracking. The new ISA allows programmers to directly communicate data secrecy with the hardware (rather than indirectly, e.g., via encryption [29]). OISA also directly exposes any potential micro-architectural side channels in the form of unsafe instruction operands. Therefore, eliminating side-channel leakage is equivalent to preventing the information flow from any secret data to unsafe operands—we demonstrate how to achieve this by presenting an implementation of OISA architecture on an existing processor. This methodology not only ensures security against traditional side-channel attacks but also the recent speculative side-channel attacks without any special handling.

This work appears in [30] and won the following awards:

- Distinguished Paper Award Honorable Mentions at NDSS'19
- CSAW'19 Applied Research Finalists
- IEEE MICRO Top Picks, 2020

Chapter 4 – Speculative Taint Tracking (STT) This chapter presents a novel protection framework named STT, which comprehensively mitigates speculative execution attacks. STT makes two key contributions: first, it presents a comprehensive classification of hardware mechanisms (referred to as covert channels) that might transmit speculative data to the attacker's execution context, in the form of data-dependent hardware resource usage. These covert channels manifest as explicit channels and implicit channels, akin to the explicit flow and the implicit flow in the program analysis literature. Second, STT applies OISA's methodology of mitigating micro-architectural side channels to speculative execution attacks only, by leveraging hardware-level information-flow tracking that prevents secret-dependent data from being consumed by identified covert channels during the execution of transmitter instructions. This prevention is implemented in a conservative manner in STT: the speculative execution of transmitter instructions that deliver secrets to covert channels is delayed until non-speculative. But like OISA, the protection is only enforced when the leakage is possible, which effectively reduces the performance cost of the overall mitigation.

This work appears in [31] and won the following awards:

- Best Paper Award at MICRO'19
- IEEE MICRO Top Picks, 2020
- Communication of ACM Research Highlight, 2021

Chapter 5 – Speculative Data-Oblivious Execution (SDO) This chapter introduces SDO, a protection framework that builds on top of STT to achieve higher performance without compromising STT's strong security guarantee. Since the performance overhead of STT is mostly due to the delay of transmitter instruction's execution when it involves covert channels consuming speculative secrets, SDO devises novel mechanisms to proceed with the execution. Inspired by OISA, SDO partially achieves this goal with data-oblivious execution hardware, whose execution is independent of sensitive data.

However, despite its security, data-oblivious execution involves significant redundant work that usually offsets the benefit of executing unsafe transmitters early. Here comes the major contribution of SDO: we for the first time demonstrate how prediction/speculation can assist in mitigating speculative attacks, despite being the root cause of those attacks. In short, SDO utilizes prediction to reduce the baseline data-oblivious execution down to a smaller set of hardware activities that still maintain data obliviousness, and eventually delivers a net performance gain.

This work appears in [32] and won first place in the Inaugural Intel Hardware Security Academic Award in 2021.

Chapter 6 – NightVision Attack This chapter presents a novel micro-architectural side-channel attack called NightVision that is capable of directly observing the exact address of arbitrary victim dynamic instructions. NightVision leverages several overlooked characteristics of Branch Target Buffers in modern processors, and shows how this predictor structure, while commonly considered only relative to control instructions, can also lead to observation of the execution of non-control instructions. NightVision's capability of extracting PCs of potentially any victim dynamic instructions naturally surpasses prior control-flow leakage attacks that infer secret information through its impact on program control-flow, and overcomes corresponding software mitigations. In addition, NightVision enables new opportunities in binary fingerprinting—allowing an attacker to identify unknown victim programs through their dynamic instruction address footprints.

This work appears in [33].

Chapter 7 – Synchronization Storage Channel (S^2C) This chapter presents a novel timer-less cache side-channel attack that exploits hardware synchronization instructions, specifically load-linked/store-conditional (LL/SC) instructions. This attack, dubbed S^2C, aims at eliminating the reliance on high-resolution timing measurements in existing cache attacks. S^2C exposes how the implementation of LL/SC on modern Apple processors allows an attacker to directly read private cache evictions as the SC return value. With this vulnerability, S^2C further develops several techniques to expand the attacker's observability toward activities within multiple shared cache sets. The end result is a cache side-channel attack with the same threat as state-of-the-art cache attacks [5, 34, 35] without depending on the existence of high-resolution timers.

This work will appear in [36].

Chapter 8 – Conclusion This chapter summarizes the thesis and addresses possible directions for future research.

CHAPTER 2: BACKGROUND

2.1 PROCESSOR OPTIMIZATIONS

Modern processors integrate a wide range of processor optimizations to enhance performance and efficiency. Unfortunately, many of those optimizations exhibit data-dependent behavior which the attacker can exploit for mounting micro-architectural side channels. Before delving into those attacks, it is important to first highlight the most common processor optimizations, as they form the foundation for most identified side-channel attacks.

2.1.1 Processor Pipelining and Out-of-order Execution

Modern processors use dynamic scheduling to execute instructions in parallel and out of program order to improve performance [37, 38]. This is achieved by dividing the processing of instructions into multiple *pipeline stages*. Initially, instructions are fetched into the processor pipeline at the *fetch stage*, and issued to reservation stations for scheduling at the *issue stage*, all following the program order. All pipeline stages up to this point are usually referred to the processor *frontend*. The execution of instructions at the *execution stage* is out of program order. Finally, at the *retire (also called commit) stage*, instructions retire as they make their operation externally visible by irrevocably modifying the architectural system state in program order. The pipeline stages starting from the execution stage until the retire stage are referred to as the processor *backend*. The in-order retirement is implemented by queuing instructions in a reorder buffer (ROB) [39] in the instruction fetch order and dequeuing a completed instruction when it reaches the ROB head.

2.1.2 Prediction and Speculation

Speculative execution improves performance by executing instructions whose validity is uncertain instead of waiting to determine their validity. If such a speculatively-executed instruction turns out to be valid, it is eventually retired; otherwise, it is *squashed* and the processor's state is rolled back to the prior state when speculation starts. As a byproduct, all the following instructions also get squashed. That is, a squash causes a large pipeline disturbance. However, if the speculation is done correctly, the processor can execute instructions that would have been delayed, leading to a significant performance improvement.

To facilitate speculative execution, modern processors employ multiple hardware prediction mechanisms for determining when and how speculation should be performed. The most

common prediction mechanism is the branch predictor for predicting the direction and target of control-transfer instructions. Branch predictors in today's processors are sophisticated hardware logic structures that can be trained to learn the program's control-flow behavior, and make predictions based on the current program counter value alone. Additionally, memory dependency prediction is another prevalent prediction mechanism for predicting dependencies between memory loads and stores before their address operands are ready. Other prediction strategies such as value prediction have also been proposed for improving processor performance [40].

2.1.3 Superscalar processing

Super-scalar processing is a fundamental performance optimization for executing multiple instructions within a single instruction stream at any clock cycle. A superscalar processor augments its hardware resources, such as buffers, execution units, and reservation stations, allowing every pipeline stage— such as fetch, decode, issue, execute and commit—to handle multiple instructions per cycle. The maximal number of instructions that can be possibly processed by a superscalar pipeline simultaneously is usually called *pipeline width*. When combined with other optimizations such as processor pipelining and speculative execution, superscalar processors achieve a high level of instruction-level parallelism and overall computation throughput.

The instruction-level parallelism achieved by superscalar processing is inherently restricted by the interdependency between instructions, as the execution of mutually-dependent instructions has to be serialized. Programs with fewer inter-instruction dependencies thus benefit more from superscalar processing. On the other hand, superscalar machines must employ dedicated hardware logic for checking instruction dependencies, whose complexity scales super-linearly with the pipeline width. Consequently, most real-world processors typically opt for modest pipeline widths (typically around 4 to 8).

2.1.4 Multi-threading, Multi-core, Multi-processor

Simultaneous multi-threading, multi-core, and multi-processor are basic performance optimization aiming at improving the resource utilization by concurrently running multiple programs at different levels of processors. Simultaneous multi-threading, or SMT [41], allows each processor core to simultaneously run multiple hardware threads, such that a hardware thread can use core resource that is not currently used by other threads. While individual thread performance may be negatively impacted by resource competition with other threads,

the overall utilization of the execution units within the core pipeline is maximized, resulting in a substantial throughput improvement.

Multi-core and multi-processor achieve a lower-level of resource sharing compared to SMT. With multi-core, processor cores can share lower-level resources, such as shared caches and memory controllers. The prevalence of multi-core processors is evident in the contemporary computing landscape, with the core count in state-of-the-art processors continuously increasing to meet the demands of modern computing tasks [42]. Multi-processor architecture aims at building large, expansive interconnected computer networks by linking a large number of processors together. This allows each processor to share data with other processors through high-speed interconnects.

2.1.5 Caching

Caching is essential optimization to alleviate the performance impact of memory (DRAM) accesses. Since accessing main memory is a slow process that may yield a significant performance bottleneck, modern processors adopt small and fast on-chip storage units called caches for storing frequently used data. By doing so, caching benefits applications with good locality by transforming slow DRAM accesses into fast cache accesses.

Processor caches operate as lookup tables, mapping addresses to corresponding data. Based on how cache access logic locates the target data, caches can be categorized into three types: *directed-mapped*, *fully-associative* and *way-associative* caches, among which the way-associative cache is the most popular option for modern processors. A way-associative cache is organized into cache *sets*, where each set is associated with a specific subset of address bits called *set index bits*. Each cache set consists of N cache lines, where N represents a number of ways, and each cache line stores a fixed number of contiguous bytes. During a cache access, the cache access logic first uses cache set index bits to locate a cache set, which contains cached data that share the same set index bits. The remaining address bits are then used to identify the target data within the set. When the data is present in the cache (called a *cache hit*), it is retrieved and returned to the pipeline. If the data is not found (a *cache miss*), the next cache level (or the memory) will be accessed.

Most processors nowadays adopt a multi-level cache hierarchy. The highest-level caches are typically private to each core and have smaller sizes compared to lower-level caches, allowing for short access latency. The lower-level caches are larger in size and are commonly shared between multiple processor cores. As a trade-off, they store more frequently-used data but come with longer access latency. The specific organization of the cache hierarchy is implementation-dependent and varies across different processors.

2.2 OVERVIEW OF MICRO-ARCHITECTURAL SIDE-CHANNEL ATTACK

Micro-architectural side-channel attacks are attacks exploiting the incidental computational side effects resulting from running programs on real processors, to infer secret information processed by the processor. Those side effects include but are not limited to timing [1], power [43], temperature [44], and hardware activity statistics reported by the hardware performance counters. Modern processors provide primitives for directly obtaining these side effects at the software level, making micro-architectural side-channel attacks possible with only software effort.

Micro-architectural side-channel attacks can be categorized into two large types based on the attacker's proximity to the victim program. First, there are *remote attacks*, where the attacker needs not co-reside with the victim software on the same computing platform. The attacker simply interacts with the victim (e.g., via a caller-callee relationship) from remote and obtains the side effect associated with the victim's computation. Timing is the most commonly exploited side effect in this case, which has inspired the remote timing attack—a common side-channel attack paradigm where the attacker can remotely invoke the vulnerable program and obtain accurate program execution timing. For instance, Paul Kocher utilized end-to-end program execution timing to learn secret data in cryptographic systems [1]. While many remote timing attacks only exploit the data-dependent behavior at the program level [45, 46, 47, 48], data-dependent micro-architectural designs also significantly influences the program timing [49, 50]. Other side effects, like power [50, 51, 52] could be exploited in a similar manner. Despite the advantage of not relying on resource sharing between the attacker and the victim, the coarse-grained, end-to-end measurement of the side effect is typically susceptible to a low signal-to-noise ratio in today's computing systems. Therefore, large numbers of repeated experiments are necessary for extracting a clear correlation between the measured side effect and the target secret value.

The second attack setting, assumed by most micro-architectural side-channel attacks, leverages the multi-tenancy in modern computing systems. Multi-tenancy is ubiquitous nowadays: browsers on personal devices can connect to websites from independent hosts; cloud computing platforms such as Amazon AWS [53] and Microsoft Azure [54] normally use the same processor for hosting services belonging to different customers for higher resource utilization. In this *co-location attack* setting, the attacker can interact with the hardware directly through their own programs. Consequently, instead of relying on coarse-grained, end-to-end side-channel measurements, the attacker can directly monitor the victim program's micro-architectural resource usage. By observing the appropriate hardware resource in a precise manner, the measured side effect establishes a much more accurate correlation with the

victim's secret information. Furthermore, direct interaction with the victim is not necessary for the co-location attack setting. This work presents mitigation methods applicable to both settings, with a primary emphasis on the co-location setting since this scenario presents a clear view of how different hardware resources function as side channels and enable information leakage.

Over the years, countless micro-architectural side-channel attacks have been developed to exploit weaknesses in the design of various components in current processors for leaking sensitive information. To implement effective countermeasures, it is crucial to understand each individual attack as well as the underlying principles behind all attacks. Therefore, in this section, we further classify co-location-based attacks into two categories: classic (non-speculative) side-channel attacks and speculative side-channel attacks. We highlight the relationship between these two categories, the individual attacks they encompass, and the common mitigations employed against these attacks. Inspired by the prior work DAWG [55], we introduce abstract models for describing the classic and speculative attack procedures, as illustrated in Figure 2.1.

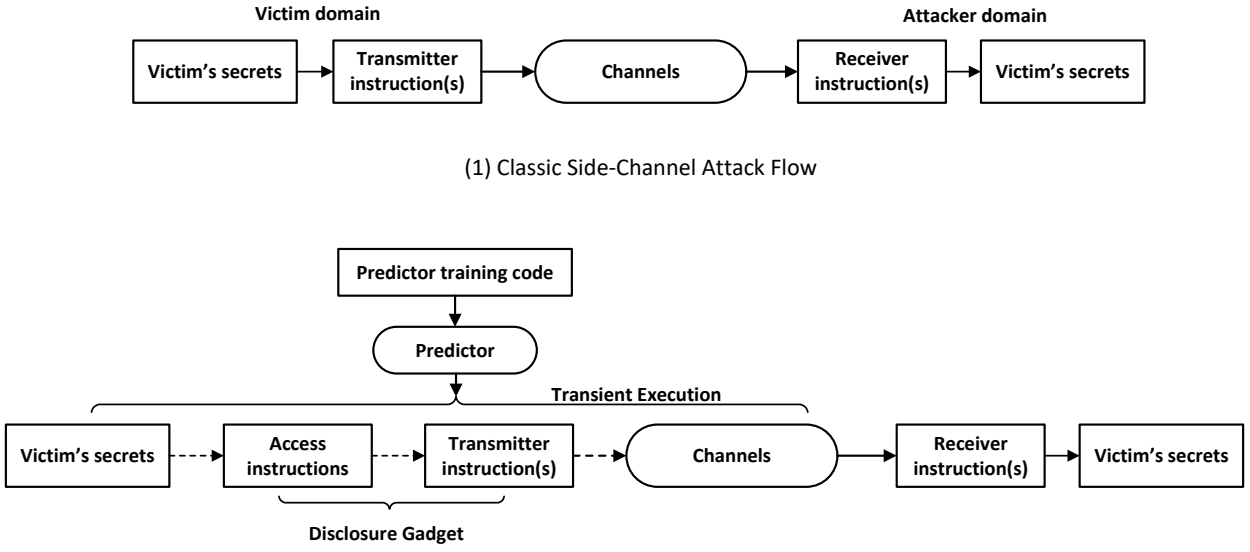


Figure 2.1: Classic and speculative side-channel attack procedures. Regular squares represent software components; rounded-corner squares are hardware components. Dotted arrows denote information flow via transient execution.

The rest of this chapter will describe classic (Section 2.3) and speculative attacks (Section 2.4) in detail, as well as the existing mitigations for both types of attacks (Section 2.5 and 2.6).

2.3 CLASSIC SIDE-CHANNEL ATTACKS

As Figure 2.1 (1) showns, a classic micro-architectural side-channel attack requires the victim program to contain *transmitter instructions* (or *transmitters*) that operate on the victim program secrets. The key to micro-architectural side-channel attacks is that, modern processors contain various hardware resources and optimizations that act as *channels*, such that the execution of transmitter instructions inadvertently transmit secrets through the *channel*. To capture the secret-dependent resource usage happened to those channels, the attacker carefully selects *receiver instructions* (or *receivers*) and *when* and *how* to execute those instructions. The receiver instructions interact with the channel, and generate computational side effects which is precisely correlated with the transmitted information.

Importantly, the transmission of secret information is never the intended purpose of the victim[1]. In the following, we describe classic side-channel attacks based on these three components: channels, transmitter instructions, and receiver instructions.

2.3.1 Channels

Micro-architectural side-channel attacks use data-dependent micro-architectural resources as channels for transmitting information stealthily. Specifically, the inner state of the micro-architectural resources is encoded with the information associated with the victim's program execution. Although this encoding does not affect the victim's program output, the state can be decoded by the attacker who accesses the same hardware and uses specific instructions to monitor the hardware state. Eventually, this enables the attacker to learn the victim's secret information.

To illustrate the concept, take the cache side-channel attack, such as Prime+Probe [5, 34, 56] as an example. In the Prime+Probe attack, the attacker preloads a specific memory address into the cache, therefore accessing to this address is a cache hit unless other addresses sharing the same cache set are loaded and evict this address from the cache. By waiting for the victim's execution and then testing whether this address is cached, the attacker can deduce whether the victim loads addresses into the same set. Based on how different address values are mapped into cache sets, the attacker can further recover a significant portion of the victim's address bits.

Over the years, various cache attacks have been created, in addition to Prime+Probe [10,

[1]Note a side channel is different from a covert channel. In a micro-architectural covert channel, the sender and the receiver are both malicious parties who agree to use a micro-architectural resource to transmit data. Although the setting is similar, a side channel attack assumes a victim program that is beyond the attacker's control.

15, 35, 57, 58, 59, 60]. Many other micro-architectural structures that are exploited in similar manners, such as branch predictors [17, 18, 19, 20, 61], instruction decoders [62], micro-op caches [21], instruction scheduler [63], execution units [22, 64, 65, 66], Translation-Lookaside Buffers (TLBs) [16], data prefetchers [23, 24], DRAMs [67, 68], and many more. These channels can be further characterized by three factors:

Channel Duration

Based on how long the transmitted information remains inside the channel, we can roughly characterize micro-architectural side channels into two types, based on the duration:

- *Ephemeral*: Ephemeral channels are micro-architectural resources that exhibit observable data-dependent usage only during the instruction's execution. For instance, many execution units, such as multiplier [64, 65] and floating-point units [22] have data-dependent behavior (e.g., timing), which allows a side-channel attack to infer knowledge about the operand value based on their behavior. However, the behavior depends only on the operand of the currently executed instruction, not on the previously executed ones. Several other attack vectors, such as execution port contention [66] and cache bank conflict [69], are also ephemeral channels. Since the leakage from ephemeral channels is present for a short duration during the instruction's execution, it typically imposes strict requirements on the attacker, such as close proximity to the victim, and concurrent execution with the victim.

- *Persistent*: Persistent channels maintain the date-dependent effects of instructions even after they have retired, thus impacting instructions executed much later in the pipeline. One prominent example of a persistent channel is the processor cache: accessing the cache causes an irreversible change to the cache state, including the eviction of existing cache lines. Subsequent memory accesses to the evicted line will experience cache misses and longer access latency, from which one can infer the confidential memory access patterns. Secret information transmitted through persistent channels persists for an extended duration compared to ephemeral channels, which significantly relaxes the attacker's requirements. For instance, if the secret-dependent micro-architectural state change persists throughout a process context switch, the attack will not necessitate concurrent execution. However, certain resources such as TLB in many processors perform state sanitization when transitioning between different security domains, thereby mitigating the advantage of persistent channels.

Channel Location

When the attacker has only end-to-end measurements of the computational side effect of the victim's computation, e.g., through a remote timing attack, the physical location of the channel within a processor becomes irrelevant in terms of the attacker's proximity. However, in the co-location attack setting where the attacker typically needs to measure the channel state, the channel placement becomes crucial as it directly affects its exploitability. For example, several *intra-core* channels, like the branch predictor [19, 20, 61], micro-op cache [21, 70], and load-store unit [71], reside within the core pipeline. Therefore the attacker must be scheduled to the same core as the target victim program to observe the data-dependent usage of those channels.

Additionally, there are also micro-architectural resources shared between processor cores, such as the shared cache [7, 11, 15, 34, 35, 56, 57, 58, 72, 73], shared directory [60, 74], and on-chip interconnect [75, 76, 77]. By utilizing these *inter-core* resources as channels, the attacker is not limited to only the victim's core, and can instead launch the attack from other cores, which significantly improves the practicality and the effectiveness of the side-channel attack.

The requirement of the attacker's location can be further relaxed by *cross-processor* channels [78], which enables side-channel leakage across processors.

Channel Information

While various micro-architectural resources exhibit data-dependent behavior and serve as channels, they transmit different types of information related to the executed instructions. The information about the instruction *type* is naturally leaked by what execution units are used for the instruction, as well as other characteristics about the instruction itself, such as the execution timing, and whether or not it accesses the memory subsystem. Although the instruction type is typically not considered confidential, as we will explain next in Section 2.3.2, it can reveal secret control-flow decisions that lead to the execution of the current instruction.

The primary type of information exposed through the execution of instructions pertains to the *operand values* of the instructions. For instance, information about the source operand of floating-point instructions can be leaked by the latency of their execution [22]; the address operand of memory loads can be partially leaked via the accessed cache set or bank [5, 69]. The specific information leaked about the instruction's operand depends on the hardware implementation. In Prime+Probe attacks, for example, only the cache set bits from the

address operand of memory instructions are leaked. However, even the leakage of a limited number of operand bits can pose a significant threat to real-world applications, such as those involving cryptographic algorithms [79].

Another significant type of information that is commonly leaked through various identified channels is the *instruction's address*, also referred to as the *Program Counter (PC)* value [80]. Several hardware structures, including the instruction cache, branch predictors, micro-op cache, and hardware prefetchers, are directly accessed and updated based on the instruction's address as part of their functionality. This enables the side-channel attack to learn information regarding the instruction address based on their behavior [4, 17, 18, 19, 20, 21, 24, 70, 81, 82, 83, 84]. As later described in Section 2.3.2, acquiring knowledge about instruction addresses assists the attacker in deducing the program control flow, which contains sensitive information in many applications, such as cryptographic codes [33].

2.3.2 Transmitters

As shown in Figure 2.1, a classic side-channel leakage starts with the victim's transmitter instructions operating using the secret information. These transmitter instructions will incur data-dependent behaviors to the exploited channels, which are later observed by the attacker. The transmitter instructions determine how the side-channel attack should be conducted and what secret information is collected through the attack. Therefore, information about the target transmitter instructions in the victim program is generally required for launching classic side-channel attacks.

Transmitter instructions are related to secret data through data flow or control flow. In other words, the transmitter instructions are either data-dependent on the secret, or control-dependent on the secret. We now use the following example in Algorithm 2.1 to explain these two kinds of transmitters.

Algorithm 2.1: A pseudocode example of the square-and-multiple algorithm for modular exponentiation, which is used by the RSA algorithm. Secret data are marked in red.

Input: [**Public**]: `base`, `modulus`; [**Private**]: `exponent`

```
1 for bit in exponent do
2 |   output ← output^2 % modulus
3 |   if bit == 1 // data-dependence
4 |    then
5 |   |   output ← output * base % modulus // control-dependence
6 |   end
7 end
```

- *Data-dependence:* A transmitter instruction that is data-dependent on the secret has its source operands data-dependent on the secret. Information about the secret is leaked when this type of transmitter instruction interacts with channels that leak operand values. For example, consider line 3 of Algorithm 2.1. The condition of the branch instruction from this line has a clear data dependence of `bit`, which corresponds to bits of the private `component`. The branch instruction's execution requires branch predictor, which is a common channel for leaking branch predicate values [17, 18, 19, 20, 61].

- *Control-dependence:* For transmitter instructions that are control-dependent on the secret, the fact that they execute or not is decided by the secret. For instance, line 5 of Algorithm 2.1 has control-dependence of the secret `exponent`. By observing whether the instructions corresponding to line 5 execute, the attacker could learn each `bit` value. Note that in practice the attacker has various channel options for exploiting transmitters with control-dependence of secrets. For example, assume that a branch with a secret condition has two sides of different types of operations, or the same type of operations but different operand values. In those cases, the attacker can utilize specific channels for differentiating instruction types or operands, as described above. A more straightforward approach is to leverage channels that leak instruction addresses, such as the instruction cache [20, 81, 82, 83]., branch predictors [17, 18, 19].

2.3.3 Receivers

In contrast to attacks like memory corruption attacks [13] where the attacker can conveniently read the victim's secret, side-channel attackers rely on carefully crafted receiver instructions to capture the victim's computational side effects. Although these side effects are not the actual target secret, they can be utilized to deduce the target secret through analysis.

The choice of receiver instructions is straightforward in remote attacks: the attacker employs primitives for measuring the side effect during their interactions with the victim. In a typical remote-timing attack, the attacker records the wall clock time when invoking and receiving responses from the vulnerable victim routine. The attacker would then use complex analysis techniques to extract the target secret [85]. The specific analysis methods utilized by the side-channel attackers are beyond the scope of this thesis.

In scenarios when the attacker co-locates with the victim, the attacker must include instructions in their receivers that use the intended hardware resources (channels). More importantly, these instructions must be properly parameterized so that their execution is

influenced by the victim's transmitters through the channel behavior. As an example, in Prime+Probe cache attacks, the attacker aims to observe whether the victim accesses a specific cache set. Therefore, the attacker would normally incorporate memory loads in their receiver instructions to probe the same cache set. The attacker also includes measurement primitives to observe the execution of those instructions, such as timing the memory accesses in Prime+Probe attacks.

2.4 SPECULATIVE SIDE-CHANNEL ATTACKS

The surge of speculative side-channel attacks (also referred to as speculative execution attacks), notably led by Spectre [4] and Meltdown [3] has opened up a new chapter of micro-architectural side-channel attacks, and greatly advanced our understanding of this type of attack. Speculative side-channel attacks overcome a major limitation of classic side-channel attacks where the attacker can only exfiltrate victim secrets that are consumed by the victim's transmitter instructions in a valid (retired) execution. In speculative side-channel attacks, the attacker exploits the various hardware speculation mechanisms in today's high-performance processors, to *transiently* execute instructions (which are invalid execution due to mis-prediction). While speculative execution never affects program correctness as the invalid execution is eventually squashed once the misprediction is detected, it introduces major security problems: the attacker can leverage speculative execution to allow transmitter instructions to consume secret data that it would not consume in normal, non-speculative execution. Even though the execution is ultimately invalidated, the transient execution of transmitter instruction facilitates secret-dependent utilization of channels (such as processor caches), thereby leaking the secrets to the potential receivers. Since prediction and speculation are pure hardware optimizations, it is impossible for programmers to reason what data can be speculatively accessed and leaked.

Figure 2.1 (2) illustrates the process of speculative side-channel attacks. Speculative side-channel attacks leverage the classic side-channel attack flow "transmitter → channel → receiver" for transmitting the victim's secrets to the attacker. On the other hand, these attacks exploit hardware prediction and speculative execution to realize "secret → transmitter" relationships that never appear in the normal program semantics.

Algorithm 2.2: A pseudocode example of the Spectre V1 [4] example, which has a branch followed by the access and transmitter instructions. secret data are marked in red.

Input: public array: `publicData`, array index: `Index`;
secret data array: `secretData`

```
1 cond ← ...
2 if cond then                    // a long-latency branch for transient execution
3 |   secret ← secretData[Index] // access instruction
4 |   temp  ← publicData[secret] // transmitter instruction
5 end
```

As a demonstration, consider the original Spectre V1 attack, illustrated by Algorithm 2.2. The vulnerable code contains a branch, which will be mispredicted by the branch predictor after the attacker trains the predictor. The branch condition, `cond`, may take a significant amount of cycles to compute, thus giving sufficient time for transiently executing the instructions inside the branch block. The access instruction on line 3 transiently accesses a secret, and the subsequent transmitter instruction uses the secret value as the memory operand, therefore leaking the secret through channels such as cache [4, 86] or TLB [87].

A speculative side-channel attack typically requires three extra components beyond a classic side-channel attack: *the predictor training code*, *a predictor*, and *a disclosure gadget* which includes the combination of access and transmitter instructions. A previous survey work by Xiong et al. uses a similar framework for classifying speculative side-channel attacks [88]. We now discuss all three components in details.

2.4.1 Predictor

There are various prediction mechanisms in modern processors that have been exploited for speculative side-channel attack purposes, which include:

- *Branch prediction:* Branch prediction is the most common hardware optimization that involves multiple hardware structures such as pattern history tables (PHTs, also called directional predictors), branch target buffers (BTBs), and return stack buffers (RSBs, also called return address stacks or RASs). All these structures have been the target of different attacks for generating misprediction. For example, the PHT has been exploited in the original Spectre V1 attack [4] and its derivatives [89, 90]; the BTB has been exploited in Spectre V2 attack [4] and its derivatives [91]; the RSB has been exploited in [92, 93].

- *Memory address dependency prediction:* Memory address dependency prediction, also

referred to as memory disambiguation, is another prediction mechanism widely used by high-performance processors to predict the dependency between memory loads and stores when their address operands are unavailable. This prediction mechanism allows speculative data flow between stores and loads, which motivates several speculative attacks such as Spectre V4 [84, 94, 95].

- *Hardware prefetching:* Differing from branch and memory dependency prediction, hardware prefetching does not lead to software instructions being speculatively executed. Instead, an activated hardware prefetcher will perform memory accesses that do not correspond to any software instructions. Attacks such as Augury [23] have found that in some processors, a hardware prefetcher can also be trained in a similar way as branch predictors and result in a similar leakage as the Spectre attack.

- *Exception:* Several existing attacks, such as the Meltdown attack [3] and ForeShadow [96, 97], are built on the observation that processors typically resolve exceptions at the retire stage. This allows the subsequent instructions to exploit the fault and perform operations that would otherwise be prohibited before the exception is resolved. So even though exceptions are fundamentally different from hardware predictions, it has a similar effect as predictions in terms of speculative execution attacks: code dependent on the instruction triggering the exception gets transiently executed.

2.4.2 Predictor Training Code

As Figure 2.1(1) shows, a successful speculative side-channel attack starts by utilizing predictor training code for training the hardware predictor into a state which produces desired misprediction for the attack. Depending on the targeted predictor structure, the predictor training code includes different instructions. For example, conditional branches are used to train PHTs, while jump instructions are employed for BTBs. In most attacks, the attacker exploits the sharing nature of hardware predictors to train the predictors using predictor training code within their own programs [3, 86, 92, 93, 95, 96, 97, 98, 99, 100, 101, 102]. However, this approach necessitates the attacker and the victim being co-located on the same processor core, and the exploited hardware predictor cannot adopt mechanisms that isolate the predictor state between the attacker and the victim. If these requirements are not satisfied, the attacker may resort to leveraging predictor training code available within the victim software routine, provided they can directly invoke such routines [4, 94]. Lastly, the predictor training code is not necessary for a successful speculative attack; the predictor state may correspond to a well-trained predictor state by pure coincidence.

2.4.3 Disclosure Gadget

Another key step in speculative execution attacks is the transient execution of the disclosure gadget, which includes access and transmitter instructions. The access instruction should transiently access the secret data by bringing it into the core pipeline, and a subsequent transmitter instruction consumes the secret and generates secret-dependent channel usage, such as cache access or data-dependent execution timing. Because the leakage must happen transiently (otherwise those instructions will be squashed before the secret-dependent channel usage is made), the attacker must create a sufficiently large *speculation window* for covering those instructions. For instance, in the Spectre Attack shown in Algorithm 2.2 where the access and transmitter instructions follow a branch, the attacker must induce a long latency for determining the branch condition, thus the access and transmitter instructions can finish their execution, and leak the secret through cache before the branch misprediction is identified. The transmission of secret information from the channel to the attacker through receiver instructions is identical to classic side-channel attacks.

In Spectre V1 and other variants, attackers exploit the access and transmitter instructions embedded within the victim code to extract the victim's secrets. However, many speculative attacks, represented by Meltdown, have showcased how access and transmitter instructions inside the attacker's code can also result in a similar effect [3, 96, 97, 99, 100, 101, 102, 103]. Ideally, processors should enforce strict isolation between distinct security domains to prevent attackers from accessing the victim's program data. Unfortunately, these attacks have found that real processors have implemented various optimizations that could temporarily violate the isolation during transient execution, which enables the attacker's access instructions to momentarily access and transmit the victim's data. For example, the original Meltdown attack demonstrated that in modern Intel processors, when a (potentially malicious) user program accesses kernel memory and triggers an exception, before the exception is handled at the retire stage of the faulting instruction, subsequent instructions can transiently consume the accessed kernel data, causing leakage of kernel data through classic side channels. Subsequent attacks like ForeShadow [96, 97], MDS [100, 101, 102], and others [103, 104, 105] exploit different conditions when speculative execution relaxes the isolation policies, such as when speculative data is in the L1 cache or in various on-chip buffers.

A speculative side-channel attack can also include a combination of a non-speculative access instruction and a speculative transmitter instruction as the disclosure gadget. However, this attack approach is relatively uncommon due to a major drawback compared to existing attacks with speculative access instruction. With a transiently-executed access instruction, data that would otherwise be inaccessible during normal execution can be transiently fetched

and leaked, which significantly enlarges the attack surface. This has been demonstrated originally by Spectre [4] and Meltdown [3] as they were able to leak the entire kernel memory from userspace.

2.5 MITIGATIONS FOR CLASSIC SIDE-CHANNEL ATTACKS

Since the appearance of classic side-channel attacks such as cache attacks, researchers have proposed various software and hardware countermeasures to mitigate those attacks, which is the focus of this section.

2.5.1 Software Mitigations

Software mitigations cannot directly fix the vulnerable micro-architectural resources (channels). Instead, they focus on software elements that constitute the classic side-channel leakage, which are the transmitter instructions and the receiver instructions.

Addressing Transmitters

Classic side-channel attacks start with the victim's transmitter instructions consuming secret data as their input operands, and the execution of the transmitters creates micro-architectural resource utilization that exhibits a correlation with the secret data. The strategy that software developers adopt to address the risk posed by transmitters is to introduce redundant program execution which obfuscates the measurement of channel usage.

For instance, consider the two scenarios depicted in Figure 2.2. In Algorithm 2.3, the memory access operation uses secret data in its address calculation (line 4), rendering it vulnerable to side channels that can divulge address access patterns, such as cache attacks [25]. To conceal the real target (`secretIndex`), as Algorithm 2.3 shows, one common approach is to perform a memory scan across all potential targets (all elements of `publicData` in the case of Algorithm 2.3) and only use the right data for subsequent computation [106, 107, 108]. This simple approach ensures that the memory access pattern no longer reveals information about the `secretIndex`. However, it comes at a significant performance cost of $O(N)$ bandwidth cost for each access. One may assume that the attacker has no observability over the cache line offset bits, which holds for most cache attacks, and only access addresses that are multiples of cache lines away from the target address. This strategy is further challenged by more subtle attacks which are capable of leaking low-order address bits [66]. To address all these attacks that collectively reveal different parts of memory addresses, researchers resort to more

Algorithm 2.3: Memory access with a secret address operand. `secretIndex` could be leaked via cache side channels.

Input: array `publicData` of size `N`;
secret index `secretIndex`
Output: returned value `tmp`

```
1 dummy ← publicData[0];
2 dummy ← publicData[1];
3 ...  ...;
4 tmp ← publicData[secretIndex] ;
5 ...  ...;
6 dummy ← publicData[N-2];
7 dummy ← publicData[N-1];
```

Algorithm 2.4: Branch with a secret predicate. `cond` could be leaked via observing the execution unit usage.

Input: integers `a`, `b`, `c`
secret branch condition `cond`
Output: returned value `tmp`

```
1 if cond then
2 |   tmp ← a * b;
3 |   tmp ← tmp + c
4 else
5 |   tmp ← a * b;
6 |   tmp ← tmp + 0
7 end
```

Figure 2.2: Two examples of how secret is leaked by transmitters taking secret input operands through micro-architectural side channels. Programmers may address this leakage with redundant operations, marked in blue, to obfuscate the data-dependent micro-architectural behavior.

strict security policies—such as memory trace obliviousness [109]—which require the memory address pattern to be independent (or oblivious) to any secret information. In fact, the problem of performing memory accesses with confidential addresses motivates the creation of Oblivious RAM [110, 111, 112, 113], which strives for achieving sub-linear bandwidth cost with the aid of some adversary-invisible data structures (such as position map in [114]). Nonetheless, these methods are rarely used in practice due to their performance degradation.

Algorithm 2.4 represents another common scenario when the secret control-flow decision is leaked through branches or other control-flow structures. In the original form, the *then* path includes one more addition than the *else* path, therefore a side-channel attacker who is capable of measuring the execution time of the branch body can easily deduce the branch decision. A simple countermeasure against this attack approach is to add a dummy addition (line 6 to balance both sides and ensure identical execution time. However, the fixed code remains vulnerable to other side-channel attacks, such as those that can observe the address sequence of executed instructions (at different granularity, such as page [115] or cache line [116]). The attacker may even learn the conditional branch direction directly by monitoring the behavior of the hardware branch execution unit [19, 20, 61]. To mitigate those attacks, software developers may adopt more aggressive code transformation techniques, such as restricting the secret-dependent control flow to the minimal observation granularity (cache/page) [117]. In high-security applications such as cryptographic libraries, a common practice is to eliminate secret-dependent branches by performing the worst-case amount of work [118].

The endeavors of programmers in search of side-channel-free programming techniques have

led to data-oblivious programming [22, 30, 106, 107, 114, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129]. Data-oblivious programming, also known as "constant-time programming" or "writing programs as circuits", aims to thwart micro-architectural side channels by ensuring secret-independent hardware resource usage. It provides a formal framework for adding redundant operations in code vulnerable to side channels, such as Algorithm 2.3 and 2.4. Despite its strong security guarantee, its significant performance overhead still impedes its practical adoption in real-world applications [118]. Moreover, the emergence of speculative execution attacks has raised concerns about the security of data-oblivious programming, which was originally designed to address classic side-channel attacks without considering out-of-order and speculative execution [30, 130, 131].

Addressing Receivers

Preventing attackers' receiver instructions from measuring secret information through the channel provides another direction for countering micro-architectural side-channel attacks. Since any micro-architectural side-channel attack, regardless of the attacker's proximity to the victim, relies on analyzing the correlation between the measured computational side effects (e.g., timing) and target secrets, introducing inaccuracies in the measurement is considered an effective mitigation applicable to any micro-architectural side-channel attacks, including speculative attacks that still utilize classic side channels for secret transmission. Since most side-channel attacks leverage high-resolution timers, possible mitigations can be reducing the timing resolution, fuzzing, or completely disabling the timing source [132, 133, 134, 135, 136]. However, injecting noise or completely disabling the timing source is impractical in many circumstances, and will result in significant performance degradation [137]. Furthermore, subsequent attacks have been proposed to completely eliminate the side-channel attack's dependence on timers [36, 138, 139, 140, 141].

Considering that many micro-architectural attacks assume the attacker's physical proximity to the victim for the direct observation of the hardware resource usage, a straightforward approach is to assign the victim application exclusive ownership of an isolated core (to mitigate intra-core attacks) [142] or even an isolated processor (to mitigate cross-core attacks). However, as the current trend of building processors that are capable of running more concurrent workloads continues, these mitigation strategies come at the expense of higher performance costs. Researchers have also proposed to sanitize the victim's secret-dependent hardware state change, such as flushing the cache [5, 143] and the branch predictor state [144] when switching between security domains. While these solutions result in less performance impact, they offer a trade-off in terms of weaker security guarantees.

2.5.2 Hardware Mitigations

Micro-architectural side-channel attacks exploit security flaws in the hardware design, therefore incorporating defense strategies during the hardware design process should address the root cause of micro-architectural side channels. Additionally, hardware mitigations generally provide better software compatibility due to less involvement by the software. However, few of those hardware mitigations are implemented in real-world processors, since they usually introduce significant design complexity as well as non-negligible performance cost.

Secure Hardware Units

Most current hardware mitigations focus on hardening individual micro-architectural resources that induce side-channel attacks, such as caches, TLBs, branch predictors, etc. These resources are commonly shared by programs from distinct security domains, allowing potentially malicious programs to learn secret information from the hardware resource usage pattern of the victim program. Based on the cause of the information leakage, most existing solutions aim for achieving *non-interference*: the execution of a program cannot create secret-dependent hardware resource usage that can be observed by the execution of another program from a different security domain. To achieve non-interference, these solutions typically follow two different methodologies (some may aim at the best of both worlds):

- *Isolation-based.* Isolation-based mitigations give the victim exclusive ownership of a portion of the vulnerable hardware resource, therefore the victim's usage of the resource cannot be observed by other tenants sharing the hardware. This strategy was first applied to the processor cache, as the researchers suggested assigning different cache ways to software from different security domains (e.g., VMs) [145, 146]. However, static partitioning faces scalability limitations due to the finite hardware resources and the potentially infinite number of tenants. Moreover, static partitioning results in poor resource utilization as the resource demand of different programs can vary. To address these challenges, most solutions adopt dynamic and fine-grained (e.g., cache-line level) cache partitioning approaches to alleviate the performance impact of the protection mechanism [55, 147, 148, 149, 150, 151, 152, 153, 154]. Specifically, these solutions follow the key observation that cache side-channel attackers must identify the victim's accesses to critical cache lines that depend on the victim's secrets through interference with those accesses (e.g., through cache evictions [5] and flushes [15]). By introducing new cache replacement logic (some even with assistance from the operating

system [154, 155], these solutions ensure that critical cache lines remain unaffected by other software, effectively isolating them from interference. The isolation-based approach is also applied to other micro-architectural resources such as TLB [26], cache directory [156], and branch predictor [157, 158].

- *Randomization-based.* Micro-architectural side-channel attacks are possible only when the hardware activities observed by the attacker exhibit a correlation with the victim's secret. To counteract this, researchers propose randomization-based solutions, which introduce randomness in the attacker-observable hardware activities with the goal of eliminating the correlation. The initial randomization-based cache defenses are designed based on fully-associative caches, which render the popular cache attacks like Prime+Probe and Flush+Reload ineffective due to the absence of cache sets [147, 159]. Since the use of the fully-associative cache introduces significant performance and complexity overhead, more recent randomization-based caches are constructed on top of the set-associative cache, by randomizing the mapping between memory addresses to cache sets [160, 161, 162, 163, 164]. Researchers have also proposed similar approaches for other hardware resources [26, 165].

In general, these mitigations offer promising security assurances against well-known attacks, such as Prime+Probe [5] and Flush+Reload [15] in terms of cache attacks. In addition, the performance overhead associated with these mitigations has been steadily reduced as new solutions are proposed. On the other hand, such mitigations are point defenses that cannot address micro-architectural side channels comprehensively. For instance, most randomization-based cache mitigations block the leakage of cache set index bits, while the sub-cache-line address bits remain vulnerable to other subtle side-channel attacks [69].

Secure Co-processors

Aside from running victim programs on shared hardware with vulnerable hardware resources fixed, other solutions propose to run victim programs exclusively on secure co-processors [166, 167]. Attackers in those cases only have coarse-grain observability that captures the processor's external pin activities. The internal state of the co-processor remains hidden from external programs, and is securely sanitized during context switches. Yet, these proposals assume simple processor pipelines and scheduling (e.g., one process per chip at a time). In contrast, our goal is to show how *existing* high-performance machines can be retrofitted to simultaneously execute extensive workloads encompassing both sensitive and non-sensitive programs, all while maintaining the high parallelism provided by modern processors.

2.6 MITIGATIONS FOR SPECULATIVE SIDE-CHANNEL ATTACKS

Since the appearance of speculative side-channel attacks, researchers have developed many defense schemes to counteract those attacks. We now brief the previously proposed speculative side-channel mitigations.

As mentioned in Section 2.4, speculative attacks still rely on the classic side channels for transmitting the secret over to the attacker. However, many existing classic side-channel mitigations are ineffective in defending against speculative attacks. First, consider software mitigations that apply software transformations to the victim code for constant micro-architectural resource usage, e.g., using data-oblivious programming. However, due to speculative execution, the hardware may execute the victim program in an unexpected way, or even generate software-agnostic hardware activities (such as hardware prefetching) [23]. In addition, in practice, many parts of the victim program may not be covered by the software mitigations, such as those legacy or third-party libraries, and any code vulnerable to side channels in those parts can be exploited by speculative attacks even if those codes do not touch any user secrets in non-speculative execution.

Most hardware-based classic side-channel mitigations also fail to address speculative attacks. As mentioned in Section 2.5.2, the security of most defense schemes comes from achieving the non-interference property between different programs sharing the same hardware. In this leakage model, the victim program must actively execute to enable any form of side-channel leakage. However, many recent speculative attacks [3, 23, 96, 97, 100, 101, 102, 103] have shown how the attacker can leverage speculative execution to access and transmit victim secret data *at rest*, i.e. secret data that is never touched by the victim. So even if the victim never executes, which ensures that any classic side-channel attacks are impossible to carry out, its secret can still be exposed to speculative attacks.

A simple solution to eliminate speculative attacks is to disable speculation, for instance, by inserting speculation barriers after branches. However, this approach completely erases the performance advantage of speculation. Next, we showcase prior works on mitigating speculative attacks by introducing new security-related mechanisms to speculative execution. Similar to the previous surveys conducted by Xiong et al. [88] and by Hu et al. [168], we categorize these works based on which step of the speculative side-channel attack flow they target.

2.6.1 Targeting Malicious Predictions

Speculative execution attacks start from the predictor being properly trained, which leads to a misprediction and eventually the speculative leakage of secrets. To ensure the attacker's training of the predictors does not result in a misprediction favoring the speculative attack, the first defense strategy is to separate the victim's predictor state from the attacker. For example, Intel/ARM both introduce security features to their branch predictor structures [169, 170] to ensure that the jump target in the branch target buffer that is updated by a program cannot be used for making predictions for programs belonging to different security domains. Researchers also proposed similar isolation mechanisms for other branch predictor structures [158, 171].

While this strategy prevents the malicious predictor training from different security domains, it fails to mitigate attacks like the original Spectre V1 [4] and NetSpectre [172] where the predictor is trained by the victim program itself.

2.6.2 Targeting Speculative Access Instructions

In speculative attacks, it is impossible to leak the secret without first accessing the secrets from the memory. This leads to the second mitigation strategy, which is to enforce specific speculation-aware access control policies to the secret data. For instance, the Meltdown attack is possible because the kernel memory can be accessed from the userspace during speculative execution. Hence, an immediate countermeasure against Meltdown, dubbed kernel address space isolation (or kernel page-table isolation), completely separates kernel address space from the user address space such that kernel memory is inaccessible from the userland even during hardware speculation [173]. Browsers also adopted this strategy and developed site isolation [174], which segregates browser sites into different address spaces.

However, completely separating programs into mutually exclusive address spaces might be overly conservative when the victim program only possesses limited private data and can benefit from sharing public data. In such scenarios, schemes that enforce speculative access control at a finer granularity, such as page-granularity as demonstrated by Intel's speculative-access protected memory (SPAM) [175], could be more desirable. Nonetheless, such schemes often rely on manual labeling of program secrets, which is often infeasible in real-world applications.

2.6.3 Targeting Speculative Transmitter Instructions

Blocking speculative accesses to secret data is a sufficient, but unnecessary mitigation strategy, as the secret also needs to be consumed by the transmitter instructions. Many defenses therefore choose to block the execution of transmitter instructions instead of access instructions [176, 177, 178, 179, 180, 181]. These defenses track data returned by speculative access instructions, and block the execution of any speculative transmitters that use those data or the dependent data as inputs. While most of them assume any data returned by speculative access instructions as secrets, some schemes (such as SpectreGuard [177] and ConTExT [179] provide extra software support for programmers to mark data as private. Alongside the hardware schemes mentioned, Speculative Load Hardening (SLH) [182] employs similar strategy at software level, with compilers introducing extra data dependence between transmitter instructions and the prior branches.

2.6.4 Targeting Channels

Another major strategy adopted by many existing mitigations is to intercept the speculative leakage at the channel. To achieve this, any micro-architectural resource (such as cache) cannot exhibit secret-dependent usage resulting from transient execution that can be eventually observed by the attacker. Most solutions focus specifically on the processor cache since it is the primary option for transmitting speculative secrets. InvisiSpec [183], SafeSpec [184], and MuonTrap [185] are based on the idea of using dedicated buffers for filtering any speculative updates to caches, such that the cache state remains unchanged from the attacker's perspective. CleanupSpec [186], on the other hand, rollbacks any updates to the cache once the processor recognizes that those updates are from squashed memory accesses.

Other mitigations follow a major observation that in most speculative attacks, attackers are from a different security domain, and such attackers can only observe speculative cache state changes if the state changes happen to the shared cache, or the private cache state change remains across context switches. Therefore, in SelectiveDelay [187], only cache accesses that miss L1 are delayed. Other defense frameworks, such as DAWG [55], MI6 [188] and Ironhide [189] combine resource partitioning with micro-architectural resource flush to ensure speculative state updates cannot be directly observed from other security domains.

A major drawback of the above schemes that target channels is that, they still create secret-dependent micro-architectural state usage. For instance, despite using special buffers to filter speculative cache updates [183, 184, 185], the cache access still reaches the cache level where the address is present, which leads to speculative leakage when the address itself

depends on speculative data.

CHAPTER 3: DATA-OBLIVIOUS ISA

This chapter presents a comprehensive solution for blocking micro-architectural side channels called Data-Oblivious ISA extension (OISA). The current effort of addressing micro-architectural side channels comprehensively, by writing program data-obliviously, is insecure and inefficient due to the absence of security guarantees from the hardware. OISA, for the first time, includes side-channel-specific security specifications at the ISA level, facilitating communication between hardware and software regarding side-channel protections. Aside from introducing OISA design principles, this work also proposes a hardware prototype implementing an instance of OISA including safe performance optimizations, and provides a formal analysis showing that programmers can write provable and efficient side-channel-free (covering both traditional and speculative side channels) programs with OISA.

3.1 INTRODUCTION

With the rise of cloud computing and internet services, *digital or microarchitectural side channel attacks* [107] have emerged as a central privacy threat. These attacks exploit how victim and adversarial programs share hardware/virtual resources on shared remote servers (e.g., an amazon EC2 cloud). Simply by co-locating to the same platform, researchers have shown how attackers can learn victim program secrets through the victim's virtual memory accesses [115, 190], hardware memory accesses [5, 15], branch predictor usage [20, 191], arithmetic pipeline usage [22, 66, 192], speculative execution [4, 193] and more. Given the many avenues to launch an attack, it is paramount for researchers to explore holistic and efficient defensive strategies.

Recently, there has been a surge of work that attempts to block *all* digital side channels, on commercial machines, by writing and compiling programs in a *data-oblivious* fashion (e.g., [22, 106, 107, 114, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129]). Data-oblivious code, a.k.a. "constant time" or "running programs as circuits," blocks side channels by disallowing private data-dependent control flow. Figure 3.1 and 3.2 give an example. Figure 3.1 leaks private information over microarchitectural side channels—namely, program execution time (the 'if-taken' case executes more instructions) and memory footprint (if `x` and `y` touch different lines in cache). To block these leakages, a data-oblivious program will evaluate both sides of the branch as shown in Figure 3.2. A ternary operator—e.g., implemented as the x86 `cmov` instruction or bitwise operations—chooses the correct final result (Figure 3.2, Line 4). Since executing each side of the branch is independent of the secret, and the ternary operator

does work independently of the secret, running the code data-obliviously does not leak the secret.

```
1  x = 0, y = 64
2  if (secret)
3    x = y
4  z = Memory[x]
```

Figure 3.1: Non-oblivious (insecure) code. The word secret denotes private data.

```
1  x = 0, y = 64
2  z = Memory[x]
3  tmp = Memory[y]
4  z = secret ? tmp : z
```

Figure 3.2: Equivalent data-oblivious code. The word secret denotes private data.

Despite the promise of data-oblivious programs to block side channels, future progress faces two key challenges.

Security First, existing Instruction Set Architectures (ISAs) provide no guarantees that instructions used in data-oblivious codes can block leakages over microarchitectural side channels. For example, if `cmov` (used as the ternary operator in [107, 114, 123, 125]) was ever implemented as the microcode sequence `branch+mov`, the secret condition would leak through branch predictor state and whether hardware speculation results in a squash. Being ISA-invisible, these changes can occur at any time. Case in point, Intel has stated that `cmov`'s behavior w.r.t. speculation may change in future processors ([194], Section 3.2).

Beyond `cmov`, the larger problem is that commercial ISAs such as x86 give engineers significant rope to perform *data-dependent* optimizations during program execution. For example, it is well known that arithmetic units can sometimes take data-dependent time [22, 192]. We provide a comprehensive background on related vulnerabilities in Section 3.3.2. Any of these software-invisible optimizations can undermine the security of prior and future work that attempts to write data-oblivious programs.

Performance Data-oblivious programming usually incurs large performance degradation. The reason, once again, is that data obliviousness does not have ISA-level support. As a result, programmers are forced to use only the simplest instructions to achieve data-oblivious execution, out of fear that other instructions will leak privacy. For example, data-oblivious codes must make two memory accesses in Figure 3.2 out of fear that a single access will reveal the address through the processor cache, or other, side channel. This overhead scales with deeper data-dependent control flow and larger data sizes.

In this work, we tackle both the security and performance aspects of this problem by developing a novel type of ISA extension which we call a *Data-Oblivious ISA* extension

(*OISA*). To our knowledge, this represents the first foundation for writing and executing secure, portable, and performant data-oblivious code on commercial-class (out-of-order, speculative) processors. Our key idea is to explicitly specify security guarantees at the ISA level, while decoupling those guarantees from the implementation details of a particular processor. Specifically, each operand of each instruction is given an ISA-level attribute specifying whether that operand is *Safe* to receive private data. If marked Safe, processor implementations (*microarchitectures*) using that ISA must hide operand-dependent side effects from other parts of the system due to that instruction's execution. Importantly, how protecting Safe operands is implemented is left to the hardware designer, who can devise efficient protections depending on each microarchitecture (e.g., by breaking the instruction into simpler data-oblivious instructions [22] or using hardware partitioning [108] or using cryptographic techniques [114]). In all cases, the programmer works with a simple, portable guarantee.

With these principles, we define a set of instructions that can serve as the foundation for the rich line of ongoing work in data-oblivious programming [22, 106, 107, 114, 119, 120, 123, 124, 125, 126, 127, 128, 129]. Beyond Turing completeness and security, we also want to reduce the performance overhead common with data-oblivious code. To that end, we provide additional instructions that implement efficient *memory oblivious* computation [106, 114] (featuring loads/stores with private addresses). Given the principles above, this extension is conceptually simple: instead of emulating memory obliviousness with dummy memory operations (Figure 3.2), we designate a new load instruction whose address operand is *Safe*, which gives hardware designers the ability to build secure and efficient implementations, e.g., using partitioning, for that specific operation.

To show that our ideas are practical, we prototype all hardware changes needed to support our ISA on top of the RISC-V BOOM processor (for "Berkeley Out-of-Order Machine") [195]. BOOM is the most sophisticated open RISC-V processor, featuring modern performance optimizations such as speculative and out-of-order execution, and is similar to commercial machines that run data-oblivious code today.[2] We evaluate our prototype in terms of hardware area and performance over a range of existing data-oblivious programs (including linear algebra, data structures, and graph traversal). Area-wise, our proposal takes $< 5\%$ the area of the unmodified BOOM processor. Performance-wise, our ISA and hardware implementation provide an $8.8\times/1.7\times$ speedup on small/large data sets, respectively, relative to data-oblivious code running on commodity machines (and with the security and portability

[2]We note that prior work [166, 167, 196] requires the use of discrete co-processors with simple microarchitecture. To match modern cloud deployments, our goal is to support *concurrent* execution of many processes on advanced microarchitectures.

benefits stated before). We also show case studies, where our ISA speeds up constant time AES [197, 198] by 4.4× and the memory oblivious ZeroTrace [114] library by 4.6× to several orders of magnitude, depending on parameters.

In parallel to our hardware prototype, we develop a formal analysis that models an abstract BOOM-class processor (out-of-order, speculative, superscalar), and describe how to map the abstract BOOM to our concrete BOOM prototype. A key insight enabling this analysis is that by applying *local* checks to each instruction as it executes, the analysis/hardware need not be aware of whether each instruction is speculative, executed out-of-order, etc.: the checks performed to maintain security are the same in all cases. Through this formalism, we prove that the ISA provides a basis to satisfy strong security definitions such as non-interference [199] on advanced machines. Importantly, we achieve this result *while* allowing high-performance hardware optimizations (e.g., out-of-order, speculative execution) to remain enabled in the common case and *without* ever requiring hardware flushes to structures such as the cache or branch predictors [19, 96].

Contributions To summarize, this chapter makes the following contributions:

- We present the design principles for data-oblivious ISA extension, as a way to facilitate communication between software and hardware for micro-architectural side-channel protections.
- We design a concrete OISA based on RISC-V ISA, and present a hardware prototype on an out-of-order, speculative processor, based on RISC-V BOOM processor, which supports our OISA. The evaluation shows less than 5% area overhead, 8.8x/1.7x speedup on small/large data sets, respectively, relative to data-oblivious code running on commodity machines.
- We perform a formal analysis showing that the abstract processor supporting OISA fulfills the non-interference property, which is a strong security property allowing for side-channel-free programming.

3.2 THREAT MODEL

We consider the setting where a victim program runs on a shared machine in the presence of adversarial software. The adversary's goal is to learn private data in the victim program through digital side channels. For example, private inputs contributed by another party or

secret program state (e.g., a cryptographic key). The program itself is considered public. We trust the processor hardware and that the victim program is correctly using the OISA.

We defend against two classes of adversaries: supervisor-level (Ring-0) and user-level (Ring-3) software. In both cases, we strive to block digital side channels that could be exploited by the standard Intel SGX adversary used in prior work on data-oblivious programming [106, 107, 114, 122, 123, 124, 125, 128]. This adversary is supervisor-level software that controls when victim threads run, and therefore can monitor/influence the victim's hardware resource utilization (e.g., monitor/prime the cache/branch predictors [4, 5, 19]) at near-perfect resolution (e.g., via [115, 190, 200]). By extension, this adversary can monitor the victim's termination time, and determine when a precise exception [107, 201] or system call [202] occurs. We don't make assumptions on where the victim runs relative to adversarial code (e.g., as an adjacent SMT context, adjacent core, etc.). If the adversary is actually user-level software, our threat model is strictly conservative.[3]

In the case of a supervisor-level adversary, we assume the victim is running within a virtual shielding system, such as an SGX enclave [203, 204], to prevent direct inspection/tampering on victim data. The OISA is orthogonal to which virtual shielding system is used, in the sense that shielded programs can execute oblivious instructions regardless of the exact shielding system implementation. We will therefore only discuss the OISA, independent of the shielding system, for the rest of the paper.

Non-goals Physical side channels (e.g., power [43] or EM [205]) are out of scope. Similar to previous works on data-oblivious programming, we also do not consider the integrity of computation. Integrity relies on orthogonal mechanisms, e.g., traditional or SGX-augmented process/memory isolation.

3.3 DATA-OBLIVIOUS EXECUTION

We now give background on data-oblivious execution and give examples for where prior work on commercial ISAs (e.g., x86) and modern machines (e.g., speculative, out-of-order) is vulnerable to attack.

[3]Note that even user-level adversaries have been shown to be surprisingly powerful in their ability to monitor digital side channels [7].

3.3.1 Security Definition

Data-oblivious execution satisfies computational indistinguishably[4] of program traces, once the trace is projected by an appropriate observability function.

Definition 3.1. (Confidential input privacy). Given a program λ with Public (non-sensitive) input x and Confidential (sensitive) input y, $\mathsf{O}(\mu Arch(\lambda(x,y))) = X = \{X_0, X_1, \ldots, X_M\}$ represents the program's observable execution trace (projected through function O) when running on a processor $\mu Arch$. What information is contained in each X_t (for each time step t) depends on the observability function O. W.l.o.g. we will treat x and y as fixed-size arrays, thus λ can accept an arbitrary number of Public and Confidential inputs. Privacy for the Confidential inputs then requires:

$$\forall x \in Data_P, \quad \forall y, y' \in Data_C : \mathsf{O}(\mu Arch(\lambda(x,y))) \simeq \mathsf{O}(\mu Arch(\lambda(x,y'))) \tag{3.1}$$

where $\simeq$ denotes computational indistinguishability, and $Data_P$ and $Data_C$ denote the space of Public and Confidential inputs, respectively.

We denote Definition 3.1 parameterized by an observability function O and a specific microarchitecture $\mu Arch$ as $Oblivious[\mathsf{O}, \mu Arch]$, dropping $\mu Arch$ when it is clear which microarchitecture we are referring to.

Existing data-oblivious programs written for commodity machines demand a rich observability function that reveals fine-grain details about processor state [106, 107, 114, 119, 120, 123, 124, 125, 126, 127, 128, 129]. The reason is that machines today are shared, and adversaries from Section 3.2 can monitor internal activity such as caches and pipeline behavior. It is therefore useful to define the most conservative observability function that could apply to adversaries from Section 3.2:

Definition 3.2. (BitCycle observability: Security labels at bit-level spatial granularity, cycle-level temporal granularity). Let $S_t = \{0,1\}^N$ denote the processor state during clock cycle t, where the state includes all on-chip storage (e.g., flip-flops, SRAM). S_t^i denotes the value of the i-th bit in cycle t. Given a program execution $\lambda(x,y)$, $\mathsf{BitCycle}(\mu Arch(\lambda(x,y))) = X = \{X_0, X_1, \ldots, X_M\}$ where $X_t = \{0,1\}^N$ and $X_t^i = 1$ indicates S_t^i contains an explicit flow[5] of

[4]Here, computational indistinguishability (adopted from the Oblivious RAM literature [112]) is synonymous with computational non-interference [206], and the definition can be easily changed to require strict non-interference [199] if the program does not require computational assumptions.

[5]Formally: Let each memory cell S^i take two inputs: data (in^i) and write enable (we^i) where both are functions (combinational logic) taking a subset of bits in S as input. For time $t = 0$: S_0 (i.e., at $t = 0$) is initialized with starting program state, $X_0^i = 1$ iff S_0^i is Confidential data. For time $t > 0$: $X_t^i = 1$ if (a) we^i outputs 0 in cycle t and $X_{t-1}^i = 1$ or (b) we^i outputs 1 in cycle t and $X_{t-1}^j = 1$ for some j in the inputs to S^i (in^i or we^i). We note that implicit flows [207] are accounted for once BitCycle is applied to Definition 3.1.

Confidential data in cycle t ($X_t^i = 0$ otherwise).

For example, writing data d to the processor cache at address a in cycle t sets bits in X_t, corresponding to cache memory cells at a, if either d or a were computed based on Confidential data. More generally, the definition implies the adversary can monitor every possible hardware resource pressure (e.g., flip-flop level pipeline utilization, cache footprint, etc.) every cycle. Our goal is to provide a basis for programs to achieve *Oblivious*[BitCycle] on advanced commercial-class machines.

3.3.2 Security Issues in Existing Data-Oblivious Code

Existing data-oblivious codes are written extremely conservatively to remove code constructs that blatently violate *Oblivious*[BitCycle]. For example, prior works rely solely on a carefully chosen subset of arithmetic operations (e.g., bitwise operations), conditional moves, branches with data-independent outcomes, jumps with public destinations, and memory instructions with data-independent addresses [22, 106, 107, 114, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129].

It is important to understand when this isn't sufficient for security. To that end, we now detail 11 possible attack vectors on today's data-oblivious code. Importantly, we do not list many popular attacks (e.g., prime+probe in the cache [5]) as these are defeated by writing programs in the style described above. Yet, attacks can still occur because the hardware can apply invisible optimizations to undermine software-level transformations. In the following, we describe attack vectors known to be implemented today, and also proposals whose implementation status is unknown. However, importantly, each optimization could be implemented at any time, breaking existing codes.

Vectors 1, 2, 3: branch, jump, memory speculation

While transient execution attacks [4, 193] are known to impact general-purpose code, their impact on data-oblivious code has not been adequately studied. We make an important observation that data-oblivious code security is undermined even by 'honest' speculative execution. By 'honest', we mean the speculation is not intentionally being controlled in a malicious way, e.g., as in [4]. The root problem is that modern ISAs have limited resources (e.g., ISA-level registers) and executing unintentional instructions can cause secrets stored in *aliased resources* to be exposed accidentally.

Consider a toy example for data-oblivious decryption, exploiting conditional branch misprediction (denoted Vector 1):

```
1    for (i = 0; i < NUM_ROUNDS; i++)
2      state = OblDecRound(state, rkey[i])
3    declassify(state)
```

Figure 3.3: An example of why data-oblivious encryption does not guarantee key privacy.

A legal data-oblivious code can implement decryption round logic data-obliviously, with the round keys rkey considered Confidential (Definition 3.1), and wrap the round in a data-independent branch to reduce code footprint. Once decryption is complete, the program may use the plaintext in a non-oblivious way, e.g., by using it as an address to lookup a record in cache (denoted `declassify(state)`). Such a non-oblivious operation can reveal information related to the *decryption key* on a speculative machine. Specifically, if the branch mispredicts "not taken" (e.g., while the predictor is training), state is prematurely exposed before all rounds complete, allowing an attacker to perform cryptanalysis on encryption round intermediate state.

Removing branches or disabling branch speculation is not sufficient to fix this issue, as other forms of speculation (e.g., unconditional branches/jumps, memory disambiguation [193]—denoted Vectors 2 and 3) cause similar issues on legal data-oblivious code.

Vectors 4, 5: sub-address optimizations

Numerous data-oblivious codes, e.g., "constant time" cryptography [208, 209], make an assumption that modulating certain bits in a memory address (e.g., the bits indicating offset within a cache line) does not create observable behaviors. This assumption doesn't hold on some microarchitectures due to hardware optimizations such as speculative store forwarding (Vector 4) and cache banking (Vector 5), and attacks exploiting these features have been shown to lead to full cryptographic breaks [69, 71].

Vector 6: input-dependent arithmetic

It is well known that complex arithmetic operations (e.g., multiply/divide, floating point square root) exhibit observable data-dependent timing based on their operands [22, 192]. While prior work can mitigate these threats by re-writing complex arithmetic using bitwise operations, this can incur over an order of magnitude performance overhead depending on the operation [22].

Vector 7: microcode

Even simple instructions may be decomposed into simpler instructions, called micro-ops, before being executed. In some cases, micro-op conversion can create data-dependent behavior. For example, `cmov` (which implements conditionals based on Confidential values [107, 114, 123, 125]) can be broken into a `branch+mov`. There is evidence to suggest that this transformation will be applied in future Intel processors ([194], Section 3.2). This breaks privacy: the branch direction will be speculatively guessed and whether a misprediction occurs changes program timing due to the squash (Section 2.1).

Vectors 8, 9, 10, 11: data-based compression, data-based speculation, silent stores

Finally, there are a number of proposals whose implementation status on commercial machines is unknown. In register file [210] and cache [211] compression (analogous to OS-level page de-duplication [212]), register file and cache pressure is a function of program data (Vectors 8 and 9, respectively). Value prediction [40] (Vector 10) speculates on the result of a memory load or long-running arithmetic operation, causing a squash if the prediction is incorrect (Section 2.1).Finally, silent stores [213] (Vector 11) remove redundant store operations (impacting cache pressure) when the hardware detects the memory already contains the same value at the same address. What all of the above have in common is that they are *program data*-centric optimizations that don't discriminate between Public and Confidential data. Thus, they can undermine any data-oblivious code written in any style.

Takeaway

Not only is writing data-oblivious code difficult, but it is also fraught with danger due to subtle ISA-invisible optimizations such as those given above. Our proposed OISA gives hardware the visibility it needs to decide when and when not to apply leaky performance optimizations (such as those above) and enables richer hardware support for data-oblivious code to speed up core operations such as oblivious memory.

3.4 DATA-OBLIVIOUS ISAS

We now describe data-oblivious ISA (OISA) design principles and give an example concrete OISA that we will later implement on top of the RISC-V BOOM.

3.4.1 Design Principles

We had two primary goals in designing an OISA. First, the ISA should expose security guarantees in a microarchitecture-independent way. A single ISA may be embodied in many different microarchitectures (within and across processor generations), each with different organizations and optimizations. It isn't reasonable to ask the software to reason about each microarchitecture: a developer who writes a data-oblivious code correctly once should have confidence that security will hold on each microarchitecture. Second, the ISA should not preclude modern hardware performance techniques, except when those techniques have a chance to leak privacy. Specifically, we want to be compatible with wide (multiple instructions fetched per cycle), speculative, out-of-order commercial-class machines, e.g., those described in Section 2.1, and also point optimizations (e.g., banked caches, data-dependent arithmetic; c.f. Section 3.3.2) that, left unchecked, cause security problems.

To achieve these goals, an OISA has the following components (with hardware support).

Dynamic tracking for Confidential (sensitive) data

We use hardware-based dynamic information flow tracking techniques (DIFT, similar to [214, 215]) to track how Confidential data propagates through the processor as the program executes. Conceptually, all data in the processor is *labeled* Confidential/Public at some granularity (e.g., word-level).[6] This gives hardware the ability to decide when to apply optimizations to data in use (e.g., attack Vectors 6-7, 10-11; c.f. Section 3.3.2) and at rest (e.g., Vectors 8-9).

Prior work does not specify precise rules for when data labeled Confidential can be processed relative to when its label is resolved. A conservative strategy is to require all {data, label} state to correspond to program order, which would preclude speculative, out-of-order execution. A more aggressive strategy is to allow speculation, and to further allow data to be used before its label is resolved.[7] Based on our use of DIFT, it will be clear the latter approach is not secure. Instead, we adopt (and prove secure in Section 3.6) a middle ground which we call *coherent labels*.

Rule 3.4.1. (Coherent labels) When reading an operand, its label must be resolved with respect to the dynamic sequence of speculative/non-speculative instructions (which does not necessarily follow program order) that have executed so far to generate that operand.

[6] 'Public' and 'Confidential' semantics are equivalent to the lattice $\{L, H\}$ ({low, high} security) where $L \sqsubseteq H$ [146, 216].

[7] For example, [215] proposes storing labels in the page table. If the processor supports speculative store-forwarding [71] (Vector 4), data will be used before the label lookup completes.

A simple implementation that satisfies Rule 3.4.1 is to physically extend each data word with a label bit, which allows normal processor dependency tracking to ensure labels are resolved on time. We use this strategy for our implementation in Section 3.5.

Instruction operand-level security specifications

In an OISA, instruction definitions specify, for each operand, whether that operand can accept Public or both Public/Confidential data. We call the former an *Unsafe* operand and the latter a *Safe* operand. Once specified, the hardware designer must handle the following cases.

Rule 3.4.2. (Confidential → Safe) When Confidential data is sent to a Safe operand: the hardware designer must add mechanisms to enforce Definition 3.1, for a specified observability function, despite that instruction's execution. For example, by disabling performance optimizations, scrubbing side effects and masking exceptions that occur as a function of Confidential operands.

Rule 3.4.3. (Confidential → Unsafe) When Confidential data is presented to an Unsafe operand: the hardware must stop (squash) that instruction's execution as soon as the label is resolved. This event is called a label violation #LV. Due to Rule 3.4.1, #LV will be signaled immediately after register/memory read, and before the execute stage begins. If the violating instruction is the next instruction to retire (i.e., is non-speculative), terminate the program. This event is called a label fault #LF.

That is, Rule 3.4.3 is similar to rules that handle badly typed programs, extended to speculative execution. Label violations (#LV) are caused by transient conditions, e.g., imperfect prediction (Section 3.3.2, Vector 1), and are correctable. Label faults (#LF) indicate a program bug or illegal typing. Fixing bugs is outside of our scope, so we will focus on #LV.

An important question is whether #LV creates a side channel based on *when* it is triggered. We prove in Section 3.6.2 that it does not, and further prove that #LV signals enable the OISA to block multiple additional attacks (Vectors 1-5; c.f. Section 3.3.2), e.g., speculation that can reveal Confidential data, on top of the vectors blocked from Section 3.4.1. Finally, Public data is handled as:

Rule 3.4.4. (Public → Safe/Unsafe) When Public data is sent to Safe or Unsafe operands, no special treatment is needed and execution can proceed without protection.

As the above definitions apply at operand granularity, the OISA permits optimizations that are functions of individual operands. For example, zero-skip multiply can be enabled if a Public operand is 0, regardless of whether other operands are Confidential.

Specifying each instruction operand as Safe/Unsafe at the ISA level is a key design feature, and provides significant flexibility to both the ISA and hardware designer while simplifying programmer-level reasoning about security. At the ISA level, an ISA designer can decide which instructions are sufficiently important to warrant Safe operands. These choices should be made carefully: On one hand, Safe operands impose a burden on hardware designers as the processor must support mechanisms to uphold Definition 3.1 for those operands. On the other hand, Safe operands do not specify an implementation strategy. Hardware designers can implement a given operation using simpler data-oblivious instructions (e.g., [22]), hardware partitioning (e.g., [108]) or cryptographic techniques (e.g., [114])—depending on what is efficient given public parameters and the specific microarchitecture. In either case, programmers work with a simple guarantee: Confidential values will not be at risk when consumed by Safe operands, and dynamic execution will be terminated when violations to this policy are detected.

3.4.2 Concrete OISA Specification

Using the principles from the previous section, we now present a concrete OISA that we will implement on top of the RISC-V BOOM processor. Figure 3.4 shows data-oblivious instruction encodings, supported instruction types, and the Safe/Unsafe characteristics for each operand (Section 3.4.1).

Label propagation

Our ISA requires word-granularity labels, tracked in the register file and memory. In most cases, label update logic follows standard taint tracking rules, given the 2-level security lattice {Public, Confidential} [216], as shown in Figure 3.4. When the result is fully determined by Public operands, regardless of other operands (e.g., zero-skip multiply), the result label is set to Public (as done in GLIFT [218], but not shown in Figure 3.4 for simplicity).

Label declassification

Declassification—downgrading data marked Confidential to Public—is a rare but necessary task needed to, e.g., return results. Our ISA supports a single serializing declassification

Base Data Oblivious ISA:	Operand label constraints (S = Safe, U = Unsafe)		Instruction functionality	Label propagation	Notation (assembly)
Arithmetic (R-type)	rs2 (S)	rs1 (S)	**R**[rd] <- **R**[rs1] op **R**[rs2]	**Lr**[rd] <- **Lr**[rs1] | **Lr**[rs2]	
Arithmetic (I-type)		rs1 (S)	**R**[rd] <- **R**[rs1] op ext(imm)	**Lr**[rd] <- **Lr**[rs1]	
Declassify (I-type) (serializing)		rs1 (S)	**R**[rd] <- **R**[rs1]	**Lr**[rd] <- 0	ounseal %rd, %rs1
Classify (I-type)		rs1 (S)	**R**[rd] <- **R**[rs1]	**Lr**[rd] <- 1	oseal %rd, %rs1
Conditional move (R-type)	rs2 (S)	rs1 (S)	**R**[rd] <- (**R**[rs1]) ? **R**[rs2] : **R**[rd]	**Lr**[rd] <- **Lr**[rs1] | **Lr**[rs2] | **Lr**[rd]	ocmov %rd, %rs1, %rs2
Branch (B-type)	rs2 (U)	rs1 (U)	if (**R**[rs1] op **R**[rs2]) PC = PC + imm	**Lr**[rd] <- 0	
Jump register (I-type)		rs2 (U)	**R**[rd] = PC + 4; PC = PC + imm	**Lr**[rd] <- 1	
RNG (J-type)			**R**[rd] <- rand()	-	orng %rd
Load (I-type)		rs1 (U)	**R**[rd] <- **M**[**R**[rs1]+ext(imm)]	**Lr**[rd] <- **Lm**[**R**[rs1]+ext(imm)]	orld %rd, imm(%rs1)
Store (S-type)	rs2 (S)	rs1 (U)	**M**[**R**[rs1] + ext(imm)] <- **R**[rs2]	**Lm**[**R**[rs1]+ext(imm)] <- **Lr**[rs2]	orst %rs2, imm(%rs1)
Oblivious Memory extension:			let addr := **R**[rs1]+ext(imm) % OSZ		
Oblivious Load (I-type)		rs1 (S)	**R**[rd] <- **M**[addr]	**Lr**[rd] <- 1	ocld %rd, imm(%rs1)
Oblivious Store (I-type)	rs2 (S)	rs1 (S)	**M**[addr] <- **R**[rs2]	-	ocst %rs2, imm(%rs1)
CPUID (J-type)			**R**[rd] <- OSZ	-	ocpuid %rd

Figure 3.4: Data-Oblivious ISA. **R**/**Lr**, **M**/**Lm** denote register file data/labels, memory data/labels, respectively. The label Public is denoted logic 0, Confidential logic 1. rs1 and rs2 denote operand registers in RISC-V instructions while rd denotes destination register. R, I, B, J, S-type refers to standard RISC-V instruction formats [217]. ext extends the immediate to the word width. If assembly notation is unspecified, it follows RISC-V with an 'o' prefix (e.g., add becomes oadd). OSZ refers to the microarchitecture-specific oblivious memory partition size (Section 3.4.2).

instruction called ounseal. Serializing instructions are not executed until all older in-flight instructions retire. This is necessary for security: declassification is the only mechanism to demote Confidential to Public, and this action under malicious speculative execution could be used to bypass label checking.

Instruction set

Our ISA supports the following instruction types, which we chose to maximize compatibility with existing data-oblivious codes and minimize hardware changes. First, all RISC-V integer and floating point arithmetic with Safe operands. This means programmers can implement floating point directly, without invoking bitwise libraries [22]. Second, random number generation, as many randomized data-oblivious codes require private random numbers (e.g., [106, 114, 219, 220, 221]). Third, a `cmov`-style ternary/conditional move operator with a Safe predicate for implementing conditionals, and branches/jumps with Unsafe operands to reduce code footprint. Fourth, load/store operations (orld and orst) with Unsafe address operands.

Lastly, we support a second flavor of load/stores (with Safe address operands) which can be used to implement oblivious memory using Confidential addresses (Section 3.4.2).

Mixing in non-oblivious instructions

Oftentimes, only a small program region should be made data-oblivious (e.g., the inner branch in modular exponentiation) to prevent unnecessary performance overheads. To support these situations, we support mixing data-oblivious instructions with instructions from the original ISA. All operands for all original instructions are considered Unsafe. All data-oblivious instructions are encoded on top of the normal RISC-V ISA by modifying existing instruction fields (e.g., the opcode and func [217]).

Putting it all together

To summarize the section, we show a version of Figure 3.2 written using our OISA in Figure 3.5. The programmer need only specify what data is Confidential via oseal. The ISA and hardware will prevent %x3 from being processed by subsequent speculative/non-speculative Unsafe operands. For example, specifying %x3 as an address to a speculative/non-speculative orld triggers a #LV/#LF, respectively.

```
1 oaddi %x1, %x0, 0
2 oaddi %x2, %x0, 64
3 oseal %x3, secret
4 orld %x1, 0(%x1) // memory access
5 orld %x2, 0(%x2) // memory access
6 ocmov %x1, %x3, %x2
```

Figure 3.5: Data-oblivious implementation of Figure 3.2 using OISA without OMP. The word secret denotes Confidential data. %xn are RISC-V general-purpose registers. %x0 is RISC-V idiom for zero.

```
1 oaddi %x1, %x0, 0
2 oaddi %x2, %x0, 64
3 oseal %x3, secret
4 ocmov %x1, %x3, %x2
5 ocld %x1, 0(%x1) // memory access
```

Figure 3.6: Data-oblivious implementation of Figure 3.2 using OISA with OMP. ocld can take secret address operands therefore one load instruction is sufficient.

Oblivious memory extension

A common bottleneck in existing data-oblivious code is the inability to use Confidential data as memory addresses [106, 107, 108, 114]. For example, Figure 3.5 needed to execute two orld instructions. More generally, looking up an array with a Confidential address requires a memory scan.

To accelerate these operations, our OISA exposes two new instructions ocld and ocst, which are analogous to orld/orst (Section 3.4.2) except with Safe address operands, and a new

variant of CPUID ocpuid which returns a microarchitecture-specific constant OSZ ("oblivious memory partition size").

Each microarchitecture is responsible for providing OSZ bytes of "fast" oblivious storage, called the *oblivious memory partition* (OMP), which only ocld and ocst can read/write. This storage can be used to speed up data-oblivious code. For example, if `x` and `y` in Figure 3.2 both fall within the OMP, then Figure 3.5 can be rewritten as Figure 3.6.

How much storage is provided (the value of OSZ) and how that storage is implemented—e.g., a dedicated scratchpad, flexible cache partition, etc.—is left to hardware designers and can be decided on an implementation-by-implementation basis. (Our prototype in Section 3.5.2 uses ways in a cache.) We note that the hardware constrains addresses sent to ocld/ocst to fall within bounds 0 to OSZ-1.

To make data-oblivious code portable across machines (each of which can specify a different OSZ), we provide the following software/programmer-level functions:

- Unsafe OblObj * obl_alloc(Unsafe int size)
- void obl_free(Unsafe OblObj * o)
- Safe int obl_read(Unsafe OblObj * o, Safe int addr)
- void obl_write(Unsafe OblObj * o, Safe int addr, Safe int data)

Safe/Unsafe qualifiers are implied based on how these functions are implemented. That is, size must be Public. obl_alloc/free dynamically allocate/free an oblivious memory object OblObj which exposes type, base and bound fields. type $= \{\mathsf{OMP}, \mathsf{ORAM}, \mathsf{SCAN}\}$ and is determined by obl_alloc under the hood using the following rules:

1. If the new object will completely fit into the OMP, based on the size argument, previous allocations, and OSZ: set type = OMP.

2. Else: depending on remaining space in the OMP and the size argument, set the type as ORAM or SCAN. Heuristics to select which are described below.

Post-allocation, users perform reads and writes to OblObjs through obl_read and obl_write, which instrument each operation based on the allocator's prescribed type, as shown in Figure 3.7. We describe the ORAM type below.

obl_alloc decides on each allocation's type based on information returned by ocpuid. In the current design, ocpuid returns OSZ, the implementation-specific size of the OMP. Future implementations may also return richer information, such as machine cache sizes/etc. to make more informed decisions. Since size and branches/jumps in our OISA are Unsafe, the

```
int obl_read(OblObj* o, int addr) {
#oblivious {
    int ret; int tmp;
    switch (o->type)
    case OMP:
        asm ("oaddi %0, %1, %2": "=r" (tmp): "r" (addr), "r" (o->base));
        asm ("ocld %0, 0(%1)": "=r" (ret): "r" (tmp));
        break;
    case ORAM:
        ret = oram("read", o, addr); break;
    case SCAN:
        for (int j = o->base, j < o->bound; j+=4) {
            asm ("orld %0, 0(%1)": "=r" (tmp): "r" (j));
            asm ("ocmov %0, %1, %2": "+r" (ret): "r" (j==addr), "r" (tmp));
        }
        break;
    return ret;
} }
```

Figure 3.7: obl_read implementation (obl_write is analogous). `#oblivious` is short-hand to indicate that the body consists only of data-oblivious instructions. `oram`'s implementation is discussed in Section 3.4.2. "=r","+r" denotes output register; "r" denotes input.

strategy selected for each allocation depends only on the program (which is Public) and the machine architecture. Lastly, we note that since the allocator makes decisions based on the order of previous allocations, more performance-sensitive objects should be allocated first.

ORAM and SCAN types. When the oblivious object does not fit into the OMP, the allocator may implement it as an Oblivious RAM [112] (ORAM) or memory scan. ORAMs are randomized algorithms which implement oblivious memory in poly-logarithmic time. For ORAM, we use the ZeroTrace library [114] which is a data-oblivious ORAM client written in our threat model. Depending on the remaining OMP space, ZeroTrace's internal sub-structures (e.g., the ORAM stash and position map [114]) can be placed in the OMP, which we show can speedup the original ZeroTrace by $> 4\times$ (Section 3.7.3). SCAN is a fallback that emulates oblivious memory using normal memory, and is implemented as a sequence of orld and ocmov instructions (Figure 3.7).

Pointed out by [107], when scan vs. ORAM is more efficient depends on the memory size and the allocator should take this into account based on the allocation size parameter.

3.4.3 Process-OS Interface

Processes interact with the OS through exception handling, context switching and system calls. We design the OISA to cause minimal friction with the existing OS-process interface.

Exceptions Exceptions leak data-dependent conditions (e.g., when a divide by zero occurs) in programs [107, 201]. When an exception occurs on instructions with all Public operands, it is handled like a normal exception. When an exception occurs on an instruction with a Confidential operand, the hardware must mask that exception (e.g., by replacing the result with a canonical value and leaving the label unchanged). In this design, the adversary may learn an exception has occurred only if resulting data is explicitly declassified with ounseal.

Context switching In the current design, the OMP (Section 3.4.2) and register file labels are added as thread state. Labels in memory are mapped to pages in a region of virtual memory that cannot be accessed directly by the program (Section 3.5.3). While adding the OMP to thread state doesn't make context switching performance-prohibitive for the OMP sizes we consider in Section 3.7, it will for sufficiently large OMPs. We leave integrating the OMP into normal process virtual memory (e.g., by using the RISC-V VLS technique [222]), as future work. Finally, if the adversary is supervisor-level (Section 3.2), we rely on the shielding system, e.g., SGX, to protect program data during context switches. For example, in an SGX setup [203], all data (Public and Confidential) would be stored within the SGX ELRANGE.

System calls We rely on orthogonal software techniques to sanitize system call arguments [202, 223].

3.5 IMPLEMENTATION

This section describes how we prototyped our OISA on the RISC-V BOOM microarchitecture. Our design augments BOOMv2, which is the most recent iteration of the BOOM design [195]. We give the exact parameters used for the architecture in Table 3.2, which corresponds to the block diagram in Figure 3.8 and is a default BOOM configuration.

3.5.1 RISC-V BOOM Summary

We first summarize unmodified BOOM (referencing Figure 3.8). These details will be used for our implementation (this section) and formal analysis (Section 3.6).

First, multiple instructions are *fetched* each cycle ❶. Based on the current program counter (PC) and decoded instructions, multiple levels of branch/jump predictors issue predictions for fetched branches/jumps. Mispredicted branches/jumps are discovered in the execute stage, and cause subsequent speculatively decoded instructions to squash (Section 2.1). Once

decoded, instructions are added to the issue windows ❷ where they wait for their operands to be ready, at which point they are *scheduled* (possibly out-of-order) to execution units. Operands become ready when they are written (or written back) to one of two register files (RFs, for floats and integers) ❸, or when an execution unit finishes early and *bypasses* the result directly to the consumer instruction. RFs contain speculative and non-speculative data.

BOOM supports a configurable number of execution units ❹, each of which contains a configurable number of primitive arithmetic/branch/etc. units, shown in Table 3.2. Each execution unit receives dedicated read/write ports to the RFs. Primitive arithmetic blocks may be *pipelined* (have input-independent latency) or *un-pipelined* (have input-dependent latency). Lastly, a load/store unit interfaces to the cache and decides whether load data should be read from the cache or store data queue (SDQ) which contains speculative stores (store-load forwarding). Loads may speculatively execute after stores whose address has not resolved [224]; address alias violations are caught and squashed at retire time. Finally, a reorder buffer (ROB) ❺ tracks in-flight instructions in-order to facilitate in-order commit (Section 2.1).

The current BOOM does not currently support SMT/hyper-threading. We note that our OISA is compatible with an SMT-enabled machine and that the hardware mechanisms discussed below need not change to support SMT.

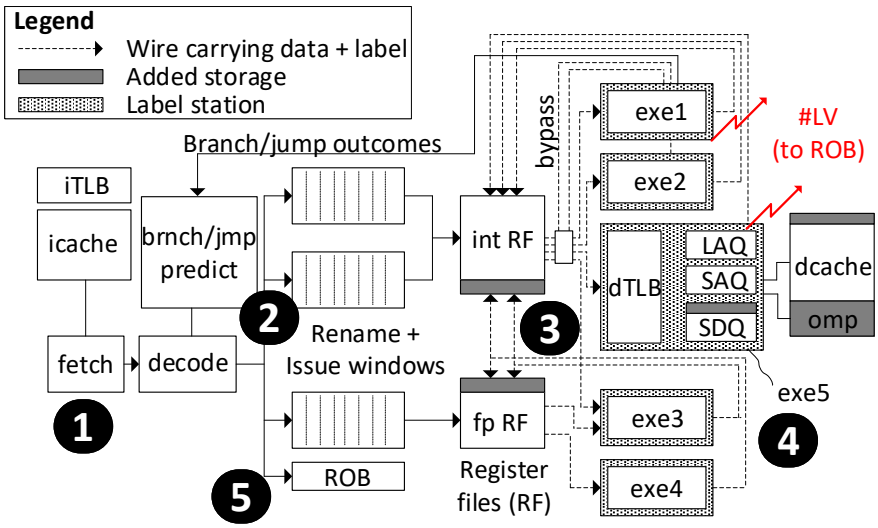


Figure 3.8: RISC-V BOOM v2 pipeline [195]. 'exeXX' are execution units, and contain arithmetic/branch/etc units stated in Table 3.2. Hardware modifications needed to support the OISA (Figure 3.4) are shown in the legend. No modifications are needed before the int/fp register files. Label stations are discussed in Section 3.5.3. 'omp' is the oblivious memory partition (Section 3.5.2).

3.5.2 Support for New Instructions

Discussed in Section 3.4.2, most instructions in the OISA have exact counterparts in RISC-V, but with additional semantics/dynamic checks for Safe/Unsafe operands. These instructions reuse existing RISC-V encodings and have altered opcode/func fields to be identified during the decode stage. Several exceptions are oseal, unseal, orng, ocmov, ocld/ocst/ocpuid which don't have RISC-V counterparts (Figure 3.4).

We implement oseal and ounseal as the RISC-V addi instruction with the immediate field set to 0 (functionally a move operation), but with modified logic to set/clear label bits. As discussed in Section 3.4.2, ounseal must also serialize (execute non-speculatively) to prevent malicious declassification. Since BOOM already implements serializing instructions, we reuse that functionality for ounseal. Our prototype implements orng as a cryptographic PRNG (iterative AES core), although a hardware TRNG [225] may be used for a production design.

ocmov presents a challenge, as conditional move requires three operands (predicate, new value and old value) whereas no RISC-V integer instruction requires three input operands. To minimize ISA-level changes, we design a single ALU (in one execution unit) to serve ocmov instructions, and add a new RF port for that execution unit. We design this ALU to support bypassing. This design is low overhead and efficient. Having one execution unit support ocmov means we only need to add a single read port to the RF (not +1 per execution unit). Through bypassing, our design can execute back-to-back dependent ocmovs, one per cycle.

Finally, our current implementation implements the oblivious memory partition (OMP) for ocld/ocst as a quarantined region of the first-level data cache. We isolate a region of the cache using way partitioning techniques [148], which are a low-complexity mechanism to divide the cache into non-interfering regions as long as the region size is a multiple of the associativity (our first-level cache is 16-way; Table 3.2). This design has low hardware overhead. If no process has allocated oblivious objects (Section 3.4.2), OMP storage can be used as normal cache memory. While an ocld/ocst instruction is looking up the OMP, all concurrent cache lookups are stalled to avoid cache bank contention [69].

3.5.3 Tracking and Checking Labels

An important component in our OISA is checking and tracking Public/Confidential labels as data flows through the pipeline and signalling #LV when violations occur. Noted in Section 3.4.2, we track labels at word granularity.

Label storage

Labels must be stored alongside each word, wherever each word resides in the processor. This includes the RF, the SDQ, the data cache hierarchy, and intermediate pipeline registers. In these structures, we treat the data label as an extra bit in each word. This makes it simpler to satisfy Rule 3.4.1: whenever a speculative or non-speculative instruction reads an operand, normal out-of-order processor dependency checking ensures the label is resolved.

Unfortunately, this strategy would require large changes to the DRAM/below memory levels because wider words would require wider DRAM lines and larger page tables. Thus, at the DRAM level, we store data and labels in separate disjoint pages and modify the hardware DRAM controller to join data and label into a widened cache line when on-chip (a similar scheme was used in [214]). This means any DRAM access in our system turns into two DRAM accesses.

Label checks

To satisfy Rules 3.4.2 and 3.4.3: once a consumer instruction indicates its intent to use an operand, that operand's label must be checked against the instruction opcode/func fields, before the use occurs. We design a parameterizable hardware module called a *label station*, which wraps each BOOM execution unit, to administer these checks. The main observation enabling the label station design is that in BOOM, all operand-dependent processor state updates are signalled from the execution units. This makes it possible to implement a shim at the input of each execution unit to perform label checks, handle label violations/faults, and disable hardware optimizations on Confidential inputs.

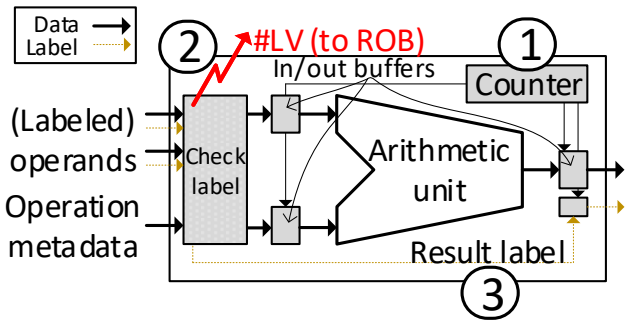


Figure 3.9: Label station (Section 3.5.3) for an execution unit with one internal arithmetic unit. A real execution unit may contain multiple arithmetic units (Table 3.2), in which case this logic is replicated as needed. Added hardware is shaded.

Specifically, the label station (visualized in Figure 3.9):

① (Rule 3.4.2: Confidential → Safe) Blocks access to/from arithmetic units so that any operation processing Safe operands takes the worst-case time. This is implemented using input/output buffers (e.g., flip-flops), a timer (counter), and operand/label decode logic ("Check label" in the figure). Variable-time arithmetic units and their worst-case times are given in Table 3.2. Lastly, any status bits set as a function of Confidential operands are set to canonical values.

② (Rule 3.4.3: Confidential → Unsafe) Checks each incoming operation for illegal label-operand violations, and signals #LV when violations are detected. All checks are performed before operands are forwarded to the execution unit. If any violation is detected, the execution unit does not receive the operation and an #LV signal is sent to the ROB, where it is interpreted as a violation (squash) or a fault (termination, #LF), respectively.

③ (Label propagation) Computes the result label based on operand labels and stages the label to travel with the result when it writes back to the RF or exits early via bypass.

Label stations are parameterized at design time based on what functionality is actually needed. For example, Execution unit 2 (Table 3.2) only supports Safe-operand arithmetic and therefore doesn't need logic to enforce Rule 3.4.3 (Confidential → Unsafe). Hence, this logic is pruned away at hardware synthesis time.

3.6 SECURITY ANALYSIS

We will show that the OISA provides a basis for satisfying *Oblivious*[BitCycle] (Section 3.3.1) by proving its security over an *abstract* out-of-order, speculative machine (AOOM), and arguing that this abstract machine can be reduced to real hardware such as the BOOM.

3.6.1 Takeaways and Main Insights

The takeaway from the analysis is that the OISA provides a basis to prove (computational) noninterference on an out-of-order, speculative processor. Importantly, we achieve this result *while* allowing hardware optimizations, such as branch predictors, to remain enabled and *without* requiring those structures to be partitioned or periodically flushed.

Informally, for this result to hold we need to show that (a) each instruction's visible execution and (b) the sequence of instructions that are executed is independent of Confidential data. (a) follows by definition, given Rules 3.4.2-3.4.4, and is enforced by label stations in our implementation (Section 3.5.3). A key insight here is that by applying these rules *locally, and to each instruction as it executes*, the analysis/hardware need not be aware of whether each instruction is speculative, executed out-of-order, etc.: the checks performed to maintain

security are the same in all cases. To show (b), we leverage a key property inherent in any OISA: *that the inter-instruction program counter (PC) never becomes a function of ("tainted by") Confidential data.*

Guaranteeing that the PC stays "untainted" involves some subtlety. On an out-of-order speculative machine, the sequence of dynamic instructions clearly depends on more than just the program and its input. For example, the PC is influenced by hardware predictors and dynamic data-dependent events such as when squashes occur. Yet, the untainted PC property (once proven) is surprisingly powerful,[8] and is the crux behind why hardware performance optimizations can remain safely enabled. For example, if the PC is untainted, branch predictor structures are also by extension untainted. In our design, this holds because only branch instructions that do not cause #LV (i.e., those based on Public decisions) are allowed to update the branch predictor, and because maliciously "priming" the branch predictor [4] can be modeled using only Public information. If the branch predictor is untainted, it can by definition remain enabled.

In general, the only new source of overhead occurs when Confidential → Unsafe events cause squashes. The analysis will show that when this occurs doesn't depend on Confidential data and, in particular, that a correctly written program should only see this event when "honest" miss-speculation (Section 3.3.2) occurs. Predictors must be high accuracy to be effective, thus honest miss-speculation should be rare.

3.6.2 ISA Level

The following analysis assumes the OISA disables the ounseal instruction (Section 3.4.2) unless otherwise stated.

Abstract machine basics

The functional model for AOOM is given in Algorithm 3.2, with notations/helper functions explained in Table 3.1 and Algorithm 3.1. Our goal was to keep the model as simple as possible, while capturing core features. Specifically, the abstract machine: (1) has a 3-stage pipeline {Fetch, Execute, Retire} where each stage is atomic and takes one unit of time, (2) has four instruction types {Arithmetic, Branch, MemLoad, MemStore}, (3) has infinite fetch bandwidth and execution units, (4) can be parameterized as an in-order or out-of-order/speculative machine. Which instruction types support Safe/Unsafe operands are encoded as conditionals checking operands for label violations (#LV). We explain how to extend the model (e.g., to

[8] Similar observations were also made in prior work [218].

$\|T\|$	Returns number of elements in T
$T[i:j]$	Returns items with index i to j (inclusive)
λ	Public program
Fetch, Execute, Retire	Instruction stages
Arithmetic, Branch MemLoad/Store	Instruction types
stage, pc, squash, update	Trace entry format
Write(addr, data, label)	Token denoting write to program memory
$Proj(T)$	Trace with updates removed
$arg_i(\mathsf{pc}, \lambda)$, $dest(\mathsf{pc}, \lambda)$	Returns instruction operand/dest fields
$op(\mathsf{pc}, \lambda)$	Returns instruction's implied arithmetic op
$T.append(e)$	Append e to end of of T
$type(\mathsf{pc}, \lambda)$	Return instruction at pc's type (Branch, etc)
$done(e, \lambda)$	Returns `true` if e.stage = Retire and e.pc is the stop PC given λ
SCHEDULE, PREDICT	Instruction scheduler and predictor functions

Table 3.1: Notations and simple helper functions.

account for variable latency instructions, cache, limited execution units, more pipeline stages, etc.) in Section 3.6.3.

Algorithm 3.1: Helper functions *meminit* and *mem*.

function: $meminit(x, y)$ `// fill memory w/ Public` x`, Confidential` y

$T := []$;
for $x_i \in x$ **do**
  $T.append((\mathsf{Execute}, \bot, \texttt{false}, \mathsf{Write}(i, x_i, \texttt{false})))$
for $y_i \in y$ **do**
  $T.append((\mathsf{Execute}, \bot, \texttt{false}, \mathsf{Write}(|x| + i, y_i, \texttt{true})))$
return T;

`/* return coherent memory snapshot, given` T`. Note, an instruction that is squashed by another instruction may still create visible state changes in the window of time before the other instruction reaches Execute. */`

function: $mem(T)$

$T' = T$ with all squashed instructions (trace entries) removed. That is, remove from T any entry that occurs in between the Fetch and Execute stage of an instruction I if I satisfies $I.\mathsf{stage} = \mathsf{Execute} \wedge I.\mathsf{squash}$ (inclusive);
mem := [$\bot$ **for** $t \in T'$]; `//` $|T'|$ `upper-bounds mem size`
for $x_i \in T'$ **do**
  up := x_i.update;
  if up.addr $\neq \bot$ **then**
    mem[up.addr] = up.data, up.label;
return mem;

Algorithm 3.2: Abstract machine definition.

```
   function: AOOM(λ, x, y)
1  T := meminit(x, y);                                          // initialize memory
2  while !done(T[|T| − 1], λ) do
3      idx := SCHEDULE(Proj(T), λ);
4      if idx = ⊥ then                                          // Fetch new instr
5          pc := PREDICT(Proj(T), λ);
6          T.append((Fetch, pc, false, Write(⊥, ⊥, false)));
7      else
8          pc := T[idx].pc;
9          stage := T[idx].stage;
10         if stage = Fetch then                                // Execute instr
11             T.append(execute(Execute, pc, T, λ));
12         else if stage = Execute then                         // Retire instr
13             T.append((Retire, pc, false, Write(⊥, ⊥, false)));
14 return T;
15
   function: execute(stage, pc, T, λ)
16 update := Write(⊥, ⊥, false); squash := false;
17 arg_{0,data}, arg_{chpo isa:0,label} := mem(T)[arg_0(pc, λ)];
18 arg_{1,data}, arg_{chpo isa:1,label} := mem(T)[arg_1(pc, λ)];
19 if type(pc, λ) = Arithmetic then
20     data := arg_{0,data} op(pc, λ) arg_{1,data};
21     label := arg_{chpo isa:0,label} ∨ arg_{chpo isa:1,label};
22     update := Write(dest(pc, λ), data, label);
23 else if type(pc, λ) = Branch then
24     if arg_{chpo isa:0,label} ∨ arg_{chpo isa:1,label} then
25         squash := true;                                      // #LV: Confidential->Unsafe
26     else
27         fidx := index of Fetch for current instr in T;
28         guess := direction for PREDICT(Proj(T[0 : fidx]), λ);
29         actual := arg_{0,data} op(pc, λ) arg_{1,data};
30         squash := guess ≠ actual;                            // mispredict
31 else
32     if arg_{chpo isa:0,label} then
33         squash := true;                                      // #LV: Confidential->Unsafe
34     else
35         if type(pc, λ) = MemLoad then
36             data, label := mem(T)[arg_{0,data}];
37             addr := dest(pc, λ)
38         else if type(pc, λ) = MemStore then
39             data, label := arg_{1,data}, arg_{chpo isa:1,label};
40             addr := arg_{0,data}
41         update := Write(addr, data, label)
42 return stage, pc, squash, update;
```

Execution traces

The abstract machine AOOM takes as input a program λ, Public input x and Confidential input y and generates a trace T where each entry T_t tracks a stage of each instruction as it executes on the machine. That is, the t-th element in T is a 4-tuple:

$$T_t = (\mathsf{stage}_t, \mathsf{pc}_t, \mathsf{squash}_t, \mathsf{update}_t) \tag{3.2}$$

stage_t denotes the instruction's stage $\{\mathsf{Fetch}, \mathsf{Execute}, \mathsf{Retire}\}$. pc_t denotes the instruction address/program counter. Different stages for the same logical instruction share the same pc. If $\mathsf{stage}_t = \mathsf{Execute}$, $\mathsf{squash}_t = \{\texttt{true}, \texttt{false}\}$ denotes whether the instruction caused a squash during speculation (Section 2.1) or due to a label violation #LV (Section 3.4.1). If $\mathsf{stage}_t \neq \mathsf{Execute}$, $\mathsf{squash}_t = \texttt{false}$. $\mathsf{update}_t = \mathsf{Write}(\mathsf{addr}, \mathsf{data}, \mathsf{label})$ where Write is a token denoting whether program memory was written, and with what addr, data and label. The Public label is logic 0, Confidential is logic 1. If no write occurs, $\mathsf{addr} = \bot$.

Modeling time

In our abstraction, entries in T are ordered in time as $\mathsf{time}(T_i) \leqslant \mathsf{time}(T_{i+j})$ for $i, j \geqslant 0$ where time is a metric for real-time (e.g., clock cycles). That is, multiple events may occur in the same clock cycle (as in a real processor) or be separated far apart. stage_t and $\mathit{type}(\mathsf{pc}_t, \lambda)$ allow us to model contention in different pipeline stages for different instruction types.

Modeling out-of-order and speculative execution

A key feature in our analysis is that AOOM is parameterized by two functions, SCHEDULE and PREDICT. SCHEDULE represents control logic in a real processor and decides which stage of which instruction should be evaluated next. It takes as input the program λ and $\mathit{Proj}(T)$, a projection of T that removes update from each entry, i.e.,

$$\mathit{Proj}(T) = \{e.\mathsf{stage}, e.\mathsf{pc}, e.\mathsf{squash} \textbf{ for } e \in T\} \tag{3.3}$$

Importantly, $\mathit{Proj}(T)$ constrains scheduling to not be a function of program data (i.e., e.update) beyond the sequence of present/past fetched instructions (e.stage, e.pc) and whether those instructions result in a squash (e.squash). SCHEDULE outputs an index $\mathsf{idx} \in [0, |T|)$ or $\bot$. If $\mathsf{idx} = \bot$, the machine will fetch the next instruction. If $\mathsf{idx} \neq \bot$, the machine will evaluate the next stage for the instruction at $T[\mathsf{idx}]$. PREDICT represents branch/jump predictor logic,

takes the same inputs as SCHEDULE and outputs the predicted next PC. W.l.o.g. we assume SCHEDULE and PREDICT are deterministic.[9]

Importantly, SCHEDULE and PREDICT are representative of modern processors and allow us to model simple in-order processors to advanced out-of-order speculative processors (details on this claim related to BOOM are in Section 3.6.3). The only assumption we will make is that SCHEDULE respects in-order Fetch and Retire, as done by machines today.

Modeling machine state

The current machine state at some point idx in the trace is determined based on the trace prefix from 0 to idx. This includes program state (register file, cache, etc.) and intermediate pipeline/machine state. Program state is calculated based on *mem* (Algorithm 3.1). We merge the register file and other memory into a single memory for simplicity. Data always travels with its label, which models Rule 3.4.1. As mentioned in Section 42, pipeline state (e.g., flip-flops/SRAM not included in program state) is modeled by the sequence of PCs and stages in the trace.

Proof of Security

We now prove that the abstract model AOOM satisfies Definition 3.1 with respect to the following observability function WordStage.

Definition 3.3. (WordStage observability: Public data and labels at Word spatial granularity, instruction stage-level temporal granularity) Given $T = \mathsf{AOOM}(\lambda, x, y)$,

$$\mathsf{WordStage}(T) = \{e.\mathsf{stage}, e.\mathsf{pc}, e.\mathsf{squash}, h(e) \textbf{ for } \mathsf{e} \in T\} \tag{3.4}$$

where $h(e)$ returns $e.\mathsf{update}$ (unmodified) when $e.\mathsf{update.label} = \texttt{false}$ holds, and returns $\mathsf{Write}(e.\mathsf{addr}, \perp, \texttt{true})$ otherwise.

That is, WordStage only removes write data from the trace if the label corresponding to that data is Confidential. Satisfying Definition 3.1 with the WordStage function implies the strongest level of privacy with respect to our abstract machine, and implies that the machine's pipeline utilization, PC sequence, set of squash events, and state w.r.t. Public data is independent of Confidential data. We proceed to show Theorem 3.1:

Theorem 3.1. *Oblivious*[WordStage, AOOM] holds.

[9] Heuristics based on randomness can be modeled with an additional seed input.

We prove Theorem 3.1 using strong induction over traces of two program executions $\mathsf{AOOM}(\lambda, x, y)$ and $\mathsf{AOOM}(\lambda, x, y')$, relying heavily on PREDICT and SCHEDULE not being functions of trace data. Details for the proof are given in the full version [226].

Extensions to randomized cryptographic algorithms

It is straightforward to extend the above analysis to support randomized cryptographic algorithms such as Oblivious RAM (ORAM) [112, 114]. For example, ORAM client logic can be written data-obliviously to satisfy *Oblivious*[WordStage, AOOM] [114, 227]. What is left is to show how the visible ORAM access pattern—which forms a subset of the trace—satisfies computational indistinguishability [112]. This reduces to the security of the ORAM protocol itself and to the OISA's mechanism to declassify private data, i.e., ounseal. For the latter, since ounseal is a serializing instruction, we know private randomness will be exposed if and only if it is intended by the protocol.

3.6.3 Implementation Level

We now map our ISA-level security analysis (Sections 3.4.2 and 3.6.2) to our prototype on BOOM (Section 3.5), referred to as BOOM.

Threat vectors in unmodified BOOM

Unmodified BOOM hardware (Section 3.5.1) supports speculation over branches, jumps and unresolved store instructions (Vectors 1-3; c.f. Section 3.3.2) as well as arithmetic units with input-dependent timing (Vector 6, Table 3.2).[10] Our implementation of the OMP (Section 3.5.2) is also susceptible to cache bank contention (Vector 5) because it uses space in the data cache.

Securing BOOM

Recall, the primary hardware mechanisms we added to get security are dynamic information flow tracking (Section 3.5.3), label stations per execution unit to implement Safe/Unsafe operand semantics (Section 3.5.3), and logic to isolate the OMP (Section 3.5.2).

[10]We note BOOM also supports load/store forwarding but is not susceptible to Vector 4 because the data TLB is accessed sequentially before checking the SAQ (Section 3.5.1).

In Section 3.6.2, we proved *Oblivious*[WordStage, AOOM]. We show how to use the proof to argue *Oblivious*[BitCycle, BOOM]—i.e., cycle-level security of our implementation—which implies that Vectors 1-3 and 5-6 are blocked. There are two steps: (1) mapping AOOM to BOOM and (2) mapping WordStage to BitCycle.

Finally, we remark that our current reduction to BOOM is best effort, and consider using formal/automated methods to improve design confidence to be important future work.

3.7 EVALUATION

We now evaluate the OISA in terms of area overhead (given our prototype on RISC-V BOOM) and performance over data-oblivious workloads. We also show two case studies, showing how the OISA secures and accelerates constant time cryptographic code and memory oblivious libraries.

3.7.1 Methodology

We evaluate our system through hardware prototyping to show area overheads and software simulation to show performance.

Core μarch	out-of-order, speculative
Fetch/issue width	4 instructions fetched/issued per cycle
Execution unit 1	iALU, Branch, iMul, iDiv (6-66)
Execution unit 2	iALU, **CondMove**
Execution unit 3	IntToFP casting
Execution unit 4	fAdd, fMul, fDiv (5-21), fSqrt (5-29), FPToInt casting
Execution unit 5	Load/store + **Omp** (memory unit)
L1 I/D cache	32 KB, 4 way/64 KB, 16 way; 64 B cache lines
I/D TLB	16/32 entries

Table 3.2: RISC-V BOOM parameters we use for our prototype and evaluation. Arithmetic units with a '(xx)' next to their name are un-pipelined (variable latency), where 'xx' denotes the worst-case latency. The prefix 'i' denotes integer, 'f' denotes floating point. **CondMove** and **Omp** denote logic for ocmov and the oblivious memory partition (Section 3.4.2), respectively, and are only present on our modified BOOM.

Hardware prototyping

We build on top of the open-source BOOM design [195] which is written in the Chisel hardware description language [228]. We parameterized the prototype according to Table 3.2 and synthesized the design using a 32 nm commercial process and the Synopsys flow. We report standard cell (logic cell) area for logic and flip-flops post-synthesis, and report SRAM area using the widely used Cacti tool [229]. BOOM maps the instruction/data caches/TLBs and branch predictor tables to SRAM. The remaining storage structures (e.g., the SDQ, RFs) are mapped to flip-flops. The BOOM word width is 64 bits.

Software simulation

The BOOM hardware only features a single-level cache, whereas commercial machines feature two- or three-level caches to reduce traffic to DRAM. Thus, to measure more realistic performance figures for our system we use Multi2Sim [230], parameterized to match Table 3.2 as closely as possible. For all experiments, we use a 256 KB 4-way level 2 cache (that is shared by data and instructions) and a 2 MB 16-way level 3 cache. This configuration is similar to a single slice on an Intel Skylake machine.

OMP usage

We use a 32 KB OMP (Section 3.4.2) that is built into the level 1 data cache. This is sufficient to store ORAM sub-structures (Section 3.4.2) and also big enough to fit tables for constant time cryptographic routines (e.g., AES T-tables and RSA multiplier tables). Some workloads do not benefit from the OMP. In this case, a bit in the thread state disables the OMP to recover cache space.

3.7.2 Hardware Prototyping and Area Results

We show area results for unmodified BOOM and BOOM extended to support our OISA in Table 3.3. Our prototype supports all instructions in Section 3.4.2 and Figure 3.4. The main hardware components needed to support the OISA are storage for DIFT, logic/storage for label stations, logic to partition the OMP, and a random number generator for orng (Section 3.5). For structures that need to store labels, we store those labels alongside the data in whatever medium the data was stored in. That is, labels in the data cache are stored in SRAM, labels in the SDQ and register files are stored in flip-flops. The largest single area overhead comes from an iterative AES core that we downloaded from OpenCores [231] to

implement orng. This unit has area 10,935 um^2 (3% of the logic area for the unmodified BOOM), and can be replaced by a hardware TRNG (whose area is negligibly small [225]) in a production design.

The takeaway is that hardware overheads are tolerable, both on the logic and SRAM side, showing the practicality of the proposal on advanced commercial-class machines.

	BOOM	**BOOM + OISA**	**Overhead**
Logic	363,900	388,658	6.80%
SRAM	384,232	391,291	1.84%
Total	748,132	779,949	4.25%

Table 3.3: Area (um^2) for baseline and modified BOOM cores.

3.7.3 Performance Results

We now perform studies to evaluate the performance overhead of running data-oblivious code securely, with and without the oblivious memory partition.

Comparison systems

We compare doisa and doisa_ompto a baseline insecure system. All three systems use the same microarchitecture (Table 3.2). Benchmarks run on insecure are written in a non-data-oblivious fashion (i.e., without the constraints in Section 3.3.2). Benchmarks run on doisa are data-oblivious, and written using only instructions in Figure 3.4 except ocld/ocst (the oblivious memory extension; c.f. Section 3.4.2). Thus, doisa will be similar performance-wise to existing data-oblivious codes, e.g., Raccoon [107], which don't have access to an OMP. Benchmarks run on doisa_omp use all instructions in Figure 3.4 including ocld/ocst.

Name	**Implementation**	**Data size (small / large)**
mat. mult	data-oblivious by default	256x256 / 1024x1024
neural network	data-oblivious by default	64-1K-8 / 1024-32K-256
findmax	data-oblivious by default	8K / 1M integers
sort	bitonic-sort, data-obliv. merge-sort	4K / 256K integers
pagerank	GraphSC [232]	1K / 16K nodes
binary search	memory scan (doisa), obl. memory (doisa_omp)	8K / 16M integers
kmeans	obl. memory for histogram	64/256 clusters, 4K/32K points
heap push	ODS [221]	8K / 32M integers in heap
heap pop	ODS [221]	8K / 32M integers in heap
sparse dijkstra	ObliVM [219]	256 / 4K vertices

Table 3.4: Benchmarks and input data sizes for comparing insecure, doisa and doisa_omp.

Workloads

We evaluate a suite of common workloads (Table 3.4) which have previously been written and evaluated data-obliviously [107, 219, 221, 232] on existing x86 machines. These codes are divided into three categories. First, codes that are nearly data-oblivious in their default form (mat mult, neural network, findmax). Second, codes that rely heavily on data-oblivious sort as a subroutine (sort and pagerank). Third, codes that rely heavily on oblivious memory (binary search, kmeans, heap, dijkstra). We will also perform case studies showing our proposal's applicability in two additional important settings—constant time cryptography and oblivious memory—in Sections 3.7.3 and 3.7.3.

Data set sizes

For each benchmark, we evaluate 'small' and 'large' data sizes. 'small' indicates the largest input size that wholly fits into the 32 KB OMP (Section 3.7.1). We use this configuration for two reasons. First, to show the benefit of having an OMP. Second, to performance compare against prior work (Raccoon [107], which uses similar data sizes). Finally, we show the 'large' data size to illustrate overheads where program data does not completely fit into the OMP. In that case, we fall back to ORAM or SCAN as described in Section 3.4.2.

Results

Figure 3.10 shows the overhead of {doisa, doisa_omp} × {small, large} relative to insecure. The main takeaway is that doisa_omp achieves significant (8.8×/1.7× for small/large data sizes) speedup over doisa. Furthermore, doisa_omp has only 3.2×/40.4× slowdown relative to insecure on the same data sizes. This shows that our OISA makes data-oblivious computing practical in cases where data fits in the OMP.

There are two avenues for future work. First, enhance the OMP to support larger sizes (e.g., beyond the level 1 data cache, see Section 3.4.3). As we see on the large data set size, the overhead for both doisa and doisa_omp can be large for workloads that depend on oblivious memory, as large data sizes cannot fit into the OMP. Second, engineer more sophisticated instructions supporting Safe operands. For example, sort is an important kernel in multiple data-oblivious codes [125, 219, 232, 233]. An OISA can support an osort instruction with Safe operands directly, and use techniques such as hardware partitioning to speed up that operation.

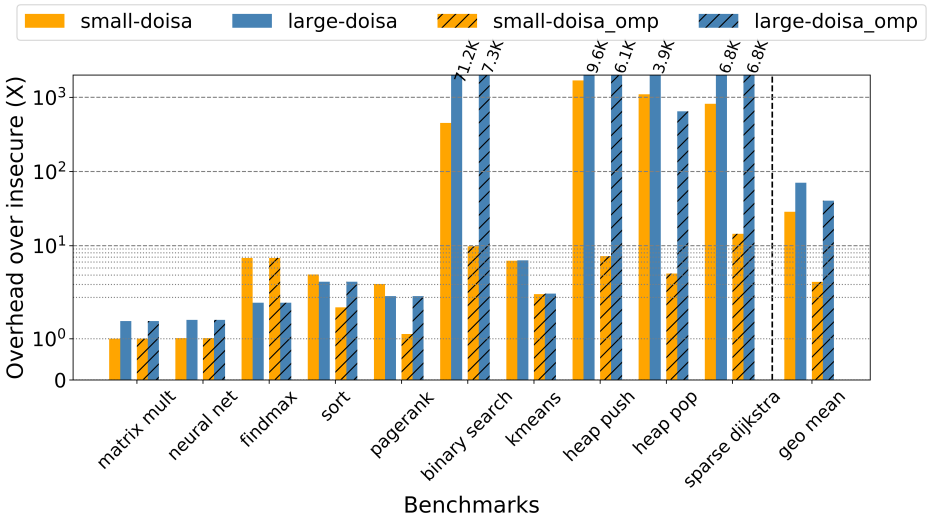


Figure 3.10: Performance comparison between doisa and doisa_omp relative to insecure for small/large data sets.

Case study: constant time AES

An important commercial use-case for data-oblivious code today is "constant time" cryptography. Many papers have demonstrated how un-protected codes—e.g., T-table AES [197] and naive modular exponentiation for RSA—leak privacy over microarchitectural side channels [5, 69, 71]. As a result, practitioners use slower codes to improve security—e.g., S-box or bitslice AES [198] and montgomery ladder exponentiation for RSA.[11]

Our OISA provides a basis for running high-performance cryptography securely. To demonstrate the benefit, we compare the performance of T-table AES [197] (high performance, low security) vs. bitslice AES [198] (low performance, high security). For this study, we retrofit T-table AES using our ISA and store the T-tables in the OMP to prevent cache attacks (the rest of the code is naturally data-oblivious). This gives us a high-performance, high-security code. The OISA can securely run both the fully unrolled code or a variant with a loop over the number of rounds, regardless of branch prediction accuracy (Section 3.3.2). We argue that on commodity machines today, highly sensitive applications will have to resort to codes like bitslice AES.

Both codes are compiled with gcc using `-O3` optimizations. Relative to an insecure T-table AES code (insecure), our data-oblivious T-table AES (doisa_omp) has a 2.17× slowdown, while bitslice AES has a 9.6× slowdown against the same baseline. Our slowdown relative to insecure is caused by the compiler not optimizing code around ocld instructions. Thus, doisa_omp can achieve an even lower slowdown with better compiler support.

[11] Discussed in Section 3.3.2, even hardened codes may be insecure due to subtle hardware optimizations.

Case Study: ZeroTrace

Beyond encryption, there is a rich literature to accelerate data structure operations data-obliviously [106, 219, 221]. These schemes typically use oblivious memory as a subroutine. We now demonstrate how the OISA can speed up this subroutine by comparing our oblivious memory API to the original ZeroTrace [114] proposal. As discussed in Section 3.4.2, our library combines ZeroTrace with the OMP to achieve speedup for different oblivious memory sizes.

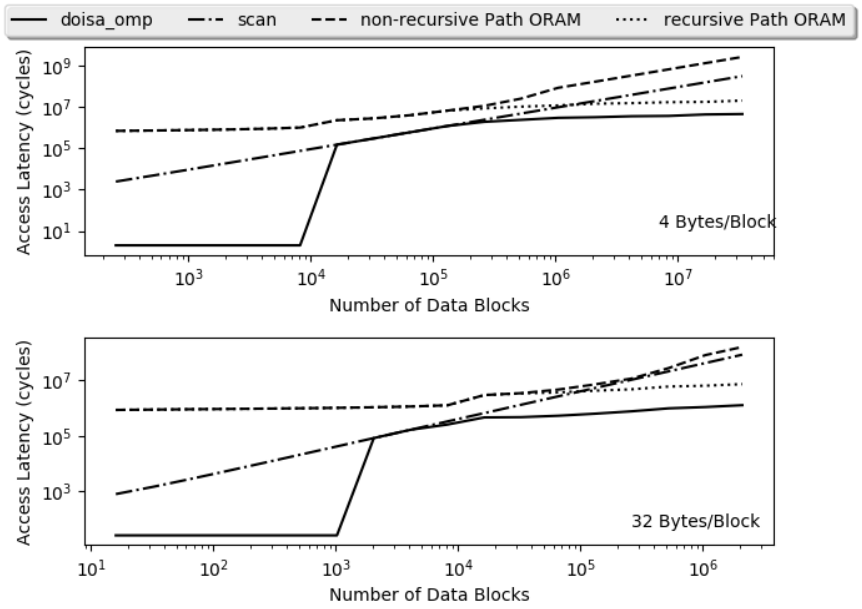


Figure 3.11: Comparison between oblivious memory primitives. Scan is the SCAN code from Figure 3.7, shown for completeness. non-recursive/recursive Path ORAM are baseline ZeroTrace [114].

Results are shown in Figure 3.11. doisa_omp provides significant speedup in all size regimes. For small data, doisa_omp places the entire memory in the OMP, providing $O(1)$ ($> 1000\times$ speedup) time access to that data. For larger data, doisa_omp uses SCAN or ORAM, depending on which strategy yields the best performance, and places the ORAM stash in the OMP in the latter case. An important finding in the ZeroTrace paper is that stash management, written data-obliviously, creates a performance bottleneck.[12] Since the stash does not grow as a function of the ORAM capacity, we can use the OMP to store the stash and manage it more efficiently, which allows us to improve over baseline ZeroTrace by $\geqslant 4.6\times$ in all regimes.

[12]We note that an alternate ORAM, Circuit ORAM [227], was designed to avoid stash management overheads. Unfortunately, Circuit ORAM has worse bandwidth—$12 * \log n$ vs. $8 * \log n$ and $3.5 * \log n$ for data size n—than Path ORAM, which relies on a stash. Since our oblivious memory extensions make stash management essentially free, our scheme based on Path ORAM will outperform Circuit ORAM.

3.8 OTHER RELATED WORK

Data-oblivious programming stack. Beyond data-oblivious code written for today's ISAs, there is a rich literature to improve algorithm/data structure [112, 219, 221, 232, 233, 234, 235, 236, 237, 238] performance in the *software circuit* abstraction. Additionally, there is rich literature to write (e.g., [220, 239]) and compile (e.g., [219, 237, 240]) programs to software circuits. An important observation is that, although many of these works target cryptographic backends such as garbled circuits, their underlying programming abstraction (software circuits) is very similar to the data-oblivious abstraction. For example, bitwise crypto can be easily mapped to integer-wide operations. Thus, our proposal can be used as a secure hardware backend for these works.

ISAs for security, type systems for information flow guarantees. ISAs for security are not new; further Safe/Unsafe operands and the use of DIFT can be viewed as performing runtime checks between a simple type system and security lattice (e.g., see [108, 146, 218, 241]). Relative to these lines of work, we view our conceptual contribution as introducing new ISA abstractions and design principles that allow software/hardware designers to trade off efficiency with implementation complexity, while leaving programmers with simple, portable security guarantees. We note that our choice of lattice and types was done for simplicity; an OISA may be combined with a more sophisticated lattice and set of operand functionalities.

Finally, we view GLIFT [108, 218] as a spiritual predecessor to this work. One of GLIFT's major insights is that at the logic gate level, implicit and explicit flows look very similar. We observe that the same is true in the data-oblivious abstraction at the program level. This allows insight from GLIFT to carry over to our domain (e.g., GLIFT's bit-level checks/transition functions perform a similar purpose for bits as label stations perform for words; c.f. Section 3.5.3).

3.9 CONCLUSION

This chapter proposes a data-oblivious ISA extension to enable secure and high-performance data-oblivious computing in the data-oblivious abstraction. We propose ISA principles, a concrete ISA, a complete prototype on an advanced microprocessor, and accompanying formal analyses for all of the above. Long term, we hope OISA serves as a step for writing and running safe, portable, and performant data-oblivious code for sensitive applications.

CHAPTER 4: SPECULATIVE TAINT TRACKING

This chapter presents Speculative Taint Tracking (STT), a comprehensive protection framework for protecting all speculatively accessed data against all types of speculative side-channel attacks. Speculative attacks present an enormous security threat, capable of accessing and transmitting arbitrary program data under malicious speculative execution. Rather than following the conservative defense approach by delaying the speculative execution of instructions that access potential secrets, STT shows that it is safe to execute and selectively forward the speculative accessed data, which improves performance, as long as the forwarded data do not reach covert channels. We conduct a throughout classification of covert channels in modern processors, and design novel mechanisms to enable protection only when speculative leakage is possible for reducing the performance overhead. STT showcases a substantial performance improvement over previous protection schemes with a provable security guarantee against speculative attacks.

4.1 INTRODUCTION

Spectre [4], Meltdown [3] and follow-up attacks [92, 93, 96, 97, 172, 183, 242, 243, 244, 245] based on speculative execution have opened a new chapter in hardware security. In these attacks, adversary-crafted sequences of transient instructions—i.e., speculative instructions bound to squash—access and then transmit sensitive program data over *microarchitectural covert channels* (e.g., the cache [15]). For example, Spectre Variant 1, shown in Figure 4.1, bypasses a bounds check due to a branch misprediction and transmits secret data behind that bounds check over a cache-based covert channel [4]. Since the address `addr` can take an arbitrary value, `val` can be any value in program memory, meaning the covert channel can reveal arbitrary program data. (In this chapter, we denote a potentially secret value in `green`.)

Prior work has pointed out that speculative execution attacks are broken into two components [55, 172]. First, a secret value is speculatively *accessed* and read into the architectural state (e.g., a register) due to adversary-controlled speculative execution. For example, load `M1` in Figure 4.1 reads `val` even if `addr` $\geqslant 10$ due to a branch misprediction. Second, that secret value is *transmitted* over a covert channel (formed using one or more younger instructions). For example, load `M2` in Figure 4.1 transmits the secret over a cache-based covert channel (displacing the attacker's line in the cache).

Using this distinction, a conservative scheme to protect *all* speculatively accessed data is

```
1 uint8 A[10];
2 uint8 B[256*64];
3 void victim (size_t addr) {
4     if (addr < 10) { // mispredicted branch
5 M1:     uint8 val = A[addr]; // secret is accessed
6 M2:     ... = B[64 * val]; // secret is transmitted
7     }
8 }
```

Figure 4.1: Spectre Variant 1 assuming a 64 byte cache line size. Variables carrying potentially secret data are colored green. If the `if` condition is predicted as true, then the cache line of B indexed by `val` is loaded to the cache (load M2) even though both loads are eventually squashed.

therefore to delay the execution of any instruction deemed capable of accessing a secret (*access instruction* for short) until it becomes non-speculative. For example, if we define access instructions to be "all loads," it isn't possible for `val` in Figure 4.1 to leak an out-of-bound value through the covert channel formed by load `M2`, since we delay executing load `M1` until the branch resolves (and squashes). On the other hand, this scheme has high overhead, as delayed execution blocks execution for all dependent instructions.

The key observation underpinning our solution is that one can improve the above conservative scheme's performance, without hurting security, by *executing and selectively forwarding* the results of speculative access instructions to younger instructions, as long as those younger instructions cannot form a covert channel. For example, suppose the microarchitect designs simple arithmetic (e.g., adds, xors) to have data-independent timing (e.g., implemented with a single-cycle ALU). Then, it is safe to execute and forward the result of load `M1` to these dependent instructions, because their execution cannot reveal the result's value. By issuing load `M1` and the arithmetic early, we improve performance if the branch resolves with a correct prediction.

This chapter designs a framework to selectively forward data in this fashion, providing an efficient mechanism to comprehensively protect all speculatively accessed data. At a high level, our scheme tracks the flow of results from access instructions, through their def-use chains in a manner similar to dynamic information flow tracking [214, 215], until those results reach an instruction, or sequence of instructions, that may form a covert channel. *Only at that later point* do we stop forwarding the access instruction-dependent value. To be secure and efficient, this approach needs to solve two key technical challenges:

Challenge 1: Blocking leakage through all covert channels. First, paramount to deciding when to forward the results of speculative access instructions (called *secrets* for short) is having a complete understanding of how instructions can form covert channels in speculative execution attacks. This is not trivial, as prior attacks have shown there to be many ways to leak a secret (e.g., through loads that interact with the cache [4], SIMD units [172], and port contention [245]).

A key contribution of our proposed framework is a comprehensive study of how instructions can be used to create covert channels and communicate data. In particular, we find that all covert channels are one of two flavors, which we call explicit and implicit channels. First, in an *explicit channel* data is directly passed to an instruction whose execution creates operand-dependent hardware resource usage, and that resource usage reveals the data. For example, how a load impacts the cache depends on the load address [4]. Second, in an *implicit channel* data indirectly influences how (or that) an instruction or several instructions execute, and these changes in resource usage reveal the data. For example, instructions executed after a branch reveal the branch predicate [172, 245].

Implicit channels are related to implicit flow from the information flow literature [207], which is notoriously difficult to deal with in side-channel research [218]. To our knowledge, we are the first to study and provide comprehensive protection for implicit channels in the speculative execution attack setting. Along the way, we also discover new ways that these channels can leak, and also find entirely new forms of implicit channels, unique to speculative microarchitectures.

Challenge 2: Disabling protection as soon as access instructions become non-speculative. Second, to be efficient, it is important to disable protection on data produced by access instructions, as soon as doing so is safe. Consider the example in Figure 4.1. Here, a simple, secure scheme is to execute and forward data from load `M1`, yet wait to issue load `M2` until load `M2` reaches the head of the ROB. This is overly conservative. In fact, it is safe to issue load `M2` as soon as the branch resolves in a correct prediction, as this is the soonest point when the data returned by load `M1` is no longer considered secret (i.e., load `M1` is no longer speculative). To reiterate: *at that earlier point*, we can issue load `M2`. This is important for performance. The later we delay issuing load `M2`, the greater the chance it delays instruction retirement in the ROB.

The general principle is that it is safe to disable protection on data, as soon as the data's producer access instruction(s) have all become non-speculative. This is technically challenging for a variety of reasons, as data can be the result of complicated def-use chains through potentially many access instructions, and other instructions such as arithmetic. Yet, our

solution requires simple hardware and can disable protection on any protected data in a data-independent number of cycles (e.g., 1 cycle), regardless of the complexity of def-use dependencies through older instructions.

Putting everything together, we call our combined protection scheme Speculative Taint Tracking, or STT for short. In addition to proposing STT itself, we provide an extensive formal analysis and prove that STT enforces a novel form of non-interference [199] with respect to speculatively accessed data, given a powerful adversary that can monitor potentially any covert channel at cycle granularity. We show how this implies that with STT enabled, *arbitrary speculative execution is only able to leak retired register file state as opposed to arbitrary program memory.* This means STT comprehensively defeats the worst Spectre attacks, e.g., those which form a universal read gadget [246] such as Spectre Variant 1. We provide an overview of our analysis in this chapter and provide proof details in a companion technical report [247].

Contributions To summarize, this chapter makes the following contributions:

- We provide a comprehensive study of covert channels on speculative microarchitectures, including the first in-depth look at implicit channels (related to implicit flow), new ways implicit channels can leak, and new forms of implicit channels not yet exploited by speculative attacks.
- Based on our study of covert channels, we propose a general framework for preventing speculatively accessed data from leaking over any covert channel.
- We propose a novel scheme to quickly disable protection on flows of data, once the data's producer access instruction(s) becomes non-speculative.
- We formalize our protection mechanisms and show they are able to achieve a strong security definition, akin to non-interference [199], with respect to data returned by speculative access instructions.
- We extensively evaluate STT on 21 SPEC and 9 PARSEC workloads, and find it adds only 8.5%/14.5% overhead (depending on the threat model) relative to an insecure machine, while reducing overhead by 4.7×/18.8× relative to the baseline secure scheme from Section 4.1.

4.2 THREAT MODEL

We assume a powerful adversary that can monitor any microarchitectural covert channel from anywhere in the system, and induce arbitrarily speculative execution to access secrets and create covert channels. (For a more formal definition, see the BitCycle adversary from [30].) For example, the attacker can monitor covert channels through the cache/memory system [4], data-dependent arithmetic [192], port contention [245], branch predictors [191], etc. As in [183], the adversary may try to induce and monitor malicious speculative execution by priming predictors, caches, etc.—from within the victim thread itself (*SameThread* [172]), or from an external context such as an SMT sibling (*SMT*) or nearby processor core (*CrossCore*).

4.2.1 Scope: Protecting Speculatively Accessed Data

A speculative execution attack consists of two components [55, 172]. First, an instruction that reads a potential secret into a register, making it accessible to younger instructions. We call this instruction the *access instruction* [55]. Second, a younger instruction or instructions that exfiltrate the secret over a covert channel. The access instruction is almost always a load [92, 93, 96, 97, 172, 242, 243, 244, 245], but some attacks use a privileged register read [193].

We further distinguish attacks based on whether the access instruction is *transient* or *non-transient*, i.e., doomed to squash or bound to retire, respectively [193]. Figure 4.2 shows the general schema. Note that the covert channel must be transient. Otherwise, all older instructions—including the access instruction—are also non-transient, which means that the attack is a traditional side channel attack (e.g., [5]) and out of scope.

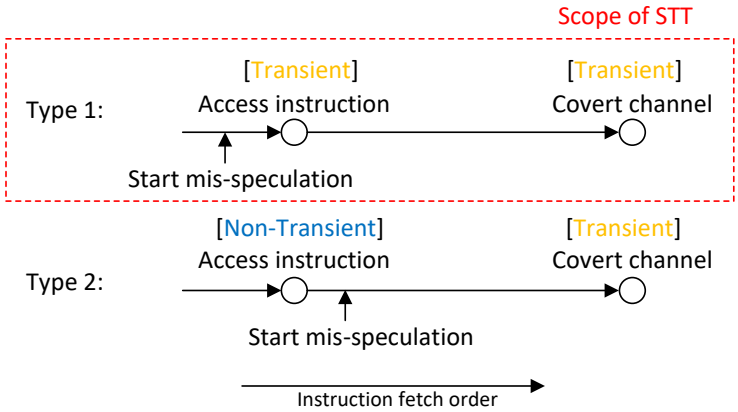


Figure 4.2: Schema for speculative execution attacks. Attacks can be classified into two types, depending on whether the instruction accessing the secret is transient (top) or non-transient (bottom). Our solution protects data returned by transient access instructions.

Our goal is to block attacks involving transient access instructions, which are arguably the most dangerous speculative execution attacks. The reason is that a transient access instruction can often be maneuvered to access data that its correct (not mis-speculated) execution would never access. The worst such attacks can read from any location in memory, which is referred to as a *universal read gadget* [246]. For example, in Spectre Variant 1 (Figure 4.1), misspeculating that a bounds check passes allows the transient access instruction—load `M1`—to read from an arbitrary out-of-bounds address. There are additional universal read gadgets that exploit different program constructs and covert channels [246].

```
1 secret = *addr; // retired (non-transient) access instruction
2 ...
3 if (...) { // mispredicted branch
4     b = B[64 * secret]; // secret is transmitted
5 }
```

Figure 4.3: Example speculative execution attack involving a non-transient access instruction. Blocking this class of attack is out of scope.

In contrast, attacks involving non-transient access instructions cannot create a universal read gadget, because they can only leak *retired (or bound to retire) register file state*. Figure 4.3 depicts such an attack. Here, a `secret` is legitimately accessed by the program, i.e., the access instruction retires. Later, a transient covert channel created through a branch misprediction exfiltrates `secret`. Although such leakage is important to address, it is clearly less dangerous than leaking all of program memory. Moreover, potential leakage of retired state can be reasoned about by programmers and compilers and blocked using complementary techniques (e.g., [30]). Therefore, we consider such leakage out of scope.

4.3 COVERT CHANNELS IN SPECULATIVE EXECUTION ATTACKS

As discussed in Section 4.1, STT executes and selectively forwards the results of speculative access instructions (which are deemed *secrets*) to younger instructions. For security, it is essential to understand how instructions, computing on secrets, can be used to create covert channels. For this, we propose a novel abstraction for covert channels in the speculative execution attack setting, shown in Figure 4.4. We note that, although our scope is protecting data read by speculative access instructions (Section 4.2.1), our analysis here applies to covert channels following non-speculative access instructions as well.

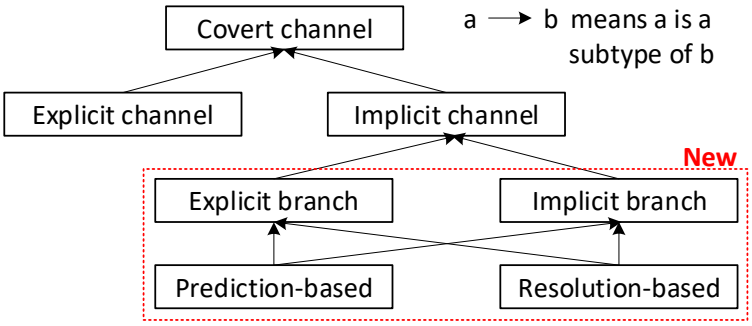


Figure 4.4: Covert channel classification on speculative microarchitectures.

4.3.1 Explicit vs. Implicit Channels

To start, we classify all covert channels as one of two types: explicit channels and implicit channels. An *explicit channel*, related to explicit flow in information flow [207, 218], is one where data (e.g., a secret) is *directly* passed to an instruction whose execution creates operand-dependent hardware resource usage, and that resource usage reveals the data. An *implicit channel*, related to implicit flow [207, 218], is one where data *indirectly* influences how (or that) an instruction or several instructions execute, and these changes in resource usage reveal the data. Examples of explicit channels are memory instructions (e.g., load `M2` from Figure 4.1), variable latency arithmetic instructions [192] and prefetch instructions. Importantly, that load `M2` executes is not secret; it is *how* the load executes (i.e., brings a line into cache at a secret-dependent set) that leaks. Recent speculative execution attacks have also started exploiting implicit channels. Examples are branches with secret-dependent predicates, which influence the instruction cache footprint, program timing [172], execution unit port usage [245], etc.

(a) Control dependency:	(b) Squash dep. (new):	(c) Alias dep. (new):

`if (secret)`
`  load rX <- (rY)`
	`if (secret)`
	`  rX += 64`
	`load rY <- (rZ)`
		`store rX -> (secret)`
		`load rY <- (rZ)`

Figure 4.5: Examples of implicit covert channels revealing `secret`. Assume an older speculative access instruction has already read `secret` into a register, e.g., `M1` in Figure 4.1. The attacker can see the sequence of load addresses sent to the memory system. rX, rY and rZ are registers. Each of these covert channels can be "plugged into" existing attacks as the "Covert channel" in Figure 4.2. For example, in Spectre V1 (Figure 4.1) we can replace load `M2` with one of (a)-(c) above.

A key contribution of this work is finding new ways that implicit channels can leak (Section 4.3.2), and finding entirely new classes of implicit channels related to what we call "implicit branches" (Section 4.3.3). Figure 4.5 gives examples of "traditional" (Figure 4.5(a))

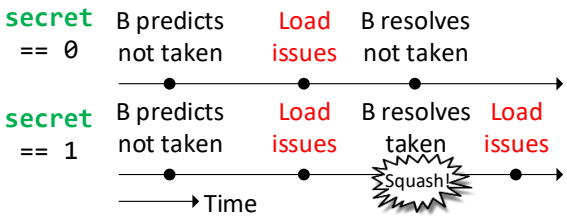


Figure 4.6: Resolution-based implicit channel for example Figure 4.5 (b). When branch (B) resolves, it leaks the secret based on whether a squash occurs. There is an analogous case when the (public) predictor state takes the branch.

and new (Figure 4.5(b)-(c)) channels. We denote the value being revealed through the channel as `secret`. The examples assume the attacker can monitor the cache-based covert channel, but in many cases (e.g., Figure 4.5(a) and (b)) the load can be replaced by an instruction *whose execution timing/etc. does not depend on its input*. Importantly, `secret` is not passed directly as the load address in any of the examples, yet still leaks.

4.3.2 Prediction- vs. Resolution-based Leakage

We make a key observation that on speculative machines, implicit channels can leak secrets at two points: when a control-flow prediction is made (if any) and when that prediction is resolved. Recall, branch prediction and resolution occur in the processor frontend and backend, respectively. This creates new types of leakage depending on the adversary's capability. In the following, consider a branch whose predicate depends on a secret.

At *prediction time*, the sequence of instructions fetched after this branch is fetched (after branch prediction but before resolution) leaks secrets if the predictor structures have been updated based on secret information at some time in the past. For example, if an attacker runs repeated experiments and the branch predictor is updated speculatively based on how the branch resolves, the branch predictor "learns" the secret and will make future predictions based on the secret.

At *resolution time*, the branch can also leak the secret *even if the predictor state has not been updated based on secret data*, because incorrect predictions will cause a pipeline squash. See the code snippet in Figure 4.5(b), whose timing is shown as a function of the secret in Figure 4.6. If the attacker knows the branch will predict not taken (e.g., by priming it beforehand [4]), a squash means the branch was actually taken. The adversary can observe the squash through different effects, e.g., program timing or the fact that the load issues twice. Importantly, Figure 4.5(b) would not be considered a leak in traditional implicit flow, because the load is control- and data-independent of the branch.

4.3.3 Explicit vs. Implicit Branches

We make a key observation that on speculative machines, non-control flow instructions that speculate can similarly influence control flow based on conditions in the pipeline. For example, in Figure 4.5(c) there is no control flow instruction and the load address seemingly does not depend on secret data. Note, stores in isolation don't form covert channels because they are not performed until they retire.

Yet, there may still be an implicit channel. For example, on a machine that performs memory dependence prediction [248], if the store address resolves after the load is issued, the load will squash based on whether `secret==rZ`, causing a similar pipeline disturbance as discussed above.[13]. In that attack, an *access instruction* reads stale data through a store bypass. Our attack is concerned with store bypass used as a covert channel. Likewise, if store-load forwarding is enabled, the load conditionally accesses the L1 cache depending on whether `secret==rZ`. Many additional hardware mechanisms, e.g., memory consistency speculation [249], value prediction [250], etc., create similar issues.

An important observation is that hardware optimizations like those above can be modeled as *implicit branches*, whereas explicit control-flow instructions like branches can be viewed as *explicit branches*. That is, the store bypass in Figure 4.5 (c) can be rewritten as "if (`secret` == rZ) { rY = rX; } else { load rY <- (rZ); }" where the "implicit branch" direction is predicted if `secret` has not yet resolved. In this sense, implicit branches may also leak at prediction and/or resolution time (Section 4.3.2), e.g., if the architecture uses a store set predictor [251].

Summary. To summarize, covert channels can be explicit or implicit, and implicit channels can be further broken down based on when they leak and their branch type. The next section uses these observations to block leakage through all channel types with a unified mechanism. For reference, Table 4.1 specifies channel types for existing attacks and a variety of hardware optimizations.

4.4 SPECULATIVE TAINT TRACKING

Speculative Taint Tracking (STT) is a low-overhead framework that protects data accessed under misspeculation, such as data obtained by an out-of-bounds array access. We refer to such data, that a non-speculative execution would never read, as *secret*.

[13]Note, this is not the already known Spectre Variant 4 (SSB) attack [94],invisispec

Channel	Spectre PoC?	Type	Branch Type
Cache timing [6, 15]	Spectre V1 [4]	Exp	-
Execution unit timing [22, 192]	-	Exp	-
SIMD utilization	NetSpectre [172]	Imp	Exp
Port contention [66]	SmotherSpectre [245]	Imp	Exp
Store-load forwarding	-	Imp	Imp
Mem. dep. prediction [248]	-	Imp	Imp
Mem. consist. speculation [249]	-	Imp	Imp
Value prediction [250]	-	Imp	Imp

Table 4.1: Classifying existing attacks and covert channel-creating hardware structures. A channel's *Type* can be either Explicit (Exp) or Implicit (Imp), c.f. Section 4.3.1. An implicit channel's *Branch Type* is likewise Exp or Imp, c.f. Section 4.3.3. Attacks utilizing implicit channels may be either prediction- or resolution-time (Section 4.3.2), thus we leave that field out.

In a manner similar to dynamic information flow tracking (DIFT) [214, 215], STT "taints" secret data. The STT framework (Section 4.4.1) defines which data should be tainted, which instructions might leak it and thus should be protected, and when protection can be disabled. Section 4.4.2 describes how STT tracks the flow of tainted data between instructions and how—in contrast to conventional DIFT schemes—it automatically "untaints" data once the instruction that produces it becomes non-speculative. Based on taint information, STT applies novel protection mechanisms to block explicit covert channels (Section 4.4.3) and implicit covert channels (Sections 4.4.4–4.4.5).

4.4.1 Framework and Concepts

STT has three characteristics, which are set at design time.

Which data should be tainted? The microarchitecture classifies instructions capable of reading secrets under speculative execution as *access instructions*. We focus on the case where access instructions are loads, as this will be sufficient to block universal read gadget attacks (Section 4.2.1). STT taints the output of any speculative access instruction.

When can data be untainted? The microarchitecture specifies when a speculative access instruction is no longer considered a security threat, referred to as the instruction's *visibility point* [183]. The visibility point depends on the attack model. In the *Spectre* model, an instruction has reached the visibility point if all older control-flow instructions have resolved. In the *Futuristic* model, an instruction has only reached this point if it cannot be squashed.

(The futuristic model protects data read by any possible hardware speculation, blocking additional attacks such as Meltdown.) Instructions reach the visibility point in the fetch order. We call access instructions before and after the visibility point *unsafe* and *safe*, respectively, as instructions which have passed the visibility point are not speculative from a security perspective. STT untaints the output of an access instruction once it becomes safe.

Who can leak secrets? The microarchitecture classifies certain instructions as *transmit instructions*. (Note that an instruction can be neither, either, or both an access and a transmit instruction.) STT considers the execution of a transmit instruction as an explicit covert channel that leaks its argument (Section 4.3). Classifying only loads as transmitters will block memory system-related explicit covert channels. Classifying all instructions that have operand-dependent hardware resource usage as transmitters will block all explicit channels. We assume that stores trigger a cache coherence invalidation only on retirement, or else are defined as transmitters.

To block implicit channels, STT requires the microarchitect to classify *explicit branch* instructions, which affect control flow, and to identify the *implicit branches* that represent additional sources of data-dependent resource usage, e.g., store-to-load forwarding, memory consistency speculation [249], etc. Section 4.4.4 discusses implicit branches in detail.

Identifying Access Instructions, Transmit Instructions and Implicit Branches

Here, we describe how microarchitects can identify access and transmit instructions, and implicit branch conditions.

An instruction should be an access instruction if it has the potential to read a secret. Except for loads, there are only a handful of such instructions (e.g., privileged/configuration register reads or I/O instructions), which can be identified manually.

An instruction should be a transmit instruction if its execution creates operand-dependent resource usage that can reveal the operand (partially or fully). Identifying implicit branches is similar: the architect must analyze whether the resource usage of some in-flight instruction changes as a function of *some other* instruction's operand. Examples of both transmitters and implicit branch-based channels are given in Table 4.1. This informal definition can be formalized by analyzing (offline) how information flows in each functional unit at the SRAM-bit and flip-flop levels to determine whether resource usage depends on the input value, in the style of the OISA [30] or GLIFT [218] formal frameworks. We leave such analysis to future work.

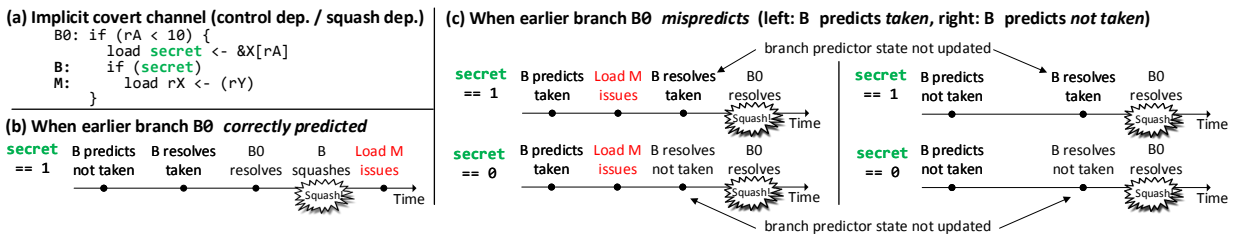


Figure 4.7: STT executing the code in (a), which includes an untainted branch `B0`, an access instruction reading `secret`, and an implicit channel.

4.4.2 Taint and Untaint Propagation

STT tracks information flow from access instructions to younger in-flight instructions. STT *taints* the output register of an unsafe access instruction and propagates taint using standard taint tracking rules, namely that an instruction's output register is tainted if any of its input registers are tainted. Unlike conventional DIFT, STT *automatically* untaints data. When an access instruction becomes safe, its output register is *untainted*. Untaint information is also propagated, so that when all the data dependencies of an instruction become untainted, the instruction's output is untainted.

We defer the details of STT's taint/untaint tracking implementation to Section 4.5. At a high level, taint propagation is piggybacked on the existing register renaming logic in a modern out-of-order core. As an instruction enters the frontend and its registers are renamed, the instruction's output register is tainted if (1) it is an access instruction or (2) any of its input (physical) registers are tainted. Tainting is therefore fast. Propagating untaint is non-trivial, because dependency chains can be long and each instruction can have many data dependencies whose taint status needs to be tracked. STT addresses these challenges with a novel fast untaint algorithm in Section 4.5. In this section, we simply assume that taint/untaint information is available.

Unlike prior DIFT schemes [30, 171, 214, 215], STT does not require tracking taint in any part of the memory system (TLB, caches, or memory) or across store-to-load forwarding. The reason is that the taint of the output of a load—which is an access instruction—is determined *only* based on whether it has reached the visibility point: If a load is unsafe, its output is always tainted. If a load is safe, every instruction on which it depends has also reached its visibility point (since this happens in-order) and so the load's output is not tainted.

4.4.3 Blocking Explicit Channels

STT blocks explicit channels by delaying the execution of any transmit instruction whose operands are tainted until they become untainted.[14] This scheme imposes relatively low overhead because it only delays the execution of transmit instructions if they have tainted operands. For example, a load that only *reads* a (potential) secret but does not transmit one—such as load `M1` in Figure 4.1—executes without delay. Load `M2`, however, will be delayed and eventually squashed, thereby defeating the attack.

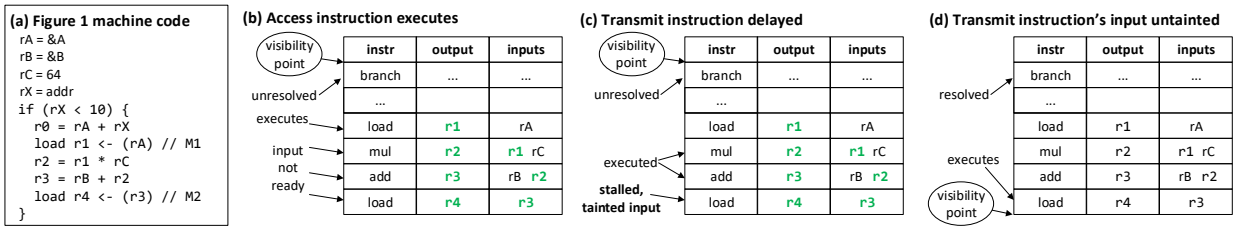


Figure 4.8: Snapshots of ROB state during the STT execution of the Spectre V1 code, in the Spectre threat model. (Tainted registers are green.)

Figure 4.8 depicts this scenario in detail. Figure 4.8(a) shows a sequence of instructions executing the Spectre V1 code; load `M1` is an access instruction. In Figure 4.8(b), the access instruction has been executed, and its output and all dependencies are tainted. Non-transmit dependent instructions can freely execute, but any transmit dependent instruction like `M2` is stalled (Figure 4.8(c)). If the speculation succeeds (i.e., `rX` $<$ `10`), the branch resolves as correct and the access instruction becomes safe (assuming the Spectre threat model defined in Section 4.4.1). In this case, its output becomes untainted and the transmit instruction is allowed to execute (Figure 4.8(d)). Although in this example the transmit instruction becomes safe together with the access instruction, this is not true in general (e.g., if there is an unresolved branch between them). Thanks to STT's untaint mechanism, however, even an unsafe transmitter (i.e., that has not reached the visibility point) whose input becomes untainted can execute without having to delay until it reaches the visibility point or head of ROB.

In contrast, if the branch is mispredicted (i.e., `rX` $\geqslant$ `10`) the transmitter remains stalled until it is eventually squashed along with the access instructions it depends on.

Protection strategies. STT can apply different protection strategies to transmit instructions with tainted arguments. We chose to delay execution for simplicity, and also because this allows us to prove non-interference (Section 4.6). Yet, other protections are possible

[14] Notice that for loads, delaying execution implies delaying the TLB lookup.

which create security-performance trade-offs. For example, one can combine STT with a scheme such as InvisiSpec [183], which would allow loads with tainted arguments to be executed earlier.

4.4.4 Eliminating Implicit Channels

STT blocks implicit channels by enforcing an invariant that the sequence of instructions fetched/executed/squashed never depends on tainted data. That is, *STT makes the program counter independent of tainted data.*

A key challenge in enforcing this invariant is how to maintain efficiency. For example, a strawman DIFT approach to block implicit channels would be to consider the execution of *any* instruction following a branch with a tainted predicate (or *tainted branch*) as an implicit channel, and delay the execution of all such instructions. This approach would impose high overhead, as it requires delaying execution of all instructions following a tainted branch until the branch predicate becomes untainted. Even then, such an approach would not block implicit channels caused by implicit branches (Section 4.3.3), which are unique to the speculative execution setting.

To efficiently maintain the STT program counter invariant, we introduce two general principles to neutralize the sources of implicit channels identified in Section 4.3.2:

- *Prediction-based channels* are eliminated by preventing tainted data from affecting the state of any predictor structure.
- *Resolution-based channels* are eliminated by delaying the effects of branch resolution until the branch's predicate becomes untainted.

In the following, we discuss how STT applies these principles to eliminate implicit channels over explicit and implicit branches. Implementation details are presented in Section 4.5.

Explicit Branches

To prevent implicit channels through explicit branches (i.e., control flow instructions), STT modifies the baseline microarchitecture as follows.

Prediction-based channels. STT requires that every frontend predictor structure be updated based only on untainted data. This makes the execution path fetched by the frontend unaffected by the output of unsafe access instructions. STT passes a branch's resolution

results to the direct/indirect branch predictors only after the branch's predicate and target address become untainted; if the branch gets squashed before this, the predictor will not be updated.

Figure 4.7(c) demonstrates the effect of STT on a speculative execution of the code snippet in Figure 4.7(a), in which the branch `B0` is mispredicted as taken. No matter how many experiments the attacker runs, the predicted direction of the branch `B` will not be a function of `secret`, because the branch predictor is not updated when `B` resolves. As a result, the execution path does not depend on `secret` (top vs. bottom)—it only depends on the predicted branch direction (left vs. right).

STT does not need to change how the *return address stack* (RAS) [252] (which predicts `RET` results) is updated; we discuss this issue shortly.

Resolution-based channels. STT delays squashing a branch that resolves as mispredicted until the branch's predicate becomes untainted. As a result, a transient branch with a tainted predicate (such as the branch `B` in Figure 4.7(c)) will never be squashed and re-executed, preventing the implicit channel leak shown in Figure 4.6. As Figure 4.7(c) shows, the transient branch `B` is eventually squashed once an older (mispredicted) branch with an untainted predicate squashes. Thus, the squash does not leak any information about the transient branch's resolution. Importantly, note that it is safe to resolve a branch *as soon as* its predicate becomes untainted, even if an *older branch with a tainted predicate* has not yet resolved.

To summarize, where the strawman DIFT approach would stall the execution of any instruction following a tainted branch, STT lets the instructions execute, and only increases the latency of *recovering* from a *tainted branch* misprediction. For example, in Figure 4.7(b), the load `M` does not execute immediately after the tainted branch `B` resolves, because `B`'s predicate is tainted at this point. This is in contrast to a modern processor, which squashes a mispredicted branch and starts executing the correct path immediately upon resolving the branch [253, 254, 255]. Fortunately, tainted branch mispredictions are only a small fraction of overall branch mispredictions (Section 4.7), which are infrequent in the first place because successful speculation requires accurate branch prediction.

Handling the RAS. With STT, the RAS is updated by the frontend as usual: as `CALL`s and `RET`s are fetched, they push and pop return addresses from the RAS. The reason is that STT makes the predicted execution path up to a `CALL`—and therefore the return address it pushes—independent of tainted data. Thus, `RET`s can safely pop from the RAS as usual, as the values they pop do not depend on tainted data.

STT does need to delay squashes due to RAS misprediction until the mispredicted `RET` reaches its visibility point, because the RAS misprediction resolution depends on the return address the `RET` reads from the stack. We assume that the baseline microarchitecture can repair the RAS after a squash to undo the effects of squashed `CALL`s [256]. We do not require a perfect RAS repair algorithm. Our only requirement is that the repair does not depend on tainted data. In particular, any repair algorithm whose input is only the RAS state [256] is fine.

Implicit Branches

Implicit branches frame microarchitectural mechanisms that observably change how the processor executes instructions as being caused by a branch "injected" into the execution (Section 4.3.3). Speculations—such as memory dependence speculation [248], value prediction [250], memory consistency speculation [249], etc.—are *predictions* of an implicit branch.

Formulating microarchitectural mechanisms as implicit branches allows STT to block leakage through them using a similar mechanism as was used to block prediction- and resolution-based channels for explicit branches. This section walks through this process for several common optimizations. While we cannot exhaustively discuss all known processor optimizations, the successful systematic application of STT's principles is evidence that they should generalize to other optimizations.

Implicit branch without prediction. Consider a store-to-load forwarding design without memory dependence speculation (e.g., [257]). In such a design, a load stalls until the addresses[15] of all older stores have resolved. As shown in Figure 4.9, store-to-load forwarding creates an implicit channel because the load `M2` accesses the cache only if it does not alias with the store S, i.e., if the secret `r2` is not 17. Figure 4.10 shows store-to-load forwarding framed as an implicit branch, denoted by impIf in the code.

This implicit branch only creates a resolution-based implicit channel, because the branch is not predicted. We eliminate this channel by delaying the resolution of the implicit branch until its predicate is untainted. In our example, this means that `M2` (and so all instructions data-dependent on it) are delayed until both `r2` and `r0` become untainted, which means they are delayed until the mispredicted explicit branch `B` squashes. (Section 4.4.5 describes an STT optimization that handles store-to-load forwarding more efficiently.)

[15]We refer to physical addresses simply as "addresses."

In general, the store-to-load forwarding implicit branch predicate checks that *every* older in-flight store does not alias with the load, i.e., it is a conjunction of the "no alias" predicate for each older store. For example, Figure 4.10 should have:

```
impIf (AND_{S | in-flight store S older than M2} S.addr ≠ &Y[r0]).
```

Crucially, the implicit branch predicate never depends on prior explicit or implicit branches. This ensures that implicit branch predicates do not grow more complicated as more speculative instructions enter the ROB. Because of STT's invariant that instructions fetched/executed/squashed thus far are independent of tainted data, we need only guarantee that subsequent instructions do not create an implicit channel by (in this case) delaying the branch's resolution until its predicate is untainted. Implementation-wise, our STT microarchitecture (Section 4.5) efficiently tracks the taint of addresses in the load-store queue (LSQ), which allows resolving the implicit branch as part of the store-to-load forwarding logic.

```
1      r0 = 17
2  B:  if (r1 < size) {    // mispredict
3  M1:   load r2 <- &X[r1] // access
4  S:    store r0 -> &Y[r2]
5  M2:   load r3 <- &Y[r0] // transimit
6      }
```

Figure 4.9: Store-to-load forwarding.

```
1      r0 = 17
2  B:  if (r1 < size) {      // mispredict
3  M1:   load r2 <- &X[r1]   // access
4  S:    store r0 -> &Y[r2]
5        impIf (r2 != r0)
6  M2:     load r3 <- &Y[r0] // transimit
7        else
8          r3 = r1
9      }
```

Figure 4.10: How store-to-load forwarding in Figure 4.9 forms an implicit branch.

Implicit branch with prediction. Consider now *memory dependence speculation*, where the processor might execute a load (read from memory) speculatively and squash it if an older store ends up aliasing with it. This is simply a *prediction* of the store-to-load forwarding implicit branch. We eliminate its predictor-based channel by requiring that the relevant predictor (e.g., a store set predictor [251]) be updated only by untainted data, i.e., only after the implicit branch predicate becomes untainted. The prediction typically also depends on the LSQ state—for example, a prediction is made only if there are older stores with unresolved addresses. In this case, eliminating the predictor-based channel also requires delaying the prediction until the relevant LSQ state becomes untainted, e.g, until the addresses of older stores become untainted. We eliminate the resolution-based channel by delaying the squashing

of the load on a misspeculation (misprediction of the implicit branch) until the branch's predicate becomes untainted.

Squashing implicit branches early-on. An advantage of implicit branches is that the microarchitecture knows the structure of their predicates. In some cases, this knowledge allows STT to untaint an implicit branch predicate early-on, based on the observation from GLIFT [218] that "the output of a logical function should only be untrusted if some untrusted input actually had an opportunity to affect the output."

For example, suppose that the memory dependence predictor predicts that the implicit branch in Figure 4.10 is taken, and that this turns out to be a misprediction. Naively, STT would need to delay squashing the implicit branch until the load's address and the addresses of *all* older stores become untainted, as the predicate is a function of them. However, the GLIFT observation implies that we need to delay the squash only until *one* term that evaluates to *false* becomes untainted—i.e., until the addresses of some older aliasing store and the load are untainted. At this point, the result of the AND becomes a function of only untainted data—the attacker only learns that an alias between untainted addresses exists.

Statically-predicted implicit branches. Several common forms of speculation can be formulated as implicit branches that are predicted statically, and therefore have no predictor-based channels. We only need to eliminate resolution-based channels by identifying the branch's predicate and, if the implicit branch is mispredicted, delay the resulting squash until the predicate becomes untainted. Below, we consider the example of memory consistency speculation in a multicore processor. Load-load ordering is another example with similar characteristics.

Memory consistency speculation. A memory consistency model (or memory model) specifies the order in which a processor's memory operations are performed and observed by other processors in the system [258]. Memory consistency speculation [249] allows the processor to maintain any required ordering between loads while still issuing them out of order. The idea is that if two loads, `M1` and `M2`, must appear to execute in program order (`M1` before `M2`), then `M2` can still execute before `M1` but will be squashed if the data loaded by `M2` is invalidated by another processor or gets evicted from the cache before `M2` retires.

Memory consistency speculation is thus a static prediction of *true* for an implicit branch predicate that the cache line a load accesses remains valid until the load retires. It turns out that with STT's delayed execution protection strategy, the resolution of this implicit branch predicate can only occur when it becomes untainted. Therefore, its resolution does not need to be delayed—i.e., consistency squashes can be performed when signaled, as usual.

The reason is that a consistency squash of load `L` can be signaled only after `L` accesses the cache, which implies that `L`'s address is untainted. In addition, the memory access that triggers the line's invalidation/replacement is independent of tainted data. This holds because such a memory access occurs either because of a load/store instruction or a hardware prefetch. With STT, loads and stores access memory only if their address argument is untainted, which also implies that hardware prefetching state at the caches is a function of untainted data. Therefore, if the implicit memory consistency branch predicate evaluates to *false*, it is due to an untainted term.

4.4.5 Optimizing Store-to-Load Forwarding

We now describe an optimization that allows resolving the store-to-load forwarding implicit branch without waiting for its predicate to become untainted. The insight is, because store-load forwarding can only result in two observable outcomes (issue the load, or forward from a prior store) it is feasible to hide which occurs. Specifically: when the load address becomes untainted, we issue the load unconditionally. (That is, we do not wait for the prior store addresses to become untainted.) If forwarding should occur, we ignore the value read from memory and use the forwarded store value as the load's output. The load's output register is written only after the memory access completes, to guarantee that the timing of younger instructions is unchanged from the "no forwarding" case.

The optimization maintains STT's property that tainted data does not influence the execution path. The reason is that the resolution of this implicit branch only determines whether the load will access memory (and its output). It does *not* influence the execution path as long as the load is unsafe. This principle is general to other implicit branches whose resolution does not determine whether instructions retire/squash.

4.5 MICROARCHITECTURE

We now present a microarchitecture for STT. The key challenge in the implementation is how to implement the automatic untaint operation (Section 4.4.2). We present a unified mechanism that implements taint and untaint, and show how it can be used to block/eliminate both explicit and implicit channels. In the following, instructions are labeled with monotonically increasing numbers, which we loosely refer to as their position in the ROB (younger instructions are assigned larger numbers).

4.5.1 Main Ideas

We make a key observation that helps implement untainting. Since instructions reach their visibility point in program order (Section 4.4.1), to untaint the arguments for an instruction i, it suffices to wait for the youngest access instruction that is causing the taint for i to reach the visibility point. We call this instruction the *Youngest Root of Taint* (YRoT) of i.

With this approach, we do not need to track exact def-use chains between instructions. Conceptually, we track the position of the YRoT for each instruction in the ROB. Then, we broadcast the ROB position of each access instruction as it reaches the visibility point. Each younger instruction whose YRoT is smaller or equal to the broadcasted value becomes untainted. If multiple access instructions reach the visibility point in the same cycle, we can broadcast the maximum index of all of them (instead of all of their indices). We note that logic indicating which instructions reach the visibility point is provided by prior work on InvisiSpec [183].

Untainting can trigger different operations. When a transmit instruction's arguments become untainted, it can execute (Section 4.4.3). When an explicit branch predicate is untainted, subsequent instructions are squashed if the branch was mispredicted (Section 4.4.4). Finally, when an implicit branch predicate is untainted, any observable effect related to the implicit branch, e.g., a squash due to memory dependence prediction (Section 4.4.4), can occur.

4.5.2 Hardware Changes to Frontend

We now describe an architecture that implements STT (Figure 4.11). Our architecture is meant to model a modern speculative out-of-order multicore with optimizations such as those described in Section 4.4.

In the processor frontend (up to dispatch to execution units) the main hardware changes are to add logic to generate the Youngest Root of Taint (Section 4.5.1) for fetched instructions. We reuse logic from the InvisiSpec paper [183] paper to generate the visibility point (VP, shown as ①). For example, in the Spectre model the VP equals the ROB index of the oldest unresolved branch.

Tracking Youngest Root of Taint. We calculate the Youngest Root of Taint (YRoT) in the processor rename stage (Figure 4.11 ②). We add two new fields to the entries in the rename table (which maps logical registers to physical registers): YRoT and the *access instruction ROB index* (AccessInstrIdx), both of which require $\log_2$(ROBSize) bits. YRoT

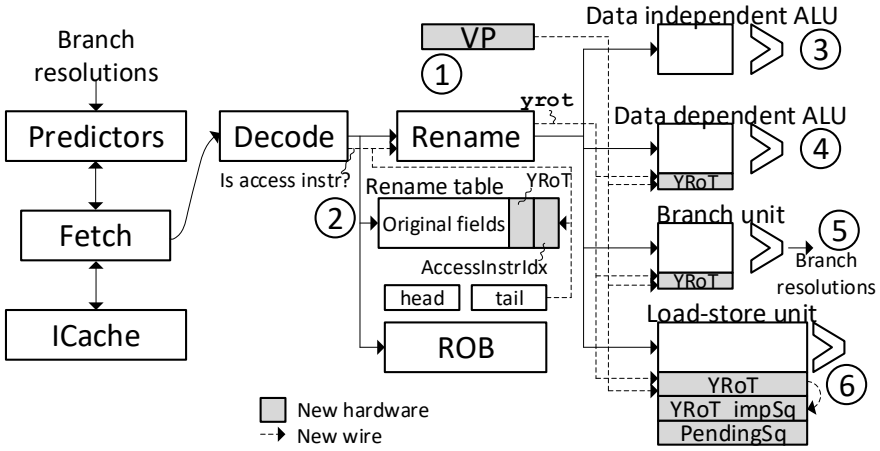


Figure 4.11: Microarchitecture with support for STT. The shaded blocks represent new hardware added to support our scheme; the dashed lines are new wires. The per-instruction Youngest Root of Taint is denoted `yrot`, whereas fields added to hardware tables are denoted YRoT.

tracks the Youngest Root of Taint of the instruction that last produced each logical register in program order. AccessInstrIdx records the ROB index of the last producer for each logical register, if that producer was an access instruction, or -1 otherwise.

Suppose a new instruction is being renamed. Then we calculate the Youngest Root of Taint for that instruction (denoted `yrot`) as the YRoT of the argument, in the rename table, which reaches the visibility point the latest. If an argument was directly produced by an access instruction, that instruction's index in the ROB directly becomes the YRoT for that argument (which is known through the AccessInstrIdx field). For example, suppose that this instruction has source registers Rs1 and Rs2. Let `RT` be short for rename table. Assume that when the source for a logical register Rx is in retired architectural state, `RT[Rx].YRoT = -1`. Then at rename time, we compute the instruction's `yrot` as:

```
yrot = max( ((RT[Rs1].AccessInstrIdx == -1) ? RT[Rs1].YRoT : RT[Rs1].AccessInstrIdx),
            ((RT[Rs2].AccessInstrIdx == -1) ? RT[Rs2].YRoT : RT[Rs2].AccessInstrIdx) );
```

Recall, younger instructions have larger ROB indices, hence the use of max. If the instruction has a different number of arguments, the same max is taken over all of them.

If the instruction has a destination register, call it Rd, then we update the rename table as `RT[Rd].YRoT = yrot`. As previously mentioned, `RT[Rd].AccessInstrIdx` is set to the instruction's index in the ROB, if the instruction is an access instruction (determined in decode), or -1 otherwise. If the instruction does not update a register (e.g., a branch or store), the rename table is not updated because no logical register is being updated.

After rename, the instruction is dispatched to a reservation station (RS) based on its type. Depending on the instruction type, `yrot` may travel with the instruction and be stored in the RS (see below).

Importantly, this design assigns each instruction a `yrot` without adding new ports to the rename table or ROB. The YRoT and AccessInstrIdx fields for each entry are read along with the logical to physical register mappings that are read per-argument in the rename table already. As with the normal logical-to-physical register mappings, the YRoT and AccessInstrIdx in each entry needs to be restored after a squash.

4.5.3 Hardware Changes to Backend

At dispatch time, each instruction travels with its `yrot`. We now describe logic changes at the reservation stations (RS) for each instruction type, based on whether those instructions can form explicit and/or implicit channels (Section 4.3).

Our goal is design simplicity, and note that many optimizations are possible. While we cannot cover every proposed microarchitectural optimization, we cover an example for instructions that cause neither, both or one of explicit/implicit channels. The mechanisms can be generalized to other instructions. Recall, the microarchitecture is responsible for denoting each instruction as potentially creating explicit and/or implicit channels (Section 4.4.1).

Data-Independent Arithmetic. In the simplest case, the instruction performs a simple task that cannot create either an explicit or implicit covert channel (e.g., single-cycle add, xor without side effects), shown in Figure 4.11 ③. In this case, there are no changes to the RS and the `yrot` is dropped. Such instructions can execute as soon as their arguments are available, even if they are tainted. This is key for performance.

Data-Dependent Arithmetic. Instructions may be capable of creating only explicit channels, such as arithmetic with data-dependent timing (e.g., a multiplier [192]), see Figure 4.11 ④. If the microarchitecture classifies these as transmitters, the `yrot` of each instruction waiting in the RS is stored alongside the instruction in a new field called YRoT. When the VP changes, we calculate for each instruction i in the RS:

$$\text{instr } i \text{ can execute } = \texttt{yrot}_i < \text{VP}. \tag{4.1}$$

Our current design performs these checks in parallel, for each RS entry, in a fashion similar to instruction wakeup logic that checks if a dependency is ready.

Recall, `yrot` is based on each instruction's def-use chains, not where each instruction appears in program order. So, instruction wakeup due to `yrot` still allows instructions to execute out of order once their arguments become untainted.

Branches. Instructions may be capable of creating only implicit channels, such as conditional/unconditional branches and jumps, see Figure 4.11 ⑤. We handle these cases with the same mechanism as in the previous case for Data-Dependent Arithmetic: the `yrot` is stored alongside each branch in the branch RS (branch unit) and wakeup occurs by performing the same comparison between `yrot` and VP. This ensures that branch resolution only occurs when the branch's predicate and target (if any) are untainted. This design also avoids any modification to the branch predictors in the frontend, because only executions based on untainted data will update the predictors.

Loads and Stores. Finally, there are instructions which can create both explicit and implicit channels such as stores and loads, see Figure 4.11 ⑥. Discussed in Section 4.3.1, memory instructions are an important type of explicit channel. At the same time, loads and stores can also create implicit channels due to hardware features such as store-to-load forwarding, memory dependence speculation and memory consistency checks across cores (Section 4.4.4).

In isolation, a store creates neither explicit or implicit channels because we assume stores are performed at retirement. Yet, stores may alias with younger loads; we thus must store the `yrot` for each store, calculated in rename, along with the store, to calculate the predicate for implicit branches.

We block channels related to loads as follows. First, loads are stored in the LSQ with their `yrot`, as with previous cases, in a new YRoT field. Loads are also assigned two additional fields: PendingSquash (1 bit), and YRoT_impSquash (same width as the YRoT field). YRoT is used in the same way as before, to notify when the load address is untainted (can no longer form an explicit channel), at which point it is safe to perform the load.

When the load address becomes untainted, it may require store-load forwarding or memory dependence speculation. We handle store-load forwarding as discussed in Section 4.4.5: we unconditionally perform the load unless all prior stores are resolved and untainted. PendingSquash and YRoT_impSquash are used to handle memory dependence speculation. After the load is performed, if it suffers an alias to an earlier store whose address resolves late, we set YRoT_impSquash to the YRoT of the store causing the alias and set the PendingSquash bit (but do not perform the squash). If PendingSquash is set and YRoT_impSquash ¡ VP, we perform the squash. (This check requires analogous logic as comparing the normal `yrot` values

to the VP.) This is equivalent to performing the squash when the implicit branch predicate becomes a function of untainted data (see Section 4.4.4). As explained in Section 4.4.4, a memory consistency violation simply squashes when it is signaled.

Multiple late resolving stores may alias with the load, resolving one after another. In this case, YRoT_impSquash is set to the min YRoT_impSquash of any store causing an alias. This is important for security. Once any memory violation occurs, it will eventually cause a squash, unless a squash is triggered beforehand by older instructions in the ROB. If the memory violation itself causes the squash, we must reveal that squash only when the implicit branch predicate is untainted. This moment is exactly when the min YRoT_impSquash of any alias reaches the visibility point. The logic for repeatedly updating the YRoT_impSquash in this fashion piggybacks off of the existing LSQ logic for detecting aliases.

4.6 SECURITY ANALYSIS

We formally prove that STT in the Spectre threat model enforces a novel notion of non-interference [199] appropriate for enforcing privacy of speculatively accessed data. (The formal proof for the futuristic model is ongoing work.) Our proof applies to a strong adversary that observes the instructions fetched, when and which functional units are busy (i.e., resource usage and port contention), and the target address of every cache/memory access. (See Section 4.2.) This section summarizes the key ideas and results; the details appear in a companion technical report [247].

We formally model processors as state machines. We define an STT machine that is a detailed model of a speculative out-of-order processor with STT. In addition to registers and memory, its *state* includes hardware structures such as branch predictors, the ROB, LSQ, etc. Its state also includes taint bits for registers. A *processor logic* defines how the state changes at every cycle. In each cycle, the processor logic performs *events* that modify the machine's state. These events model microarchitectural events such as instruction fetch, execution by a functional unit, squashes, retirement, tainting/untainting, etc. The STT machine models the protections described in Section 4.4 (e.g., delaying execution of tainted transmit instructions) and the fast untaint mechanism described in Section 4.5.

We show that the STT machine provides the following non-interference security guarantee: at each step of its execution, the value of a *doomed* register, that is, a register written to by a speculative access instruction that is bound to squash, does not influence future visible events in the execution. The key challenge is that when an instruction executes, we do not know whether it is going to squash or not. We address this by considering a simple in-order processor model, which we use to verify the STT machine's branch predictions against their

true outcome, obtained from the in-order processor. Specifically, at each step of the STT machine's execution of a program, in our formal analysis we maintain an auxiliary bit of state, mispredicted, that is only set to `true` if the prediction of one of the preceding branches differs from the outcome of a corresponding branch in the in-order execution of the program.

With the way to identify whether the STT machine mispredicts at each point of the execution, we are able to distinguish doomed registers: a register r is doomed in a state σ with a given mispredicted flag, if it gets tainted in the shadow of what the in-order processor identifies as a misprediction, i.e., while mispredicted = `true`. For each register, we also maintain an auxiliary state indicating whether it is doomed. We refer to the STT machine state coupled with its auxiliary state as an *extended* state. Given two extended STT states, κ_1 and κ_2, we say that $\kappa_1 \approx \kappa_2$ holds if κ_1 and κ_2 only differ by values of doomed registers.

We prove the following theorem, which states that values of doomed registers neither influence which events the STT machine executes at each cycle nor gets leaked into the rest of the state by those events. We parameterize the theorem by an *observability function* view, which models the adversary's view [30], i.e., it projects event traces onto the parts the adversary can observe. The theorem holds in particular for the strong adversary described above, which observes the entire event trace.

Theorem 4.1. At any cycle t, given two extended states κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds, if τ_1 and τ_2 are the sequences of events the STT processor logic performs at the cycle t from κ_1 and κ_2 respectively, then the following holds:

1. $\mathsf{view}(\tau_1) = \mathsf{view}(\tau_2)$ holds, and
2. for the extended states κ_1' and κ_2' resulting from executing τ_1 and τ_2 respectively, $\kappa_1' \approx \kappa_2'$ holds.

The theorem proves that $\kappa_1 \approx \kappa_2$ is an invariant preserved by each cycle of the machine execution. Since this invariant is inductive, we obtain the following corollary: at any cycle t in any extended state κ, changes to doomed registers do not influence the future of the execution of the machine. In particular, they never influence the program counter's value, which is sufficient to eliminate traditional implicit covert channels.

4.7.1 Experimental Setup

Simulator setup. We evaluate STT with the Gem5 [259] simulator, which models the performance implications of speculative instructions. We run SPEC CPU2006 [260] and PARSEC 3.0 [261] benchmarks, as representatives of both single-threaded and multi-threaded programs. For SPEC, we use the *reference* input size, and launch detailed simulation for 1 billion instructions after skipping the first 10 billion instructions. PARSEC benchmarks are all run with eight cores with the *simmedium* input size (except x264, whose input size is *simsmall*). A detailed architecture specification used for all schemes is shown in Table 4.2.

Parameter	Value
Architecture	1 core (SPEC) or 8 cores (PARSEC) at 2.0GHz
Core	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

Table 4.2: Parameters of the simulated architecture.

Microarchitectural features modeled in Gem5. In Gem5, instructions that create explicit channels are loads. The simulator also models the implicit channels discussed in Section 4.4, including direct/indirect branches, jumps, calls/returns, as well as implicit branches formed by store-load forwarding, memory dependence speculation (with a store set predictor), memory consistency checks and load-load ordering.

Configurations. We evaluate the following design variants, as shown in Table 4.3. We evaluate a baseline scheme DelayExecute which conservatively delays execution for all access instructions until they reach the visibility point. This models the strawman, secure scheme from Section 4.1. Our main proposal DelayExecute+STT only applies this protection to tainted transmitters, and additionally eliminates implicit channels.

Configuration	Description
Unsafe	An unmodified insecure Gem5 processor as the baseline.
DelayExecute	Delay the execution of every transmit instruction until it reaches the visibility point.
DelayExecute+STT	STT implemented on top of DelayExecute, therefore only transmitters with tainted arguments are delayed.
DelayExecute+STT-ExpOnly	DelayExecute+STT without handling implicit channels. Thus this configuration has weaker security.

Table 4.3: Evaluated configurations.

For each configuration, we evaluate the two visibility points from [183], namely Spectre and Futuristic, together with both Total Store Ordering (TSO) and Release Consistency (RC) memory consistency models.

Penetration testing. Prior to performance modeling, we evaluated whether our framework blocked Spectre variants, such as Spectre V1 (Figure 4.1) and confirmed the attack was blocked.

4.7.2 Main Performance Result

Figures 4.12 and 4.13 compare the execution time of configurations Unsafe, DelayExecute and DelayExecute+STT on the single-threaded SPEC and multi-threaded PARSEC applications, respectively. The goal is to show the performance overhead of DelayExecute+STT (our complete proposal) relative to Unsafe and performance improvement relative to the naive but secure DelayExecute scheme. All individual benchmark results use the TSO model, and execution times are normalized to Unsafe for each memory model.

SPEC analysis. With the Spectre threat model, STT improves overhead on average relative to naive DelayExecute from over 40% without STT to 8.5% with STT, in TSO. RC results are similar. The main reason is that only a small portion of all speculative loads (transmitters) are tainted due to older speculative loads (access instructions).

The saving is even more pronounced using the Futuristic model, where overhead drops from around 3× to 14.5%. This makes sense because Futuristic is a more restrictive model that forces longer delays before loads can execute. The fact that Futuristic overhead is close to Spectre overhead (14.5% to 8.5%) is an important result. Futuristic was designed in [183] to be a holistic threat model, taking into account all possible reasons for an instruction

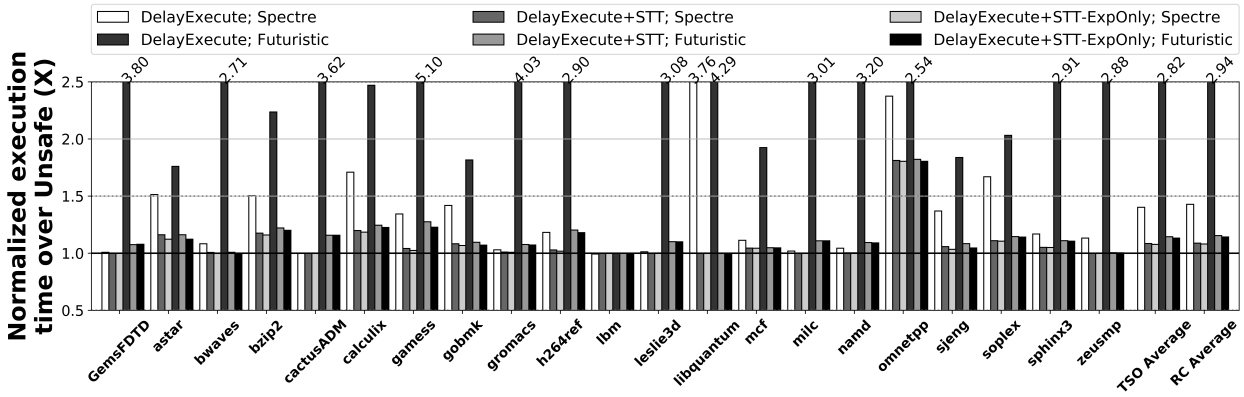


Figure 4.12: Execution time (normalized to UNSAFE) of SPEC benchmarks with the schemes DELAYEXECUTE, DELAYEXECUTE+STT, DELAYEXECUTE+STT-EXPONLY, in two visibility points (Spectre and Futuristic). All per-benchmark results assume TSO; averages for TSO and RC are given on the right.

to be speculative. The small difference between the two models suggests that supporting comprehensive security definitions is viable with STT without sacrificing much performance.

PARSEC analysis. The multi-threaded PARSEC workloads in Figure 4.13 exhibit the same trends seen in the SPEC workloads. For the Spectre model, DELAYEXECUTE+STT reduces the overhead of DELAYEXECUTE from 78% to 24% in TSO model, and from 103% to 30% in RC. For the Futuristic model, DELAYEXECUTE+STT lowers the overhead (over 3x in both TSO and RC) of DELAYEXECUTE to 27% and 36% for TSO and RC, which are close to the weaker Spectre model.

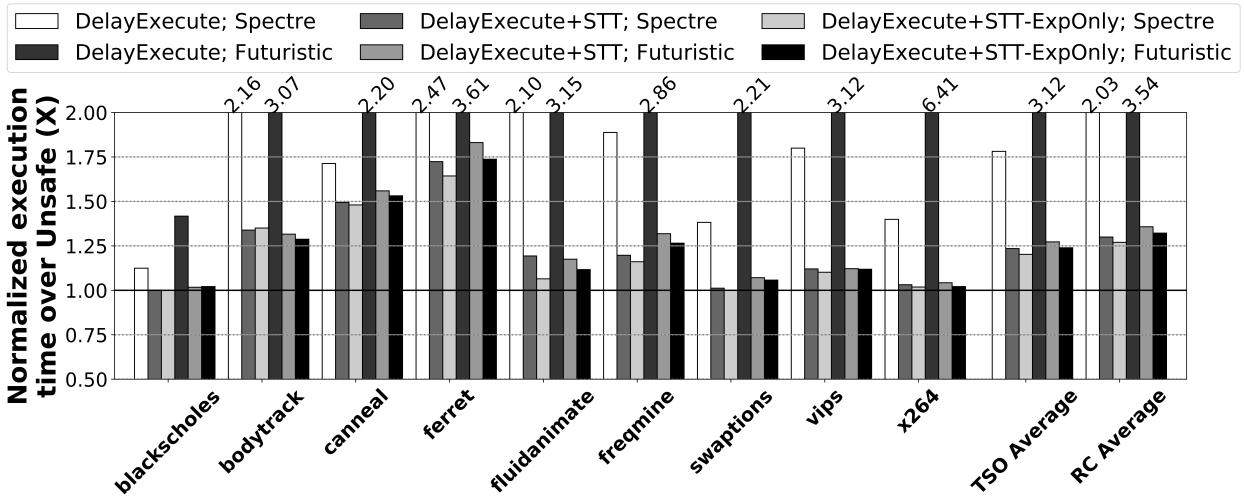


Figure 4.13: Execution time (normalized to the UNSAFE baseline) of PARSEC benchmarks. Results shown and methodology follow Figure 4.12.

Overhead from implicit branch protection. A central component in STT is mechanisms that eliminate implicit channels by delaying predictor updates and explicit/implicit branch resolutions until branch predicates are untainted. Figure 4.12 and 4.13 include results for DelayExecute+STT-ExpOnly, which shows performance for a weaker security guarantee that ignores implicit channels. For SPEC workloads, ignoring implicit branches reduces overhead on average relative to DelayExecute+STT by around 1%, for all memory models and visibility points. The number increases to 3% for PARSEC workloads. The takeaway is that protection against implicit branches using our mechanisms is very cheap.

For more insight into implicit channel overhead, Table 4.4 shows the explicit and implicit branch misprediction rates for both the Unsafe baseline and DelayExecute+STT, under TSO in the Futuristic model. We see that, delaying branch predictor updates only increases the explicit branch misprediction rate by 0.2% relative to the unsafe baseline, and the rate of memory violations (implicit branch misprediction) is close for all schemes. Second, the percentage of branch mispredictions where STT would delay resolution—and thus may incur performance overhead—is small. These are the explicit and implicit branch mispredictions that occur when the branch is tainted. For example, with DelayExecute+STT on SPEC, only 0.8% of all dynamic branches are both tainted and mispredicted (8.87% of the 9.04% mispredicted branches). The situation is even more apparent for memory violations, as they are very rare (< 0.1%, regardless of whether the implicit branch predicate is tainted). Results for PARSEC are similar.

	SPEC2006		**PARSEC**	
	Unsafe	DelayExecute+STT	Unsafe	DelayExecute+STT
# explicit br. misp. / # explicit branches	8.81%	9.04%	3.62%	3.85%
# tainted explicit br. misp. / # explicit br. misp.	N/A	8.87%	N/A	28.81%
# implicit br. misp. / # implicit branches	0.008%	0.01%	0.022%	0.018%
# tainted implicit br. misp. / # implicit br. misp.	N/A	15.5%	N/A	7.74%

Table 4.4: The effect of delaying predictor updates and resolution (for explicit and implicit branches). All numbers assume TSO + Futuristic.

4.7.3 InvisiSpec vs. STT

InvisiSpec [183] is a prior hardware mechanism for blocking speculative execution attacks. The two schemes have different security trade-offs: On one hand, InvisiSpec only blocks

covert channels through the cache hierarchy, whereas STT can block any covert channel. On the other hand, STT does not prevent leaking secrets which are part of the retired state (Section 4.2.1) whereas InvisiSpec does handle this case. As we mentioned, STT is sufficient to prevent the universal read gadget, which is the most dangerous class of attacks.

We compare the overhead of InvisiSpec and STT running SPEC2006, using the Spectre and Futuristic threat models. We find InvisiSpec and STT have 7.6% and 8.5% overhead relative to UNSAFE for the Spectre model, respectively; and 18.2% and 14.5% overhead in the Futuristic model, respectively.

4.8 CONCLUSION

This chapter proposes Speculative Taint Tracking (STT), a novel protection framework to comprehensively protect speculatively accessed data from speculative execution attacks. STT has two key novelties: a framework to eliminate both explicit and implicit covert channels, and a new taint/untaint procedure capable of waking instructions up early. Together, these mechanisms enable a high-performance and high-security design, able to enforce strong non-interference properties with respect to speculatively accessed data.

CHAPTER 5: SPECULATIVE DATA-OBLIVIOUS EXECUTION

This chapter presents another efficient protection framework against speculative execution attacks, called Speculative Data-Oblivious Execution (SDO), built on top of the previous work Speculative Taint Tracking (STT, Chapter 4). The major performance bottleneck of STT and other similar schemes is due to delaying the execution of unsafe operations (identified as explicit channels by STT) and their dependent instructions. The premise of SDO is to safely execute these instructions by executing them data-obliviously in hardware. However, replacing delaying with data-oblivious execution does not necessarily improve performance, as data-oblivious execution introduces redundant work. SDO leverages the safe prediction provided by STT to predict only a subset of data-oblivious execution to perform, yielding a significant performance improvement over STT without compromising STT's performance properties.

5.1 INTRODUCTION

As described in prior chapters, a successful speculative execution attack must force the speculative secret to be *transmitted* to the attacker through a *transmitter instruction*. Therefore, prior work has endeavored to block leakage through transmitters. On one hand, *invisible speculation* schemes [184, 186, 187, 262] attempts to execute transmitters "invisibly," e.g., by not modifying cache state for the load transmitter. This is efficient, but has security holes, e.g., due to data-dependent timing effects. On the other hand, *delayed execution* [31, 176, 178] schemes, such as STT present in Chapter 4, endeavor to completely block all covert channels by monitoring how secrets flow through instructions and delaying the execution of transmitters until their operands become a function of non-speculative data. While this idea can enable strong security, it can incur high overhead as delaying execution can stall the pipeline.

The goal of this chapter is to get the best of both of the above worlds: high performance and high security. The key idea is that it is safe to execute transmitters early, i.e., while their *operands* are still a function of speculative (sensitive) data, as long as their *execution* is made to not require operand-dependent hardware resource usage. We call this *speculative data-oblivious execution (SDO)*.

We start with a state-of-the-art defense framework against speculative execution attacks, called STT [31]. STT provides comprehensive protection for speculatively accessed data, but incurs overhead from delaying the execution of transmitters until their operands are a function of non-speculative data as discussed earlier. We will apply SDO to transmitters

in STT, retaining that work's strong security properties while dramatically improving its performance. (See Section 5.2 for detailed background on STT.)

To illustrate SDO, consider a simple example where the transmitter is implemented as a floating-point instruction. Floating-point instructions typically have operand-dependent behavior: if the operand is subnormal, the instruction may execute on a slow path (e.g., in microcode); otherwise, it executes on a fast hardware floating point unit [22]. To simplify the example, we assume this creates two execution equivalence classes: slow and fast, respectively. This forms a covert channel. An attacker can infer which equivalence class a floating point operation belongs to by, e.g., monitoring program runtime [22, 172] and hardware resource usage, and this reveals information about the operand.

To be speculative data-oblivious, the starting point is to execute multiple copies—one for each equivalence class—of any transmitter that computes on speculative data. In our example, we execute two copies of speculative floating point instructions: one for the fast and one for the slow execution. Once both are complete, it is safe to forward the result of whichever one was correct to younger, dependent instructions.

While the above idea is sufficient for security, it is low performance. Not only must we execute two versions of the floating point instruction, we must wait to forward the result until the slowest (subnormal) mode completes, to hide which version was actually needed.

The key idea to address this issue is to *predict* that one equivalence class (or a subset of classes) will be correct, and execute only those classes. An obvious pitfall with this idea is security: can the prediction itself reveal private information? Since we build on STT, however, the answer is no. STT automatically ensures that predictors' predictions, and when those predictions are resolved, are completely independent of speculative data.

With an "equivalence class predictor" in hand, we can safely speed up our floating point example. Assuming subnormal inputs are rare, we might statically predict the input will be normal. If the prediction was correct, we execute the floating point operation without delay and incur no overhead. If the prediction was incorrect (hopefully rare), we squash when the inputs become non-speculative and suffer a performance hit.

A remaining issue is: if we predict an equivalence class, as opposed to executing all equivalence classes (the naïve strategy), we will have computed an incorrect result if the prediction is incorrect. For security, we must ensure that any instructions receiving the result do not reveal whether it is correct or not. Fortunately, STT also addresses this issue through its tainting mechanism: by marking the output as tainted, no attacker will be able to learn any bit of the return value.

The above ideas are general, and apply to other types of transmit instructions and more sophisticated predictors. Beyond our simplified floating-point example, the bulk of this

chapter is to design a novel speculative data-oblivious load operation. This is important for performance as prior work shows the lion share of overhead in blocking speculative execution attacks is due to loads [31, 183, 184, 187]. It is also non-trivial, as there are many ways a load can execute that can create covert channels. For example, a load may hit or miss at any cache or TLB level, contend for different resources such as uncore buses, cache banks, lookup DRAM, etc. To address these challenges, we develop a new speculative data-oblivious load operation (an "Obl-Ld" operation, for "oblivious load"), a novel "location predictor" to improve performance for Obl-Ld operations, and additional optimizations that allow us to safely return and forward load data to the pipeline earlier.

Contributions To summarize, this chapter makes the following contributions:

- We propose a novel framework enabling Speculative Data-Oblivious Execution (SDO), enabling safe execution of unsafe transmitters.
- We propose a novel implementation of an SDO operation for speculative loads.
- We evaluate SDO as an optimization to prior work STT, treating both loads and loads plus floating point operations as transmitters. We find that STT+SDO improves STT's performance on SPEC17 workloads by an average 36.3% to 55.1%, depending on the microarchitecture and attack model—and without changing STT's security guarantees.

5.2 MOTIVATION: IMPROVING SPECULATIVE TAINT TRACKING

Threat Model With the main goal of improving upon STT, we adopt STT's threat model, i.e. the adversary that can monitor any microarchitectural covert channel and induce arbitrarily speculative execution from anywhere in the system. In addition, we only protect speculatively-accessed data. More details are described in Section 4.2.

Source of STT's performance overhead The lion share of STT's overhead is due to blocking explicit channels, specifically, delaying execution of tainted loads. The STT paper reports that ignoring implicit channels reduces STT's average overhead on SPEC [260] and PARSEC [261] applications by just 1%–3% over the original 8.5%/14.5% overhead (depending on the attack model). The effect on outlier applications with higher-than-average overhead is similar. This result shows that STT's implicit channel protection mechanism is cheap and that obtaining better performance requires avoiding the delayed execution of tainted transmitters, particularly loads.

Key observation: STT Makes Prediction Safe An important corollary of the above is that predictor structures can remain enabled without leaking privacy. The reason is two-fold. Consider branch prediction as an example (an analogous argument follows for other predictors). First, STT ensures that the branch predictor state is never a function of secret data. Further, mis-predictions are squashed only when the prediction's outcome becomes safe to reveal. This policy means the predicted branch directions do not leak privacy and that "what instructions are fetched due to the predictions" is public information. In other words, the only possible source of privacy leakage is through transmitters. Second, STT ensures that if any transmitter fetched on the predicted path can leak privacy, then that transmitter is delayed until the threat passes. Note, this argument does not depend on whether the predictor mispredicts by mistake, is intentionally mistrained, etc.

Our goal is to enable transmitters to safely execute early. That is, we will build on STT's mechanisms to block implicit channels and propagate untaint, and implement a new scheme for handling select high-overhead tainted transmitters (loads and floating point operations).

5.3 SPECULATIVE DATA-OBLIVIOUS EXECUTION

We now present a general methodology for designing an *SDO operation*, which is an SDO implementation for an arbitrary transmit instruction (transmitter). At a high level, an SDO operation is a new implementation of the transmitter that can be executed safely, even if its operands are tainted (Section 5.2).

The microarchitect starts with a transmitter f which takes operands args and returns result, denoted $\mathsf{result} \leftarrow \mathsf{f}(\mathsf{args})$. We will construct an SDO operation for f, denoted $\mathsf{Obl\text{-}f}$. This is a two-step process (next two sub-sections).

5.3.1 Design Data-Oblivious (DO) Variants

First, the microarchitect designs N data-oblivious variants (called *DO variants*) of f, denoted $\mathsf{Obl\text{-}f1}, \ldots \mathsf{Obl\text{-}fN}$. We will see how to choose N and how to design each variant create a new design space with performance/design complexity trade-offs. Each variant i for $1 \leqslant i \leqslant N$ has the following signature:

$$\mathsf{success?}, \mathsf{presult} \leftarrow \mathsf{Obl\text{-}fi}(\mathsf{args}) \tag{5.1}$$

where success? is a boolean and presult is whatever datatype is returned by this instruction, i.e., the same return type as the original f. We will abbreviate success? ≡ true as success and success? ≡ false as "fail," where ≡ denotes an equality check that returns true/false.

Informally, each variant must be data oblivious, i.e., not reveal its argument over microarchitectural side channels. A given variant may not be able to return the correct result. For example, if the floating point operand is subnormal the variant for evaluating normal operands will return the wrong result. Each variant indicates whether it has returned the correct data with the success? flag.

We now formalize these requirements in two definitions:

Definition 5.1. (Functional correctness). Consider Equation 5.1 for $1 \leqslant i \leqslant N$. For all possible args: If Obl-fi(args) returns success, then presult ≡ f(args) must hold. If this function returns fail, then presult is undefined (we assume presult is set to $\perp$).

Definition 5.2. (Security). Consider Equation 5.1 for $1 \leqslant i \leqslant N$. For any two operand assignments args and args′: the execution of Obl-fi(args) and Obl-fi(args′) must create the same hardware resource interference (i.e., which is measurable as a microarchitectural side channel).

Definition 5.1 is also important for security, as we will see in Section 5.6.

Note, these definitions do not require that for some argument args, there must exist an Obl-fi that will return success. For example, in our floating point example, we may have $N = 1$, e.g., a DO variant for the fast mode and no DO variant for the slow mode. This is allowed because our scope only considers speculative execution attacks. Continuing the above example, having a single DO variant for floating point operations means we will necessarily see fail on a subnormal input. We will, however, be able to correct that situation by squashing the incorrect speculation when the input becomes non-speculative (see Section 5.3.3 for details).

5.3.2 Design Predictor To Select Which DO Variant

Second, the microarchitect designs a *DO predictor*, which selects some DO variant Obl-fi ($1 \leqslant i \leqslant N$) to execute in place of f. Executing the DO variant in place of f will ensure args does not leak. That is, a DO predictor is a pair of functions:

$$i \leftarrow \mathsf{predict}(\mathsf{inp}) \tag{5.2}$$

$$\mathsf{update}((\mathsf{inp},\ \textit{actual}\ i)) \tag{5.3}$$

where $1 \leqslant i \leqslant N$. predict, given an input (e.g., the PC), predicts which of the N DO variants is most likely and update updates the predictor state (if any) to influence future predictions. A prediction is considered correct iff Obl-fi returns success.

Importantly, the predictions made by the predictor, and the updates to the predictor must be a function of non-sensitive information. We will see how this is achieved in the context of malicious speculative execution next, in Section 5.3.3. The predictor's internal implementation can be arbitrarily simple or complex, and is free to select a different i for each dynamic instance of a given transmitter f.

5.3.3 Putting It All Together: Protecting Speculative Data

To protect speculative data, we combine DO variants and the DO predictor with STT (Section 5.2). Pseudo-code for the complete construction, called an *SDO operation*, is given in Figure 5.1.

First, instead of delaying the execution of a transmitter with tainted operands as in STT, the tainted transmitter f is unconditionally predicted on by the DO predictor and issued as a DO variant (Lines 3-7). The case statement in the pseudo-code therefore corresponds to an implicit branch (Section 5.2) with predicate success?. Likewise, each DO variant corresponds to a non-transmitter by Definition 5.2. We follow STT's principles (Section 4.4.4) to make the DO predictor/implicit branch not leak privacy. That is, we require that predictor updates be a function of untainted data, e.g., the program counter/PC, and that predictor resolution/squashes be delayed until the implicit branch predicate becomes untainted (Lines 11-16).

Second, results are unconditionally forwarded to dependent instructions (regardless of success?) and tainted/protected by STT (Line 8). Once args becomes untainted, it is safe to reveal success? because it is a function of the prediction and args. By extension, it is safe to update the predictor and/or squash/re-issue the transmitter (since the implicit branch predicate is now untainted).

At a high level, this construction combined with STT's mechanisms ensures that the predictor always makes predictions based on public information and that args never leaks before becoming untainted. Section 5.6 formally argues how this does not change STT's security guarantee.

```
// Part 1: On issue of f with PC pc, with tainted operands args
Obl-f(args):
    switch( predict(pc) ):
        case 1: success?, presult ← Obl-f1(args) break;
        case 2: success?, presult ← Obl-f2(args) break;
        ...
        case N: success?, presult ← Obl-fN(args) break;
    return presult;

// Part 2: When args becomes untainted
if (success?): // Note: success? no longer tainted
    update((pc, prediction))
else
    // Required: squash instructions starting at pc
    // Optional: call update, if correct prediction is known
    return f(args)
```

Figure 5.1: Pseudo-code for translating a transmitter f(args) into an SDO operation Obl-f(args). Sensitive/tainted values are colored green. The predictor input is assumed to be the transmitter's PC, since this is what we evaluate in the paper. Part 1 occurs when the SDO operation issues. Part 2 occurs when args becomes untainted.

5.4 SPECULATIVE DATA-OBLIVIOUS LOADS

We now use the framework from Section 5.3 to build a high-performance SDO operation for loads. We call it an Obl-Ld operation. Loads are notorious transmitters, leaking privacy over the cache/memory side channel. They are also notoriously high-overhead to protect [31, 183, 184, 187]. For example, STT reports nearly all overhead comes from delaying the execution of loads until their arguments are untainted [31].

While the focus of this section is on loads, the framework in Section 5.3 applies to any transmitter. Thus, to demonstrate generality, the evaluation shows how SDO improves overhead when the framework is applied to both loads and loads plus floating point operations.

5.4.1 High-level Design Overview

To design the Obl-Ld, we must decide what DO variants to design (Section 5.3.1). This is not trivial, as there are many distinct ways a load can execute from a covert/side-channel perspective. For example, a load may hit or miss at any cache or TLB level, contend for different resources such as uncore buses, cache banks, lookup DRAM, etc.—and each of these implies different hardware resource usages, timings, etc.

For this chapter, we decided to design DO variants which are capable of looking up specified level(s) of cache in an address-independent fashion.

First, this leads to a relatively simple design. We need to design one DO variant per cache level. Importantly, by the SDO operation semantics in Section 5.3.3, which DO variant we predict is public information. That is, it is public knowledge *which level* cache tag/data arrays we are accessing and we only need to hide the load address while looking up that specific cache tag/data array. Thus, to implement the DO variants for monolithic private caches (e.g., the L1 or L2), a straightforward design might serialize access to the cache and only check but not update state (to hide whether a hit/miss occurred, and to eliminate timing variations from, e.g., cache bank conflicts [69]). For shared caches (such as the sliced L3) and DRAM, this is more challenging and discussed in Section 5.5.2.

Second, this design allows us to potentially eliminate most of the overhead from protecting loads. For example, if a given load has data in the L2 and the DO predictor correctly predicts "L2", then the Obl-Ld latency will be comparable to that of an insecure load. That is, a majority of the lookup time is simply to send a request to the L2, not to hide minor timing behaviors such as bank conflicts. Note, our goal in designing Obl-Ld variants for every level in the cache hierarchy plus the DRAM is to explore the design space. Our evaluation finds that, in terms of the associated performance/complexity trade-offs, systems may be best served by implementing only Obl-Ld variants for the caches (not DRAM).

5.4.2 Basic `Obl-Ld` Design

To implement the Obl-Ld, we design a DO variant for each cache level (i.e., Obl-Ld1 for predicting data is in the L1, Obl-Ld2 for L2, etc.). When a load with a tainted address issues, we look up a DO predictor (called the *location predictor*, see Section 5.4.4) to predict a level/variant. Predictions are made based on the load's PC. The chosen DO variant proceeds to lookup *all* cache levels from the L1 to the level predicted. For example, predicting data is in the L3 creates lookups in the L1, L2 and L3. Each lookup only checks if there is a tag match in that cache level and returns either data/$\perp$ accordingly. Importantly, a lookup makes no address-dependent state changes to the cache.

We will assume each DO variant is securely implemented (i.e., satisfies Definition 5.2) for now. In other words, when accessing a specific cache level, Obl-Ld does not create observable address-dependent hardware resource interference. We will describe how this is implemented in Section 5.5.2.

Designing the Obl-Ld to look up all levels from the L1 to the predicted level is important for functional correctness and security. Consider an alternate design that only looks up

the predicted level (e.g., looks up only the L2 if we predict L2). Such a design violates Definition 5.1: if there is dirty data in the L1, such a design will return stale data. As we will see in Section 5.6, this violates STT's semantics for access instructions and creates a security problem.

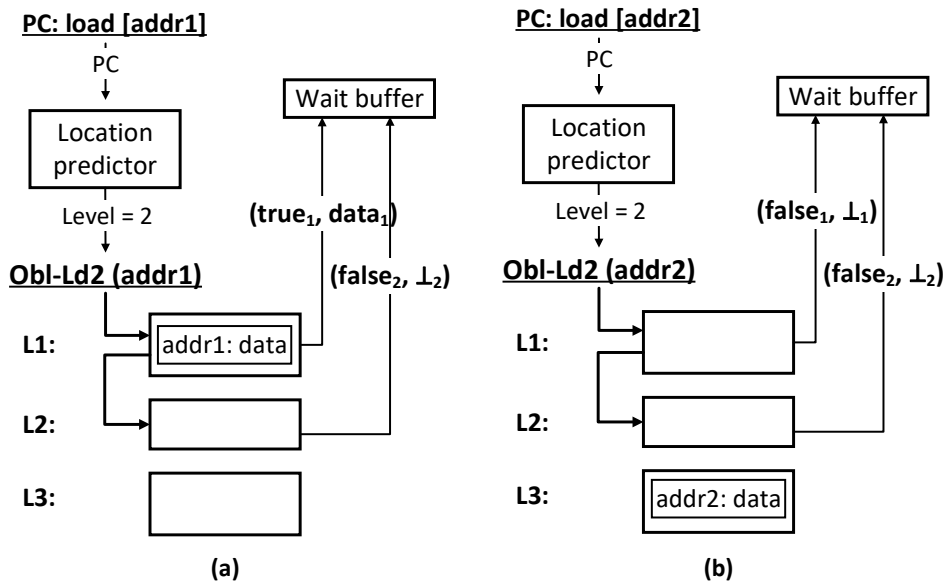


Figure 5.2: Example Obl-Ld operation given two different addresses addr1 (present in L1) and addr2 (present in L3). In both cases, resource contention and operation timing is a function of the prediction, not the address.

A structure at the core called the *wait buffer* receives responses from each cache level. Each cache that is looked up returns either (success, presult) if data was present in that level, akin to a cache hit, or (fail, $\perp$) otherwise. In our current design, each DO lookup returns a word, not the whole cache line. We denote a response from cache level i with subscripts, e.g., (success_2, presult_2) for a response from the L2. Once all responses are received, the wait buffer forwards to the pipeline:

- presult_i: for the smallest i such that we have success_i
- $\perp$: if we have fail_i for all i

That is, we forward the correct data from the cache level closest to the core, as in a regular cache.

Putting it all together, an example Obl-Ld is shown in Figure 5.2. In (a) and (b), we have a load with different addresses but the same location prediction. The takeaway is that the resource usage (cache lookups, timing, etc.) is a function of the prediction, not the address.

When the load address becomes untainted, we follow the framework described in Section 5.3.3. If the prediction was correct, we proceed with execution and update the DO predictor (if applicable). Else, we squash execution starting at the load and re-issue as a regular load.

Virtual memory. Every load first needs to consult the TLB/page tables for address translation, and hits/misses in the TLB can also leak privacy [263]. As observed by prior work, L1 TLB miss rates are low [187]. Thus, we adopt a simplified strategy based on the same ideas as the main load operation. Conceptually, we design a single DO variant that looks up the L1 TLB. On an L1 TLB hit (success), the rest of the access proceeds as discussed above. On an L1 TLB miss (fail), we likewise continue with the prediction and Obl-Ld access, but with $\perp$ as the translation. That is, we do not consult the L2 TLB until the address becomes untainted and execution squashes.

5.4.3 Advanced `Obl-Ld` Design

Memory Consistency Issues

In a standard multiprocessor system, memory consistency is maintained using speculation [249]. Loads execute out of order—speculating that their output satisfies the memory consistency model rules—and bring the accessed cache lines into the core's L1. Memory consistency violations are detected based on invalidations of these cache lines from the L1. Under consistency model-specific conditions, a load is squashed and re-issued if the cache line it read gets invalidated from the core's L1. This mechanism has correctness and security implications for the Obl-Ld implementation.

Correctness: Handling Missed Invalidations. Unlike a standard load, an Obl-Ld may read from a cache line that is not in the core's private L1. In this case, the core will not get notified if the line is later invalidated (due to a coherence transaction or cache eviction), which can lead to an undetected consistency violation. To address this problem, we adopt InvisiSpec's validation/exposure mechanism [183, 264]. Specifically, when an Obl-Ld that did not obtain its value from the L1 becomes *safe* (i.e., its address is untainted), its output is *validated*. A validation performs a standard cache access, bringing the line read by the Obl-Ld into the L1, and compares the word read by the Obl-Ld to its up-to-date value. If they match, the Obl-Ld's output is valid; otherwise, the Obl-Ld is squashed and re-issued (this time as a standard load, since it is safe). Following an Obl-Ld validation, the core

receives future invalidations of the line and is able to maintain memory consistency as usual. Because validation delays a load's retirement, we adopt InvisiSpec's *exposure* optimization, which avoids validating loads that the memory consistency model would not have required squashing had an invalidation been received. For such loads, validation is replaced by an expose operation that brings the accessed line to the L1 asynchronously, without delaying the load's retirement. (InvisiSpec [183] details the conditions under which validation can be replaced by exposure.)

Security: Handling Consistency Squashes. Our attacker model assumes that squashes are observed by the adversary, e.g., through the re-execution of the load. Squashing a load due to an invalidation could therefore leak the load's address. To prevent such a leak, we delay consistency squashes until the affected load's address becomes untainted. This delay is simply an application of STT's implicit channel protection rule to the implicit branch created by the memory consistency check, whose predicate compares the addresses of the load being squashed and of the memory access that triggers the invalidation [31]. The fact that an invalidation occurs implies that the access triggering it has an untainted address [31], so we only need to wait for the Obl-Ld to become untainted.

Event Interleavings

Modern out-of-order processors maintain multiple in-flight operations concurrently, meaning that the different steps in an Obl-Ld can occur in a variety of orders and interleavings. We now discuss the different possible cases.

Starting when the Obl-Ld executes, there are four events which control the behavior of the load:

A: The load is ready to issue but is unsafe/tainted, hence issues as an Obl-Ld.

B: The Obl-Ld operation completes when the responses from all predicted cache levels reach the wait buffer. At this point, data is safe to forward to dependent instructions.

C: The Obl-Ld becomes safe, i.e., its address is untainted.

D: The validation for the load completes.

As discussed in Section 5.4.3, validations are needed to enforce memory consistency if the Obl-Ld returns success from a lower-level cache, and the load must be re-issued if the Obl-Ld returns fail. We will refer to both of these loads as validations (event D) for simplicity.

The above events are partially ordered. In the following, we say $X \prec Y$ if event X happens before Y in time. For example, we must have $A \prec B$ because an Obl-Ld must issue before it completes. $C \prec D$ also holds since the validation is sent after the Obl-Ld becomes safe. On the other hand, $B \prec C$ may not hold, because the load may become safe before its Obl-Ld completes.

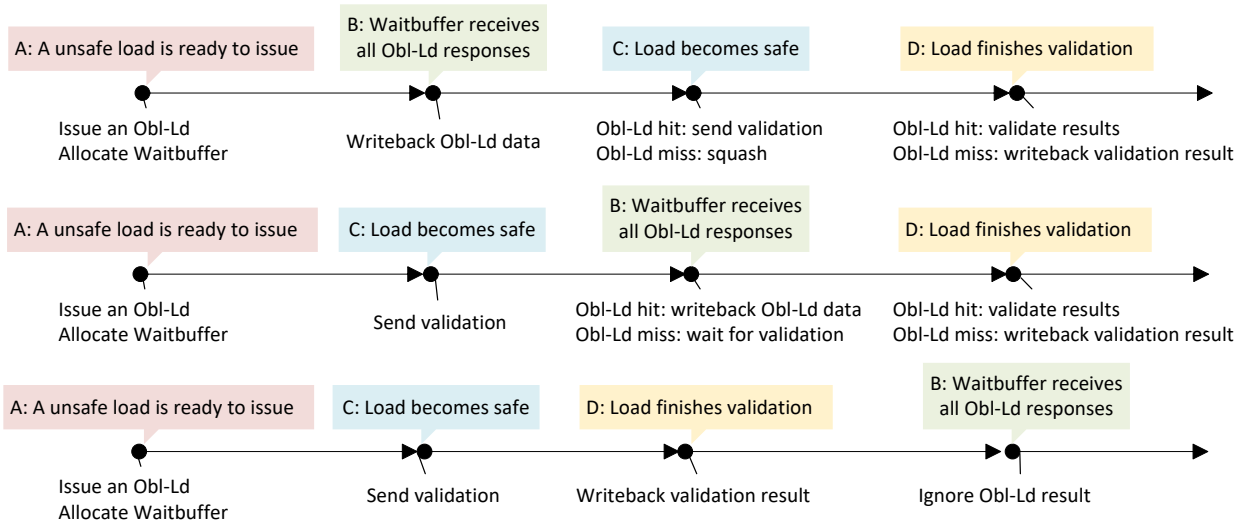


Figure 5.3: Obl-Ld operation flow with the three possible event sequences (top: $A \prec B \prec C \prec D$; middle: $A \prec C \prec B \prec D$; bottom: $A \prec C \prec D \prec B$) and corresponding actions.

In this way, there are only three possible event orderings that can occur: $A \prec B \prec C \prec D$, $A \prec C \prec B \prec D$ and $A \prec C \prec D \prec B$, discussed in detail below and shown in Figure 5.3.

1. Event ordering is $A \prec B \prec C \prec D$:
   [A] When an unsafe load issues an Obl-Ld request, it allocates a wait buffer entry. [B] When the wait buffer entry receives all responses from the accessed cache levels, the Obl-Ld request completes and the result is written to the register file/forwarded to dependent instructions. [C] Later, when the load eventually becomes safe, we issue a validation. If the Obl-Ld returned fail, we also immediately (i.e., before untainting) squash all subsequent instructions since their execution is based on an incorrect value. [D] When the validation completes: If the Obl-Ld returns success: SDO compares the result from the corresponding Obl-Ld and the current validation. If the values differ, a consistency violation might occur, therefore all subsequent instructions are squashed and the result of the validation is forwarded to re-issued dependent instructions. If the Obl-Ld returned fail (and we squashed): the validation result is forwarded to dependent instructions.

2. Event ordering is $A \prec C \prec B \prec D$:
   $[A]$ Same as Case 1. $[C]$ Before the wait buffer receives all Obl-Ld responses, the load becomes safe. Instead of waiting for Obl-Ld completion, SDO issues a validation at this moment. $[B]$ When the Obl-Ld completes: If the Obl-Ld returns success, SDO writes back the Obl-Ld result, and wakes up dependent instructions. If the Obl-Ld returned fail, forwarding incorrect data is unnecessary since it is now safe to reveal that the fail occurred. Thus, SDO drops the Obl-Ld result and waits for the result from the validation. $[D]$ Same as Case 1.

3. Event ordering is $A \prec C \prec D \prec B$:
   $[A]$ and $[C]$: Same as Case 2. $[D]$ Since the validation completes earlier than the Obl-Ld, and the validation result is always correct (i.e., a 'guaranteed success'), SDO writes back the validation result directly to complete the execution of the load. $[B]$ Since the load was completed by its validation, the Obl-Ld result is ignored.

Importantly, an Obl-Ld returning fail will only result in a squash (which hurts performance) in Case 1: when the Obl-Ld completes before the load becomes safe.

Additional performance optimization: Early forwarding from wait buffer In the scheme described so far, an Obl-Ld operation is considered to have completed when all expected responses have reached the wait buffer. Only then can the wait buffer forward its result to dependent instructions.

We can improve this scheme's performance by leveraging the observation from before: when a load becomes safe, its address and success/fail status is safe to reveal to the attacker. Thus, if a load becomes safe, it is safe to forward data from the wait buffer early, i.e., as soon as the wait buffer detects there has been a success.[16]

Other Considerations

Store-to-Load Forwarding. An Obl-Ld may get its data from the store queue due to store-to-load forwarding. Here, we adopt STT's policy: the Obl-Ld issues unconditionally, but the correct data is forwarded from the store queue instead of from the wait buffer once all Obl-Ld responses have returned.

[16]Here, we assume levels of the cache respond in order, i.e., L1 first, L2 second, etc. If responses can arrive out of order, success_i cannot be forwarded to the pipeline until all responses j for $1 \leqslant j < i$ arrive at the wait buffer.

Updating the location predictor. We update the location predictor when the load becomes safe, if its Obl-Ld returned success (per Section 5.3). If the Obl-Ld returns fail, we cannot perform the location predictor update as the actual location of the load data is still unknown. In this case, we wait for the validation and update the predictor with the level that the validation finds data in.

5.4.4 Predictor Design

We now describe a design for several location predictors. Suppose a load required data from cache level i and the predictor predicts j. We say the predictor is accurate and precise if $i == j$, accurate but imprecise if $i < j$ and not accurate if $i > j$. The goal is to be accurate and precise. When accurate but imprecise, the load incurs a larger delay because we must wait for the level j request to return. When not accurate, we may incur a squash, depending on when the load becomes safe (Section 5.4.3).

The goal of this work is to show the SDO framework is viable, not to invent a state-of-the-art predictor. We therefore first evaluate several simple static predictors that always predict to a specific cache level. Intuitively, static predictors to larger j (lower cache levels) will become more accurate but less precise. We also designed a simple dynamic predictor based on analyzing benchmark traces of high-overhead loads in vanilla STT, in particular to what levels loads hit and in what order. We observed that a given static load's cache level access pattern falls into one of two categories:

1. The cache level access pattern changes at a coarse granularity. That is, there are regions of continuous hits to a single cache level, i.e., low spatial locality.

2. The cache level access pattern consists of mostly L1 hits with predictable, singular lower level hits in between, i.e., high spatial locality. One common pattern in this category is accessing memory sequentially with a constant stride, i.e., one L1 miss per N memory accesses.

To capture both patterns, we design a *hybrid location predictor* which internally chooses between 2 predictors—a greedy predictor (greedy) and loop predictor (loop)—on each lookup, based on a per-load saturating confidence counter. Our predictor follows the template in Equation 5.2. The load's static PC (which is public in STT; Section 5.2) is used as the predictor's input. greedy and loop are designed to capture access pattern 1 and 2, respectively. greedy predicts the lowest cache level (highest j) that has been seen in the last m dynamic instances of a given load. That is, it favors imprecision over inaccuracy to avoid potential

mis-predictions. loop's behavior resembles a loop branch predictor: it predicts the frequency of lower-level accesses and tries to predict whether the next access will be an L1 hit or a hit in that lower level.

5.5 MICROARCHITECTURE

Here, we describe microarchitecture changes to support SDO. We assume a baseline STT microarchitecture. We make modest changes to the processor pipeline to support Obl-Ld operations and enforce their consistency (Section 5.5.1). We then propose implementations for data-oblivious access to the various levels of the cache/memory hierarchy (Section 5.5.2). Figure 5.4 illustrates the baseline architecture and the modifications made for SDO.

5.5.1 Changes to Processor Pipeline

Our starting point is the STT microarchitecture, which extends a modern speculative out-of-order pipeline with (1) taint propagation, which is piggybacked on the existing register renaming logic; (2) a fast (single cycle) untaint mechanism; and (3) STT's protection rules for blocking implicit and explicit channels (Section 5.2).

On top of STT, we add the location predictor (Section 5.4.4) as well as control logic to perform the Obl-Ld execution events (Section 5.4.3). To implement Obl-Ld execution logic, we extend each load queue entry with the following fields.

1. Obl-Ld State (4 bits): Stores whether the load is an Obl-Ld and if so, the current state of the Obl-Ld execution state machine described in Section 5.4.3.

2. Actual Level (2 bits): The highest cache level for which a DO variant succeeded. This is used to update the location predictor once the load becomes safe.

3. Validation/Exposure (1 bit): Indicates if, when the Obl-Ld becomes safe, an expose should be performed instead of a validation. This field indicates Exposure in one of two cases: if the Exposure condition defined in InvisiSpec [183] is satisfied when the Obl-Ld is issued,[17] or if the Obl-Ld's lookup in the L1 succeeds.

4. Pending Squash (1 bit): Indicates if this load should be squashed once it becomes safe. (Also needed by STT, but written here because SDO adds a new case where it is required.)

[17]This is a condition over the state of the load queue that identifies when the load cannot possibly be reordered with older memory operations. (Importantly, we do not require InvisiSpec hardware to detect when the condition is satisfied.) Refer to [183, Appendix A] for details.

The wait buffer (which stores data returned by DO variants) is implemented using the output register of the Obl-Ld. That is, the output register is repeatedly overwritten (with the ready bit not yet set) when each cache level access returns.

5.5.2 Changes to Memory Subsystem

The crux of the Obl-Ld implementation are the DO load variants, which perform a data-oblivious lookup of some level of the cache/memory hierarchy. To satisfy SDO's security definition (Definition 5.2), an Obl-Ld must not have any address-dependent hardware resource usage; its resource usage can only be a function of untainted (public) state. This means that an Obl-Ld cannot make any address-dependent change to the cache state, but also requires avoiding every other type of address-dependent resource contention. We generally achieve this property by partitioning an Obl-Ld from other loads (standard or DO), either spatially or temporally (by serializing resource access). As we shall see, while an Obl-Ld may contend for resources with other loads, the contention results only from the fact that the Obl-Ld is executing—which is public, due to STT's implicit channel protections—and not from the Obl-Ld's address.

Below, we discuss what a DO lookup entails for each level in the memory hierarchy, for completeness. We find, however, that limiting Obl-Lds to the on-chip caches suffices to reap most of SDO's performance gains (Section 5.7). We thus posit that a microarchitecture which implements only on-chip cache DO variants is a sweet spot from a complexity/performance trade-off standpoint.

Baseline Memory Subsystem Model

We assume the following baseline, which is based on commercial processors. Each core has private L1 instruction and data caches and a private L2 data cache. The L3 cache is shared. The caches are all physically tagged set-associative caches and are write-back, write-allocate. All caches are *banked*, i.e., the data array is arranged in several banks. Concurrent accesses to different cache lines that target the same bank are serialized; otherwise, they are served in parallel. The shared L3 cache is distributed: it is organized in multiple set-associative slices (one slice per core). A hash function (set at design time) determines the slice associated with a cache line. Caches are kept coherent with a MESI-style [258] coherence protocol.

The caches can sustain multiple concurrent misses. Each cache maintains an array of miss status holding registers (MSHRs), each of which stores all information related to an outstanding miss. A cache miss on a line allocates an MSHR if there is no outstanding miss

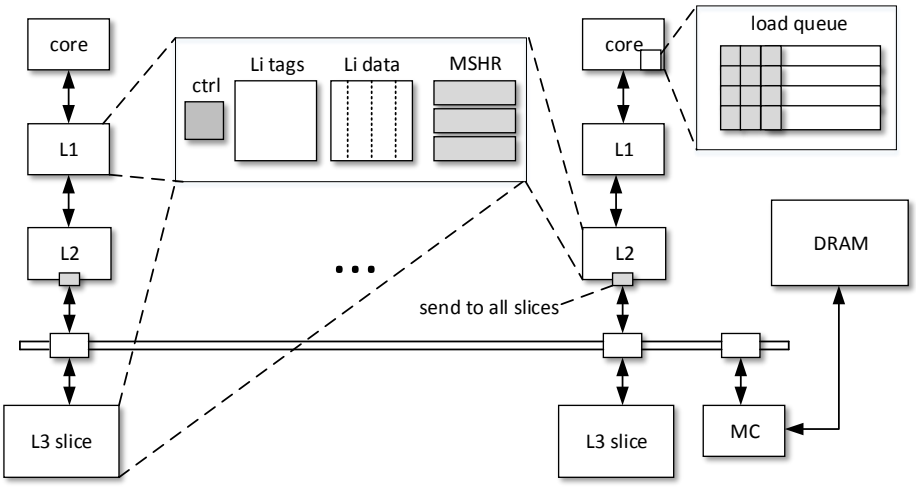


Figure 5.4: Microarchitecture with SDO support. Shaded blocks represent modified hardware.

on that line; otherwise, the information in the MSHR is augmented with the new miss and no new request is issued to the next cache level.

A single, shared memory controller (MC) connects the system to DRAM. The MC schedules memory accesses (by L3 misses) to maximize DRAM row buffer hits and bank/rank parallelism. Therefore, DRAM access latency is a function of recent and outstanding requests. The caches and memory controller communicate over a ring interconnect.

Modifications for `Obl-Ld`

We describe how to avoid address-dependent resource contention throughout the path of a cache/memory access.

Issuing Obl-Ld requests to the cache controller. In a baseline microarchitecture, address-dependent scheduling can affect when a load gets issued to the cache hierarchy, e.g., as a result of scheduling loads to minimize bank or port contention. To address this, the Obl-Ld scheduling logic must be address-independent, e.g., an Obl-Ld should be issued as soon as it is ready. Similarly, the choice of which cache port to access should also be address-independent. Note that an Obl-Ld request might not successfully issue immediately when ready due to other regular or Obl-Ld loads that are also pending to access the same cache. Importantly, such contention doesn't reveal any information about the Obl-Ld's address, because our design maintains the invariant that resource contention is a function of public (untainted) state.

Cache bank access. Accessing only the bank(s) dictated by a load's address constitutes address-dependent resource usage. Specifically, it risks creating a bank conflict with another request, with the resulting contention leaking the address. Our solution is therefore for an Obl-Ld to access all cache banks. Namely, after the Obl-Ld enters the cache, all succeeding requests are blocked until the Obl-Ld request completes its lookup. This approach guarantees that the resource use—and hence, effect on other requests—is independent of the Obl-Ld's address.

Storage of outstanding Obl-Ld miss state. We require the following to avoid address-dependent MSHR usage. First, every Obl-Ld must allocate an MSHR; it cannot share an MSHR with any other request. Second, the MSHR choice must be address-independent (e.g., the first available MSHR). In this way, any MSHR contention created by an Obl-Ld follows only from the fact that the Obl-Ld is executing and accessing a specific cache level, both of which are public (untainted) information.

LLC slice access. Similarly to bank access, an Obl-Ld cannot access only the LLC slice dictated by its address. Consequently, the Obl-Ld variant accessing the L3 must send a request to all LLC slices. The correct slice returns success or fail, depending on whether the request hits or misses; all other slices necessarily return fail. The MSHR between the L2 and L3 is de-allocated when all responses arrive, at which point it forwards a single response back to the core.

DRAM modules access. Implementing data-oblivious DRAM accesses not only requires changes to the on-chip memory controller (similar to [265]), but also to the DRAM modules themselves. For example, an Obl-Ld cannot directly fetch data from the row buffer, which has shorter access latency compared to accessing un-buffered rows [266].

As mentioned above, we find that designing a DO variant for DRAM is unnecessary, because data usually resides in the cache (Section 5.7). However, simply limiting predictions to the L3 would result in a failed Obl-Ld (and subsequent squash) for data that does reside only in DRAM. We avoid this problem by allowing the DO predictor to predict that data is in DRAM, and delaying the issue of that load, until it is safe, in that case (i.e., reverting to STT's default protection). In this way, we do not squash unnecessarily.

5.6 SECURITY

We build SDO on top of STT, and denote our combined scheme STT+SDO. Our security goal is to preserve STT's security guarantee. Cited from the STT [31] paper: "at each step of its execution, the value of a *doomed* (transient) register, that is, a register written to by a speculative access instruction that is bound to squash, does not influence future visible events in the execution." This implies blocking leakage through all microarchitectural covert channels including pressure in the cache, arithmetic unit ports, total program execution time, etc.

To prove security, we argue that STT+SDO does not change STT's security semantics for transmitters and access instructions, Claims 5.1 and 5.2 below, respectively.

5.6.1 Security for Transmit Instructions

We first analyze SDO operations in general, and then analyze our Obl-Ld operation.

Claim 5.1. Implementing transmitter f(args) as SDO operation Obl-f(args) (Figure 5.1) leaks equivalent privacy as delay-executing f(args) until args are untainted.

Proof. In STT's terminology: Obl-f(args) is equivalent to executing a non-transmitter in the shadow of an implicit branch, which STT guarantees cannot violate security. From Figure 5.1, the case statement on Line 3 is by definition an implicit branch whose (i) predictions and (ii) updates/resolutions are a function of non-speculative data. (i) holds by Equation 5.2. (ii) holds by Figure 5.1, Lines 11-16, i.e., namely we adopt STT's policy for delayed predictor update/resolution until args is untainted. Finally, by Definition 5.2 the DO variant executed in the shadow of the branch—Obl-fi(args) for some i—is a non-transmitter. QED.

The Obl-Ld operation (Section 5.4) is a valid SDO operation from Claim 5.1. First, the location predictor (Section 5.4.4) takes the load PC as input, which is a function of non-speculative data due to STT [31, 247]. The predictor's delayed resolution/updates follow the same argument as above. Second, the design for each Obl-Ld variant–for each memory level (Section 5.5.2)–satisfies Definition 5.2.

5.6.2 Security for Access Instructions

STT untaints the output of an access instruction when the instruction reaches its *visibility point*, which is the point at which the instruction is considered non-speculative with respect to the threat model (Section 5.2). Younger transmitters may reach their visibility point at

the same time as their producer access instruction(s). For example, in the Spectre model, if there are no unresolved branches between access and dependent transmit instructions. STT's protection no longer applies to such transmitters. It is therefore important to establish that the output of the access instruction—which may be forwarded to these transmitters—is the result of a correctly-speculated execution and hence not a secret. This property holds for STT, because STT only delays execution of instructions, without changing how they execute. Without careful attention, one might conclude that the property does not hold for STT+SDO, because a DO variant might return fail. Here, we prove this property does in fact hold for STT+SDO.

Claim 5.2. In STT+SDO, data returned by an access instruction is untainted only if that data corresponds to the correct speculation, for the given attack model.

Proof. We consider Obl-Ld operations, since these are the only access instruction changed in this work. Suppose an Obl-Ld access instruction, denoted ainstr, reaches its visibility point. Let X be ainstr's output. We now proceed in cases.

Case 5.1 (ainstr has forwarded X to younger instructions) . We have two sub-cases:

1. ainstr **has returned in** success: By Definition 5.1, success implies X is correct with respect to the current speculative path. For a given attack model, having reached the visibility point implies that the current speculative path is correct speculation, thus the claim holds.[18]

2. ainstr **has returned in** fail: X may or may not correspond to correct speculation. As described in Sections 5.3.3 and 5.4.2, ainstr (and younger instructions) are squashed at the same moment that the data is untainted. Thus, the claim holds.

Case 5.2 (ainstr has not yet returned X / has returned but not yet forwarded) : X may or may not correspond to correct speculation. As described in Section 5.4.3, we will (a) forward X on success or (b) drop/re-issue ainstr as a normal load on fail. (a) becomes Case 1-i above. In (b), the result will be produced by a normal load, so the claim follows from STT's properties. QED.

[18]Note, in the case of loads X may eventually cause a consistency violation. Depending on the attack model (e.g., Spectre, Futuristic), a consistency violation may or may not constitute incorrect speculation. In the Spectre model, there is no issue (consistency violations are out-of-scope). In the Futuristic model, having reached the visibility point implies that a consistency violation can no longer occur (i.e., an access instruction is unsquashable after reaching the visibility point in the Futuristic model).

5.7 EVALUATION

We now evaluate the performance of STT+SDO, for a variety of attack models and SDO design variants, relative to vanilla STT and an insecure baseline.

5.7.1 Experimental Setup

Simulation setup. We evaluate the performance of SDO using the Gem5 [259] simulator, which models cache port, bank and MSHR contention. We change the simulator to model the additional contention-related overheads caused by SDO. For example, by granting Obl-Ld operations exclusive access to all banks once they access one cache level. Table 5.1 details the simulated architecture. We use the x86-like Total Store Ordering (TSO) memory consistency model. We run SPEC CPU2017 [267] benchmarks with the *reference* input size. To obtain representative results for SPEC benchmark performance, we use SimPoint analysis [268] to identify execution fragments representing program phases. For each benchmark, we simulate 10 million instructions of each such fragment, and sum each reported metric (e.g., cycles) over all simulations, weighting each fragment according to its execution phase.

HW Components	**Parameters**
Pipeline	8 fetch/decode/issue/commit, 32/32 SQ/LQ entries, 192 ROB, 16 MSHRs, Tournament branch predictor
L1 I-Cache	32KB, 64B line, 4-way, 2-cycle latency
L1 D-Cache	32KB, 64B line, 8-way, 2-cycle latency
L2 Cache	256KB, 64B line, 8-way, 12-cycle latency
L3 Cache	2MB, 64B line, 8-way, 40-cycle latency
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	50ns latency after L2

Table 5.1: Simulated architecture parameters.

Configurations. We evaluate the following design variants, listed in Table 5.2. We compare STT+SDO (with different predictors, treating both loads and floating point (FP) operations as transmitters with architected DO operations) to STT. Two STT configurations are evaluated: STT{ld} considers only loads to be transmitters; STT{ld+fp} also considers FP operations to be transmitters. For STT+SDO, as discussed in Section 5.4.4, we evaluate both static predictors (always predicting a specific cache level) and the hybrid location predictor (*Hybrid*, which predicts L1, L2 or L3). The size of the Hybrid predictor's internal state is

4 KB. Finally, we evaluate a *Perfect* predictor which always predicts the correct cache level. All SDO configurations protect subnormal FP inputs by statically predicting FP inputs to be normal (as described in Section 5.1). All SDO configurations also mitigate leakage through virtual memory translation in the fashion described in Section 5.4.2. For each configuration, we evaluate both the Spectre and Futuristic attack models (Section 5.2).

Configuration	Description
Unsafe	An unmodified insecure processor
STT{ld}	STT, delaying the execution of unsafe loads only
STT{ld+fp}	STT, delaying the execution of unsafe loads and fmult/fdiv/fsqrt micro-ops
Static L1	SDO with predictor always predicting L1 D-Cache
Static L2	SDO with predictor always predicting L2
Static L3	SDO with predictor always predicting L3
Hybrid	SDO with proposed hybrid location predictor (Section 5.4.4)
Perfect	SDO with oracle predictor always predicting the correct level

Table 5.2: Evaluated design variants.

Penetration testing. Prior to performance modeling, we confirmed that all SDO design variants block the Spectre V1 attack, to which the Unsafe baseline is vulnerable.

5.7.2 Main Result: Performance of STT+SDO vs. STT

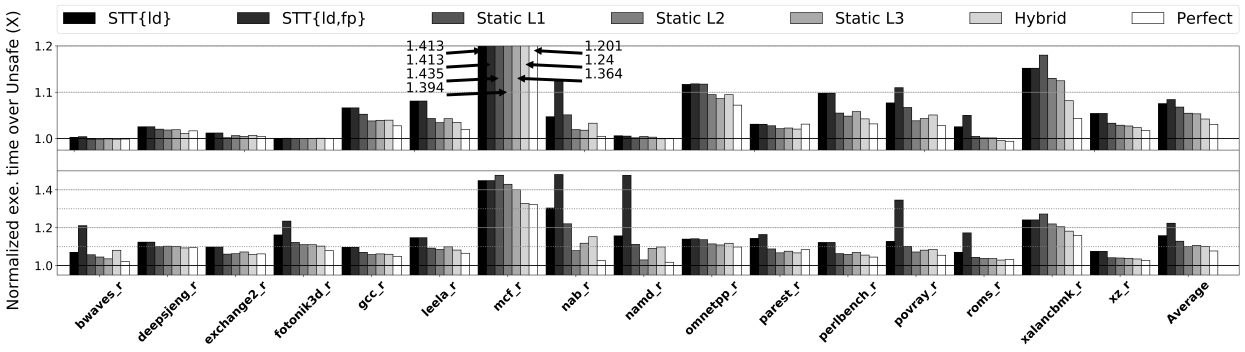


Figure 5.5: Execution time (normalized to Unsafe) of SPEC2017 benchmarks with STT and the proposed SDO design variants (STT+SDO) in Table 5.2. Averages are given on the right. (Upper half: Spectre model; lower half: Futuristic model.)

Figure 5.5 compares the execution time, normalized to Unsafe, of STT and the SDO variants on the evaluated SPEC2017 benchmarks. STT+SDO outperforms STT with both Static and Hybrid predictors in all cases. The summary is that (1) our Hybrid predictor outperforms Static predictors in the Spectre model, obtaining an average overhead of 4.19%, which

translates to a 44.4%/50.1% improvement relative to STT{ld}/STT{ld+fp}, respectively; and (2) Static L2 is the lowest overhead predictor in the Futuristic model, obtaining an overhead of 10.05%, which is a 36.3%/55.1% improvement relative to STT{ld}/STT{ld+fp}. We now discuss these results in more detail.

Static Predictors. In both the Spectre and Futuristic models, Static L1 has the highest overhead of any SDO variant. The reason is that while predicting higher cache levels (e.g., L1) yields faster Obl-Ld operations, it also incurs more frequent squashes (due to Obl-Ld operations returning fail). As shown in Section 5.7.5, performance overhead is strongly correlated to the number of squashes. Relative to Static L1, Static L2 and Static L3 have similar overheads on average, indicating that their relative differences in accuracy (which impacts squash rate) and precision (which impacts memory latency) balance each other out. (See Section 5.4.4 for definitions of accuracy and precision.)

Hybrid Predictor. While the Hybrid predictor can achieve high accuracy and precision when access patterns fall into the categories discussed in Section 5.4.4, we have found that it causes extra squashes (due to Obl-Ld failures) when accesses unpredictably miss in the L1. In the Spectre model, the overhead of these squashes can be hidden by overlapping computation, which allows the Hybrid predictor to achieve speedup over the Static predictors through its increased precision. See Figure 5.7 (left): the number of squashes for the Hybrid and Static L2 predictors is similar, but the Hybrid predictor achieves significantly lower overhead. Table 5.3 further confirms that the Hybrid predictor has significantly greater precision than the Static L2 predictor. In the Futuristic model, however, performance is more tightly correlated to the number of squashes. Indeed, Figure 5.7 (right) shows once again that the Hybrid and Static L2 predictors have a similar squash rate, but now with similar overheads.

Perfect Predictor. We show a perfect accuracy location predictor to show what performance potential is possible with SDO, beyond that achievable by our imperfect predictors. Perfect prediction improves performance by 59.5%/63.7% relative to STT{ld}/STT{ld+fp} in the Spectre model, and 51.3%/65.6% relative to STT{ld}/STT{ld+fp} in the Futuristic model. Interestingly, there is still performance overhead, even if the location predictor is perfect. We perform a more detailed breakdown of the sources of remaining overhead in the next sub-section.

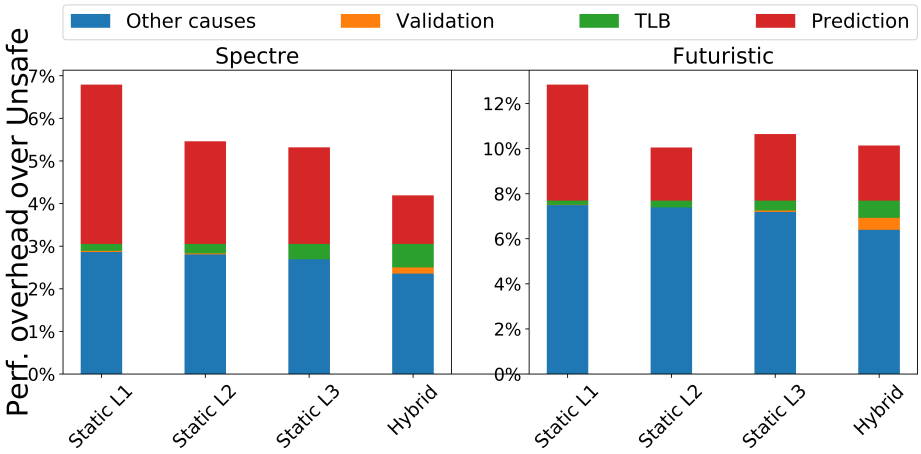


Figure 5.6: Performance overhead breakdown (vs. Unsafe) for evaluated SDO variants, averaged over SPEC17 workloads.

5.7.3 Sources of Slowdown

To provide more insight, Figure 5.6 shows a breakdown of what design components contribute what % of the total slowdown. Inaccurate and imprecise cache level prediction is a major source of overhead. Sections 5.7.4 and 5.7.5 below perform a deeper analysis of how prediction impacts performance. Validation stall and TLB/virtual memory protection constitute a small portion of the overhead. The remaining slowdown is due to (1) Obl-Ld operations not changing cache state, which leads to more cache misses; (2) implicit channel handling; and (3) the additional memory system contention caused by SDO requests.

5.7.4 Predictor Accuracy and Precision

We now measure the accuracy and precision of each SDO predictor. Table 5.3 shows that our Hybrid predictor has the highest precision, followed by Static L1, since Static L1 can usually satisfy most memory accesses. Static L2 and L3 have low precision ($< 8\%$), although their prediction tends to be more accurate (leading to fewer squashes) than Static L1/Hybrid.

5.7.5 Relationship between Performance and Squashes

We now quantify the performance loss due to inaccurate location prediction, i.e., the pipeline squashes that occur when an Obl-Ld returns fail. Figure 5.7 shows the correlation between the number of squashes and the performance overhead. The takeaway is that performance overhead is roughly proportional to the number of squashes. An exception to

	Spectre		Futuristic	
Configuration	**Precision**	**Accuracy**	**Precision**	**Accuracy**
Static L1	71.87%	71.87%	75.48%	75.48%
Static L2	7.01%	78.74%	6.58%	83.39%
Static L3	4.60%	85.04%	3.71%	89.25%
Hybrid	84.30%	86.49%	84.34%	87.18%

Table 5.3: *Precision* and *Accuracy* of evaluated SDO predictors in the Spectre/Futuristic models, averaged over SPEC17 workloads.

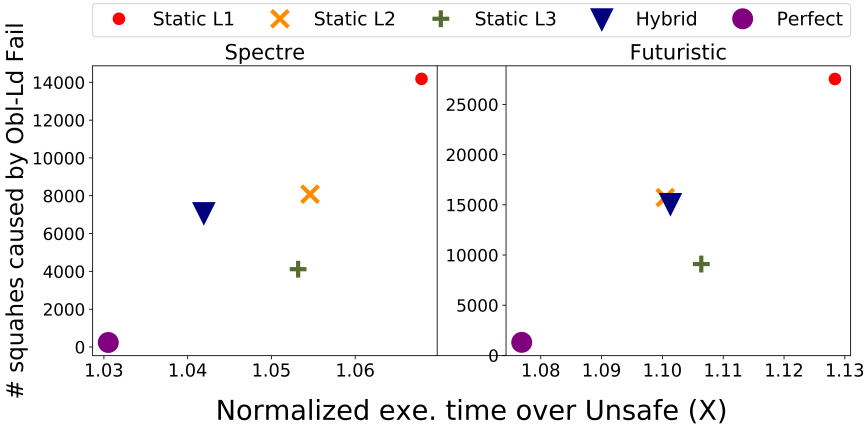


Figure 5.7: Relationship between the number of squashes and execution time (normalized to UNSAFE) of all evaluated SDO variants, averaged over SPEC17 workloads.

this trend is Static L3, which has the fewest squashes because its predictions are relatively accurate. In this case, fewer squashes are offset by imprecision (longer latency loads).

5.8 CONCLUSION

This chapter proposes speculative data-oblivious execution (SDO), a new primitive which can be used to mitigate speculative execution attacks in a high-performance and high-security fashion. The key idea is that it is safe to execute an instruction which can form a covert channel, as long as that instruction's execution is independent of sensitive data, i.e., is data oblivious. To reduce the performance overhead of this idea, we extend prior work on STT to design *safe predictors* that specify what data-oblivious behavior is needed to satisfy common case program behavior. Putting it all together, we show that augmenting STT with SDO significantly speeds up vanilla STT without altering security.

CHAPTER 6: NIGHTVISION ATTACK

This chapter presents a novel attack framework named NightVision*, which is capable of extracting the precise byte-granularity PC information of arbitrary instructions in a victim program. Leaking a program's instruction address (PC) pattern, completely and precisely, has long been a desired, but unattainable capability for micro-architectural side-channel attackers.* NightVision *achieves this goal by exploiting several previously overlooked characteristics in modern Branch Target Buffers (BTBs). The core observation is counter-intuitive: despite being a structure related to control-flow prediction, the BTB incurs observable state changes by potentially any instruction types, not just control instructions.* NightVision*'s precise PC extraction enables powerful control-flow leakage attack that bypasses prior defenses, and also can be utilized to reverse-engineer private programs under some attack scenarios.*

6.1 INTRODUCTION

Micro-architectural side-channel attacks have emerged as a critical security threat. By co-locating to the same processor, these attacks enable attackers to deduce program characteristics (e.g., control-flow decisions, the set of addresses touched in data memory) by observing the micro-architectural effects stemming from a victim program's execution (e.g., through the cache, branch predictors and more [5, 15, 17, 20, 22, 61, 66, 269]).

One such program characteristic of fundamental interest to side-channel attackers is the victim's *dynamic PC trace*. That is, the sequence of PCs corresponding to the victim's dynamic instructions over the course of its execution. If attackers are able to *directly* learn the victim program's PC trace, they would be able to beat state-of-the-art defenses for current attacks, and even effectuate new types of attacks. For example, consider control-flow leakage attacks whereby an attacker tries to learn secret-dependent branch decisions in otherwise public programs (e.g., RSA) [19, 20, 61, 62, 270, 271]. There has been significant work to mitigate these attacks, e.g., through branch balancing [62, 270] and control-flow randomization [272], while keeping the secret-dependent control flow intact for performance reasons (i.e., not converting to data-oblivious code [30, 273]). Yet, all of these defenses immediately fail if the attacker is able to directly learn the victim program's PC trace. Going beyond control-flow leakage attacks, knowing the victim program's PC trace enables new attacks on *private programs*. For example, extracting a significant portion of the victim's dynamic PC trace enables code fingerprinting on otherwise private code.

Yet, no existing side channel can precisely and directly extract any given instruction's PC,

or for that matter the program's PC trace. For example, controlled-channel attacks and their variants [115, 116, 274, 275] learn the PC trace at a page or cache-level granularity. This is too coarse to be useful against basic defenses, which impose simple restrictions to confine control flow within the minimal observation granularity (e.g., a single cache line) [117]. Several other attacks are capable of extracting PCs of particular instruction types (e.g., loads/stores [16, 276], jumps [18, 19]) or for specific code patterns (e.g., Frontal [62]). But this is not sufficiently general to infer arbitrary secret-dependent control flow, let alone recover the entire PC trace for attacks such as code fingerprinting.

We demonstrate the first micro-architectural side-channel attack capable of leaking the byte-granular PC of any victim dynamic instruction. Our attack is enabled by new findings related to how Branch Target Buffers (BTBs) are implemented in modern Intel processors. In a nutshell: It's well known that *control-transfer instructions* update the BTB, as a function of their PC and predicted target. Indeed, this behavior underpins current attacks [4, 18, 19, 277]. Our key observation is that modern BTBs, i.e., implemented with pipelined superscalar cores in mind, *are also* updated based on the execution of *non-control-transfer instructions*. We show how this enables the BTB to leak the exact PCs of even non-control-transfer instructions, e.g., of the instructions in the shadow of a branch or even the instructions in straight-line code.

The above might seem counter-intuitive. The BTB only maps branch/jump PCs to predicted target PCs, which is seemingly irrelevant to non-control-transfer instructions. However, in modern pipelined processors, the BTB is accessed without knowledge of the fetched instruction except for its PC. Additionally, modern BTB tag checks usually leave out the highest-order bits to save on area [19, 277]. Hence, it is possible for a BTB tag check to 'succeed' but to provide a prediction corresponding to a different instruction, even when the instruction corresponds to a non-control-transfer instruction.

Such *false hits* overwhelmingly result in pipeline squashes. To counteract this potential performance loss, we find that modern Intel processors *deallocate* the involved BTB entry upon detecting a false hit—as soon as instruction decoding finishes and even if the instruction causing the false hit doesn't retire. Since BTB entries correspond to branches, a BTB hit on a PC that decodes to a non-control-transfer instruction is necessarily a false hit. Thus, *non-control-transfer instructions that alias with existing entries in the BTB create observable BTB state changes.*

We find that the above effects are further amplified by how the BTB is implemented to handle superscalar fetch. Since modern superscalar processors fetch instructions in bundles, BTB lookups in fetch have range query-like semantics. That is, the BTB compares the fetch PC with all BTB entries whose PCs are greater than *but nearby* the fetch PC. Effectively, this

means that BTB deallocations can be an indicator of not only whether a fetched instruction matches a certain PC value, but also whether it falls within a BTB entry-defined address range.

Based on the above, we propose an attack framework called NightVision. The core of NightVision is a novel BTB Prime+Probe-like primitive which captures updates made by a co-located victim's execution, using attacker-allocated BTB entries. Using the false hit-induced deallocation effect, we show how NightVision can determine the PCs of (in the best case) individual victim dynamic instructions. By further exploiting BTB range query semantics, we show how the attacker can efficiently binary search through larger address ranges to recover said PCs.

We demonstrate how to use NightVision to attack victim programs in user-/supervisor-level attacker settings, where we construct control-flow leakage attacks for leaking secret data influencing program control flow. Our attacks circumvent defenses that mitigate prior control-flow leakage attacks [18, 19, 62, 270, 272]. When equipped with supervisor-level capabilities, such as managing system resources like virtual memory and interrupts, we showcase how NightVision can deduce the exact PC of every victim dynamic instruction. With the extracted PC trace, we show how NightVision can be used to reverse-engineer private programs with function fingerprinting techniques, in settings where TEEs (e.g., we evaluate on SGX) would otherwise provide code confidentiality protection.

We evaluate NightVision using two existing cryptographic functions with secret-dependent control flow: big number comparison in Intel's IPP Cryptographic library [278] and the Greatest Common Divisor in MbedTLS [279], both evaluated by a recent work [62]. We show that several prior software defense mechanisms, like branch balancing [270], basic block alignment [62] and control-flow randomization [272], and hardware mitigations (IBRS/IBPB) are ineffective at mitigating our attack. We further demonstrate how NightVision with supervisor privileges manages to identify both of the above functions from a corpus of 175K other functions by applying its function fingerprinting on the recovered dynamic PC traces, when code privacy is enforced by Intel SGX.

Contributions To summarize, this chapter makes the following contributions:

- We show how modern BTB design enables learning the PCs of arbitrary victim dynamic instructions.
- We implement an attack framework, named NightVision, and demonstrate how NightVision can observe select victim instructions' PCs, or the victim's entire dynamic PC trace, depending on the attacker's capability.

- We show how variants of NightVision can be used to leak secret data influencing program control flows even with prior software/hardware mitigations enabled, and reverse-engineer private binaries using function fingerprinting.

6.2 BTB MECHANISM

6.2.1 Existing BTB Reverse-Engineering Takeaways

Branch prediction is an essential performance optimization in modern processors to reduce the delay caused by control-transfer instructions. The core of branch prediction is a Branch Target Buffer (BTB), which is a lookup table mapping branch/jump addresses (PCs) to their target PCs [280]. Since the BTB is shared by all software running on the same core, prior work has attempted to reverse-engineer the BTB and use it to mount side-channel attacks [4, 18, 19, 277]. Next, we highlight the BTB mechanism disclosed by prior work.

In a nutshell, the BTB mechanism comprises two key actions: *access* and *update*. At the instruction fetch stage, the BTB is *accessed* to select an entry for predicting the target of the fetched branch/jump, which becomes the next instruction to fetch. If no BTB entry is selected or the selected BTB entry contains an incorrect target, the BTB is later *updated* with the correct target of the branch/jump, e.g., by *allocating* a new BTB entry or *replacing* an existing entry.

The BTB's organization on modern processors resembles a set-associative cache. Every BTB access uses tag, set index, and offset fields computed from the current instruction pointer. The set index field determines the BTB set, and the tag and the offset are used for selecting the matched BTB entry in the set. The offset field is usually 5 bits [277, 281], meaning branches within the same 32-byte-aligned block are mapped to the same set. Unlike the processor cache which uses all higher-order address bits as the tag, BTB tags tend to be truncated to reduce the BTB size[19], since the BTB is a predictor and needs not to guarantee correctness. This causes branches with the same lower-order bits to *collide* on the same BTB entry: the first branch's execution allocates a BTB entry, which is later used to predict for a different instruction with the same lower-order address bits.

Previous BTB side-channel attacks focus on BTB collisions between an attacker branch and a victim branch in two different ways. First, by executing after the victim branch, an attacker branch may access the BTB entry allocated by the victim branch. This enables the

[19]The actual number of lower-order PC bits used for BTB lookup depends on the processor generation. Based on our reverse engineering, BTBs in Intel SkyLake/KabyLake/CoffeeLake/CascadeLake CPUs ignore address bits 33 and above, while IceLake BTB ignores address bits 34 and above.

attacker to learn, through timing channels, the victim branch decision [18, 20]. Second, the attacker can execute branches first so as to corrupt the BTB and influence the prediction of the victim branch that executes after, thereby facilitating transient execution attacks (e.g., Spectre V2) [4, 86, 277]. While attacks such as Spectre V2 are mitigated with recent hardware defenses (e.g., Intel IBRS/IBPB) by preventing the collisions between indirect jumps, as we will elaborate in Section 6.4.1, collisions remain achievable in general and these lead to more fundamental attacks like NightVision.

6.2.2 Overview of Unexplored BTB Behaviors

Existing BTB side-channel attacks focus on the BTB behavior in response to only branches. This view of the BTB, however, oversimplifies the BTB mechanism and misses critical details about BTBs in modern processors. In this section, we conduct experiments to reveal two previously unexplored BTB behaviors.

Firstly and counter-intuitively, we learn that not only control-transfer instructions, but also non-control-transfer instructions can update the BTB. Since modern processors are deeply pipelined, instructions are unrecognized until they are decoded, which happens several cycles after instruction fetch. When the instruction pointer points to a non-control-transfer instruction, with no information about the instruction except its PC, the BTB must be accessed as usual. However, because the BTB lookup disregards higher-order PC bits, a *false hit* may occur, making a control transfer for the non-control-transfer instruction, eventually causing a pipeline squash after the instruction is decoded. This is where a non-control-transfer instruction can affect the BTB state—we find that the chosen BTB entry gets *deallocated* to prevent misprediction on the next encounter of the same non-control-transfer instruction. We demonstrate this behavior in Section 6.2.3.

Our second investigation examines the impact of superscalar pipelines on BTB accesses. Modern Intel CPUs fetch several consecutive instructions within a 32-byte aligned fetch block every cycle [70, 277, 281]. This bundle of fetched instructions, known as a *prediction window* or *PW* [70, 282], consists of either non-control-transfer instructions with a trailing taken branch/jump, or purely non-control-transfer instructions that end at the 32-byte boundary. Since the instructions within the bundle are not known until the decode stage, the BTB access logic must operate at PW granularity rather than instruction granularity. Specifically, the BTB access logic must first predict the location of the next branch/jump after the current PC (end of the current PW), and then the target of that branch/jump (beginning of the next PW). To achieve this, as we will show in Section 6.2.4, the BTB access hits an entry if that BTB entry has the same tag and the set index, and *the same or larger offset* compared to

```
1  F1: jmp L1; // address range: [F1, F1+1]
2      ...
3  L1: ret;
4      ...
5  < 4/8 GB padding >
6      ...
7  F2: nop; nop; ... nop; // address range: [F2, L2-1]
8  L2: ret;
9
10 /* Experiment1 */
11 for (i = 1 ... 1000) {
12     flushBTB();
13     F1();  // allocate a BTB entry
14     F2();  // may update the allocated BTB entry
15     F1();  // observe the BTB prediction outcome
16 }
```

Figure 6.1: BTB experiment in Section 6.2.3 for showing how non-control-transfer instructions update (deallocate) BTB entries.

the current PC offset. When multiple BTB hits are present, the one with the smallest offset, yet no smaller than the current PC offset is selected.

6.2.3 BTB Updates with Non-branches

We use the code in Figure 6.1 to observe the BTB update made by non-control-transfer instructions. In each loop iteration, we first flush the BTB (line 12) using the BTB cleanup routine from [20]. The code then calls `F1` to allocate a BTB entry representing the jump in `F1` (line 1). The returns on line 3 and 8 are used for returning to the caller and are not the focus of the experiment. `F2` contains a series of `nops` that may affect the allocated BTB entry state. In the second call to `F1` on line 15, we measure the prediction outcome of `jmp L1` to identify whether a BTB entry update occurs.

Given that the BTB uses the lower 32 (or 33, based on the CPU version) address bits for indexing, we place `F1` and `F2` in different 4 GB (or 8 GB) regions in memory, allowing instructions belonging to `F1` and `F2` to possibly create BTB collisions, i.e. some `nop` in `F2` may have identical lower 32 address bits as `jmp L1`. For simplicity, we only specify the lower-order bits for all addresses and ignore the higher-order bits. To ensure the BTB entry allocated by `jmp L1` is only affected by `nops` in `F2`, when exploring different `L1` and `L2` values in this experiment, we always maintain `F1` $\leqslant$ `L2-2` (note that `jmp L1` is 2-bytes long) hence `jmp L1` can only collide with `nops`.

Experimental Methodology We perform the experiment on a series of Intel CPU architectures, including SkyLake (Xeon 8124), KabyLake (Core 7700), CoffeeLake (Core 9700/9900), CascadeLake (Xeon 8252/8259), and IceLake (Xeon 8375). To capture BTB updates, we use Last Branch Record (LBR)[20], a feature available in all modern mainstream Intel processors that logs runtime information of all retired control-transfer instructions, including the branch PC, the predicted direction (only valid for conditional branches), and the elapsed cycles between the retire of the last recorded branch to the retire of the current branch. To measure the prediction outcome of `jmp L1` during the second call of `F1` on line 15, we retrieve the elapsed cycles reported for the subsequent return (line 3), which represents the duration between retiring `jmp L1` and the subsequent `ret`. When `jmp L1` is predicted correctly, `jmp L1` and the following `ret` should fetch/execute/commit back-to-back, therefore this duration is expected to be smaller compared to when `jmp L1` is mispredicted.

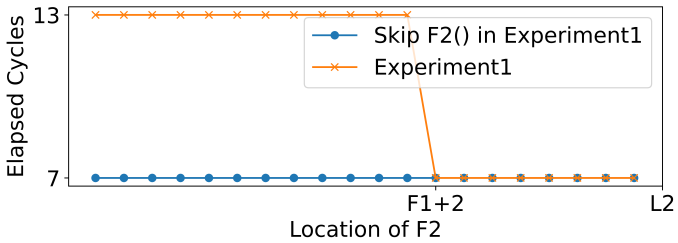


Figure 6.2: The averaged elapsed cycles between the retire of `jmp L1` (line 1 in Figure 6.1) and the subsequent return (line 3) as we change the value of `F2`. The orange line is the reported cycle count by Experiment 1 in Figure 6.1, whereas the blue line is the cycle count if we remove the call to `F2` on line 14. The gap between the two lines when `F2` < `F1`+2 indicates that when `jmp L1` collides with the `nops` in `F2`, its BTB entry is updated (deallocated) thus leading to a misprediction.

Result and Takeaway The yellow line in Figure 6.2 shows the elapsed cycles between the retire of `jmp L1` and the subsequent `ret` when varying `F2`, the starting address of the `nops`. When `F2` starts at an address prior to `F1+2`, i.e. when `jmp L1` collides with a `nop` in `F2`, we see a larger cycle count compared to when `jmp L1` does not collide with any `nop`. For reference, we plot the elapsed cycles in Figure 6.2 when removing the call to `F2` on line 14 in Figure 6.1 as the blue line. The difference between the two lines illustrates how the execution of `F2` influences the BTB entry allocated by `jmp L1`. The same pattern remains when varying `F1` and `L2`, or replacing `nops` with other non-branches such as adds. Also, the observation is consistent across all tested Intel CPUs. Thus we conclude that:

[20]Here, LBR could be replaced with a traditional timestamp counter (`rdtsc`)-based measurement or Intel Process Trace (PT). We opted to use LBR since it is orders-of-magnitude less noisy, as pointed out by [19].

Takeaway 1: The BTB update (deallocation) can occur when only non-control-transfer instructions collide with the branch recorded by the BTB entry.

6.2.4 BTB Accesses with PWs

```
1  F1: nop; ... nop; // F1 ∈ [0, 0x1e]; # of nops = 0x1e - F1
2  J1: jmp L1; // address range [0x1e, 0x1f]
3      ...
4  L1: ret;
5      ...
6  < 4 / 8 GB padding >
7      ...
8  F2: jmp L2; // address range [F2, F2+1] (F2 ∈ [0, 0x1c])
9      ...
10 L2: ret;
11
12 /* Experiment2 */
13 for (i = 1 ... 1000) {
14     flushBTB();
15     J1();  // allocate a BTB entry
16     F2();  // allocate another BTB entry
17     F1();  // observe the BTB prediction outcome
18 }
```

Figure 6.3: The BTB experiment used in Section 6.2.4 for studying the BTB access behavior.

We use Figure 6.3 to study the BTB access logic of modern superscalar processors. Similar to the previous experiment, we start each test by flushing the BTB (line 14). The code executes two different jumps (`jmp L1` on line 2 and `jmp L2` on line 8) by calling `J1` and `F2` back-to-back on line 15 and 16. Importantly, we fix the address range of `jmp L1` at [`0x1e`, `0x1f`] and limit the starting address of `jmp L2`, `F2`, to an arbitrary value within [`0`, `0x1c`] (notice both direct jumps are 2-bytes long, and they share the same set index bits and tag bits, but not offset bits and bits higher than bit 33), so the two jumps do not collide but allocate entries within the same BTB set. Then, on line 17 we jump to a series of `nops` before `jmp L1`, and execute a longer PW code (from the first `nop` at `F1` to `jmp L1`). In this experiment, we observe *any misprediction when processing this PW* as we vary `F1` and `F2` without violating the above constraints.

Experimental Methodology We test this experiment on the same Intel machines as the previous experiment, and use LBR to measure the prediction outcome. Different from

the previous experiment which measures the prediction of a single jump, this experiment requires inferring the prediction decision during the execution of the entire PW (line 1 to 2). Therefore, we use LBR to extract the total elapsed cycles between the call to `F1` (line 17) and the return after `jmp L1` (line 4). This cycle count reflects the prediction decision made for any preceding `nops` as well as `jmp L1`.

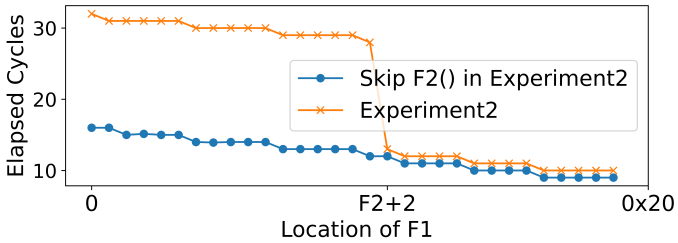


Figure 6.4: The average elapsed cycles between the retire of the call to `F1` (line 17 in Figure 6.3) and the subsequent return after `jmp L1` (line 4) as we change `F1` between `[0, 0x1e]`. The orange line is the reported cycle count by Experiment 2 in Figure 6.3, whereas the blue is the reported cycle count when we remove the call to `F2` (line 16). The gap between the two lines when `F1` < `F2`+2 indicates that the BTB entry allocated by `jmp L2` is used when fetching `nops` at `F1`, leading to misprediction.

Result and Takeaway As before, we plot two lines in Figure 6.4. The yellow line plots the elapsed cycles between the retire of line 17 and line 4 as we change the value of `F1`. The blue line reports the same measurement with the call to `F2` (on line 16) skipped, reflecting the execution of the prediction window from `F1` to `jmp L1` without any misprediction. The blue line gradually decreases as `F1` increases due to fewer executed instructions (`nops`). By comparing the two lines, we observe that when the PW base address `F1` has an offset larger than that of `jmp L2`, the execution of the PW behaves as if the call to `F2` does not exist. This implies that calling `F2` has no impact on the BTB entry allocated for `jmp L1` during the call to `J1`. In contrast, when the PW starts at an address with an offset equal to or less than `jmp L2`, misprediction occurs, leading to a constant increase in the elapsed cycle, as shown in Figure 6.4. Given the above observation that the execution of `jmp L2` should not affect the BTB entry allocated for `jmp L1`, the misprediction can only be attributed to the use of the BTB entry allocated for `jmp L2`. Specifically, in this case, when predicting the next fetch target with the current instruction pointer pointing towards `F1`, the BTB entry representing `jmp L2`, rather than the entry representing `jmp L1`, is selected. Similar to the previous experiment, this observation is also consistent among the tested machines. From this result, we can deduce the following takeaway:

Takeaway 2: Due to superscalar fetch, a BTB hit requires an identical tag/set index. The offset of the selected entry must be equal to/greater than the current PC offset. If multiple BTB hits are present, the one with the smallest offset is selected.

Lastly, we note that the above two takeaways are consistent with an earlier Intel Patent [283] that specifies the implementation details of a set-associative BTB structure (Takeaways 1 and 2 are mentioned in columns 20 and 12, respectively).

6.2.5 Differences from Prior BTB Reverse-Engineering Efforts

Our reverse-engineering approach differs from prior BTB attacks [4, 18, 19, 86, 277] in two respects. First, while prior efforts limit the scope to branches, our insight is that superscalar processors fetch PWs every cycle, therefore the BTB access must exhibit range semantics instead of simply PC matches. Second, while prior efforts mostly focus on how one branch allocates BTB entry which affects the future branch predictions, we additionally investigate how allocated BTB entries can be deallocated, displaying the role of non-control-transfer instructions in the BTB mechanism.

6.3 ATTACK MODELS AND NIGHTVISION OVERVIEW

In this work, we consider two common attacker models adopted by existing side-channel research:

- **User-level**: the attacker controls one or several user-space processes, which can co-locate with the victim process on the same CPU core via context switches (SMT is not mandatory), hence sharing the same BTB.
- **Supervisor-level**: the attacker has full control over the OS kernel, and can arbitrarily interrupt the victim program's (e.g., an SGX enclave's) execution at a fine temporal granularity, i.e., per instruction, and monitor/manipulate system resources, such as page tables.

We now overview the NightVision attack. NightVision extracts the dynamic PCs visited by the victim program. For either of the above attacker models, the attacker can extract the PCs of both control-transfer and non-control-transfer instructions at byte granularity. That said, depending on the attacker model, the attacker may be able to extract the full trace (i.e., every PC belonging to every dynamic instruction) or only a subset of it (i.e., depending on the temporal granularity at which the victim program is context switched).

The core of NightVision is a BTB Prime+Probe style primitive that is built on top of Takeaways 1 and 2 from Section 6.2. This primitive, dubbed NightVision-Core or NV-Core in short, determines if a fragment (i.e., the instructions executed during one context switch) of the victim program's execution contains instruction bytes overlapping with a specified virtual address range. We construct two NightVision variants from NV-Core, depending on the attacker model, namely NightVision-User (or NV-U) for the user-level attacker and NightVision-Supervisor (or NV-S) for the supervisor-level attacker. Both variants repeatedly apply NV-Core to every fragment of the victim's execution. As described above, NV-U is (practically) capable of recovering a subset of elements in the PC trace, whereas NV-S can achieve much finer-grain measurement, ideally recovering the PC of every victim dynamic instruction. Section 6.4 explains the design details of NV-Core and both variants.

We later demonstrate two attack applications using the two NightVision variants. Section 6.5 will explain how NV-U can serve as a control-flow leakage attack that aims at leaking secret data influencing program control-flow, even when the program is protected against prior control-flow leakage attacks. Section 6.6 further highlights how NV-S enables binary fingerprinting to reverse-engineer private enclave binaries, using full PC-trace extraction.

6.4 NIGHTVISION ATTACK DESIGN

6.4.1 NightVision-Core (NV-Core)

Inspired by the two takeaways from Section 6.2, we first introduce NV-Core, which is a BTB Prime+Probe style primitive to determine if instructions executed by the victim overlap with an attacker-specified prediction window. The attacker starts by creating a PW code snippet, with a sequence of `nop`s followed by a direct jump. Based on the definition of PW, the address range of this code snippet is restricted to within a 32-byte aligned block. The attacker first executes the PW code to allocate a BTB entry, then allows the victim program to run for a certain period of time, and finally executes the same PW code again and measures the predictions during this second execution of the PW code, similar to the technique in Section 6.2.4.

Our key insight is that, as the victim's execution also fetches PWs, when the attacker PW address range overlaps with a victim PW's address range, i.e., some instruction bytes from the attacker PW share the same lower-order (lowest 32 or 33) address bits as some instruction bytes from the victim PW, the second execution of the attacker's PW will incur an observable misprediction.

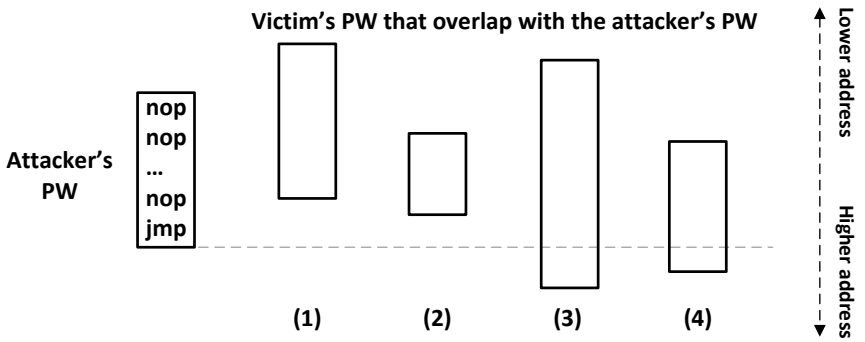


Figure 6.5: Different ways that the victim's PW overlaps with the attacker's PW. In those cases, the second execution of the attacker's PW in NV-Core incurs observable mispredictions.

To illustrate this point, Figure 6.5 shows all four scenarios when the attacker PW and the victim PW overlap. In (1) and (2), the victim PW ends at an address aligned with the middle of the attacker's PW range. In these cases, the victim PW must end with a taken branch/jump otherwise the PW will extend to (and be truncated at) the 32-byte boundary. Since the victim PW includes a branch/jump before the attacker's jump, the execution of the attacker's PW must suffer from a misprediction caused by the victim's branch/jump, in the same ways as we showed in Section 6.2.4. In the other two cases (3) and (4), the attacker PW ends at an address aligned with the middle of the victim PW range. Since the overlapping part of the victim PW is non-control-transfer instructions, fetching those instructions necessarily deallocates the BTB entry allocated at the prime step, resembling the experiment in Section 6.2.3.

We can optimize NV-Core by priming and probing multiple contiguous, non-overlapping PW ranges at once. Figure 6.7 compares the basic NV-Core monitoring one PW with an optimized NV-Core which monitors two PW ranges. In the optimized one, both the prime and the probe execute the chained PW snippets, and the prediction outcomes of all direct jumps are measured during the probe. This optimization not only allows NV-Core to simultaneously monitor multiple address ranges, but also expands the overall coverage.

Intel's Recent BTB Mitigations Intel recently introduced Indirect Branch Restricted Speculation (IBRS) [170] and Indirect Branch Predictor Barrier (IBPB) [169] to mitigate Spectre V2 [4]. Although conventional wisdom might suggest that these schemes simply flush the BTB (which, if true, naturally defeats NV-Core), we tested NV-Core with both schemes enabled and still observe an update made by the victim's execution to the BTB entry state established by the attacker-controlled PW code. Our finding reveals that IBRS and IBPB only change state for part of the BTB, i.e., entries corresponding to indirect branches,

```
// p is a fragment of victim's execution
bool NV-Core(PW, p):
  Prime BTB with PW
  execute p
  match = Probe BTB with PW
  return match

// optimized NV-Core, by priming/probing multiple PWs
bool[] NV-Core(PWs[], p)

// P is the victim program
bool[][] NV-U(PWs[], P):
  match = []
  while (P is not finished) {
    p = next fragment of P to execute
    match.append(NV-Core(PWs, p))
  }
  return match

bool[][] NV-S(PWs[], P):
  match = []
  while (P is not finished) {
    i = next instruction of P to execute
    match.append(NV-Core(PWs, i))
  }
  return match
```

Figure 6.6: The basic workflow of NV-Core, NV-U, and NV-S. NV-U measures a time slice of the victim's execution, whereas NV-S leverages supervisor privilege to measure every dynamic instruction. Both NV-U and NV-S can benefit from optimized NV-Core to monitor multiple PWs at a time.

rather than flushing the entire BTB. This is in line with the official security claims of IBRS and IBPB, which state that both schemes only apply to indirect branches [169, 170], and consistent with its goal to mitigate Spectre attacks: as direct jump targets are resolved early in decode, they cannot result in a large enough speculation window to enable Spectre attacks.

6.4.2 NightVision-User (NV-U)

As shown in Figure 6.6, NV-U invokes NV-Core for each victim execution "fragment", i.e., each time the victim is context switched onto/off of the core hosting the attacker-controlled BTB. This enables the attacker to learn dynamic PC information at scheduling-epoch granularity.

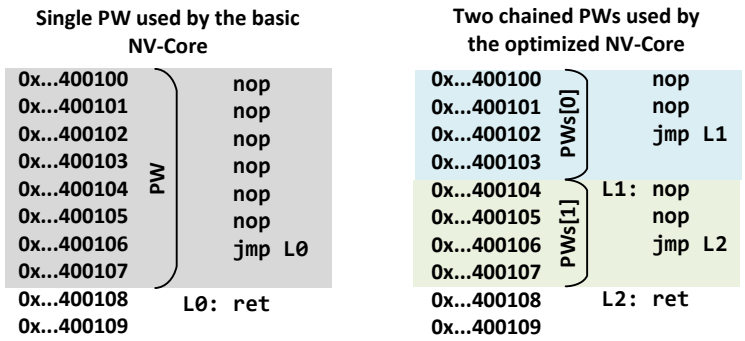


Figure 6.7: Comparison between a single PW and two chained PWs.

Since the BTB's size is limited, each context switch should run as few instructions as possible to minimize the chance that attacker BTB entries are evicted. NV-U leverages existing user-space preemptive scheduling attacks [7, 20, 81, 284, 285, 286, 287, 288, 289] to drastically reduce this victim time slice duration to on-order hundreds of cycles. In a nutshell, these preemptive scheduling attacks exploit the process scheduling mechanism adopted by modern Operating Systems (like Linux) by mounting a denial-of-service attack with hundreds of attacker-spawned child processes[21]. The attacker can (roughly) control the victim time slice by carefully controlling when the attacker processes unblock and yield to each other. We show in Section 6.5 how NV-U leaks victim control flow for programs hardened to withstand prior control-flow side-channel attacks.

6.4.3 NightVision-Supervisor (NV-S)

Same as NV-U, NV-S continuously invokes NV-Core across the entire victim's execution. Other than relying on the coarse-grained user-level scheduling technique, as a supervisor attacker, NV-S can leverage interrupts or signals (such as [270] and [290]) to *single-step* the victim's execution. In other words, every fragment is exactly one victim dynamic instruction, as indicated by Figure 6.6. This allows NV-S to determine if each victim dynamic instruction resides within specified PW(s) range(s). Although this information trivially enables the control-flow attack we describe in Section 6.5, it uniquely enables a new attack on private code fingerprinting that we will describe in Section 6.6.

[21]The implementation of the preemptive scheduling attack is orthogonal to our work and has been demonstrated by prior work (e.g., [7]). It is a common ingredient in side-channel attacks (including NightVision) for acquiring fine-grained side-channel measurements. We discuss its use and potential impact to NightVision in Section 6.6.3 and Section 6.8.1.

6.5 USE CASE 1: CONTROL-FLOW LEAKAGE ATTACKS

6.5.1 Motivation and Threat Model

We assume a user-level attacker described in Section 6.3 who wishes to extract the victim's secret data. The secret must affect the victim program's control flow, e.g., as a branch predicate. The victim code is assumed to be *public*, but can use software mitigations such as branch balancing or control-flow randomization (CFR) [272]—see below—to block existing control-flow attacks. The victim could also use SGX to provide an extra layer of data protection.

The Control-flow Leakage Arms Race Prior control-flow leakage attacks [18, 19, 20, 61, 62, 115, 116, 270, 274, 275] share the same attacker goal and have similar assumptions. However, the side channels that they exploit can be mitigated by incremental defenses that block various tell-tales of control-transfer instructions. For example, existing attacks on the BTB and the directional predictor [19, 20, 61] infer conditional branch outcomes by observing branch predictions. Correspondingly, CFR [272] mitigates those attacks by replacing observable secret branches with randomized jumps allocated at runtime, and indirect jumps are not exploitable with Intel's mitigation in place (Section 6.4.1). Several attacks [62, 270, 271] in turn leverage observed features in the code executed in the shadow of the branch, e.g., instruction count [270], type [271] or alignment [62], to leak control-flow decisions. Yet these attacks fail against branch balancing-style defenses, e.g., padding both sides of the branch to the same instruction count, type, or alignment [62].

All of the above defenses fail against NightVision, due to its ability to extract victim PCs directly. For example: CFR fails because it tries to protect the branch decision, but NightVision does not rely on the branch decision. Balancing fails because it tries to make the execution of both sides of the branch look the same, but NightVision directly extracts the PCs of instructions shadowing the branch.

6.5.2 Control-flow Leakage Attack Procedure Using NV-U

For a control-flow leakage attack, with the knowledge of the victim's program, the attacker first selects one or several consecutive victim instructions that control-depend on the secret, i.e. instructions that execute if and only if the branch swings in a particular direction. The attacker then creates a PW code snippet, and ensures the PW range is within the virtual address range of the chosen victim instructions. Finally, the attacker employs NV-U along

with the victim's execution to determine whether and roughly when (within which time slices) the victim executes the chosen instructions during its execution, and deduces the secret value based on that information.

NightVision achieves a byte-granularity observation since the PW can start at any byte address. Given the shortest PW snippet contains a 2-byte direct jump, any instruction longer than one byte (almost all x86 instructions except `ret` and `nop`[22]) or any number of consecutive instructions can be measured with a PW. For example, one may use NV-U to observe whether a basic block executes, by making the PW a sub-range of the address range of that basic block.

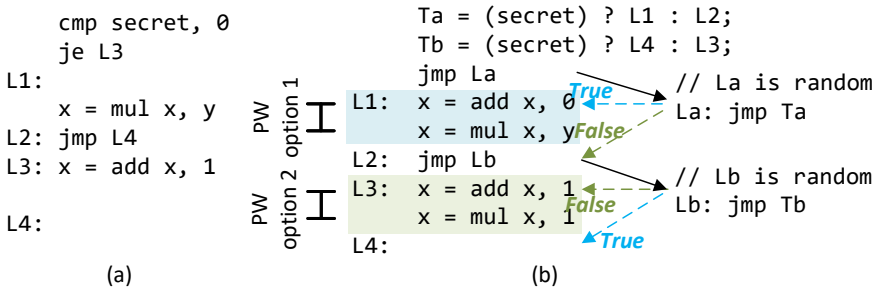


Figure 6.8: (a) The original leaky code (b) The hardened code with branch balancing and control-flow randomization [272]. NV-U defeats both defenses by observing which side of the branch (L1 or L3) is executed with a PW that is either a sub-interval of the address range of L1 (PW option 1) or L3 (PW option 2).

Figure 6.8 demonstrates the control-flow leakage attack when the victim program is using branch balancing and CFR to protect the secret branch condition. Given the secret-dependent control flow, i.e., either the then side (L1) or the else side (L3) will be executed, NV-U can create a PW code snippet, and ensure that the PW range is either within the address range of L1 or L3. By observing whether L1 or L3 is executed, NV-U can deduce the value of the secret.

Note that the attacker often needs to measure the decisions of multiple dynamic instances of the same branch inside a loop. As mentioned in Section 6.4.2, NightVision relies on a preemptive scheduling attack, a technique applied by many existing attacks for achieving fine-grain temporal resolution side-channel measurements (e.g., per loop iteration) [75, 103, 276, 287, 288, 289]. This technique does have a limitation: it does not provide synchronization between the attacker and victim, since it does not ensure the preemption of the victim exactly once per loop iteration. To help overcome this limitation, we note that NV-U provides additional opportunities to deduce the victim's execution progress. For example,

[22]And some rarely used instructions, such as halt (`hlt`), complement carry flag (`cmc`).

in Figure 6.8, if the attacker monitors both L1 and L3 with both PW options 1 and 2, it is possible to detect excessive preemptions occurring within one loop iteration—NV-U performed in those excessive preemptions will show that neither L1 nor L3 is executed.

6.6 USE CASE 2: FINGERPRINTING PRIVATE CODE

6.6.1 Intel SGX

Intel SGX [291, 292] is a hardware-based trusted execution environment solution that is widely deployed by major cloud vendors today. It protects the integrity and confidentiality of (a part of) a user-level application from a powerful attacker who controls the entire software stack (including the OS, hypervisor, and BIOS), with a new CPU mode called *enclave*. Enclaves leverage hardware-managed memory isolation to guarantee that the enclave state can only be accessed by its owner enclave. However, SGX enclaves still rely on an untrusted OS for managing resources, such as page tables and interrupts, enabling a privileged attacker to interrupt enclave execution, as shown by controlled-channel attacks [115], microarchitectural replay attacks [293], SGXStep [294], etc. Aside from the typical data protection, Intel SGX also provides code confidentiality via Protected Code Loader (SGX PCL) [295], or similar mechanisms [296, 297, 298, 299], by keeping the enclave binary encrypted until it is loaded into the enclave. SGX also serves as a design paradigm for future TEE solutions, such as Intel TDX [300].

6.6.2 Threat Model and Motivation

We assume an untrusted, privileged attacker whose goal is to obtain the victim's code. Given this strong attacker model, we make a realistic assumption that the victim relies on a TEE mechanism, in this work Intel SGX. SGX provides data confidentiality such that the victim's program state is inaccessible to the attacker, and processor features such as LBR, Intel Processor Trace, and performance counters are disabled in enclave mode. SGX also provides code confidentiality (using PCL [295] or similar mechanisms [296, 297, 298, 299]), therefore the attacker has no knowledge of the victim enclave code.

Different from Section 6.5 which leaks data in a public program setting, this section demonstrates how NV-S enables reverse-engineering of private programs. Existing side-channel attacks mostly consider programs as public and available for offline analysis. However, an attacker without knowledge of the code, for instance, cannot locate Spectre gadgets [86], deduce that certain types of side-channel vulnerabilities exist [301, 302], nor determine how

to monitor the side-channel [303]. Although this may imply that keeping the program private could be a holistic defense strategy by breaking an essential requirement for side-channel attacks, this "security by obscurity" fails with NV-S, which extracts every element in the dynamic PC trace with high accuracy. Although instruction addresses cannot directly reflect the binary content, a sufficiently long sequence of PCs may contain enough entropy to identify a specific function. Using this insight, we design a function fingerprinting approach that identifies functions of interest from the extracted PC trace. NightVision thus complements existing side-channel attacks by satisfying their assumption about victim programs being public, even when the victim enclave program is unreadable+unwritable and only executable.

6.6.3 Dynamic PC Trace Extraction Using NV-S

NV-S infers the complete byte-granularity control-flow information of an enclave program's execution, in the form of a sequence of PCs of every retired dynamic enclave instruction. No existing control-flow leakage attack achieves this goal: they either only recover coarse spatial-granularity control-flow information [115, 274, 275] or only leak the addresses of specific instruction types/patterns [16, 19, 62, 276].

Figure 6.9 illustrates the entire attack flow. In the following, we focus on how to deduce a victim instructions' page offset bits, since the victim instruction virtual page numbers can be leaked through complementary work on controlled-channel attacks [115, 274, 275] (line 2-4 in Figure 6.9) that induce page faults to learn page-level information. The attack repeatedly invokes NV-S (line 1), and every invocation of NV-S continuously applies the optimized NV-Core to measure every dynamic instruction against multiple PW ranges (line 5-6). To execute exactly one enclave instruction during each call of NV-Core, we use SGXStep [294] to single-step the enclave, by choosing a proper timer interrupt interval such that the enclave execution is interrupted precisely after each dynamic instruction is retired. Once an instruction retires, the timer interrupt delivers the control to NV-Core inside the interrupt handler. By probing the BTB with the PW code snippet (line 11), NV-Core determines if the instruction overlaps with the tested PW ranges (line 12-14), and then launches NV-Core for the next instruction by choosing and creating the PW code snippet for the next instruction and starting the prime step before resuming the enclave execution (line 8-9). How NV-S chooses the PWs is described in the next paragraph. When the enclave execution finishes, for each dynamic instruction, NightVision collects the PWs that overlap with the instruction. If for any dynamic instruction, its matched PWs so far are insufficient for determining its PC, NightVision performs another round of NV-S (line 17).

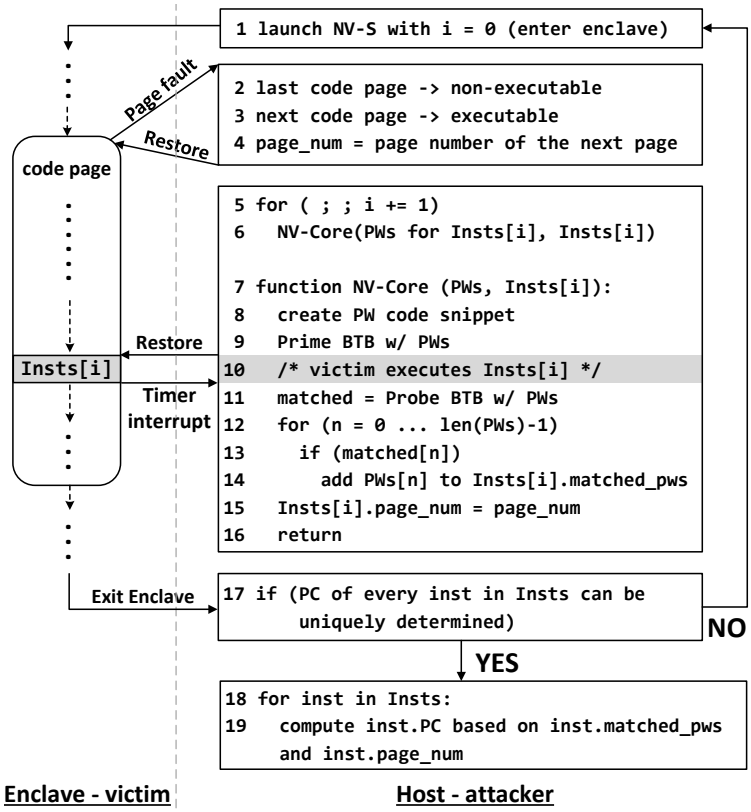


Figure 6.9: How NV-S infers the address of every dynamic enclave instruction. Definitions of keywords: `Insts`: the sequence of dynamic enclave instructions; `Insts[i].pc`: the PC of `Insts[i]`; `Insts[i].matched_pws`: set of PWs which collide with `Insts[i]`; `Insts[i].page_num`: page number of the code page containing `Insts[i]`.

PW Traversal We now explain how to choose a proper set of PW ranges for NV-Core to measure each instruction during every NV-S call. We leverage the range semantics of PWs to perform a binary search through progressively-smaller PWs for each victim instruction. Searching for the instruction PC is divided into multiple passes. Each pass splits the matched PW range in the previous pass, and measures which sub-PW range contains the base address of the instruction. As shown in Figure 6.10, the measurement of every dynamic instruction starts by dividing the 4 KB page size range into 128 mutually disjoint 32-byte PW ranges. Suppose every call to NV-Core tests `N` PW ranges. The first pass takes 128/`N` enclave executions (NV-S calls), after which NightVision determines inside which 32-byte PW each dynamic enclave instruction starts. Next, this 32-byte PW is split into `N` sub-PWs, and with one more NV-S call, NightVision can determine which sub-PW each dynamic instruction starts. This process is repeated until it determines the base address of the target instruction.

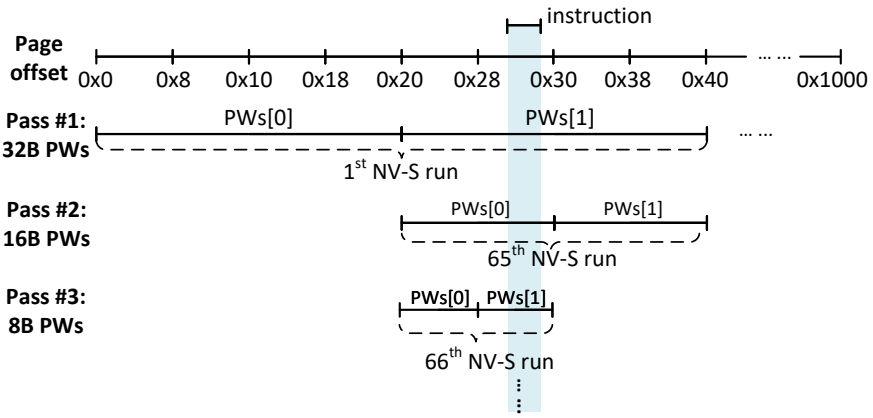


Figure 6.10: The PW traversal approach is explained in Section 6.6.3. NightVision first generates 128 mutually disjoint 32-byte PWs. The matched PW is further converted to smaller PWs recursively. We assume $N = 2$, therefore each NV-Core tests the current instruction with two PWs.

Impact of Speculative Execution The enclave single-stepping retires exactly one instruction per interrupt. However, succeeding instructions may speculatively execute, and update the BTB state before the current instruction retires. So instead of measuring only the address range of the single-stepped instruction, NightVision may measure the address ranges of all executed instructions, including the succeeding speculatively-executed ones. When speculation does not involve control transfer instructions, the base of the (extended) measured range still corresponds to the PC of the target instruction. However, when speculation involves control transfers, NV-S may produce multiple candidate PCs: one being the PC of the target instruction, and the others being the target PCs of the succeeding control-transfer instructions which execute speculatively. The correct one can be deduced after the next single step, when NV-S again generates a set of candidate PCs for the next instruction, including the false PCs corresponding to the control-transfer targets. The actual PCs belonging to the two single-stepped instructions are captured by comparing the two PC sets and ruling out the repeated candidates.

6.6.4 Function Fingerprinting for Private Code

Although NightVision only collects PC traces, which have no information about the actual instruction bytes, PC traces of sufficient length can be used to fingerprint known PC sequences corresponding to functions in existing code bases. This technique is called function fingerprinting [304] and is employed by NightVision for deducing whether the enclave execution contains functions from a known binary file (these functions are called *reference*

functions). In general, it is impractical to assume the attacker owns a set of binaries that will always include enclave code. However, we deem this reasonable for at least cryptographic code since most enclave programs use several popular off-the-shelf, open-source cryptographic libraries, several of which contain vulnerable functions that have been exploited by existing side-channel attacks. With the function fingerprinting, such use of vulnerable functions can be identified in private programs.

The attack proceeds in two steps, as shown in Figure 6.11. First, we collect the victim PC trace through NV-S and pre-process the trace. Second, we compare the *similarity* of victim functions included in the PC trace to reference functions.

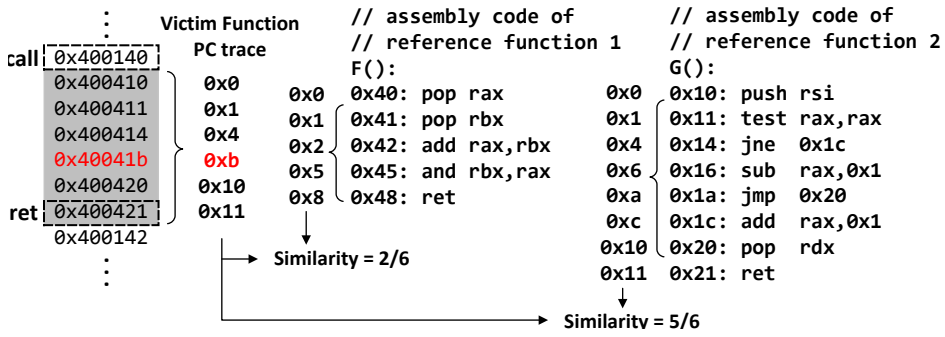


Figure 6.11: NIGHTVISION function fingerprinting computes the similarity between the victim function and two reference functions `F` and `G`. Red PC value represents an incorrect measurement.

Step 1: Victim PC trace preprocessing NIGHTVISION uses the complete attack flow described in Section 6.6.3 to obtain a victim PC trace. As illustrated by Figure 6.11, it partitions the whole PC trace at function call boundaries into per-function traces. Each function-level trace is normalized to be position-independent by subtracting the PC of the function start from all PCs in the trace, thus each function-level trace starts with zero. Every function-level trace represents an invocation of an unknown *victim function*, and will then be considered separately during function similarity matching (Step 2).

Function-level traces are sliced by locating `call`/`ret` pairs in the original trace, using the following approach to identify PCs corresponding to `calls` and `rets`. First, we capture jumps between PCs that are greater than 16 bytes, which indicates a control-transfer instruction. Second, unlike other control-transfer instructions, `call`/`ret` also access data memory. Thus, we additionally monitor whether a suspected `call`/`ret` accesses a data page (through a controlled-channel attack [115]). Note for this work, we assume functions are only entered/exited via `calls`/`rets`.

Step 2: PC trace similarity test Next, we compare the function-level PC traces collected during Step 1 that represent unknown victim functions with *sets* of PCs corresponding to reference functions. Importantly, the PC traces collected during Step 1 are *dynamic* instruction sequences. To avoid having to build a large (potentially exponential) number of dynamic PC traces for reference functions, we simply analyze the PCs corresponding to static instructions within each reference function. Testing similarity between victim and reference functions then proceeds as follows, as illustrated by Figure 6.11.

1. (One time, offline) For each reference function `f*`, collect the static PCs in that function (relative to the entry PC, to maintain position independence) into a set called `S*`.

2. For each victim function-level PC trace `t` (from Step 1), convert `t` to a set called `S` and compute similarity = $(|\mathtt{S} \cap \mathtt{S*}|)/|\mathtt{S}|$) for each reference PC set `S*`.

The percentage of PCs that survive the intersection indicates the similarity between the victim function and the reference function. We note that this heuristic makes use of x86's variable-length instruction encoding in an essential way: due to the nature of variable-length instruction encoding, the instruction length is directly influenced by the instruction semantics. For example, x86 uses different byte lengths for different opcodes and addressing modes. Additionally, variable instruction lengths add extra entropy to the PC trace, which serves as the fingerprint in our case. Also, notice that the similarity based on set intersection sacrifices information such as the ordering of PCs for simplicity. Section 6.8.3 discusses a more sophisticated approach that considers instruction ordering.

6.7 EVALUATION

6.7.1 Experimental Methodology

We perform the evaluation on Intel 9700 and 9900(K) CPUs, all running Ubuntu 18.04 kernel version 4.15.0, with Hyper-threading disabled. All tested cryptographic programs are compiled with gcc 7.5.0 (more compiler versions are used when evaluating function fingerprinting in Section 6.7.3). When evaluating the control-flow leakage attack in Section 6.7.2, the attacker and the victim run separated userspace processes. For the fingerprinting experiment in Section 6.7.3, the victim programs are written with Intel SGX SDK [305] and run inside the enclave.

6.7.2 Evaluating Control-flow Leakage Attack (Use Case 1)

We first evaluate whether NightVision can leak secret data in common cryptographic code through vulnerable functions with secret-dependent control flow. Specifically, we focus on leaking the secret key during the RSA key generation procedure in mbedTLS [279] version 3.0 by inferring the secret-dependent control-flow behavior in Great Common Divisor (`GCD`) function. `GCD` contains a loop, in which a perfectly-balanced branch repeatedly evaluates the secret key values. Recovering the secret key requires determining the direction of the balanced branch at each loop iteration. The recent Frontal attack [62] has already exploited this vulnerable function. On the other hand, Frontal can be mitigated by aligning two sides of the branch to the same base address modulo the instruction fetch window size (16-byte in Intel CPUs) using a simple compiler flag `-falign-jumps=16`. This flag is applied in our experiment.

We implement a proof-of-concept control-flow leakage attack by simulating the preemptive scheduling attack, following the same methodology as prior work [20, 75, 276, 287, 288, 289]. Specifically, we make the victim call `sched_yield()` system call after the branch body to yield to the attacker process. The attacker process executes the NV-U routine to deduce the secret control-flow decision of the current victim loop iteration, followed by a `sched_yield()` to transfer the control back to the victim for the next loop iteration. We notice that the per-victim-loop-iteration time is over 300 cycles, which is greater than the minimal time slice achievable by the preemptive scheduling attack [7, 284, 285]. Therefore, although our proof-of-concept attack simulates the attack preemption with `sched_yield()`, per-loop-iteration measurement is possible for `GCD`in practice.

We apply the strategy described in Section 6.5.2 to infer the direction of the balanced branch. Because the instructions on the *then* path occupy address range [`0x5940`, `0x597c`], and the instructions on the *else* path occupy address range [`0x5980`, `0x59bc`], we simply apply NV-U oracle with a PW range [`0x5980`, `0x598f`] that only overlaps with the first several instructions on the *else* path. When the attacker observes a misprediction in the probe step of NV-U, its BTB entry is updated (deallocated), meaning that the else path should be taken. We repeat the attack on 100 different victim process executions. Each run calls the RSA key generation function for generating a new key, and on average loops over the vulnerable branch 30 times in `GCD`. NightVision achieves 99.3% accuracy in measuring the direction of the vulnerable branch.

We additionally use NightVision to infer the secret predicate of a similarly balanced branch in the big number comparison (`bn_cmp`) function in Intel's IPP-Crypto [278] v2020. Frontal also evaluates this function. Similar to `GCD` above, Frontal cannot succeed when the

basic block alignment flag is enabled. NightVision again is able to achieve 100% accuracy in inferring the branch direction across 100 different runs.

6.7.3 Evaluating Function Fingerprinting (Use Case 2)

We now evaluate the effectiveness of NightVision's function fingerprinting in the private program setting. Since the goal of function fingerprinting is to reveal the use of vulnerable functions in the private victim binary, here we show that it is possible to use the proposed function fingerprinting mechanism to detect the use of `GCD` and `bn_cmp` evaluated in Section 6.7.2.

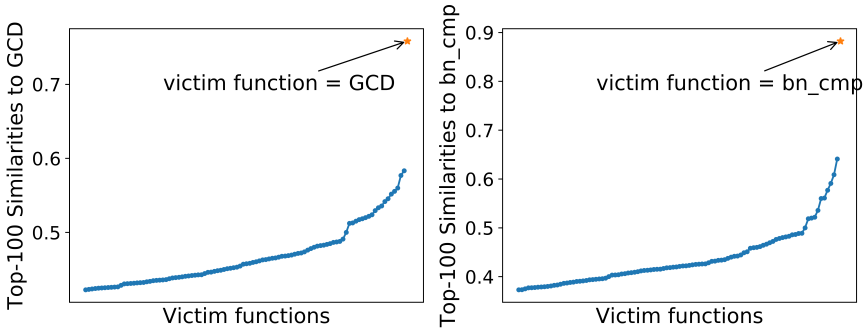


Figure 6.12: The top-100 highest similarity of the measured victim functions with respect to `GCD` (left) and `bn_cmp` (right). When matching against `GCD`as the reference function, the highest similarity is 75.8% when the victim function is also `GCD`. The highest similarity to `bn_cmp` is 88.2% (when the victim function is `bn_cmp`).

Effective fingerprinting should only match an unknown victim function to a reference function when the victim function is indeed the reference function. To validate that NightVision fulfills this requirement, we generate victim PC traces for `GCD`and `bn_cmp` as well as other 175,168 additional functions from many open-source SGX projects listed in [306]. Then we compute the similarity of all tested victim functions to the two reference functions `GCD` and `bn_cmp`. Figure 6.12 demonstrates that for both `GCD` and `bn_cmp`, we can observe the highest similarity when the victim function is indeed `GCD` or `bn_cmp`, whereas all other functions that are not the reference function exhibit lower similarity. This shows that NightVision can identify the two vulnerable functions in a large group of unknown, executed victim functions.

We notice that the similarities of `GCD` and `bn_cmp`with themselves are not even 100%. We compare PC traces, with their originating assembly code, and notice that nearly all incorrectly measured instructions correspond to *macro-fusion* structures [307, 308]. Since macro-fusion combines adjacent instructions (usually arithmetic-branch or load-arithmetic-branch) into a single, executable macro-op, one single step actually executes and retires all instructions in

a macro-fusion structure. Therefore, NightVision only manages to measure the leading instruction in a macro-op structure. The impact of macro-fusion on enclave single-stepping attacks has also been observed and studied by prior work, e.g., CopyCat [270].

Figure 6.13: (Left) The similarities of `GCD` in eight different MbedTLS versions (2.5-3.1) to `GCD` in the same/different versions computed by NightVision's fingerprinting mechanism. (Right) The similarity of `GCD` compiled with three different optimization levels to `GCD` with the same/different optimization levels.

NightVision's fingerprinting only handles functions in binary form, meaning that the version of the library to which the function belongs, and the compiler configuration, can both influence fingerprinting results. To evaluate how robust our fingerprinting is to these effects, we compile `GCD` by tuning three different parameters:

1. MbedTLS library version: 8 different versions as shown in Figure 6.13 (left)
2. GCC version: 7.5, 8.4, 9.4, 10.3
3. Compiler optimization flag: -O0, -O2, -O3

We then run each compiled `GCD` with NightVision and compute the similarity of each specific `GCD` with all compiled versions of `GCD`. We draw the following conclusions. First, the impact of the library versions on fingerprinting depends on whether the source code is changed across the library versions. We found that, for example, the `GCD` function source code does not change across MbedTLS versions 2.5-2.15, but version 2.16 has a different implementation than previous versions. This is reflected by Figure 6.13 (left): the similarity amongst versions 2.5-2.15 is much higher than the similarity between a version before 2.16 and one after 2.16. Second, the compiler version alone usually does not affect the function binary. Third, as demonstrated by Figure 6.13 (right), compiler flags can significantly impact the fingerprinting result. The similarity between the victim function and the target function may not be high enough when the victim function and the target function are compiled

with largely-different compilation options. In summary, to successfully identify the use of a specific target function in an unknown binary, the attacker should compile the function from different library versions, and with different compilation options.

6.8 DISCUSSION

6.8.1 Limitations

Affected CPUs In this work, we examined a limited set of Intel desktop/server CPUs listed in Section 6.2.3. However, any CPU that facilitates BTB updates by non-control instructions is potentially vulnerable to the same attack. Since deep processor pipelining and superscalar design are the underlying causes factors motivating this behavior, NightVision can also likely affect other existing high-performance CPUs. We leave the analysis of CPUs from other vendors as future work.

Limitations of Control-flow Leakage As explained in Section 6.5.2, NV-U relies on the preemptive scheduling attack to achieve fine-grained measurement of the victim's execution, similar to many existing side-channel attacks [7, 20, 75, 81, 276, 284, 285, 286, 287, 288, 289]. This technique is infeasible in restricted environments such as browsers, which limit the attacker's capability of arbitrarily spawning threads. Additionally, the preemptive scheduling attack does not provide perfect synchronization between the NV-U attacker and the victim, thus the attacker needs complementary techniques (such as the example in Section 6.5.2) to deduce the victim's execution progress.

Limitations of Binary Fingerprinting Our fingerprinting attack is based on several assumptions mentioned in Section 6.6. First, the attack is useful primarily in the context of variable-length ISAs (like x86), as the variety in the instruction length amplifies the entropy in the PC sequence, resulting in fewer false positives/negatives in fingerprinting. Second, the fingerprinted function itself must have a sufficiently long PC trace to produce enough entropy to differentiate it from incorrect candidates. Third, the attacker must possess knowledge of the target function in assembly form, which may require non-trivial effort from the attacker to prepare different possible assembly forms for the target function, similar to Section 6.7.3.

6.8.2 Mitigations

Data-Oblivious Programming The only reliable software mitigation for NightVision is to ensure the program control flow never depends on secret information, using data-oblivious programming. Achieving data-obliviousness for normal programs requires engineering effort to port existing applications [107, 219, 273, 309] and suffers from significant performance overhead [30].

We have seen existing cryptographic libraries gradually adopt data-oblivious programming for eliminating secret-dependent control-flows [278] and design efficient data-oblivious implementations for critical cryptographic operations [310]. That said, non-data-oblivious cryptographic libraries used by legacy binaries are still susceptible to NightVision. In addition, NightVision's function fingerprinting capability is unaffected by data-oblivious programming.

BTB Hardening NightVision can be mitigated by constantly flushing BTB state [19], or enforcing strict isolation between security domains [311, 312]. However, neither approach has been adopted by current processors, due to the performance cost and implementation complexity. The only effort in this direction is IBRS [170] and IBPB [169] which only prevent indirect branches belonging to different security domains from influencing each other, as stated in Section 6.4.1. Future processor generations could extend those defense proposals to block attacks like NightVision.

6.8.3 Improving Function Fingerprinting

NightVision's fingerprinting technique constructs function fingerprints by compressing sequences of dynamic PCs into sets of static PCs. This simplifies the fingerprint-matching problem to performing set intersections, but sacrifices information about the original PC sequences such as the instruction order (e.g., loops). An alternative fingerprinting mechanism could directly use the dynamic PC trace as the function fingerprint. In this case, matching the PC sequence to reference functions becomes a more-complicated pattern-matching problem: the PC sequence must obey the control flow of the original function, modulo the measurement error. We note that this process is similar to genomic (DNA) sequence matching, which pattern-matches a DNA sequence against several sample DNA sequences, and at the same time, circumvents the interference from mutated genes. We leave applying related approaches to improve NightVision's fingerprinting as future work.

Existing Side Channels Extracting PC-related Information Prior side-channel attacks leak partial information about victim's dynamic PCs. First, controlled-channel attacks manipulate page permissions and attributes (access/dirty bits) to observe the page-granularity PC trace [115, 274, 275]. Succeeding attacks leverage the instruction cache to obtain cache-granularity PC information [116]. These attacks potentially leak information of all dynamic PCs, but at a coarse spatial granularity. Correspondingly, several mitigations have been proposed to confine secret-dependent control flow, e.g., inside a single page or cache line [116, 117]. The more recent Frontal attack [62] exploits instruction decoding timing for deducing information about basic block alignment, i.e., code address offsets relative to 16 B blocks. However, the timing channel discovered by the Frontal attack is not sufficiently precise for determining byte-granularity PC information (use case 2 in our case); rather, it uses differential analysis to differentiate branch directions (use case 1 in our case, which can be mitigated as described in Section 6.7.2). Additionally, Frontal requires memory writes to be present inside the basic block.

Several other attacks achieve more fine-grained PC measurements for specific PC types. For instance, prior attacks targeting the BTB [18, 19] leverage collisions between control-transfer instructions to deduce if a branch/jump at a specific PC executes at runtime. Although these side-channel approaches have proved to successfully achieve different attack goals (e.g., control-flow leakage attacks, breaking ASLR, crafting Spectre attacks), control-flow randomization [272] converts conditional branches into randomized indirect jumps to thwart attacks targeting conditional branches, and recent hardware mitigations such as IBRS further provide protection for indirect jumps. Note that a prior attack BranchShadowing [19] is similar to NV-S in that both leverage SGX single-stepping and BTB Prime+Probe. However, NV-S's BTB Prime+Probe mechanism is inherently different from BranchShadowing (Section 6.2.5), hence NV-S can infer the PC of every dynamic enclave instructions while BranchShadowing is limited to behaviors of branches with known PCs. Other hardware structures, e.g., the hardware prefetcher [276], TLB [16, 275], reflect the PC information of memory loads and stores. NightVision is the first side-channel capable of leaking the precise byte-granular PC for all instructions types, and in the ideal case, for every single victim dynamic instruction.

Other Control-flow Leakage Attacks Many control-flow leakage attacks also do not rely on extracting dynamic PCs directly. Attacks which exploit directional branch predictors [20, 61] are well-known for leaking secret branch conditions. Similarly, such attacks can be mitigated with software defenses protecting conditional branches [272]. Recent attacks,

such as CopyCat [270] and Nemesis [271], assume a privileged attacker who can single-step the victim's execution, which is the same as the supervisor-level attacker model in this work. They show how to acquire secret-dependent control flow by counting instructions or deducing the executed instructions' types. However, such observations (instruction counting/types) are insufficient for control-flow leakage attacks if the victim deploys defenses such as branch balancing, and contain much less entropy for binary fingerprinting when compared to NightVision.

6.10 CONCLUSION

NightVision is the first micro-architectural side-channel attack that extracts dynamic, byte-granular PCs from the victim program's execution—in the best case, for every victim instruction. NightVision reveals victim dynamic PCs *directly*, thereby bypassing prior defenses that attempt to block channels leaking partial/indirect information about the PC sequence (such as the length of subsequent basic blocks). NightVision also showcases its ability to identify unknown programs, challenging the notion of "security through obscurity" while complementing existing side-channel attacks which by default require public code.

CHAPTER 7: SYNCHRONIZATION STORAGE CHANNELS

This chapter presents a novel technique, called Synchronization Storage Channels (or S^2C*), for mounting timer-less cache attacks targeting Apple M1 CPUs. Shared caches have been a prime target for mounting cross-core side-channel attacks. However, most cache attacks rely on timing measurements to indirectly infer cache state, hence their success hinges on the reliability and availability of accurate timing sources. Our key observation is that the implementation of synchronization instructions, specifically Load-Linked/Store-Conditional (LL/SC) allows attackers to directly observe cache state changes on architectures like Apple M1. Based on key insight, we reverse engineer the M1 cache organization and overcome several challenges in using LL/SC for monitoring cross-core cache access patterns. The end product* S^2C *empowers a single-threaded user process to perform standard cache attacks, such as breaking cryptographic implementations, without any reliance on timers.*

7.1 INTRODUCTION

The increasing complexity of modern processors has led to a plethora of micro-architectural side channels that can be exploited to infer sensitive information. Despite being one of the earliest targets to mount such attacks, the shared cache is still the most prominent, due to its shared use among all tenants on the same processor, the relative ease with which it can be monitored, and the richness of information that can be gleaned through it. By measuring the cache contention within the victim process, an attacker obtains information about the victim's memory access pattern, which can be useful in breaking cryptographic implementations [5, 15, 34], key logging [10, 73, 313], browser fingerprinting [314], model stealing [315], and aiding transient execution attacks [3, 4, 96].

A fundamental requirement of any cache side-channel attack is a way to accurately measure cache state changes. Most existing attacks *indirectly* observe the cache state by measuring memory access latencies with high-resolution timers. These can be used to deduce the cache level an address resides in, and to further deduce the presence of the victim's address(s) in those cache levels. Even if only provided a low-resolution timer, techniques have been proposed to enhance the effective resolution [134, 141, 316, 317]. Regardless, precise timing measurement is prone to (or can be aggravated by adding) noise [132, 135], requires micro-architecture-specific profiling (e.g., to ascertain cache latencies), and can be fully mitigated by blocking the use of explicit timers. The attacker can also craft 'implicit timers' with a counter incremented by sibling threads in the absence of explicit timers [140, 316, 318],

but this requires additional attacker capabilities such as running multiple attack threads concurrently.

To circumvent the limitations/defenses associated with timers, an attacker ideally would like a way to *directly* and *precisely* measure cache state without relying on timers. Yet, there is scant literature on such *timer-less* attacks, and at present all known methods have limitations. For instance, cache storage channels [288] rely on uncacheable memory, which can only be exploited by privileged attackers; Prime+Abort [59] and its variant [60] exploit Hardware Transactional Memory, which is a rare and even deprecated feature implemented only by specific vendors such as Intel [319]. Currently, no general-purpose primitive is available that allows userspace attackers to *directly* and *precisely* observe the shared cache state on modern CPUs.

This chapter's key insight is that the implementation of *hardware synchronization primitives* on modern CPUs, specifically Load-Linked/Store-Conditional (LL/SC) instructions on the Apple M1, can be exploited to directly measure whether cache evictions have occurred.

LL/SC are general-purpose instructions in many common ISAs (such as ARM and RISC-V) for implementing synchronization/mutual exclusion. In a nutshell: LL loads an address and marks it as 'exclusive'[23] in the memory system. SC is a store that a) 'succeeds', i.e., performs the store, if and only if its address has been marked 'exclusive' (by some older LL) and b) writes to a register whether it succeeded. If any store (including an SC) from any processor core writes to an address marked exclusive, the exclusive state is cleared. That is, LL-SC implements atomic read-modify-write when there isn't an intervening write to the target address, and performs a NOP when there is an intervening write/atomicity violation. *In both cases, it makes architectural state changes (by writing the success bit to a register).* In the normal use of LL-SC, if an SC fails, the thread using LL-SC will retry the LL-SC sequence until it reports success (e.g., lock acquired).

Ideally, only shared variables will be exclusive, and the variable's exclusivity status is maintained regardless of where the variable is cached in the memory hierarchy. However, we found that the implementation of LL/SC in the recent Apple M1 drops the exclusive semantic when the address is evicted from the L1 data cache, causing the later SC to fail conservatively. In practice, this design does not compromise correctness, as a benign use of LL/SC involves retrying when SC fails. However, it *does* enable attackers to directly measure whether a local L1 eviction has taken place—using only a single user-space attacker thread and without the use of any timer—by monitoring the result of the SC.

Following this idea, we propose *Synchronization Storage Channels (*S^2C*)*, the first timer-less

[23]Not to be confused with the Exclusive (E) state in modern cache coherence protocols, e.g., MESI, which is a related but different concept.

cache side-channel attack technique on the Apple M1, and also the first micro-architectural attack to exploit hardware synchronization instructions (specifically LL/SC). At a high level, S^2C aims to build a cross-core side channel to monitor memory accesses of other processes sharing the same cache as the attacker.

The technical core of the paper proposes techniques to overcome two main limitations in LL/SC semantics that impede shared cache attacks. First, LL/SC on the M1 only indicates L1 evictions, but to perform cross-core attacks, notifications from the shared L2 cache (L2 evictions) are required (both for building L2 eviction sets and performing the actual attack). Second, LL/SC only allows the attacker to monitor evictions on a single cache line, as opposed to traditional timer-based cache attacks that pre-occupy a cache set with multiple attacker-controlled lines and monitor all those lines. Sometimes, the attacker may also need to simultaneously monitor evictions happening to multiple cache sets.

To enable S^2C-based attacks that can monitor a single L2 set, we reverse-engineer a variety of new features related to the L2 cache on the M1 (for example, the L2 is inclusive, implements the AutoLock optimization [320] and uses a dynamic replacement policy). We then develop techniques that exploit said features to enable single-threaded, unprivileged Prime+Probe-like attacks through the L2 using only LL/SC.

To enable S^2C-based attacks that can simultaneously monitor multiple L2 cache sets, the insight is to view LL/SC *as a general-purpose single-bit communication channel*, as opposed to *just* a means to monitor whether a victim accessed the particular cache set. Following this, we construct a weird circuit [321] that micro-architecturally computes the logical-OR of whether the victim accessed at least one of an attacker-specified set of L2 cache sets—and communicates the single-bit result of this logical-OR through LL/SC to the attacker's architectural state. While weird circuit constructions are not the main focus of the paper, our weird circuit is relatively simple conceptually (relies purely on out-of-order execution and LL/SC as opposed to requiring speculative execution/Intel's TSX [321]), and thus may be of independent interest.

Contributions In summary, this chapter makes the following contributions.

- We present the first timer-less cross-core cache attack technique that exploits hardware synchronization instructions, namely Load-Linked/Store-Conditional (LL/SC) on the Apple M1. We call this technique Synchronization Storage Channels (S^2C).
- We identify that LL/SC serves as a *direct* and *precise* architectural observation channel to monitor micro-architectural events (L1 evictions).

- We conduct a detailed reverse-engineering of the M1's L2 caches, which is necessary to exploit S^{2}C and may benefit future attacks against the M1.
- We develop techniques to overcome challenges in using LL/SC to perform cache attacks. In particular, we develop methods to monitor L2 evictions using LL/SC and methods to simultaneously monitor evictions on multiple L2 sets (although LL/SC natively tracks L1 evictions of only a single line at a time).
- We show that S^{2}C can simultaneously monitor up to 11 L2 sets with high accuracy, and can be used for building covert channels as well as attacking cryptographic implementations such as T-table AES.

7.2 CACHE SIDE-CHANNEL ATTACKS

The goal of a cache side-channel attack is to infer a victim program's secret-dependent memory access pattern by monitoring the victim's use of a shared cache. In general, cache attacks can be categorized into the following two types.

The first type, known as *flush-based*, works by the attacker flushing a line corresponding to a shared address from the cache and monitoring whether the victim re-reads said address (refilling the cache with the corresponding line). This type includes techniques such as Flush+Reload [7, 15] and its variants Evict+Reload [10] and Flush+Flush [57]. These attacks are only capable of learning memory access to data that is shared between the attacker and the victim. Nonetheless, they have the advantage of inferring the precise cache line-granular address accessed by the victim.

The second type, known as *contention-based*, relaxes the requirement for shared memory by monitoring how the victim's cache lines (addresses) contend for space in the cache with the attacker's cache lines (addresses). Our attack falls into this category. All contention-based attacks, such as Prime+Probe [5, 34, 56, 72], Prime+Abort [59, 60], Reload+Refresh [58], and Prime+Scope [35], follow a similar attack procedure. First, the attacker *primes* the cache by filling a target shared cache set with attacker-controlled lines. Later, the attacker *probes* the same cache set to observe whether any of its lines have been evicted, and from this deduces if the victim line(s) has been accessed. By monitoring shared cache contention, contention-based attacks do not require shared memory, but they only learn a subset of a victim address bits (i.e., those bits used to choose the cache set).

An accurate method for determining the location of a specified cache line within the cache hierarchy, is crucial for any cache attack. Most methods are based on timing measurements

(*timer-based* attacks), yet several techniques achieve this without relying on timing (*timer-less* attacks).

Timer-based attacks Most cache attacks rely on high-resolution timers to measure the latency of accessing a specific cache line, by either reading the cycle count register directly (e.g., via `rdtscp` in x86), or exploiting special instructions that interact with the cycle counter register (e.g., `monitor`/`mwait` in x86 [138]). When only low-resolution timers are accessible, the attacker can also leverage existing techniques to amplify small access latency differences so that they are detectable by the timer [134, 141, 316, 317]. Despite these efforts, a fuzzy or inaccessible timing source can still impede attacks that rely on timers [132, 134, 135]. When an explicit timer is absent, a counter incremented by a sibling thread can serve as an *implicit* timer [140, 316, 318]. However, this requires the attacker's ability such as spawning and concurrently running multiple threads, which may not be feasible in practice. For instance, Javascript programs in browsers are single-threaded and sandboxed [139, 141].

Timer-less attacks The limitations of timers can be overcome if the attacker has the ability to directly measure cache state, i.e., directly convert micro-architectural state changes in the cache to architectural state changes in the register file. However, existing methods for doing so are limited and scarce. One method is cache storage channels [288], which directly returns to the attacker whether its data is cached or not. This primitive, however, is not available to normal user-space attackers as it requires configuring non-cacheable memory. Hardware Transactional Memory is another feature that communicates specific types of cache misses/evictions as transactions abort when their data is evicted from the shared cache [59, 60]. Yet, this feature is only available on some Intel products and has been deprecated. Intel has also disabled the use of TSX by default on CPUs that support it through a microcode update [322].

7.3 APPLE M1

Apple has recently started using a new processor architecture on its laptop, desktop, and tablet devices. The new processor design, including the M1 and the latest M2, is based on the ARMv8-A ISA (a variant of ARM64). We have confirmed that our findings about LL/SC in Apple processors that are later discussed (Section 7.4) apply to both the M1 and the newest M2, but in this work, we mainly focus on the M1.

7.3.1 Apple M1 Cache Organization

	Level	Ways	Sets	Line Size	Total size
P-core	L1D	8	256	64 B	128 KB
	L2	12	8192	128 B	12 MB
E-core	L1D	8	128	64 B	64 KB
	L2	16	2048	128 B	4 MB

Table 7.1: Apple M1 cache parameters (from Table 2 of [87])

An M1 CPU consists of four performance-oriented cores (P-cores) and four energy-oriented cores (E-cores). Based on prior works [87], each P-core/E-core has its own private L1 data cache (L1). There are two separated L2 caches, one shared among all four P-cores, and the other shared among all four E-cores. The associativity, number of sets, line size, and total size of each cache is shown in Table 7.1. The M1 supports regular pages of size 16 KB and 32 MB huge pages natively. Notice that M1 does not implement Simultaneous Multi-Threading (SMT), thus each core only runs one thread.

Previous studies [140, 323] have highlighted two difficulties in performing cache attacks on ARM processors. By adopting the ARM ISA, the M1 also inherits both difficulties. First, the cycle count register on the M1 is only accessible by privileged software, unlike x86 processors where the corresponding register can be read by unprivileged instructions. Second, ARM does not have dedicated instructions for flushing a specific address from caches (like `clflush` on x86), making attack methods that rely on those instructions, such as Flush+Reload [15] and Flush+Flush [57], ineffective on the M1.

7.3.2 Load-Linked/Store-Conditional in ARM64

```
1 Retry: lock_val = LDREX [lock_addr]
2 if (lock_val == FREE) {
3   fail = STREX BUSY, [lock_addr]
4   if (fail) { goto Retry; }
5 }
```

Figure 7.1: A test-and-set-style lock implemented with ldrex/strex. Even when all threads perform ldrex simultaneously and see a `FREE` lock, only one thread will successfully perform strex and acquire the lock. All other threads will encounter failed strex and retry.

Load-Linked and Store-Conditional, also referred to as Load-Exclusive and Store-Exclusive (ldrex/strex) by ARM [324], are used in common RISC ISAs such as ARM, MIPS, and RISC-V

for implementing synchronization and mutual exclusion. A ldrex loads a specified address while marking it as *exclusive with respect to the current core* in memory. This means that an address can be exclusive to multiple cores at a given time when multiple sibling threads running on different cores execute ldrex concurrently (e.g., for competing for a lock like Figure 7.1). One core can track *at most one exclusive address*, meaning a younger ldrex will overturn the exclusive address marked by an older ldrex on this core.

A strex only performs the store operation when the address is *exclusive to the current core* and returns a bit in its output operand indicating whether the store is performed. Whether a strex succeeds or not, it clears all exclusive states associated with its address for every core. This ensures that when multiple threads simultaneously perform ldrex, such as competing for a lock in Figure 7.1, only the first thread to perform strex will succeed, forcing all other threads to lose their exclusive access to the lock and retry the procedure. Lastly, a regular store behaves like an always-succeeding strex and it also changes the address back to non-exclusive for all cores.

The exclusive states of cached addresses and how they respond to different memory operations are managed by an *exclusive monitor* [325]. The exclusive monitor tracks exclusiveness at a granularity called *Exclusive Reservation Granule (ERG)*. Whether the exclusive monitor is implemented as a dedicated unit or integrated with existing cache logic such as cache coherence protocol, as well as the ERG size, is design-specific.

Exclusive address vs. the Exclusive cache-coherent state. Modern cache-coherence protocols, e.g., MESI, use an Exclusive (E) state to reduce bus invalidations when Shared+Clean (S) data is Modified (M). Although the exclusive monitor implementation may piggyback on top of the coherence protocol, it has different semantics when compared to the E cache-coherence state. E in cache coherence means "clean, owned by a single core". However, in the context of ldrex/strex, a single address can be marked exclusive simultaneously by multiple cores. For the rest of the paper, we refer to the target address of ldrex/strex as the *exclusive address*.

7.4 NEW ATTACK PRIMITIVE ON M1 USING LL/SC

This section introduces the attack primitive associated with the behavior of ldrex and strex on the Apple M1, which enables the S^2C attack technique proposed in this work.

7.4.1 Micro-architectural strex Failures

As explained in Section 7.3.2: A strex instruction, immediately following a ldrex, might return 'failed' when multiple threads are competing to write to the same shared data. This requires the implementation of the exclusive monitor to track the address's exclusiveness regardless of its location in the cache hierarchy. However, the official ARM specification states that in some implementations, strex might return fail for micro-architectural reasons, e.g., cache evictions [326]:

> *An implementation might clear an exclusive monitor between the ldrex and the strex, without any application-related cause. For example, this might happen because of* cache evictions.

We investigated whether such an implementation exists, looking specifically at the widely-used Apple M1. Our questions are: *Can a single-threaded program, that executes ldrex-strex to its own private data, see strex failures due to cache evictions—even if it performs no action that is known to revoke the exclusive state (from Section 7.3.2)? If so, from what level(s) of the cache can said evictions lead to strex failures?*

The rest of this section answers these questions. To summarize: in a ldrex-strex sequence, even when ldrex and strex instructions are applied to private data, strex can fail when the exclusive address accessed by ldrex is evicted from the L1 cache before strex executes.

7.4.2 Experiment Design and Methodology

To answer the questions in Section 7.4.1, we designed the experiment shown in Algorithm 7.1. The code contains a ldrex (line 3) and a strex (line 6) to addr, which points to a local variable. Between the ldrex and strex, we traverse a set of random addresses S (S never includes addr) which may evict addr from the L1 cache (line 4). To identify the location of addr after the possible cache eviction, we load addr right before strex (line 5), and compare its access latency with the L1 access latency L1_Latency using timers. The experiment, therefore, studies how strex failures correlate with L1 evictions.

We test the above code on both P-cores and E-cores, by configuring the thread quality of service [327]. The load latency is measured by reading the M1's cycle count register before and after the load and computing the difference. We use a custom kernel extension (similar to the prior Pacman Attack [87]). Note that we read timers in this experiment, not in the actual attack technique. L1_Latency can be obtained by executing two consecutive loads to the same address (with a memory barrier in between for maintaining ordering) and measuring the latency of the second load.

Algorithm 7.1: Code for testing the correlation between strex failures and L1 evictions of the exclusive address.

```
   Input: addr: a selected target exclusive address
 1 for i = 1 to 1000 do
 2     Generate a set of random addresses S
 3     val = LDREX [addr]
 4     traverse S // may or may not evict addr from L1
 5     latency = measure latency of load [addr]
 6     fail = STREX val, [addr]
 7     evicted_from_L1 = latency ¿ L1_Latency
 8     print fail, evicted_from_L1
 9 // An example of counting the output:
10 // fail = True, evicted_from_L1 = True:  518
11 // fail = True, evicted_from_L1 = False:  0
12 // fail = False, evicted_from_L1 = True:  0
13 // fail = False, evicted_from_L1 = False:  482
```

7.4.3 Result and Takeaway

Line 10-13 in Algorithm 7.1 shows an example of counting the output of the experiment. We further ran Algorithm 7.1 over 10 times, and consistently observed that fail correlates perfectly with evict_from_L1. In other words, strex to local exclusive address never fails when the address still remains in the L1, and always fails when the address resides only in lower levels of the cache. This observation aligns with the information in ARM's documentation (Section 7.4.1). Therefore, we can confirm that on Apple M1 CPUs, when a ldrex-strex pair is applied to a local address, strex can fail if the address is evicted from the L1 in between when the ldrex and strex are performed. We discuss the potential implementation of the exclusive monitor on the M1 that can produce this behavior in Section 7.9.

We performed another experiment to determine the ERG size. The experiment consists of only a single ldrex followed immediately by a strex, but they have different addresses. We observed that strex can succeed even if ldrex and strex addresses differ. Also, strex only succeeds when the two addresses are on the same 64-byte-aligned block. The fact that ERG size is 64-byte (L1 line size) not 128-byte (L2 line size) implies that the exclusive monitor may be implemented to only track exclusiveness for L1 lines, which results in our above finding about strex failures. We also conduct experiments to verify that the behavior of ldrex/strex follows ARM's specification (Section 7.3.2). For example, we find that each core can indeed only monitor one exclusive address at a time.

Microarchitectural factors influencing the result of strex might seem like a bug that affects correctness. However, in practice, this behavior is acceptable. Regular programs always use

ldrex and strex for thread synchronization purposes, which are inherently non-deterministic due to unpredictable thread interleavings. Programmers will normally use the output of strex to create higher-level synchronization primitives, for example by retrying the code until the strex succeeds as shown in Figure 7.1. Therefore, a well-written program should not break given this additional cause of strex failures, apart from possibly wasting cycles from unnecessary code retries.[24]

From the security perspective, this observation is relevant in the context of cache side-channel attacks. As explained in Section 7.2, the ability to directly and accurately observe cache state circumvents the defenses and other challenges associated with timer-based attacks. Utilizing ldrex and strex instructions, an attacker can detect L1 evictions, which can be caused by contention in lower-level shared caches, such as the L2 cache in M1 processors. On the other hand, such detection is limited to one exclusive address at a time, since each core can only monitor one exclusive address. The rest of the paper builds upon this idea and presents S^2C, a timer-less attack technique aimed at the M1's shared L2 caches.

7.5 REVERSE-ENGINEERING M1'S SHARED L2 CACHE

The goal of the S^2C attack technique is to leverage ldrex and strex to expose information leakage through the shared L2 cache, which requires detailed knowledge about the M1's L2 cache. Given the lack of such information in existing research, in this work, we present the first detailed reverse-engineering of the M1's shared L2 caches, which encompasses various key characteristics of the L2 cache, such as the inclusion policy (Section 7.5.3), the replacement policy (Section 7.5.4) and the set index mapping (Section 7.5.5). Due to the observability of ldrex/strex being limited to the L1 (Section 7.4.3), these details are essential for a successful S^2C-based attack. The information in this section may also benefit other cache attacks.

We describe the reverse-engineering process with a primary emphasis on the caches used by P-cores, as the results obtained from E-cores are similar. This section's reverse engineering makes use of all capabilities required, such as making extensive use of the privileged cycle count register, and executing experimental code on multiple cores simultaneously. For the actual attack technique, starting in Section 7.6, we limit attackers to single-threaded execution without access to timers.

[24]Note that livelock can occur when too many memory accesses are placed between ldrex and strex, causing the exclusive address to be constantly evicted from L1. There is no guarantee that this won't occur, but programmer guidance (which says to minimize the use of instructions between a ldrex and strex pair, plus several other rules of thumb) tries to minimize its likelihood in practice [326]. We discuss this further in Section 7.9.

Terminology We define addresses mapped to the same cache set as *congruent* addresses, and further define addresses mapped to the same L1 set as *L1-congruent*. Correspondingly, *L2-congruent* addresses are mapped to the same L2 set. An *L1/L2 eviction set* for a target address is defined as a set of addresses that are L1-/L2-congruent to the target address, with the size of at least the L1/L2 associativity. Such sets can be used to evict the target address from L1 or L2, respectively.

7.5.1 Reverse-Engineering Private L1s

Before reverse-engineering the L2 cache, we first investigate the private L1 cache. We suspect that, like most modern CPUs, the M1's L1 cache is also virtually-indexed/physically-tagged, with a Least Recently Used (LRU) replacement policy. In this case, the L1 set index will correspond to the regular page offset subtracting the L1 line offset bits.

To test this, we first choose a random address and create a candidate L1 eviction set comprised of eight addresses (since L1 is 8-way-associative) that share the same L1 set index bits as the target address. We start by accessing the target address, then traverse all 8 addresses in the candidate eviction set, and lastly measure the access latency to the target address and compare the latency with the L1 latency obtained from Section 7.4.2.

The results show that the candidate L1 eviction set always evicts the target address out of the L1 cache. Also if we drop one address from the eight-element eviction set, the target address will never be evicted. This confirms our hypothesis:

Observation 1. L1 sets are indexed by the page offset bits subtracting the L1 line offset bits. The L1 cache adopts an LRU-based replacement policy for choosing evicted lines.

7.5.2 L2 Eviction Set Generation with a Timer

Our L2 cache reverse-engineering process relies heavily on L2 evictions. Since ARM does not provide cache flush instructions, we generate L2 eviction sets for inducing L2 evictions, similar to previous studies [140, 323, 328, 329]. However, identifying L2-congruent addresses for building the L2 eviction set is challenging given that the L2 set index may depend on physical page number bits.

Vila et al.'s algorithm [330] is the most popular eviction set generation algorithm that overcomes this challenge. The algorithm starts by adding randomly generated virtual addresses to a candidate set, and tests whether the candidate set can evict the previously loaded target address from the L2, until enough L2-congruent addresses are included and

the eviction appears. At this moment, the candidate set is an *L2 eviction set superset*, meaning that it contains a valid L2 eviction set, but also a significant number of redundant, non-L2-congruent addresses. One thing to notice is that the L2's inclusion policy is not yet known. Assuming the L2 is exclusive of the L1, the first loaded target address will only be cached in the L1, not L2, therefore a valid L2 eviction set cannot evict the target address from the L2 unless it evicts the target address from the L1 first. So we force Vila et al.'s algorithm to generate only addresses with identical L1 set indices as the target address, ensuring that a valid L2 eviction set can always evict the target address from the L1 to the L2 first, and subsequently from the L2. Once obtaining the L2 eviction set superset, the algorithm prunes the superset iteratively and checks whether the remaining is still capable of evicting the target address until the set size reaches the L2 associativity, and the superset is now reduced down to a minimal L2 eviction set.

7.5.3 L2 Cache Inclusion Policy and AutoLock

Understanding the cache inclusion policy is crucial for cache attacks, especially our new S^2C attack technique, where the attacker can only observe the L1 cache state. Case in point, if the shared L2 is inclusive of the private L1, contention within the L2 will cause L1 evictions, allowing the attacker to detect cross-core activities by monitoring the L1.

One straightforward method for determining whether the L2 cache is inclusive of the L1 works as follows. First, we randomly select an address and create its L2 eviction set using the technique in Section 7.5.2. Next, we access this address on a processor core, and then traverse the eviction set on a different core. If the L2 cache is inclusive, traversing the eviction set will evict the address from the L2, and correspondingly from L1. If the cache is non-inclusive/exclusive, such L1 eviction will not happen. Hence whether the L2 is inclusive can be determined by measuring the access latency to the address in the end and comparing it with L1_Latency.

After running this experiment with different addresses, we always observe an L1 hit. In fact, a recent attack on the Apple M1 by Hetterich et al. [329] also observed this phenomenon and suggested that it could be due to several factors, namely an exclusive/non-inclusive policy or a hardware optimization used in some ARM CPUs called *AutoLock*. *AutoLock* is a common optimization used by ARM CPUs in conjunction with inclusive caches [320]. When evicting data from the shared inclusive L2, AutoLock prioritizes data that is not present in L1 caches, effectively *locking* those that are present in the L1, since they are likely to be frequently reused. However, Hetterich et al. did not confirm whether the M1's L2 cache is inclusive with AutoLock or exclusive/non-inclusive [329]. The rest of Section 7.5.3 explains

how we determine that the L2 cache is actually inclusive with AutoLock, by leveraging a technique called *eviction set splitting*.

Eviction Set Splitting

We first eliminate the impact of AutoLock on our analysis of the inclusion policy. A standard AutoLock mechanism chooses L2 lines with no L1 copies to evict over those with L1 copies. Therefore, for an L2 eviction set to evict the target address cached in another core's L1 (hence AutoLocked), we must ensure that the L2 eviction set addresses are also AutoLocked. In this way, the replacement logic must evict AutoLocked lines for bringing in new lines. The original eviction set generated from Section 7.5.2 cannot achieve this, because all eviction set elements are mapped to the same L1 set, so there are always eviction set elements cached only by the L2, as shown in Figure 7.2 (left).

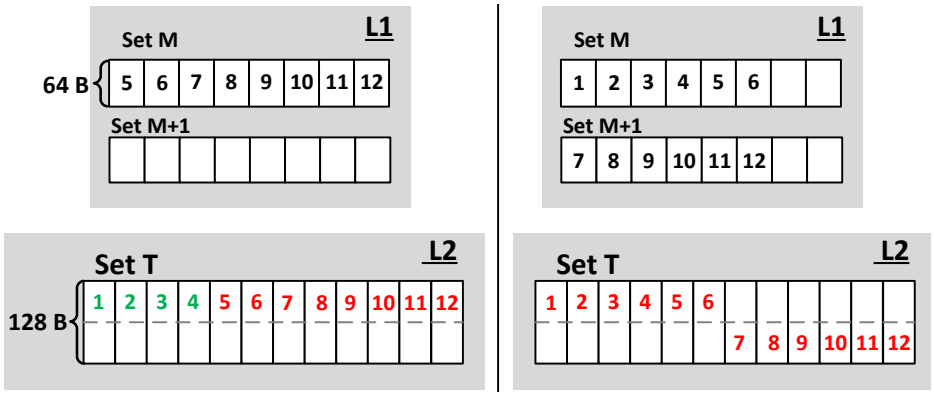


Figure 7.2: How eviction set splitting circumvents AutoLock. Every number represents a different address, and the values represent the access order. Green marks addresses without L1 copies (non-AutoLocked). Red marks addresses with L1 copies (AutoLocked). The right part shows eviction set splitting: the minimal 12-element eviction set is distributed over two adjacent L1 sets, ensuring all elements are AutoLocked.

Yet, we notice that the L2 line size in the M1 is 128 B, twice the size of L1 lines. This means that data in one L2 set can reside in two adjacent L1 sets. By toggling bit 6 in half of the eviction set addresses, as shown in Figure 7.2 (right), those addresses can be moved to the other half of the L2 lines, and their L1 copies are moved to the adjacent L1 set. Now, since all addresses own L1 copies, AutoLock, assuming it exists, can no longer interfere with the L2 eviction.

L2 Inclusion Policy

We use Algorithm 7.2 to determine the inclusion policy of L2. The algorithm first generates L2-congruent addresses following Section 7.5.2, and chooses 1-4 addresses to form a set T. The rest of the addresses, serving as the L2 eviction set of T, undergo eviction set splitting and form S. Hence, all lines in S and T are AutoLocked. The algorithm then traverses T on core 1, and subsequently traverses S on a different core 2 and measures the latency of traversing S.

Algorithm 7.2: Code for testing L2's inclusion policy.

Input: T: A set of addresses mapped to the same L2 set
S: A **split** L2 eviction set of T (T, S are in the same L2 set)
Output: Cycles spent by the traversal of S

```
1 Function Is_Inclusive(T, S):
2     [Core 1] traverse T
3     [Core 2] t = measure latency of traversing S
4     return t
```

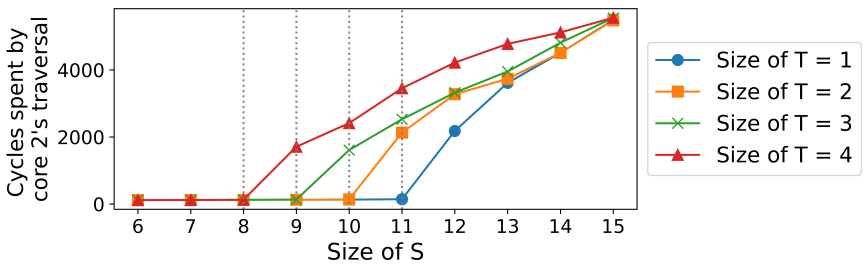


Figure 7.3: Time spent to traverse the split S when varying the size of S and T in Algorithm 7.2. This result shows that the L2 cache is inclusive of the L1.

Figure 7.3 shows the result of the experiment. This result revealed that the L1 cache is inclusive, because, from core 2's perspective, the entire S can be cached in L1 only if the total size of T and S does not exceed the L2 associativity. In other words, the contention with T in the L2 causes S to be evicted from the L1, which is only possible in an inclusive L2.

AutoLock

We further use Algorithm 7.3 to verify that the L2 indeed implements the AutoLock mechanism. The experiment starts by accessing addr, and then traverses its eviction set S on a different core. We explore different ways of splitting S by varying the number of addresses

Algorithm 7.3: Code for verifying AutoLock's effect.

Input: `addr`: A randomly selected address
S: A minimal L2 eviction set of `addr` generated by Section 7.5.2
Output: Latency of the 2nd access to `addr`

```
1 Function Check_AutoLock(addr, S):
2     Split S into n on one L1 set and |S| − n on the other L1 set
3     [Core 1] load [addr]
4     [Core 2] traverse S
5     [Core 1] t = measure latency of load [addr]
6     return t
```

in S to be flipped to the other L1 set. We observe that addr can be evicted when the smaller half of S has at least 3 addresses (meaning 3 or 9 addresses are flipped). This proves the existence of AutoLock: when fewer than 3 eviction set elements reside in an L1 set on core 2, the other L1 set on core 2 can only hold 8 eviction set addresses. Hence, at most 11 addresses out of 12 in the L2 set are AutoLocked, including at most $2 + 8 = 10$ eviction set elements plus addr, thus addr is never evicted. Hence, we can conclude that:

Observation 2. M1's shared L2 is inclusive of private L1s. M1 also employs the AutoLock optimization [320] to prioritize data without L1 copies for eviction from the L2.

7.5.4 L2 Replacement Policy

Algorithm 7.3 can also be utilized to investigate the L2 replacement policy. After applying eviction set splitting to S, addr and addresses in S are all AutoLocked thus evictable from the L2. We examine the replacement policy by measuring how difficult it is to evict addr as we vary the size of S and the number of times traverse function iterates over S.

The L2 eviction rate with different eviction set sizes and iteration counts is displayed in Figure 7.4 (a). The L2 eviction rate is always 0% when the size of S is less than the L2 associativity (due to AutoLock), and it increases to 100% as S grows. In addition, by iterating over S more times, a smaller S can guarantee to evict addr. For example, iterating over the minimal eviction set at least 8 times guarantees the eviction of addr in practice. These observations imply that a non-LRU replacement policy is adopted. In fact, prior studies have shown that a pseudo-random replacement policy is adopted by the L2 cache in previous Apple ARM CPUs [140, 331, 332].

Since the experiment above uses cache lines belonging to different cores, we ask whether the replacement policy behaves differently for lines belonging to the same core. Thus, we

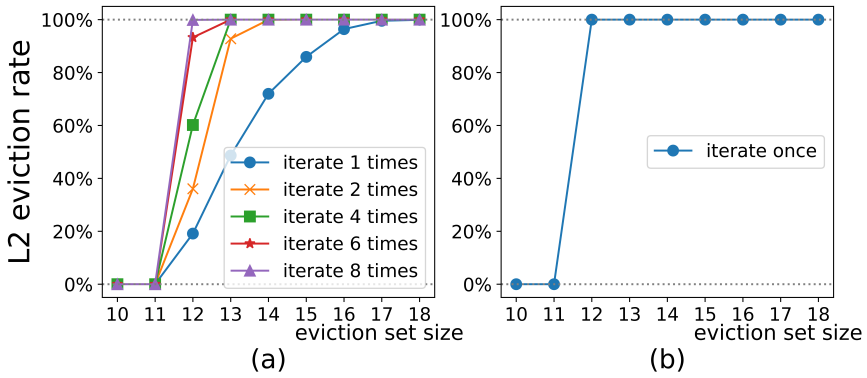


Figure 7.4: (a) The L2 eviction rate of addr when we run Algorithm 7.3 but allow Core 2 to traverse S multiple times. (b) The L2 eviction rate of addr when we run Algorithm 7.3 but traverse S on Core 1 (the same core as load [addr]) instead of Core 2 and only iterate over S once.

change line 4 in Algorithm 7.3 such that the eviction set S is accessed by the same core as addr. Also, traverse only iterates over S once. The result shown in Figure 7.4 (b) indicates that even one iteration over the minimal L2 eviction set S can guarantee to evict addr. This is possible only when an LRU-based policy is used. We make the following observation:

Observation 3. The L2 replacement policy behaves as LRU when eviction candidates are lines belonging to the same core, and non-LRU (possibly pseudo-random) when lines belonging to different cores can be evicted.

7.5.5 L2 Cache Set Index Mapping

The L2 eviction set generation technique Section 7.5.2 is agnostic about the actual L2 set index mapping function. However, this technique is not applicable when the cache state is measured with ldrex/strex instead of timers, as shown in Section 7.6.3. Here, we aim to learn the L2 set index mapping which is later used by S^2C for generating L2 eviction sets.

We reverse-engineer the undocumented L2 set index hash function by inspecting the physical addresses of L2-congruent addresses, which can be retrieved by `/proc/self/pagemap` on Linux. Based on previous research on reverse-engineering Intel's undocumented set/slice mapping function [333, 334, 335], we speculate that on M1, every L2 set index bit is also computed through a reduction operation with exclusive-or (xor) against a specific set of physical address bits. To verify this, we generate a large number of mutually L2-congruent addresses and compute the xor value of different combinations of physical address bits. When a combination is actually used for computing an L2 set index bit, the xor reduction will

produce the same bit value for every L2-congruent physical address. We demonstrate how every L2 set index bit is formed from physical address bits in Figure 7.5 based on our experiment results[25]. Notice that 11 out of 13 set index bits are directly mapped from huge page offset bits. The other 2 bits are computed over a complex xor operation involving the huge page number bits.

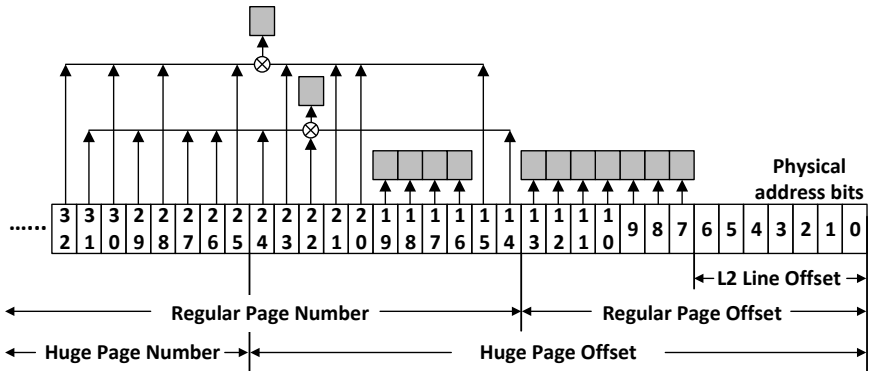


Figure 7.5: L2 set index bit mapping based on our reverse engineering (Section 7.5.5). Eleven L2 set index bits are directly mapped from huge page offset bits. Two L2 set index bits on the top are XOR-ed from multiple regular page number bits ($\otimes$ denotes the XOR operation).

7.6 MONITORING A SINGLE CACHE SET

We now present a protocol that enables the attacker (the receiver) to use S^2C to monitor a single L2 cache set that the victim may access. Section 7.7 will describe how to generalize the protocol to simultaneously monitor multiple L2 sets.

7.6.1 Attacker Model and Overview

We assume an attacker who co-locates with a victim process on the same Apple M1 processor, and shares the same L2 cache. The attacker can run arbitrary unprivileged code, but the attacker has no access to any timing source (such as `PMC0`) and is limited to using a single core. Such an attacker cannot use timing measurement techniques mentioned in Section 7.2, including the implicit timer since no two attacker-controlled threads can run concurrently. The attacker can allocate huge pages, which is required for generating eviction sets in Section 7.6.3.

[25]This pattern resembles the address mapping in Intel CPUs, in which the LLC set index is directly mapped from huge page offset bits (subtracting cache line offset bits) and LLC slice index bits are xor-ed from the high-order page number bits [334, 335]. It is possible that M1 also divides the L2 cache into as many slices as cores, resulting in a 2-bit L2 slice index, and each slice owns $8192/4 = 2^{11}$ sets.

The attacker's goal is to monitor the victim's memory access patterns, specifically, whether/how the victim accesses a target address that maps to a specific L2 set. As a contention-based cache attack, S^{2}C cannot differentiate victim addresses that are mapped to the same L2 set, as mentioned in Section 7.2.

Similar to existing cache attacks [5, 10, 15, 35, 57, 59], a complete S^{2}C-based attack has two phases: a preparation phase when the attacker generates the L2 eviction set for the target victim address, and an attack phase when the attacker detects the victim's access to the target address in real-time. We explain the attack phase first in Section 7.6.2 assuming minimal L2 eviction sets for the target address are available, and describe how to generate the L2 eviction set using ldrex/strex instead of relying on timing measurements in Section 7.6.3.

7.6.2 Attack Phase

A strex leaks 1-bit of information about whether an attacker-controlled exclusive address is evicted from the L1 cache. To relate this bit to the victim's activities on a single target address, an S^{2}C attacker can simply co-locate the exclusive address with the victim target address in the same L2 set, with additional efforts to ensure that the strex fails if and only if the victim reads/writes to the victim target address.

This strategy faces two unique challenges. First, strex only indicates data's presence in L1, unlike timing measurements which reveal the exact cache level through concrete latency numbers. To avoid false positives, the exclusive address visited by ldrex must remain in the L1 until the expected L2 set contention occurs, which implies a victim's access to the target address. Second, unlike normal Prime+Probe which can measure the access latency to multiple addresses, ldrex/strex only observes one specific address. This necessitates that our attack performs a very delicate balancing act: we must ensure that the exclusive address is not only evictable (not constrained by AutoLock), but that *it* is the line that gets evicted by the victim's access to the target address.

We now explain how S^{2}C addresses both challenges, using Algorithm 7.4 as a guide. The attacker can obtain an exclusive address P that is L2-congruent with the target address addr, by choosing an arbitrary address from addr's minimal L2 eviction set S. The rest of the eviction set S^P is traversed on the same core after ldrex completes. Importantly, we must apply eviction set splitting to this L2 eviction set S. This guarantees that every L2 eviction set element, including P and all of S^P, are cached in the L1. This addresses the first challenge. For this exact same reason, P and S^P are AutoLocked and occupy the entire L2 set, meaning P can be chosen by the L2 replacement logic, according to the mechanism of AutoLock Section 4. Additionally, our study of the L2 replacement policy in Section 7.5.4

Algorithm 7.4: How S^{2}C monitors the victim's access to a single target address `addr` using ldrex/strex.

Input: P: exclusive address
S^P: remaining eviction set of `addr` excluding P
Output: Boolean value indicating if `addr` is accessed

```
1 Function Monitor_Single_Addr(P, S^P):
2     [attacker on core X] val = LDREX [P]
3     [attacker on core X] traverse S^P
4     /* Now P should be the next to evict in L2, but still cached in L1 (i.e.
         AutoLocked)                                                        */
5     [victim on core Y] may or may not access [addr]
6     /* Accessing [addr] should evict P from the L2, and also from L1 due to
         inclusive cache                                                    */
7     [attacker on core X] fail = STREX val, [P]
8     return fail
```

points out that the L2 uses an LRU-based policy for evicting L2 cache lines belonging to the same core. Since ldrex happens strictly before traversing S^P, the L2 line where P is located will become LRU after line 3 since P and S^P are from the same core. The attacker then waits for the victim's action by spinning a loop a number of times. Whenever addr is accessed, P will be evicted from both L1 and L2, causing the strex on line 7 to fail.

7.6.3 Preparation Phase

During the preparation phase, S^{2}C generates L2 eviction sets for the target address utilizing ldrex/strex only. As mentioned in Section 7.5.2, Vila's algorithm (or similar techniques such as [336]) is widely used by cache attacks due to the weak assumptions it makes on the attacker — no knowledge about the address bits beyond the regular page offset bits is required.

However, we found that using Vila's algorithm (or any similar techniques based on pruning eviction set supersets) by replacing timing measurement with ldrex/strex is unfeasible due to AutoLock. To start, ldrex/strex cannot distinguish between L1 evictions caused by L1 cache contention and those caused by L2 evictions. Therefore, building the L2 eviction set superset should not use candidate addresses that are L1-congruent with the target address addr. Given addr is AutoLocked and hence can only be evicted from the L2 when the other 11 lines in the same L2 set are also AutoLocked, the traversal of an L2 eviction set superset S can evict addr only if at one point, all 12 L2-congruent addresses in S are cached in L1. This is clearly impossible: they compete for one single L1 set, meaning at most 8 addresses

can be AutoLocked with addr in the L2 set.[26] The outcome is that we can never identify an L2 eviction set superset, let alone reduce it to obtain the actual L2 eviction set.

Inspired by previous works [34, 56, 333], S^{2}C instead utilizes the reverse-engineering result of the L2 set index mapping and huge pages[27] to directly compute the minimal L2 eviction set. Since 11 out of 13 L2 set index bits are huge page offset bits, after allocating a huge page, the attacker can easily identify addresses within this huge page that share those 11 set index bits as the target address. Although the remaining two bits cannot be determined due to the unknown huge page number, we can easily determine whether two arbitrary addresses within the huge page share the same value for these two bits, because the huge page number is identical.

With this observation, our L2 eviction set generation works as follows. Given a target address, we allocate one huge page, and collect all $2^7 = 128$ addresses on this page that share the same regular page offset and address bits [19:16] as the target address. All these address bits except the L2 line offset bits are required to match the target address to achieve L2 congruence. Next, we group the 128 addresses into four groups, such that addresses within each group share the same two XOR-ed bits. Since the huge page number of these addresses, as well as the target address, is unknown, we cannot determine which address group has the exact same L2 set index as the target address, but it is guaranteed that one of these four address groups will be an L2 eviction set of the target address.

We leverage Algorithm 7.4 to identify which group is actually the L2 eviction set. For each group, we choose 12 addresses from the total 32 addresses as the candidate minimal eviction set, and apply eviction set splitting so that those 12 addresses are distributed to two sibling L1 sets evenly. As for Algorithm 7.4, one address is chosen as the address for ldrex/strex, and the remaining are traversed after ldrex. Unlike the attack, when testing eviction sets, the attacker must trigger access to the target address (which could be done by a victim-provided API call that is known to access the target address). Only when the tested address group is L2-congruent with the target address will the strex fail.

[26] Even if the attacker owns multiple cores to traverse the candidate set separately, it is still infeasible for a candidate L2 eviction set S to evict addr: the L2-congruent addresses constitute only around $1/2^6 \approx 1.6\%$ of S according to Figure 7.5. Therefore, it is almost impossible to see 12 L2-congruent addresses cached in the L1.

[27] Although allocating huge pages requires only user-level privilege, it may not be available when the attacker is limited to a sandboxed environment, e.g., browsers. In those scenarios, the attacker cannot proactively allocate huge pages via system calls. Instead, the attacker must rely on the runtime allocating huge pages automatically, e.g., via OS features such as Transparent Huge Pages in Linux.

7.7 MONITORING MULTIPLE CACHE SETS

We now generalize the protocol from Section 7.6 to enable the attacker to simultaneously monitor multiple victim L2 cache sets. The attacker model is otherwise the same as that presented in Section 7.6.1: the attacker is unprivileged, runs on a single thread, etc. As with Section 7.6, we do not require modifications in the victim's code.

The challenge here is that ldrex/strex only allow the single-threaded attacker to monitor evictions on a single address/cache line at a time. To work around this limitation, the insight is to view ldrex/strex *as a general-purpose single-bit communication channel* that can communicate the result of an arbitrary 1-bit function computed in micro-architectural space. With this in mind, we construct a micro-architectural weird circuit (μWC) [321] that computes the logical-OR of whether the victim accessed at least one of several attacker-specified L2 sets—and communicates the 1-bit result of this logical-OR through ldrex/strex to the attacker's architectural state, namely the result of strex. We remark that while weird circuit constructions are not the main focus of the paper, our weird circuit is relatively simple conceptually compared to the original proposals in [321] (which focuses specifically on speculative execution), and may be of independent interest. We also remark that while we compute logical-OR due its useful semantics, other functions are of course possible (e.g., one to compute hops in a binary search to *localize* which victim cache set was accessed). We leave such investigations to future work.

7.7.1 Construction

We explain the μWC assuming the attacker wishes to monitor two victim target addresses a and b located at L2 sets A and B for simplicity, and generalize to monitoring N L2 sets $A_0, A_1, \ldots, A_{N-1}$ at the end.

To start, the attacker uses the procedure from Section 7.6.3 to construct minimal L2 eviction sets for target addresses a and b, and also a third eviction set for another address x allocated by the attacker itself. The attacker ensures that x is located at an L2 set X different from A and B, and X is not used by the victim. The attacker further builds a linked list $Pa \rightarrow Pb \rightarrow Px$, where Pa, Pb, Px are addresses randomly chosen from the eviction sets generated for a, b, and x, respectively.

The attacker is interested in whether the victim accesses a line in sets A or B. At a high level, it can infer activity on these sets by observing the eviction of Px with ldrex/strex, and using out-of-order execution to create a race between a) traversing the linked list $Pa \rightarrow Pb \rightarrow Px$ and b) accessing x. Depending on the outcome of this race, Px could be

evicted by x when it is accessed by the attacker's strex, which indicates whether the victim displaced data in sets A or B.

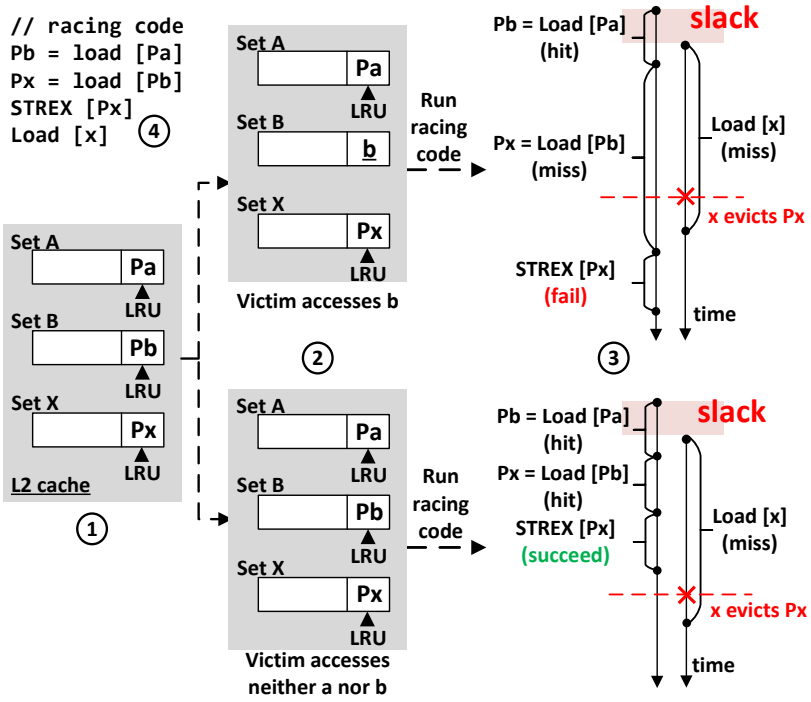


Figure 7.6: An example of how S^{2}C constructs a μWC to monitor victim addresses (called a and b) that map to L2 cache sets A and B, respectively. See text in Section 7.7.1 for a detailed walkthrough.

In more detail: In the "Prime" step (Figure 7.6 ①), the attacker uses the techniques in Section 7.6.2 to bring Pa, Pb, Px into the cache, position each in the LRU position of each respective L2 set and ensure that each is evictable (using eviction set splitting; Section 7.5.3). It further evicts x from all levels of cache and monitors Px using ldrex. It then waits for the victim to make an access (see Figure 7.6 ②), same as in Section 7.6. In the "Probe" step, the attacker begins the race (Figure 7.6 ③) by simultaneously a) traversing $Pa \rightarrow Pb \rightarrow Px$ and b) making an access to x. x will always result in a miss. It accesses Pa, Pb with normal loads and accesses Px using strex[28]. There are two possible outcomes, depending on the victim's access pattern:

- If the victim accessed neither A nor B (Figure 7.6 ②-③, bottom), traversing $Pa \rightarrow Pb \rightarrow Px$ will result in all hits and complete *before* x. Since Px will still be cached, strex returns 0 (success).

[28]This racing code is actually a "racing gadget" from a parallel work HackyRacer [317], which amplifies small timing difference to make it detectable by low-resolution timers. Both our uWC and HackyRacer leverage out-of-order execution to create a race condition between two instruction paths.

- Otherwise (Figure 7.6 ②-③, top), traversing $Pa \rightarrow Pb \rightarrow Px$ will result in at least one miss and complete *after* x. Since Px will be evicted (by x), strex returns 1 (fail).

Monitoring N victim addresses. The above generalizes to monitoring N victim addresses $a_0, a_1, \ldots, a_{N-1}$ located at different L2 sets using N eviction sets, a separate set X (which serves the same function as before) and a linked list that traverses $P_{A_0} \rightarrow P_{A_1} \rightarrow \cdots \rightarrow P_{A_{N-1}} \rightarrow Px$.

Implementation considerations. Our implementation matches closely with the above description. That said, we needed to place load[x] after the strex in program order (Figure 7.6 ④). This is because strex is not performed until it reaches the head of the reorder buffer, and thus would not execute until load[x] is completed if the load was placed before it. Another important factor is that, as N increases, it is necessary to delay the load to x to ensure that traversing the linked list is faster than accessing x, if all linked list accesses are hits. This delay, called 'slack' in Figure 7.6, is implemented by spinning in a loop a specific number of times.

7.8 EVALUATION

Our evaluation is performed on an M1 Mac Mini, running Asahi Linux (Linux version 6.1.0). We cannot conduct the full evaluation on MacOS since the M1 version of MacOS does not support huge pages. Following the attacker model described in Section 7.6.1, the S^{2}C attacker is single-threaded and does not use timers.

7.8.1 Monitoring Multiple Cache Sets

We now evaluate S^{2}C's ability to simultaneously monitor multiple cache sets. Here the victim may access a set of addresses that map to different L2 cache sets, and the attacker uses the method described in Section 7.7 to detect accesses to those sets. Ideally, any victim access to those sets should result in a 100% strex fail rate; if the victim accesses none of the aforementioned sets, we expect a 0% fail rate, i.e., the difference in the strex fail rate should be close to 100%.

Figure 7.7 shows the probability that the attacker's strex fails when the victim 1) accesses one target address, 2) performs no memory accesses, and 3) accesses addresses belonging to other L2 sets. When monitoring less than 12 addresses, we can always find a suitable slack so that the difference in the strex fail rates between the victim accessing the target

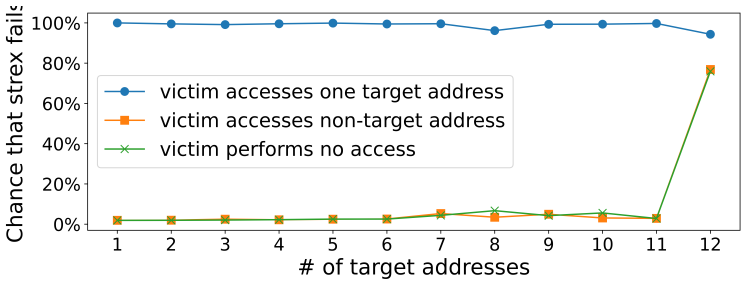


Figure 7.7: The chance that strex fails when the victim accesses different addresses, given S^{2}C monitoring multiple target addresses.

address versus accessing no/other addresses is close to 100% (the difference is at least 93%). This showcases S^{2}C's effectiveness in monitoring up to 11 L2 sets. For more than 12 target addresses, we cannot find a slack value that achieves a favorable difference in the strex fail rate. This results in significant false positives, as shown in Figure 7.7.

7.8.2 Covert Channel

S^{2}C, like previous cache side channels, can be used to establish cross-process covert channels. However, as the strex output only conveys 1-bit of information, the channel only transmits 1-bit at a time. Another challenge in building covert channels with S^{2}C is that it requires synchronization between two processes without relying on timing measurements, which are used by prior cache attacks [34, 35, 57, 337].

The transmission of each bit consists of two stages: the standby stage and the transmission stage. Both stages employ Algorithm 7.4, but in opposite directions. The sender and the receiver each designate an address located in different L2 sets, and both generate an L2 eviction set for the other party's address. The address owned by the sender, dubbed $\texttt{addr}_{\texttt{trans}}$, is used in the transmission stage. The receiver prepares an exclusive address (to be monitored by strex) which is L2-congruent to $\texttt{addr}_{\texttt{trans}}$, and uses it along with the remaining eviction set to detect the sender's accesses to $\texttt{addr}_{\texttt{trans}}$. To transmit 1 bit, the sender accesses $\texttt{addr}_{\texttt{trans}}$ (and otherwise makes no access). The receiver pauses by iterating over a busy-waiting loop eight thousand times to give the sender sufficient time for accessing the targeted L2 set, and then checks whether $\texttt{addr}_{\texttt{trans}}$ was accessed using strex.

During the standby stage, the sender waits for a signal from the receiver indicating that the receiver is ready. Likewise, this is achieved by the sender monitoring the receiver's access to the address $\texttt{addr}_{\texttt{ready}}$. Different from the transmission stage, the receiver will always access $\texttt{addr}_{\texttt{ready}}$ to indicate readiness. The sender repeatedly calls Algorithm 7.4 until an access to $\texttt{addr}_{\texttt{ready}}$ is observed, at which point the sender can proceed to the transmission stage.

After tuning the covert channel code, we achieve a bandwidth of approximately 185 Kbit per second with a 98.5% accuracy. The small error rate is due to the sender's access to $\texttt{addr}_{\texttt{trans}}$ overlapping with the receiver's ldrex/strex, creating a *blindspot* that has been studied in previous works such as Prime+Scope [35]. Since our synchronization also relies on S^{2}C instead of a precise timing source, the blindspot can also cause de-synchronization. For example, if the receiver sends a ready signal that is missed by the sender, the sender will get stuck waiting. In the next round, the receiver will interpret that as the sender transmitting 0, at which point it will send another ready signal (which, hopefully, the receiver will now see). This de-synchronization has a 0.2% chance of affecting each bit transmission.

Comparison with Prime+Probe We also implement a similar 1-bit covert channel using Prime+Probe, after enabling the cycle count register from the kernel space. For the Prime+Probe covert channel, the receiver owns the address $\texttt{addr}_{\texttt{trans}}$, and the sender controls the L2 eviction set of $\texttt{addr}_{\texttt{trans}}$. To transmit a bit, the receiver first accesses $\texttt{addr}_{\texttt{trans}}$, and the sender traverses the eviction set when the transmitted bit is 1 (otherwise does not traverse the eviction set). The receiver pauses for a certain duration to allow the eviction set traversal to finish, and then times the access to $\texttt{addr}_{\texttt{trans}}$ to determine whether it has been evicted. Unlike S^{2}C which relies on cache contention during the standby stage for synchronization, the Prime+Probe covert channel utilizes the timer directly for synchronization. This not only reduces the activities during each transmission, leading to a significant increase in bandwidth, but also ensures precise synchronization between the attacker and the sender, eliminating the problem of de-synchronization. This 1-bit Prime+Probe covert channel is capable of transmitting approximately 382 Kbit per second with over 99% accuracy.

7.8.3 Attacking T-table AES

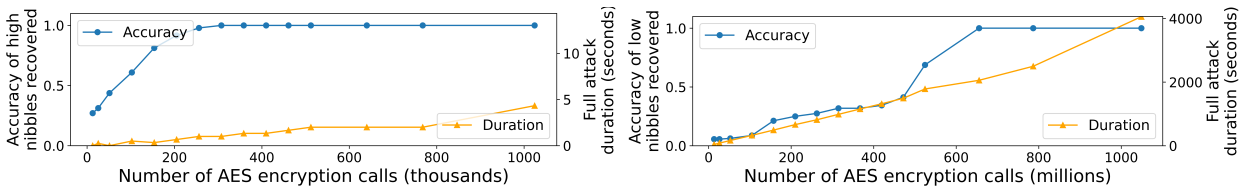
We now demonstrate how S^{2}C can be used to perform full key extraction on T-table AES, based on the classical chosen-plaintext attack due to Osvik et. al [5].

T-table AES is a popular benchmark for evaluating cache side-channel attacks because it creates secret key-dependent T-table access patterns, which can be used to infer information about the secret key [5, 35, 57, 59]. We adapt the attack by Osvik et. al [5], which proceeds in two steps: targeting high-key nibbles and low-key nibbles of key bytes, respectively. Specifically, the first round of T-table AES performs bitwise XOR between the plaintext and the private key, and the output bytes are used directly as indices for T-table lookups. To exploit this behavior, the attacker repeatedly interacts with the victim. In each interaction, the attacker tries to guess the high nibble of one key byte by creating a specific plaintext.

When the guess is correct, the XOR produces an index value $< 2^4$, thereby creating an observable access to a specific cache line where the looked-up T-table is based. While this cache line may also be accessed by other rounds, it is guaranteed to be accessed when the high nibble from the plaintext byte and the target key byte match. Repeating for each key byte, this allows the attacker to recover the high nibble of each key byte.

The low nibbles can be retrieved similarly by exploiting the second round. In the second round, every T-table access index depends on four distinct key bytes, instead of one key byte in the first round. Since the high key nibbles are known, the attacker can guess possible values for the four low nibbles corresponding to the four key bytes used by the T-table index. It validates these guesses in the same way as before: by submitting plaintexts and monitoring whether the T-table target cache line is accessed during the AES operation. This process can be repeated to learn the values of nibbles in other sets of four key bytes.

Here we show that S^{2}C can leverage the above attack technique to infer the full 16-byte AES key. Our experiment uses OpenSSL's T-table AES implementation [338]. The victim process maintains a secret key and exposes an AES encryption/decryption API to the attacker. The attacker process performs the chosen-plaintext attack and leverages S^{2}C (instead of Prime+Probe as in the original attack [5]) to monitor accesses to targeted cache lines.



(a) First-round attack result.

(b) Second-round attack result.

Figure 7.8: Accuracy of recovered high nibbles and the attack duration of both the first-round and the second-round attack, as we increase the number of total AES calls for the full first-/second-round attack. The kink in the accuracy plot in (b) is due to measurement noise.

Figure 7.8 shows the proportion of nibbles that can be accurately recovered in the first-round and the second-round attack, as we increase the number of plaintext samples used for testing each possible nibble value (or four nibble values in the second-round attack). Each data is averaged over ten independent attacks, where each attack performs one first-round and one second-round attack to leak one randomly-generated private key. The figure also shows the attack latency as a function of the number of AES calls. The high nibbles can be recovered with 100% accuracy with around 300K encryption calls, which takes around 1 second. However, to recover the low nibbles with 100% accuracy, we need around 700M encryption calls, which is roughly 40 minutes. Notice that recovering low nibbles requires significantly more time than high nibbles because, in the second-round attack, the attacker

must guess four nibbles together, and measuring a cache access only indicates if all four nibbles are guessed correctly.[29] This difference is mentioned by the original attack [5].

Since most recent cache attacks mostly only evaluate the first-round attack [35, 57, 59, 60], we can compare the effectiveness of our version of the first-round attack with them. For instance, Flush+Flush [57] shows 250 encryptions are required for recovering all high nibbles, whereas Prime+Scope [35] reports a similar number (around 200). Therefore, S^{2}C demonstrates a similar capability in retrieving the AES key bytes albeit using a different cache measurement technique and targeting Apple CPUs.

7.9 DISCUSSION

7.9.1 Impact on Other Processors

Load-Linked/Store-Conditional is an ISA-agnostic primitive for performing efficient mutual exclusion and synchronization. We now discuss its support in other existing ISAs, and where we believe those implementations may enable S^{2}C-based attacks.

CISC/x86. CISC architectures such as x86 do not support LL/SC as part of their ISAs. We have further performed experiments to check whether native x86 hardware atomics (`cmpxchg`) are implemented using LL/SC-like microcode under the hood and haven't found evidence to support this theory.

ARM. In 2022, Apple introduced the M2 which continues using the same ARMv8-A ISA as M1. We are able to reproduce the same attack primitive that we explained in Section 7.4 on M2 CPUs. Future work is needed to reverse engineer the M2's cache configuration (akin to that in Section 7.5) to enable S^{2}C-based attacks. It is possible that other ARM CPUs follow the LL/SC semantics specified in ARM's manual, making them vulnerable to the basic S^{2}C mechanism. But more reverse engineering is needed to confirm on which parts this holds.

RISC-V. RISC-V is a relatively new and rapidly growing RISC architecture. Different from ARM, RISC-V recognizes that allowing store-conditional to fail on cache evictions might impede LL/SC progress indefinitely. Therefore, the official RISC-V manual suggests that

[29] To illustrate, assume N samples are needed to verify each guess. The first-round attack, therefore, requires 16 (16 high nibbles) * 16 (16 possible values for each high nibble) * N = 256 * N AES calls, whereas the second-round attack requires 4 (4 four-nibble combinations) * 16^4 (guess values for 4 low nibbles) * N = 262,144 * N AES calls.

"reservations (exclusive addresses) are tracked independently of evictions from any shared cache" [339], which, if implemented in real life, can effectively mitigate S^2C.

MIPS. MIPS is an older RISC ISA. The MIPS manual suggests that the base observation in S^2C may apply. We quote: "...load or store may cause a cache eviction between the LL and SC that results in SC failure" [340].

7.9.2 Mitigations

Restricting access to the attack primitive is a straightforward yet effective method for mitigating many side-channel attacks [134, 135, 138, 319, 341]. Because LL/SC is only leveraged by the attacker and not required to be used by the victim, one would seemingly need to summarily disallow LL/SC. However, LL/SC are basic instructions that are impractical to disable. Given this, we propose several mitigation strategies.

Changing Exclusive Monitor Implementation. Because the exclusive monitor only tracks addresses in the L1 cache, we speculate that the exclusive monitor is implemented by piggybacking on the cache coherence protocol. The implementation may use a dedicated coherence state for exclusive addresses accessed by LL instructions. The behavior of SC aligns with this design: an SC succeeds when the target address is in the new state; and like normal stores, an SC invalidates the address in other private L1 caches, forcing other threads to lose their exclusiveness to that address. To patch this vulnerable implementation, the safe exclusive monitor should keep track of exclusive addresses independent of the location of the addresses in the cache hierarchy.

Software Mitigations for Cache Side-Channels. Mitigating cache side-channel attacks have been an important topic in side-channel research. Software developers generally use constant-time programming to eliminate secret-dependent memory access patterns [30, 106, 107, 114, 123, 125, 273]. Constant-time programming has been applied to many modern cryptographic libraries for protecting critical assets such as keys [342, 343]. However, many cryptographic libraries, as well as general-purpose programming still opt out of the constant-time property for better performance [118].

Hardware Mitigations for Cache Side-Channels. Researchers have also proposed hardware-based mitigations for defeating cache side channels, most of which are based on cache partitioning or randomization. Cache partitioning splits the cache into partitions,

and each partition can be used by a security domain without any interference from other domains [147, 148, 150, 153]. Cache randomization introduces randomness into the cache set mappings, hindering the attacker from creating cache contention with victim lines [160, 161, 162, 163]. Since LL/SC is a contention-based attack, both of these approaches would apply in principle.

Finally, we admit that disabling huge pages does mitigate our proposed attack on Apple M1 by preventing the eviction set generation. For this reason, the current M1's MacOS is immune to S^2C in practice. However, we note that previous MacOS versions on Intel CPUs do support huge pages. Given M1 CPU natively supports huge pages, future MacOS versions on Apple CPUs may re-adopt huge pages, making it susceptible to S^2C.

7.10 CONCLUSION

This chapter proposes Synchronization Storage Channels (S^2C), the first timer-less cross-core cache attack that exploits Load-Linked/Store-Conditional (LL/SC) instructions on the Apple M1. The key insight of S^2C is that the implementation of LL/SC enables direct observation of the cache activities, i.e., the L1 evictions of the address accessed by LL/SC. With several novel techniques to circumvent the limitation in the single address observation by LL/SC, S^2C achieves similar attack capability as prior contention-based cache attacks but without the dependence on timing measurements.

CHAPTER 8: CONCLUSIONS

Micro-architectural side-channel attacks have emerged as a significant threat to computer security as people discover more attack instances, which demonstrate how critical information can be leaked due to the inherent complexity of modern processors. This thesis undertook a thorough investigation of the current attacks and mitigation approaches, and highlights two essential principles in designing mitigation solutions: strong and comprehensive security guarantees, combined with low-performance overhead. Unfortunately, existing mitigation schemes typically prioritize one of these two goals at the expense of the other.

The primary contribution of this thesis is the proposed mitigation frameworks that offer comprehensive and efficient protections against generic micro-architectural side-channel attacks and the recent speculative execution attacks. The underlying methodology behind the proposed mitigations is that, side-channel-free can be provably achieved once the secret information and the side channels are precisely identified, and any leakage of secret information over the side channels is properly prevented. Following this key insight, we embed the side-channel security specification into existing ISAs into the data-oblivious ISA, which enables provable side-channel-free programming while maintaining flexibility for designing hardware optimizations to enhance performance. In response to the surge of speculative execution attacks, we present Speculative Taint Tracking and Speculative Data-Oblivious Execution, which covers all types of speculative side channels with modest performance overhead.

In addition, this thesis addressed the limitation of the point defenses employed in existing systems and presented novel micro-architectural side-channel attacks. Although these attacks may be mitigated with future point defenses, our central argument is that micro-architectural attacks and point defenses built for specific attack variants perpetuate a cat-and-mouse game in which the attacker inevitably finds new avenues to exploit. Therefore, future side-channel-proof systems must adopt protection schemes with comprehensive and robust security properties, as exemplified by the solutions proposed in this thesis.

As hardware complexity continues to advance, it is inevitable that new micro-architectural side-channel attacks will undoubtedly continue to surprise people in the near future. Fortunately, in recent years, not only the researchers but also major hardware vendors and software companies have paid more attention to this problem. However, this only marks the beginning of our battle with micro-architectural side-channel attacks. Looking ahead, we anticipate several critical challenges that must be overcome to secure victory in this ongoing battle.

First, existing side-channel mitigation solutions are typically evaluated in an ideal, ex-

perimental setting. However, the development of real-world hardware is a rigorous and sophisticated process consisting of many stages. Evaluating side-channel security properties will likely be integrated into multiple stages, such as in the early stages to prove the absence of certain side channels, and in the post-design stages to verify that the implementation adheres to the prescribed security specifications. In addition, there is a significant gap between the evaluation methodologies employed in the research area and the industrial manufacturing process. Therefore, existing industrial tool chains must be extended to properly evaluate the security property and the performance overhead of proposed mitigation schemes.

Second, the trend of modern systems is shifting towards heterogeneity, with the emergence of new hardware components such as GPU and domain-specific accelerators, alongside new computing paradigms like machine learning and virtual/mixed reality. Consequently, future research must address the side-channel vulnerabilities arising in heterogeneous systems. This entails not only studying the new side channels enabled by non-CPU processing elements, but also comprehending how the synergistic interaction between different processing elements may enable new information leakage. Developing comprehensive side-channel protection in heterogeneous systems is likely to be more complicated and requires collaborative efforts among designers responsible for different system components.

Lastly, side channels are not limited to processors or hardware alone. Modern software stacks incorporate various optimizations, such as caching, that may expose side-channel vulnerabilities akin to micro-architectural side channels. Future research should examine the security implication of those optimizations, and develop proper protection techniques if any side-channel threat is identified.

REFERENCES

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*. Springer, 1996, pp. 104–113.

[2] D. PAGE, "Theoretical use of cache memory as a cryptanalytic side-channel," *Cryptology ePrint Archive 2002/169*, 2002.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom et al., "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[4] 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," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 2006, pp. 1–20.

[6] C. Percival, "Cache missing for fun and profit," 2005.

[7] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 490–505.

[8] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 990–1003.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical." in *WOOT*, 2017, pp. 11–11.

[10] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security'15)*, 2015, pp. 897–912.

[11] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1406–1418.

[12] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security'19)*, 2019, pp. 639–656.

[13] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[14] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[15] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security'14)*, 2014, pp. 719–732.

[16] B. Gras, K. Razavi, H. Bos, C. Giuffrida et al., "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks." in *USENIX Security Symposium*, vol. 216, 2018, pp. 955–972.

[17] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 2006, pp. 225–242.

[18] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[19] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security'17)*, 2017, pp. 557–574.

[20] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 693–707.

[21] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 361–374.

[22] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *IEEE S&P'15*, 2015.

[23] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1491–1505.

[24] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 131–145.

[25] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, vol. 2, pp. 33–50, 2018.

[26] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 346–359.

[27] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams et al., "The landscape of parallel computing research: A view from berkeley," 2006.

[28] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[29] S. Gueron and M. E. Kounavis, "Intel advanced encryption standard (aes) new instructions set," *Intel Corporation*, 2010. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[30] J. Yu, L. Hsiung, M. El'Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high-performance computing," in *The Network and Distributed System Security Symposium (NDSS)*, 2019.

[31] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[32] 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.

[33] J. Yu, T. Jaeger, and C. W. Fletcher, "All your pc are belong to us: Exploiting non-control-transfer instruction btb updates for dynamic pc extraction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589100

[34] 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*. IEEE, 2015, pp. 605–622.

[35] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+scope: Overcoming the observer effect for high-precision cache contention attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2906–2920.

[36] J. Yu, A. D. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization storage channels (s2c): Timer-less cache side-channel attacks on the apple m1 via hardware synchronization instructions," in *32nd USENIX Security Symposium (USENIX Security'23)*, 2023, to appear.

[37] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[38] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[39] M. Johnson and M. Johnson, *Superscalar microprocessor design*. Prentice Hall Englewood Cliffs, New Jersey, 1991, vol. 77.

[40] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *SIGPLAN Not.*, vol. 31, no. 9, pp. 138–147, Sep. 1996. [Online]. Available: http://doi.acm.org/10.1145/248209.237173

[41] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[42] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek et al., "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[43] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 1999, pp. 388–397.

[44] T. Kim and Y. Shin, "Thermalbleed: A practical thermal side-channel attack," *IEEE Access*, vol. 10, pp. 25 718–25 731, 2022.

[45] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[46] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. Springer, 2011, pp. 355–371.

[47] D. X. Song, D. A. Wagner, X. Tian et al., "Timing analysis of keystrokes and timing attacks on ssh." in *USENIX Security Symposium*, vol. 2001, 2001.

[48] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000, pp. 25–32.

[49] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the aes," in *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 2006, pp. 271–286.

[50] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86," in *31st USENIX Security Symposium (USENIX Security'22)*, 2022, pp. 679–697.

[51] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani, "On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme," in *Advances in Cryptology–CRYPTO 2008: 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings 28*. Springer, 2008, pp. 203–220.

[52] M. Zhao and G. E. Suh, "Fpga-based remote power side-channel attacks," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 229–244.

[53] Amazon, "Amazon web services," https://aws.amazon.com, 2023, accessed: May 1, 2023.

[54] Microsoft, "Microsoft azure," https://azure.microsoft.com, 2023, accessed: May 1, 2023.

[55] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[56] 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 *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 591–604.

[57] 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*. Springer, 2016, pp. 279–299.

[58] 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 (USENIX Security'20)*, 2020, pp. 1967–1984.

[59] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision l3 cache attack using intel tsx," in *26th USENIX Security Symposium (USENIX Security'17)*, 2017, pp. 51–67.

[60] S. Kim, M. Han, and W. Baek, "Dprime+dabort: A high-precision and timer-free directory-based side-channel attack in non-inclusive cache hierarchies using intel tsx," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 67–81.

[61] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 321–347, 2020.

[62] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal attack: Leaking control-flow in sgx via the cpu frontend," in *30th USENIX Security Symposium (USENIX Security'21)*, 2021, pp. 663–680.

[63] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, "Squip: Exploiting the scheduler queue contention side channel," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 468–484.

[64] 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.

[65] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *Information, Security and Cryptology–ICISC 2009: 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers 12*. Springer, 2010, pp. 176–192.

[66] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

[67] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[68] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[69] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, pp. 99–112, 2017.

[70] J. Kim, H. Jang, H. Lee, S. Lee, and J. Kim, "Uc-check: Characterizing micro-operation caches in x86 processors and implications in security and performance," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 550–564.

[71] 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, pp. 538–570, 2019.

[72] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[73] 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 *Proceedings of the 16th ACM Conference on Computer and communications security*, 2009, pp. 199–212.

[74] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[75] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical." in *USENIX Security Symposium*, 2021, pp. 645–662.

[76] J. Wan, Y. Bi, Z. Zhou, and Z. Li, "Meshup: Stateless cache side-channel attack on cpu mesh," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1506–1524.

[77] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around:side-channel attacks and mitigations on mesh interconnects," in *31st USENIX Security Symposium (USENIX Security'22)*, 2022, pp. 2857–2874.

[78] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 353–364.

[79] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[80] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology-ICISC 2005: 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers 8*. Springer, 2006, pp. 156–168.

[81] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM Workshop on Computer security architecture*, 2007, pp. 11–18.

[82] O. Acıiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl," in *Topics in Cryptology–CT-RSA 2008: The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*. Springer, 2008, pp. 256–273.

[83] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*. Springer, 2010, pp. 110–124.

[84] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-leak forwarding: leaking data on meltdown-resistant cpus (updated and extended version)," *arXiv preprint arXiv:1905.05725*, 2019.

[85] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[86] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, vol. abs/1802.09085. IEEE, 2019. [Online]. Available: http://arxiv.org/abs/1802.09085 pp. 142–157.

[87] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution." in *ISCA*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3470496.3527429 pp. 685–698.

[88] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–36, 2021.

[89] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "Branchspec: Information leakage attacks exploiting speculative branch instruction executions," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 529–536.

[90] Bitdefender, "SWAPGS Attack," Bitdefender.com, 2019.

[91] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 667–682.

[92] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer." in *WOOT@ USENIX Security Symposium*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[93] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243761 pp. 2109–2122.

[94] Google Project-Zero mailing list, "Speculative execution, variant 4: Speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[95] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 54–72.

[96] 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.

[97] 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.

[98] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *USENIX Security*, Aug. 2022, intel Bounty Reward. [Online]. Available: Paper=http://download.vusec.net/papers/bhi-spectre-bhb_sec22.pdfWeb=https://www.vusec.net/projects/bhi-spectre-bhbCode=https://github.com/vusec/bhi-spectre-bhb

[99] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[100] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[101] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[102] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading kernel writes from user space," *arXiv preprint arXiv:1905.12701*, 2019.

[103] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 339–354.

[104] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1852–1867.

[105] I. Corporation, "Intel TSX: Asynchronous Abort," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-tsx-asynchronous-abort.html, [Online; accessed May 29, 2023].

[106] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 279–296.

[107] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security'15)*, 2015, pp. 431–446.

[108] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 493–504.

[109] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *2013 IEEE 26th Computer Security Foundations Symposium*. IEEE, 2013, pp. 51–65.

[110] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the nineteenth annual ACM Symposium on Theory of computing*, 1987, pp. 182–194.

[111] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[112] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," *Journal of the ACM (JACM)*, vol. 65, no. 4, pp. 1–26, 2018.

[113] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," in *Theory of Cryptography: 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II 13*. Springer, 2016, pp. 145–174.

[114] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," *Cryptology ePrint Archive*, 2017.

[115] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[116] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC'17)*, 2017, pp. 299–312.

[117] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 317–328.

[118] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""they're not that hard to mitigate": What cryptographic library developers think about timing attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 632–649.

[119] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer, 2006, pp. 207–228.

[120] D. J. Bernstein, "The poly1305-aes message-authentication code," in *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers 12*. Springer, 2005, pp. 32–49.

[121] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology-ICISC 2005: 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers 8*. Springer, 2006, pp. 156–168.

[122] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: functional encryption using intel sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 765–782.

[123] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors." in *USENIX Security Symposium*, vol. 16, 2016, pp. 10–12.

[124] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1211–1228.

[125] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform." in *NSDI*, vol. 17, 2017, pp. 283–298.

[126] S. Eskandarian and M. Zaharia, "An oblivious general-purpose sql database for the cloud," *CoRR abs/1710.00458*, vol. 1710, 2017.

[127] S. Tople and P. Saxena, "On the trade-offs in oblivious execution techniques," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 25–47.

[128] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx." in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[129] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 45–60.

[130] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 913–926.

[131] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, "High-assurance cryptography in the spectre era," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1884–1901.

[132] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[133] P. Li, D. Gao, and M. K. Reiter, "Stopwatch: a cloud architecture for timing channel mitigation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 2, pp. 1–28, 2014.

[134] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *25th USENIX Security Symposium (USENIX Security'16)*, 2016, pp. 463–480.

[135] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 118–129.

[136] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, 2011, pp. 41–46.

[137] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 570–581.

[138] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(m) wait for it: Bridging the gap between microarchitectural and architectural side channels," in *USENIX Security*, 2023.

[139] M. Schwarz, M. Lipp, and D. Gruss, "Javascript zero: Real javascript and zero side-channel attacks." in *The Network and Distributed System Security Symposium (NDSS)*, vol. 18, 2018, p. 12.

[140] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security'16)*, 2016, pp. 549–564.

[141] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses," in *30th USENIX Security Symposium (USENIX Security'21)*, 2021, pp. 2863–2880.

[142] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," Cryptology ePrint Archive, Report 2016/613, 2016, https://eprint.iacr.org/2016/613.

[143] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *23rd USENIX Security Symposium (USENIX Security'14)*, 2014, pp. 687–702.

[144] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, 2007, pp. 312–320.

[145] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 341–353.

[146] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 99–110.

[147] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 83–93.

[148] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–21, 2012.

[149] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 57–68.

[150] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[151] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation." in *USENIX Security Symposium*, 2016, pp. 857–874.

[152] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 451–468.

[153] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[154] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp) defending against cache-based side channel atacks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 347–360, 2017.

[155] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud." in *USENIX Security Symposium*, 2012, pp. 189–204.

[156] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 332–345.

[157] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 466–477.

[158] 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 *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1267–1272.

[159] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 494–505.

[160] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[161] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 360–371.

[162] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization." in *USENIX Security Symposium*, 2019, pp. 675–692.

[163] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization." in *The Network and Distributed System Security Symposium (NDSS)*, 2020.

[164] T. Unterluggauer, A. Harris, S. Constable, F. Liu, and C. Rozas, "Chameleon cache: Approximating fully associative caches with random replacement to prevent contention-based cache attacks," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 13–24.

[165] L. Zhao, P. Li, R. Hou, M. C. Huang, X. Qian, L. Zhang, and D. Meng, "Hybp: Hybrid isolation-randomization secure branch predictor," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2022, pp. 346–359.

[166] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," *SIGPLAN Not.*, vol. 50, no. 4, pp. 87–101, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775054.2694385

[167] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the seventh ACM Workshop on Scalable trusted computing*, 2012, pp. 3–8.

[168] G. Hu, Z. He, and R. B. Lee, "Sok: Hardware defenses against speculative execution attacks," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021, pp. 108–120.

[169] Intel, "Indirect branch predictor barrier," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html, 2018.

[170] Intel, "Indirect branch restricted speculation," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, 2018.

[171] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 395–410.

[172] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer, 2019, pp. 279–299.

[173] J. Nider, M. Rapoport, and J. Bottomley, "Address space isolation in the linux kernel," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019, pp. 194–194.

[174] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: process separation for web sites within the browser," in *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019, pp. 1661–1678.

[175] K. Sun, R. Branco, and K. Hu, "A new memory type against speculative side channel attacks," *Intel—STrategic Offensive Research & Mitigations (STORM)*, 2019.

[176] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

[177] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[178] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 151–164.

[179] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "Context: Leakage-free transient execution," *arXiv preprint arXiv:1905.09100*, 2019.

[180] 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.

[181] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability." in *USENIX Security Symposium*, 2021, pp. 1397–1414.

[182] C. Carruth, "Speculative Load Hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, [Online; accessed May 29, 2023].

[183] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[184] 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.

[185] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 132–144.

[186] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[187] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 723–735.

[188] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 42–56.

[189] H. Omar and O. Khan, "Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 111–122.

[190] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[191] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 2006, pp. 225–242.

[192] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *Information, Security and Cryptology–ICISC 2009: 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers 12*. Springer, 2010, pp. 176–192.

[193] 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," CoRR'18.

[194] "Speculative execution side channel mitigations," https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, revision 1.0, January 2018.

[195] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic, "Boomv2: an open-source out-of-order risc-v core," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[196] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. V. Lokam, E. Shi, and V. Goyal, "Hop: Hardware makes obfuscation practical." in *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[197] "T-table AES (OpenSSL)," https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c.

[198] "Bitslice AES (Bitcoin)," https://github.com/bitcoin-core/ctaes.

[199] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*. Wiley & Sons, 2007.

[200] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Springer, 2017, pp. 69–90.

[201] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[202] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.

[203] Intel, "Intel Software Guard Extensions Programming Reference," software.intel.com/sites/default/files/329298-001.pdf, 2013.

[204] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2435–2450.

[205] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 333–346.

[206] M. Backes and B. Pfitzmann, "Computational probabilistic noninterference," *International Journal of Information Security*, 2004.

[207] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[208] S. Gueron, "Efficient software implementations of modular exponentiation," *Journal of Cryptographic Engineering*, vol. 2, pp. 31–43, 2012.

[209] Intel, "Intel Software Guard Extensions Software Development Kit," https://software.intel.com/en-us/sgx-sdk.

[210] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 216–225.

[211] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 212–, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/1028176.1006719

[212] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[213] K. M. Lepak and M. H. Lipasti, "Silent stores for free," in *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 22–31.

[214] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 482–493, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273440.1250722

[215] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, pp. 85–96, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1037947.1024404

[216] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

[217] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[218] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 109–120, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/2528521.1508258

[219] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 359–376.

[220] D. Darais, I. Sweet, C. Liu, and M. Hicks, "A language for probabilistically oblivious computation," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–31, 2019.

[221] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 215–226.

[222] H. Cook, K. Asanovi, and D. A. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," 2009.

[223] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

[224] G. B. Bell and M. H. Lipasti, "Deconstructing commit," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2004, pp. 68–77.

[225] V. Fischer, "Random number generators for cryptography (design and evaluation," https://summerschool-croatia.cs.ru.nl/2014/slides/Random%20Number%20Generators%20for%20Cryptography.pdf.

[226] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," *Cryptology ePrint Archive*, 2018.

[227] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 850–861.

[228] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225.

[229] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."

[230] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 335–344.

[231] "Open cores," https://opencores.org/.

[232] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 377–394.

[233] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013, pp. 207–218.

[234] "Bitonic sort," https://en.wikipedia.org/wiki/Bitonic_sorter.

[235] J. Doerner, D. Evans, and A. Shelat, "Secure stable matching at scale," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1602–1613.

[236] T. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Cache-oblivious and data-oblivious sorting and applications," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018, pp. 2201–2220.

[237] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 411–428.

[238] S. Zahur and D. Evans, "Circuit structures for improving efficiency of security and privacy tools," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 493–507.

[239] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 69–76.

[240] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," *Cryptology ePrint Archive*, 2015.

[241] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1583–1600.

[242] "Speculative buffer overflows: Attacks and defenses," https://arxiv.org/abs/1807.03757.

[243] "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=15282018.

[244] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *CoRR*, vol. abs/1802.09085, 2018. [Online]. Available: http://arxiv.org/abs/1802.09085

[245] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[246] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv*, 2019.

[247] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Formal Analysis," University of Illinois at Urbana-Champaign and Tel Aviv University, Tech. Rep., 2019, http://cwfletcher.net/Content/Publications/Academics/TechReport/stt-formal-tr_micro19.pdf.

[248] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31, 1998.

[249] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *Proceedings of 20th International Conference on Parallel Processing*, ser. ICPP '91, 1991.

[250] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII, 1996.

[251] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 142–153, 1998.

[252] D. R. Kaeli and P. G. Emma, "Branch history table prediction of moving target branches due to subroutine returns," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 34–42.

[253] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[254] "Intel 64 and IA-32 Architectures Optimization Reference Manual," https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf, 2018.

[255] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003.

[256] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 259–271.

[257] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, vol. 5, 2001.

[258] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis lectures on computer architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[259] 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*, no. 2, pp. 1–7, 2011.

[260] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, no. 4, pp. 1–17, 2006.

[261] 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.

[262] "Invisispec-1.0 simulator bug fix," https://github.com/mjyan0720/InvisiSpec-1.0/commit/f29164ba510b92397a26d8958fd87c0a2b636b0c, Aug. 2019.

[263] 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 Security 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras pp. 955–972.

[264] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 90, 2004.

[265] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 225–236.

[266] 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 Security 16)*. Austin, TX: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl pp. 565–581.

[267] "SPEC CPU2017," https://www.spec.org/cpu2017.

[268] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[269] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening pandora's box: A systematic study of new ways microarchitecture can leak private data," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 347–360.

[270] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "Copycat: Controlled instruction-level attacks on enclaves," in *29th USENIX Security Symposium (USENIX Security'20)*, 2020, pp. 469–486.

[271] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 178–195.

[272] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating branch-shadowing attacks on intel sgx using control flow randomization," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 42–47.

[273] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 69–76.

[274] J. Van 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 *26th USENIX Security Symposium (USENIX Security'17)*, 2017, pp. 1041–1056.

[275] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[276] Y. Chen, L. Pei, and T. E. Carlson, "Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 16–32.

[277] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 667–682.

[278] Intel, "Intel ipp crypto library (2020)," https://github.com/intel/ipp-crypto/tree/ipp-crypto_2020, 2020.

[279] "Mbed-tls: An open source, portable, easy to use, readable and flexible ssl library," https://github.com/ARMmbed/mbedtls, 2022.

[280] J. L. Hennessy and D. A. Patterson, "Advanced techniques for instruction delivery and speculation," in *Computer architecture: a quantitative approach*. Elsevier, 2011, ch. 3.9, pp. 203–206.

[281] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 172–182.

[282] J. B. Kotra and J. Kalamatianos, "Improving the utilization of micro-operation caches in x86 processors," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 160–172.

[283] B. D. Hoyt, G. J. Hinton, D. B. Papworth, A. K. Gupta, M. A. Fetterman, S. Natarajan, S. Shenoy, and R. V. D'Sa, "Method and apparatus for implementing a set associative branch target buffer," U.S. Patent 5 574 871A, 1994.

[284] C. Ashokkumar, R. P. Giri, and B. Menezes, "Highly efficient algorithms for aes key retrieval in cache access attacks," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 261–275.

[285] B. Roy, R. P. Giri, C. Ashokkumar, and B. Menezes, "Design and implementation of an espionage network for cache-based side channel attacks on aes," in *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, vol. 4. IEEE, 2015, pp. 441–447.

[286] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Secretly monopolizing the cpu without superuser privileges." in *USENIX Security Symposium*, 2007, pp. 239–256.

[287] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 323–345.

[288] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 38–55.

[289] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "Papp: Prefetcher-aware prime and probe side-channel attack," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[290] Y. Chen, J. Li, G. Xu, Y. Zhou, Z. Wang, C. Wang, and K. Ren, "Sgxlock: Towards efficiently establishing mutual distrust between host application and enclave for sgx," in *31st USENIX Security Symposium (USENIX Security'22)*, 2022, pp. 4129–4146.

[291] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, pp. 1–1.

[292] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, vol. 13. ACM New York, NY, USA, 2013, p. 7.

[293] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 318–331.

[294] J. Van 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*, 2017, pp. 1–6.

[295] Intel, "Intel® software guard extensions (intel® sgx) protected code loader (pcl) for linux," 2018.

[296] J. Yu, X. Ge, T. Jaeger, C. W. Fletcher, and W. Cui, "Pagoda: Towards binary code privacy protection with sgx-based execute-only memory," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 133–144.

[297] T. Lazard, J. Götzfried, T. Müller, G. Santinelli, and V. Lefebvre, "Teeshift: Protecting code confidentiality by selectively shifting functions into tees," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 14–19.

[298] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "Sgxelide: enabling enclave code secrecy via self-modification," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 75–86.

[299] I. Puddu, M. Schneider, D. Lain, S. Boschetto, and S. Čapkun, "On (the lack of) code confidentiality in trusted execution environments," *arXiv preprint arXiv:2212.07899*, 2022.

[300] Intel, "Intel trust domain extensions," https://software.intel.com/content/dam/develop/external/us/en/documents/tdxwhitepaper-v4.pdf, 2022.

[301] 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.

[302] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, pp. 1–32, 2015.

[303] 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.

[304] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "Binsign: Fingerprinting binary functions to support automated analysis of code executables," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 341–355.

[305] I. Corporation, "Intel® software guard extensions sdk for linux* os (developer reference)," 2016.

[306] "Awesome sgx open source projects," https://github.com/Maxul/Awesome-SGX-Open-Source, 2022.

[307] Intel, "Intel 64 and ia-32 architectures optimization reference manual." https://intel.ly/2UbLwk2, 2018.

[308] WikiChip, "Macro-operation fusion (mop fusion)." https://en.wikichip.org/wiki/macro-operation_fusion., 2020.

[309] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation." *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1153, 2015.

[310] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 340–398, 2019.

[311] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–25, 2020.

[312] 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 *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1267–1272.

[313] 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 (SP)*. IEEE, 2020, pp. 20–38.

[314] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Website fingerprinting through the cache occupancy channel and its real world practicality," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2042–2060, 2020.

[315] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn dnn architectures," in *29th USENIX Security Symposium (USENIX Security'20)*, 2020, pp. 2003–2020.

[316] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 247–267.

[317] H. Xiao and S. Ainsworth, "Hacky racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers," *arXiv preprint arXiv:2211.14647*, 2022.

[318] J. C. Wray, "An analysis of covert timing channels," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 219–232, 1992.

[319] Intel, "Performance monitoring impact of intel transactional synchronization extension memory ordering issue," 2021.

[320] B. D. Williamson, "Line allocation in multi-level hierarchical data stores," 2012, uS Patent 8,271,733.

[321] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, "Computing with time: Microarchitectural weird machines," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 758–772.

[322] Intel, "Intel transactional synchronization extensions (intel® tsx) memory and performance monitoring update for intel processors," https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html, 2022.

[323] S. Deng, N. Matyunin, W. Xiong, S. Katzenbeisser, and J. Szefer, "Evaluation of cache attacks on arm processors and secure caches," *IEEE Transactions on Computers*, vol. 71, no. 9, pp. 2248–2262, 2021.

[324] ARM, "Synchronization," https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Synchronization, 2023.

[325] ARM, "Exclusive monitor system location," https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Exclusive-monitor-system-location, 2023.

[326] ARM, "Load-exclusive and store-exclusive usage restrictions," https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Synchronization-and-semaphores/Load-Exclusive-and-Store-Exclusive-usage-restrictions, 2023.

[327] Apple, "Energy efficiency guide for mac apps: Prioritize work at the task level," https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html, 2020.

[328] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "Autolock: Why cache attacks on arm are harder than you think," in *26th USENIX Security Symposium (USENIX Security'17)*, 2017, pp. 1075–1091.

[329] L. Hetterich and M. Schwarz, "Branch different-spectre attacks on apple silicon," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 19th International Conference, DIMVA 2022, Cagliari, Italy, June 29–July 1, 2022, Proceedings*. Springer, 2022, pp. 116–135.

[330] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 39–54.

[331] G. Haas, S. Potluri, and A. Aysu, "itimed: Cache attacks on the apple a10 fusion soc," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 80–90.

[332] M. Handley, "M1 explained," https://github.com/name99-org/AArch64-Explore, 2022.

[333] 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: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 2016, pp. 300–321.

[334] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer, 2015, pp. 48–65.

[335] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 191–205.

[336] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the llc." in *RAID*, 2019, pp. 427–442.

[337] W. Zhenyu, X. Zhang, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *USENIX Security Symposium*, 2012, pp. 159–173.

[338] "Openssl," https://www.openssl.org/.

[339] A. Waterman and K. Asanovic, Eds., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 2019.

[340] *MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual*, 2016.

[341] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 355–371.

[342] T. Pornin, "Bearssl," https://www.bearssl.org/constanttime.html.

[343] S. Gueron, "Efficient software implementations of modular exponentiation," *Cryptology EPrint Archive*, 2011.