DETECTING AND MITIGATING CACHE-BASED SIDE-CHANNELS

A Dissertation in
Computer Science and Engineering
by
Robert Brotzman

© 2021 Robert Brotzman

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2021
The dissertation of Robert Brotzman was reviewed and approved by the following:

Gang Tan  
Professor of Computer Science and Engineering  
Dissertation Co-Advisor, Co-Chair of Committee

Danfeng Zhang  
Assistant Professor of Computer Science and Engineering  
Dissertation Co-Advisor, Co-Chair of Committee

Mahmut Kandemir  
Professor of Computer Science and Engineering  
Committee Member

Dinghao Wu  
Associate Professor of Information Sciences and Technology  
Outside Committee Member

Chita R. Das  
Professor of Computer Science and Engineering  
Department Head of Computer Science and Engineering
Abstract

Identifying cache-based side channels in software is a particularly daunting task even for highly trained professionals. This is a consequence of side channels manifesting in a very subtle manner due to the interaction between the software and hardware. To compound the issue, once these vulnerabilities are detected, there are not many approaches available to mitigate them on commodity hardware today with reasonable overhead.

To make matters worse, new attack vectors are being uncovered seemingly all the time. This is a result of new features frequently being added to hardware to improve performance. This is most apparent with the revelation of Spectre and Meltdown, demonstrating that common optimizations found on nearly all modern CPUs can be used to construct new side-channel attacks that are costly to mitigate. With every new hardware feature and optimization brings new challenges to accurately model the relationship between the software and how it interacts with hardware.

Given these issues, this dissertation seeks to address two core problems of first the detection and second the mitigation of cache-based side channels. To address the issue of detecting cache-based side channels, we develop a tool called CaSym that automatically detects and pinpoints cache-based side channels in software or verifies their ab-
sence. CaSym uses symbolic execution to reason about programs and introduces novel cache models to achieve a more robust modeling of the CPU cache. To address new vulnerabilities discovered leveraging speculative execution, we extended CaSym in a new tool called SpecSafe to also detect leakage caused speculative execution. This is accomplished by introducing program transformation to soundly capture program behavior manifesting from speculative execution while still being able to identify conventional cache based side channels.

To address the issue of mitigating side-channels, we propose a novel user-space side channel mitigation targeting the CPU cache called Ghost Thread that is both reasonably efficient and works on virtually all commodity hardware. Ghost Thread uses threads to introduce noise into the CPU’s cache state making it drastically more difficult to successfully launch a cache-based side channel. Lastly, we survey existing user-space mitigations for Spectre attacks to provide a roadmap to developers to select the most appropriate mitigation given their circumstances.
# Table of Contents

List of Figures ix
List of Tables xii
Acknowledgments xiv

Chapter 1
Introduction 1

Chapter 2
Background 7
2.1 CPU Cache .................................................. 7
2.2 Speculative Execution ...................................... 8
2.3 Attacks ...................................................... 9
  2.3.1 Cache Side Channels ................................. 9
  2.3.2 Transient Execution Attacks ....................... 13

Chapter 3
Related Work 18
3.1 Mitigation Techniques ..................................... 18
  3.1.1 Software ................................................. 18
  3.1.2 Hardware ............................................... 20
3.2 Detection Techniques ...................................... 20
  3.2.1 Conventional Cache Attacks Detection .......... 20
  3.2.2 Spectre Side-Channel Detection ................. 21
  3.2.3 Timing Channel Detection ......................... 22
  3.2.4 Other Attack Detection ............................. 23
## List of Figures

2.1 Illustration of how an adversary can learn secret data using a conventional cache side channel. We assume A is an array of 1-byte elements and the cache line size is 64 bytes. .................................................. 9

2.2 Spectre Variant 1 sample and its control flow graph .......................... 14

2.3 Spectre Variant 2 example ................................................................. 15

2.4 Spectre Variant 4 sample ................................................................. 16

4.1 An overview of how Ghost Thread can be used to provide protection to an application. ................................................................. 27

4.2 An example showing how a user can modify an application to provide protection. The highlighted lines of code would be added by the user. Green indicates necessary code modification, yellow indicates code added to reduce overhead. ................................................................. 29

4.3 The number of threads required to obtain 99.9%, 99%, 95%, or 90% overlap with a protected thread, when various background tasks are running at the same time on a 6 core machine. These values assume that the scheduler assigns every task with the same probability of being scheduled. 34

4.4 Pseudo code of the AES attack .......................................................... 43

4.5 AES Attack without and with Ghost Thread ...................................... 44

4.6 RSA Attack without and with Ghost Thread ...................................... 45

4.7 The throughput overheads of concurrent users accessing a webpage hosted on an Apache server with both the AES and RSA routines protected by Ghost Thread. ................................................................. 48

4.8 The latency overheads of Ghost Thread on an AES implementation from libgcrypt 1.8.3 with various configurations. The boxes show the additional normalized samples needed. ................................................................. 49

4.9 The latency overheads of Ghost Thread on an RSA implementation from GnuPG 1.4.23 with various configurations. The boxes show the additional normalized samples needed. ................................................................. 50
4.10 How adjusting the frequency impacts the execution time and leakage of an libgcrypt’s AES implementation. Due to space constraints, we only show results for up to 9 background processes. .......................... 55
4.11 Percent overhead on bzip2 when a program protected by Ghost Thread is running concurrently. ........................................... 57
4.12 The percent overhead on gobmk from the SPECCPU 2006 benchmark when a program protected by Ghost Thread is running concurrently. ... 57

5.1 Example program composed of snippets of real code which demonstrate the two kinds of side channels. ............................ 63
5.2 System flow of CaSym. ................................................. 64
5.3 Syntax for a path language and symbolic execution over a path. .... 70
5.4 Sample program illustrating how path merging is handled using the infinite cache model. ..................................................... 72
5.5 Shared path computed from the model generated by the SMT solver for the code in Fig. 5.1. ................................................. 85

6.1 Cache side channels that elude existing security notions. key is a one-bit secret in all the examples. ................................. 106
6.2 Language Syntax ........................................................ 111
6.3 SpecSafe’s workflow. .................................................... 119
6.4 Transformation rules for composition, if and while with transaction length parameter $w$. The helper function $[S]_w$ statically extracts $w$ instructions from $s$, unless a fence is encountered during that. Furthermore, $s_*$ is used to denote the sub-program right after the sub-program being analyzed (i.e., the extra instructions being padded in case of mis-prediction). ......................................................... 121
6.5 Speculative-aware symbolic execution of SpecSafe assuming that the input $\sigma$ is $\overline{X} = \overline{e} \land C = c \land \Psi$. The symbolic execution also maintains a global list of symbolic memory snapshots $\text{mem}_s$, indexed by transaction identifiers. ......................................................... 125

7.1 Examples of Protection by Execution Barrier. ......................... 143
7.2 Examples of data dependency defenses. ............................. 146
7.3 Examples of PDC defenses. .......................................... 150
7.4 Average overheads reported from external sources that provide Spectre mitigations. .................................................... 160
7.5 Normalized execution time w.r.t. an unmitigated baseline to mitigate Spectre V1 1 for SPECint 2017, nginx and thttpd. .......... 160
7.6 Normalized execution time w.r.t. an unmitigated baseline to mitigate Spectre V2 for SPECint 2017, nginx and thttpd. .......................... 161
7.7 Normalized execution time w.r.t. an unmitigated baseline to mitigate Spectre V1 and V2 for SPECint 2017, nginx and thttpd. ............ 161
7.8 Normalized binary size w.r.t. an unmitigated baseline to mitigate Spec-
tre variant 1 for SPECint 2017, nginx, and thttpd. ....................... 165
7.9 Normalized binary size w.r.t. an unmitigated baseline to mitigate Spec-
tre variant 2 for SPECint 2017, nginx, and thttpd. ....................... 165
7.10 Normalized binary size w.r.t. an unmitigated baseline to mitigate Spec-
tre variants 1 and 2 for SPECint 2017, nginx, and thttpd. ............... 166
7.11 Developer guidance, summarizing the results presented in this chapter.
In this figure, the orange points require manual effort to identify sensi-
tive data while green points do not. ............................................ 166

A.1 Speculative-Aware semantics given an oracle $O$ and a transaction length $w$. ................................................................. 176
List of Tables

2.1 This table summarizes window sizes used in various cache side-channel attacks. Note that the window size for successfully launching the synchronous attacks depends on the execution time of the victim code. When the data is unavailable in the paper, we simply write “victim access time” in the table. ........................................... 12

4.1 The average number of memory accesses that a ghost thread can inject in a window of 2,000 clock cycles. ........................................... 36

4.2 Normalized samples with various amounts of noise accesses for AES and RSA. ........................................... 41

4.3 Overheads of Cloak [1] and one single Ghost Thread. .................. 48

4.4 A summary of how long it takes to collect sufficient number of samples on our test machine to launch the corresponding cache attacks without and with Ghost Thread. ........................................... 53

5.1 Evaluation results for access and trace-based attackers for crypto benchmarks. For access-based results, ☹ means a side channel identified and ✓ means no side channel identified; it also reports the amount of time in seconds for CaSym to perform side-channel checking. For trace-based results, the TP column identifies the number of true positives found. Similarly FP is the number of false positives. The third column for each models depicts the amount of time (in seconds) for each test. ........... 89

5.2 Evaluation results for access and trace based attackers for the PostgreSQL database. ........................................... 90

5.3 Evaluation results for symmetric ciphers with table preloading and pinning for trace-based attackers. ........................................... 96

5.4 Evaluation results using the LRU model; the table shows results for an access based attacker on the left and a trace based attacker on the right. 98
6.1 The throughput (bits per second) of the new side channels in Figure 6.1 as well as AES-128 embedded with Instance 1 at different levels of accuracy. ................................................................. 110

6.2 Results of SpecSafe on micro-benchmarks of both vulnerable code as well as their corresponding protected versions. Spectre examples includes all 15 from [2]. The numbers in the table are SpecSafe’s execution times. ♦ means no side channel is found. ▲ means side channels are detected. ................................................................. 127

6.3 Comparison with existing tools for detecting speculative side channels. A ✓ indicates the tool correctly judges the security/insecurity of the code (i.e. marking it secure/insecure if it is secure/insecure). A ✗ indicates the tool either has a false negative (when the code is vulnerable) or a false positive (when the code is secure). n/a applies when the tool fails to detect a vulnerability in the original code and thus cannot reason about mitigations. ................................................................. 128

6.4 Evaluation results on libgcrypt ciphers. Here, the first three columns indicate the number of lines of LLVM IR are in each module, the number of functions found in the module, the time to analyze the module. The First column under taint analysis indicates how long the taint analysis ran. The next three columns indicate the number of functions found containing those code patterns described in Section 6.7. The last column under taint analysis is the union of the previous three columns indicating the number of potentially vulnerable functions. The CNI and Spec columns indicate the number of functions the symbolic execution found containing those kinds of vulnerabilities. The SANI column contains all functions found that violate our SANI definition, along with the total analysis time for symbolic execution. ................................................................. 129

7.1 Summary of when certain classes of defenses are effective against different Spectre variants as well as the characteristics of mitigation classes. We use ● to indicate that all defenses in the class apply, ◆ to indicate a mitigation first proposed in this work, ○ to indicate some of the defenses apply, and ◐ to indicate none of the defenses apply. Note that Spectre Variant 3 (Meltdown) and Spectre RSB are not listed since they are beyond the scope of user-level defenses. ................................................................. 141
Acknowledgments

An undertaking such as this is not possible in solitude, but requires a tremendous amount of support from many people along the way. With that said, I would like to give thanks to the people that I have been very fortunate to have guide and support this endeavor over the past six years.

First, I thank my advisors Prof. Gang Tan and Prof. Danfeng Zhang for their unwavering support. Over the years, their mentorship has been invaluable in my development as a researcher, writer, and scholar. I am truly grateful for the countless hours each of you have graciously given, whether that be the late nights working on papers, tips for improving a presentation, or guidance in research pursuits. I cannot think of better role models to prepare for a research career.

I also would like to thank my other committee members, Prof. Mahmut Kandemir and Prof. Dinghao Wu, for their insightful feedback and discussions we have had during this process. I recognize that this can be a rather time consuming endeavor and truly appreciate your involvement. Furthermore, I would like specifically to thank Prof. Kandemir for bringing his years of experience and expertise with respect to computing hardware to this research. I have always appreciated and valued these discussions. Additionally, I appreciate Prof. Wu bringing additional expertise to this project and his constructive feedback.

Additionally, I would like to thank my mentor at Microsoft Research Ben Niu. I truly appreciated your guidance and fellowship over the summer and appreciate all of the time you spent to make the experience special. Furthermore, I would like to thank Weidong Cui and Xinyang Ge for their astute insights and useful lessons throughout the experience.

It has been an absolute joy to work with so many great and friendly people at Penn State. I am truly grateful for the friendships I have gained over the years that have certainly made the successes greater and the failures more endurable. Specifically, I would like to thank my lab mates Shen Liu, Dongrui Zeng, Sun Hyoung Kim, Michael Norris, Yongzhe Huang, Ashish Kumar, Xiaodong Jia, Yuxin Wang, Zeyu Ding, and Peixuan Li for all of our interesting discussions and just making the last six years fun.
Lastly, I would not have made it this far without the love and support of family and friends throughout my entire life. First, I would like to thank my wife Brittany for all of her love, encouragement, and steadfast support over the last 12 years. I would also like to thank my Mom and Dad for instilling in me a strong work ethic, showing me the value in education, and most of all giving me the opportunity to pursue my dreams.

**Funding Acknowledgement**  The work presented in this thesis is supported by NSF CNS-1956032, NSF CCF-1723571, and a gift from Intel. The findings and conclusions presented in this document are those of the authors and do not necessarily reflect the views of the organizations listed previously.
Dedication

To my family.
Chapter 1

Introduction

In computing systems, side-channel attacks manifest when the attacker can monitor the interactions between software and hardware to glean information about data being processed by observing hardware characteristics. Multiple attack vectors have been proposed, exploiting machine features such as power consumption [3], CPU cache [4, 5], electromagnetic field [6, 7] and execution time [8]. Cache-based side-channel attacks have become a popular vector due to their applicability in cloud settings [9], as well as their ability to leak data out of browser sandboxes [10].

Side channel vulnerabilities have plagued developers of security-critical software for decades due to their subtle nature as well as the ever-changing attack surface due to new hardware developments. A recent example of this is the discovery of Spectre [11] and Meltdown [12] that leverage the out-of-order execution of transient instructions to cause sensitive data that may not ordinarily be accessed to leave a footprint in the microarchitectural state and permit covert channels to uncover the sensitive data.

Much of the work done to eliminate cache-based side channels from programs has come from developers painstakingly removing them from security-critical programs. This is common in applications using cryptographic ciphers such as RSA, AES, Elga-
mal, etc. Unfortunately, this methodology is apt to miss vulnerabilities and is demonstrated every time new vulnerabilities are found in existing implementations. Fortunately, approaches have gained traction over the past decade to analyze programs to detect cache-based side-channel vulnerabilities. Works such as CacheAudit [13, 14] and CacheD [15] are able to automatically detect side channels in C programs.

With the advent of vulnerabilities resulting from speculative execution, new models are required to address these new threats. Work using pattern matching [16] and symbolic execution [17] have been proposed to detect and mitigate vulnerabilities resulting from speculative execution. Compared to tools that only reason about programs running on hardware without speculative execution, these approaches are a step in the right direction; however, they both check conservative security properties that do not precisely model real hardware behavior.

While exiting work has made great strides towards automating the detection of cache-based side channels, they lack several key features that prevent them from being more generally applicable. Some of the most significant features absent in existing work are: soundness, error-localization, multi-architecture applicability, and flexibility.

Existing tools focus on either being sound or identifying the location of a vulnerability. Tools such as CacheAudit [13, 14] propose techniques that are sound, meaning they will not miss a vulnerability, but lack mechanisms to localize where the leakage is coming from in the program. The ability to determine the line of code in a program causing the problem is an important issue to make tools practical for everyday users that may not be experts on side channels. On the other hand, techniques such as CacheD [15] are not sound, meaning they may miss vulnerabilities, but are able to analyze large programs and localize vulnerabilities in a program. Lacking soundness is particularly problematic because even after the tools report no vulnerabilities, there may still be attack vectors available to adversaries. More recent approaches that capture leakage via the CPU cache
using speculative execution share similar limitations [16–19].

Cache side-channel mitigations are often hardware specific to achieve efficiency or hardware agnostic to achieve greater applicability. Approaches that use hardware specific tools such as Intel’s TSX [1] or Intel’s Cache Allocation Technology (CAT) [20] are not generally available on existing hardware. Other mitigations that make use of resources available on virtually all hardware involve adding noise or making the resource usage uniform. Unfortunately, these approaches are either insecure or do not scale due to high performance overhead.

Lastly, current analysis techniques typically assume a strong attack model that is not currently feasible in practice. Tools such as Spectector [17] assume an adversary can observe individual addresses accessed as well as the program counter. While this assumption is common, it can lead to false positives. A better approach is to more accurately model how actual hardware behaves to reduce false positives, while abstracting away enough detail to remain general and sound.

Given these issues, it is clear that an exploration is warranted to first investigate how program analysis techniques can be constructed to be flexible, and sound while still being able to pin-point where in the program the problem is. Second, there is a clear need for a solution to cache-based side channels that do not require hardware specific features but are both secure and efficient. To address these issues, the central question investigated in this dissertation is: can we develop user-space defenses to cache-based side channel attacks that provide strong security while remaining generally applicable across multiple hardware implementations and practical in real-world scenarios. To this end, this dissertation addresses these points by proposing a novel analysis that is both sound and flexible but is also able to identify why and where the leakage occurs. Furthermore, we propose a user-space mitigation for cache-based side channels that uses features that are available on virtually every commodity system that is both efficient and
provides strong security.

Firstly, we propose a novel user-space cache side-channel mitigation called Ghost Thread [21]. Ghost Thread works by using other CPU cores to inject memory accesses to disrupt an adversary’s ability to uncover the victim’s memory access pattern. We show that Ghost Thread can successfully mitigate leakage by requiring absurd amounts of additional samples for a successful attack. Furthermore, we test Ghost Thread on widely used cryptographic libraries along with a web server to evaluate its performance impact. The results indicate that systems without a heavy load will experience virtually no overhead from Ghost Thread, while systems that are heavily loaded can experience moderate overhead due to the additional thread.

In order to provide a sound and flexible analysis that is able to pinpoint leakage, we developed CaSym [22]. CaSym uses symbolic execution along with novel cache abstractions to compute the state of the cache in a flexible yet accurate manner. Our key insight is we can use symbolic execution to provide solutions that allow us to pinpoint the cause of the leakage while still being able to encode cache models of various granularities to support different threat models. The combination of multiple cache models and error localization makes CaSym a great candidate for developers to both detect and remove cache-based side channels from their programs. We applied CaSym to multiple cryptographic ciphers as well as a widely used database application. During our experiments, we discovered over 300 unique locations of cache-based leakage in the analyzed programs.

With the advent of Spectre [11] and its variants [12, 23, 24], we extended CaSym to also be able to detect leakage that occurs during speculative execution in a new tool called SpecSafe. During our investigation, we discovered three new instances of Spectre attacks that were previously unknown and demonstrated their applicability by using these new gadgets to successfully create covert channels. To account for the newly
discovered variants, we propose a more precise security definition that is able to accommodate multiple cache models. To evaluate our tool, we apply SpecSafe to well-known cryptographic ciphers. As a result of our experiments, we discovered two new potential Spectre vulnerabilities and one conventional cache-based side channel vulnerability.

Due to the lack of comparisons amongst existing user-level mitigations for Spectre attacks, we surveyed current mitigations and developed an in-depth comparison evaluating key characteristics of each approach. Firstly, we provide a novel systematization that highlights key attributes that developers may want in a mitigation. Second, we evaluate as many techniques as possible based on availability. For techniques that were easily implementable, we implemented them to provide the most comprehensive comparison among existing techniques. We uncover a significant trade-off between mitigations that require user-intervention to mark sensitive data and techniques that are agnostic to sensitive data. Furthermore, we find an interesting trade-off between how general a technique is and its performance.

**Outline**

The rest of this dissertation is organized in the following manner. Chapter 2 presents the relevant background information. This section will discuss the basics regarding how the CPU cache works, what speculative execution is, and how side channels can manifest. In order to provide a better context for the work presented in this dissertation, related works are detailed in Chapter 3. Chapter 4 presents our user-space cache side channel mitigation tool called Ghost Thread. Chapter 5 will present a detailed discussion regarding our experience with CaSym. Chapter 6 introduces our work extending CaSym to incorporate speculative execution. Chapter 7 describes our work comparing existing user-space Spectre mitigations. Lastly, we provide our thoughts for interesting lines of
future work in Chapter 8 and provide some closing remarks in Chapter 9.
Background

Much work has been done to detect and mitigate cache-based side-channel attacks. In the next sections, we will briefly introduce necessary information to understand both past works and the work presented in this dissertation. Moreover, we will go over related work regarding both detection and mitigation techniques.

2.1 CPU Cache

The CPU cache is a hardware feature that facilitates faster memory accesses by storing data close to the CPU. To accommodate this, modern CPUs typically have multiple levels of cache (e.g., L1, L2, and L3). Each level of cache will be shared by a different number of cores and have different access latencies. For instance, the L1 usually fetches data in under 5 CPU cycles and may serve one physical CPU core, while L3 may take about 100 cycles but will serve all CPU cores on a processor. All of these access latencies are significantly faster than if the CPU had to wait for memory to be fetched from the main memory, which would take up to 2 order of magnitude more time than the CPU cache.
When the CPU needs data from the memory, it first checks if it is present in the CPU cache. During the search, the CPU goes through each level of the cache to see if the data is present. When data is in any level of the cache, it is known as a *cache hit*, while data not in any level of the cache is denoted as a *cache miss*. Note that due to the limited size of the CPU cache, it is likely that a program requires more memory than available in the cache. Hence, a cache replacement policy uses heuristics based on past execution to infer the best candidates to keep in the cache. Unfortunately, since these heuristics depend on the program’s execution, they can lead to side channels if the state of the cache becomes dependent on sensitive program data.

### 2.2 Speculative Execution

Virtually every modern CPU applies many optimizations that are designed to reduce the number of wasted CPU cycles. One such optimization is known as speculative execution. Speculative execution happens when an instruction has to wait for some other task to complete, such as a memory fetch, prior to completion. Instead of waiting for the operation to finish, the CPU will speculatively execute predicted future instructions to avoid wasting cycles. The CPU ensures that regardless of the order the instructions are speculatively executed, they will be retired in order, which ensures the integrity of the computation.

Sometimes speculative execution executes instructions that should not have been executed and thus the result of those instructions needs to be undone. Such instructions are called *transient instructions*. One common example of this is during branch prediction, during which the processor guesses which branch is more likely to be executed. Sometimes it guesses incorrectly; the processor is then responsible for rolling back the changes made during the misprediction. However, this contract is limited to the archi-
2.3 Attacks

2.3.1 Cache Side Channels

Cache side-channel attacks typically consist of three phases depicted in Figure 2.1. The first phase, the adversary attempts to get the cache into a known configuration using one of many techniques we will discuss shortly. Next, the adversary waits to allow the victim program to modify the cache. Lastly, the adversary collects information about the current state of the cache. The adversary is able to learn information based on how architectural state (e.g., registers, memory). Other microarchitecture components, such as the CPU cache, are not rolled back and thus contain side effects of instructions that were transiently executed.
the victim program modified the cache.

The most common side-channel attacks used are variants of the *Prime+Probe* and *Flush+Reload*. Other cache side-channel attacks such as *Evict+Time*, *Flush+Flush*, and *Prime+Abort* share many similar characteristics but differ in either how they check the cache state or get it into a known state.

The *Prime+Probe* [5, 25] technique consists of three phases. During the first phase, the adversary fills target cache sets with data. Then they will give the victim program time to run and use the cache. Lastly, the adversary probes the cache sets it previously filled. The adversary can infer that if their data is still in the cache set, it is less likely the victim program used that cache set. Similarly, they can learn the victim likely used a cache set if their data is not in the cache.

Similar to *Prime+Probe*, the *Flush+Reload* [26, 27] attack also is carried out in three parts. In the first part, the adversary will flush data from the cache using the `clflush` instruction and then give the victim program time to access the cache. The adversary will then reload the data previously flushed to see if it is cached or not. Cached data means the victim program used the data. If it is not cached the data was not used by the victim.

The *Flush+Reload* is generally more accurate than other attack techniques because the `clflush` instruction targets a specific cache line opposed to attacking a cache set. The improved granularity comes at the cost of being less general since in order for the *Flush+Reload* attack to work, the victim and attacker need to share a physical memory page. This sharing allows the attacker to flush the victim’s data from the cache [27]. Attacks such as *Prime+Probe* do not have such a limitation since they target the cache set being used by the victim making them more general, but also less precise.

The *Evict+Time* [5, 28] attacks similarly attempt to get data out of the cache, but typically do this without the `clflush` instruction. Essentially it evicts target cache sets
from memory, then times how long the victim program runs. This style of attack will have much lower bandwidth than the Prime+Probe or Flush+Reload since it does not try to learn information about specific cache sets or cache lines, but instead about the overall program.

Similar to the Flush+Reload, the Flush+Flush attack also initially uses the clflush instruction to flush target cache lines shared with a victim process. After the adversary waits for the victim, instead of reloading the data, this time the adversary will flush the data again from the cache. The key insight in this attack is the time it takes to flush data from the cache depends on whether the data is cached or not.

Lastly, the more recent Prime+Abort attack uses Intel’s TSX feature to create an attack that does not require a timer to be successful. Similar to the Prime+Probe attack, the Prime+Abort attack first loads data into target cache sets and waits. The novel part is instead of simply timing how long it takes to reload data in those cache sets, the attacker will wrap the memory accesses in a transaction. If the data is still cached, the transaction will succeed otherwise it will fail. A benefit of this attack is it does not rely on a fine grain timer to work, which is problematic since a common defense is to add noise to these timers to prevent side-channel attacks.

Attack Settings

Typically, attacks are separated into two categories: synchronous and asynchronous. A synchronous attacker controls when the target victim code starts executing and stops [5]. This style of attack allows an adversary to get precise information about cache behavior. Since it allows accurate measurements, it is often used to demonstrate new attacks can successfully learn information [5,29,31,32]. This style of attack assumes a strong adversary that is often unrealistic due to their fine-grained control over when the
<table>
<thead>
<tr>
<th></th>
<th>Window (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchronous</strong></td>
<td></td>
</tr>
<tr>
<td>Flush + Flush</td>
<td>200+(victim access time) [29]¹</td>
</tr>
<tr>
<td>Flush + Reload</td>
<td>520 [33]</td>
</tr>
<tr>
<td>Prime + Probe</td>
<td>500+(victim access time) [34]</td>
</tr>
<tr>
<td><strong>Asynchronous</strong></td>
<td></td>
</tr>
<tr>
<td>Flush + Reload</td>
<td>2500 [27]</td>
</tr>
<tr>
<td>Prime + Probe</td>
<td>5000 [25]</td>
</tr>
</tbody>
</table>

Table 2.1: This table summarizes window sizes used in various cache side-channel attacks. Note that the window size for successfully launching the synchronous attacks depends on the execution time of the victim code. When the data is unavailable in the paper, we simply write “victim access time” in the table.

Unlike synchronous attackers, an asynchronous attacker [5, 25–27] does not have control over the start or stop of the victim program, making it a more realistic scenario. These attacks need to infer when the victim begins running. Once the victim starts running, the adversary will make observations of the cache state using fixed time intervals called windows.

**Observation Windows**

All attack techniques previously described require getting the cache into a known state, wait for the victim to execute, and then observe the state of the cache again to see what the victim modified. We call the time interval to perform these steps once an attack window.

Table 2.1 summarizes the window sizes reported in various cache-based side-channel

¹This work does not report a specific window size. However, the attack requires two clflush instructions to be executed. We note that the execution time of a single clflush instruction varies on different CPUs. We use 200 since [29] reported that a single clflush instruction takes a minimum of 100 CPU cycles on various CPUs to complete.
attacks. Typically, for a *synchronous* attack, the window size accounts for the time to flush/reload/prime/probe, plus the time it takes for the victim code to run. With strong but unrealistic assumption that the adversary controls the beginning and end of a few instructions of interest, victim access time as low as 100 cycles has been reported [34]. But for more realistic settings where the attacker only controls the begin and end of one encryption, one encryption itself takes around 320 cycles [33]. For the less restrictive *asynchronous* attacks, the window size is typically thousands of cycles.

### 2.3.2 Transient Execution Attacks

Transient execution attacks leverage speculative execution to reveal confidential data. While it is also possible to use other hardware features to construct side channels, we focus on transient execution attacks that leverage the CPU cache. We follow a recent systematization of transient execution attacks by Canella et al. [35] to classify different variants by the attack surface that is being exploited. The classification not only avoids confusing naming schemes but also allows us to discuss defenses for each class of attacks that exploit the same attack surface in a sensible way.

**Spectre Variant 1 (aka. Spectre-PHT [35])**  
Spectre Variant 1 (V1) exploits the Pattern History Table (PHT) that predicts the outcome of conditional branches. By manipulating the PHT, an attacker is able to reveal data from a victim’s address space by accessing memory that should not be accessed during sequential execution [11]. Notably, it allows “bounds check bypass” attacks that access out-of-bound memory that is not allowed in normal execution.

We use the code fragment in Figure 2.2 with its control flow graph on the right to illustrate Spectre variant 1 attack. In this example, taken from [11], both arrays *arry* and *arry2* are of fixed size and only contain public information. During in-order execution,
idx = input;  
if (idx<arry.size)  
{   
t = arry[idx];   
...   
res = arry2[t];  
}   
...  

Figure 2.2: Spectre Variant 1 sample and its control flow graph

the program is secure since it only reads public information with idx<arry.size.

However, an adversary who can control the value of input may train the branch predictor so that the hardware predicts that the true branch will be taken. Then, the adversary feeds an input that is out-of-bounds for array arry which will trigger a mis-prediction: speculative execution loads the value of arry[idx], which refers to an arbitrary piece of memory since the true branch is predicted and arry[idx] is within the speculation window. Although the value of t is erased when the speculative execution is rolled back, as long as arry2[t] is executed during the speculation window, it will leave a secret-dependent footprint in the data cache, allowing conventional cache attack techniques such as Flush+Reload [26, 27] and Prime+Probe [5, 25] to reveal the value of arry[idx] (as illustrated in Figure 2.1).

Spectre Variant 2 (aka. Spectre-BTB [35]) Spectre Variant 2 (V2) exploits the Branch Target Buffer (BTB) that predicts the target of indirect branches and indirect
calls. By modifying the BTB, an attacker is able to speculatively execute an arbitrary piece of code as predicted by the branch target buffer [11].

We use the code fragment in Figure 2.3 to illustrate Spectre V2 attack. In this example, when an indirect branch is encountered at line 1 and the data stored at address eax is not ready yet, the processor will speculatively begin to execute an instruction predicted by BTB, say BTB_guess, which is dictated by the attacker by updating the entries in the BTB. The processor will eventually detect a misspeculation and rollback mispredicted instructions, but like under a Spectre V1 attack, the microarchitectural footprints of those instructions might still reveal secrets. Since arbitrary gadgets can be targeted, this opens Pandora’s box in terms of what the adversary can do and allows the adversary to execute arbitrary code speculatively.

**Spectre Variant 3 (aka. Meltdown-type [35])** Meltdown exploits transient execution following a CPU exception: since exceptions are only raised at the retirement of a faulting instruction, transient instructions following an exception enables the adversary to read data that would otherwise be illegal with the transient instructions, such as reading kernel memory.

We note that Meltdown [12] and its variants [36–39] cannot be effectively mitigated using user-level defenses since they exploit out-of-order execution following exceptions. Instead, Meltdown is commonly mitigated using a combination of OS-level mitigation,

```c
jmp *eax;
BTB_guess:
    leak_secret;
...
```

```c
correct_target:
    benign;
...
```

Figure 2.3: Spectre Variant 2 example.
\begin{verbatim}
*mem = benign;
benign2 = *mem2;
...
\end{verbatim}

Figure 2.4: Spectre Variant 4 sample

such as KAISER \cite{12,40}, and hardware mitigation, such as microcode updates \cite{35,41}. Since we focus on user-level defenses in this dissertation, Meltdown-type attacks are outside of the scope.

**Spectre Variant 4 (aka. Spectre-STL \cite{35})** Spectre Variant 4 (also known as Spectre-STL or Spectre-SSB \cite{23,42}) exploit hardware units that predict dependencies in the data flow, such as the Store to Load (STL) dependency. An illustrative example is shown in Figure 2.4. In the example, \texttt{mem} and \texttt{mem2} are pointers that are aliases and \texttt{benign} is non-sensitive data. If the store at line 1 is delayed due to some effect from a previous instruction, the load at line 2 is speculatively executed. When the memory disambiguation unit does not detect that \texttt{mem} and \texttt{mem2} are aliases, it allows the load at line 2 to read the value stored at the location that both \texttt{mem} and \texttt{mem2} point to before line 1 overwrites the value. If that value contains sensitive data, it can be encoded into the cache state in the following instructions (e.g., through a memory access using \texttt{benign2} as the address) and then extracted using standard techniques for cache side channels.

**Spectre-RSB (aka. ret2spec)** Spectre-RSB \cite{24,43} exploits the Return Stack Buffer (RSB), a per-core buffer that stores virtual addresses following the most recent call instructions. Since RSB is updated in a predictable way, an adversary can use this predictability to insert arbitrary code location to be speculatively executed \cite{24,43}. The transient instructions targeted by the attacker may leak sensitive data via side channels. The defenses for this threat are limited since an attacker can set the state of the RSB at will via a context switch. Thus the commonly used defense is to fill the RSB on every
context switch [44], which is handled by the operating system.
Chapter 3

Related Work

This chapter introduces work related to what is presented in this dissertation. To facilitate this discussion, the chapter is broken into two sections. The first section presents existing techniques used to mitigate cache-based side channel attacks. The second section describes existing methods to detect cache-based side channels.

3.1 Mitigation Techniques

3.1.1 Software

Many cache-based side-channel mitigations have been proposed in the literature. At the application level, one of the more recent approaches, called Cloak [1], leverages hardware transactional memory (HTM) to prevent cache-based side channel leakage. Cloak uses HTM to ensure that once data is cached, it cannot be evicted until the end of the transaction or an abort occurs.

Other approaches, such as diversification [45] and obfuscation [46] introduce randomness to confuse attackers. Diversification generates a replica that preserves the original program semantics but differs at the level of machine instructions; hence, by select-
ing a replica to execute at random, it alleviates side channels. Obfuscation confuses the attacker by pretending that additional program paths are executed than what really was. Lastly, another common method seen is to rewrite the source code so that memory accesses are independent of confidential data. A common approach in cryptographic applications is to use constant time implementations of specific algorithms [47] or hardening an existing algorithm to remove memory accesses dependent on sensitive data as discussed in [14]. Applying the constant-time programming paradigm properly is challenging for non-expert developers to get correct. Compiler-aided transformation exists [48–51]; however, those techniques only remove sensitive branches in the source code. They do not prevent sensitive cache-line accesses in general.

New software defenses have been proposed to prevent vulnerabilities manifesting from speculative execution. In the work discovering Spectre [11] style attacks, Kocher et al. propose using the \textit{lfence} to prevent speculation. Applying \textit{lfence} throughout a program can cause significant overhead, thus other approaches causing artificial data dependencies on only the sensitive data have been proposed [52,53]. These artificial data dependencies prevent the processor from loading sensitive data into the cache. Although this approach has also been shown to have high overhead in some cases (over 40% in some cases), it is still better than \textit{lfence} in terms of performance [52]. We further explore software-based mitigation for speculative execution leakage in Chapter 7.

At the OS/VM level, one related work is Düppel [54], which injects random delays into the memory access time to obfuscate the cache state (i.e., a cache hit may incur a long latency by the injected delay from OS). Schwarz et al. propose KeyDrown [55], a kernel mechanism that prevents timing attacks on keystrokes by injecting fake keystrokes. However, KeyDrown does not generally mitigate cache-based side channels, and requires modifications to the operating system.
3.1.2 Hardware

At the hardware level, randomizing cache accesses has been explored in prior work. Some system randomize the cache mapping [56] and others change the cache replacement policy to evict and fill in a randomly picked cache line [57, 58]. Implementation of Oblivious RAM (ORAM) [59–61], a technique that cryptographically obfuscates the memory access patterns, typically add large amounts of overhead when done via software, and even with some hardware support, such as secure enclaves, can still have significant overhead [62].

Another approach to prevent cache-based side channels is to isolate the victim from the attacker. Previous work has achieved isolation that partitions resources to disallow sharing between an attacker and the victim [20, 63–68]. However, resource isolation is typically very challenging at the application level. Moreover, they all incur overhead even in a system with low resource utilization.

Researchers have also developed hardware prototypes to prevent speculative vulnerabilities. Some work proposes hiding cache effects caused by speculative execution until it is safe to release them [69, 70]. Other work uses protection domains to isolate processes so that changes to the cache state are not visible across protection domains preventing side channels caused by speculation and standard ones [68].

3.2 Detection Techniques

3.2.1 Conventional Cache Attacks Detection

In order to ensure all cache-based side channels are mitigated, program analysis techniques are required to detect them. To this end, many mechanisms have been proposed. One such technique, called CacheD [15], takes a concrete program execution trace
as input and symbolizes secret values during a symbolic execution to identify secret-dependent memory accesses. Since all values except secrets are concrete in the analysis, CacheD is likely to be more precise than a static program analysis (i.e., have fewer false positives). However, CacheD explores only the same execution path as the input dynamic trace. Hence, vulnerabilities in the unexplored code or those that are due to secret-dependent branches cannot be detected by CacheD.

Another common approach is to use abstract interpretation to reason about the cache state [13, 14, 71]. CacheAudit [13, 14] uses abstract interpretation to compute an upper bound on the amount of leakage possible via side channels using the CPU cache. This is achieved by using a variety of cache and attack models. Bacelar et al. [72] propose to use abstract interpretation in combination with a 2-safety property to verify programs to be constant-time. Recent work CacheS [71] uses scalable abstract interpretation to pinpoint where in a program leakage happens.

### 3.2.2 Spectre Side-Channel Detection

More recently, automatic static analysis for Spectre-style attacks has been proposed. One of the first such attempts is oo7 [16]. It uses taint analysis along with predefined code patterns to detect potential Spectre variant 1 vulnerabilities in programs, but lacks a formal security definition of what kinds of side channels it can identify and may miss some Spectre-PHT vulnerabilities. Another tool called Spectector [17] offers both a formal security definition called speculative non-interference (SNI) and uses symbolic execution to detect violations of SNI; however, it assumes a conservative attack model where the program counter is observable. A parallel work to Spectector [19] presents a similar security definition, called trace property-dependent observational determinism (TPOD). While this definition has been specified as a 4-safety property, it is noted
that it is similar to SNI. Another recent work called InSpectre [73] builds on these ideas and incorporates additional reasoning to detect multiple variants of Spectre vulnerabilities not captured by other existing works.

Additionally, SpecFuzz [18] uses dynamic symbolic execution to detect bounds-check bypass vulnerabilities in programs using a technique called speculative exposure that modifies an IR to simulate branch mispredictions. Their technique only detects bounds-check bypass during a branch misprediction, a subclass of Spectre Variant 1 attacks.

### 3.2.3 Timing Channel Detection

Program analysis [74–79] also has been used to identify timing channels, where an attacker reveals confidential information via analyzing program execution time. However, most of them ignore both cache and speculative execution in their model of timing. Hence, they might miss conventional as well as speculative cache side channels. For example, recent systems [78, 79] use symbolic execution to detect leakage via timing channels as well as to synthesize inputs that result in the maximum amount of leakage via timing channels. However, their timing model ignores cache: they use the number of instructions to measure execution time.

On the other hand, the constant-time programming discipline offers strong security against timing channels and hence, is widely used in modern cryptography implementations. Following this discipline, constant-time programs cannot (1) branch on secrets, (2) access secret-dependent memory addresses, or (3) use secret data as inputs to any variable-time operation. Program analysis [80–83] has been proposed to enforce the discipline. Notably, parallel works [81, 83] recently extend the discipline to both define and verify constant-time execution in the presence of speculative and out-of-order ex-
execution. However, the discipline is too strong against cache attacks: it can come with unnecessary performance penalties [51], and reject permissive countermeasures against cache attacks, such as aligning branches with a sensitive condition into a single line of the instruction cache or preloading lookup tables [14].

### 3.2.4 Other Attack Detection

Detecting vulnerabilities prior to the programs running is clearly desirable, however, may not always be possible. Thus, detecting when adversaries are actively trying to exploit vulnerable code can be beneficial since it may allow other defenses to be deployed to protect applications.

The most common method [84,85] for detecting active cache-side channel attacks is to use the Performance Counter Monitor (PCM) and look for abnormal cache behavior. The PCM provides accurate measurements for recent architectural events such as the number of L1 cache hits/misses, L2 cache hits/misses, LLC cache hits/misses, page faults, etc. Most works using this technique apply strategies to identify when an attack is taking place, usually by looking at cache miss rates at various levels of the cache. By observing these counters, they are able to accurately detect attacks in progress.

Recent work [86] using Intel’s TSX to mitigate cache side-channel attacks also found that the same technique could be used to detect attacks in progress. This is because when they use a transaction to protect a sequence of instructions, they find that when the system is under attack the number of failed transactions significantly increases.
A User-Space Cache Side Channel Mitigation

4.1 Motivation

Many defenses have been proposed to mitigate cache-based side channel attacks, with the goal of eliminating or reducing the effects of sensitive data on cache. However, a big challenge is compatibility: it is difficult to protect vulnerable applications on commodity hardware/OSes. Many mechanisms [20, 54, 56–58, 63–67, 87, 88] require substantial changes to operating systems and/or hardware. Such mechanisms are appealing in the long term since vulnerable applications can be protected with few or no changes, but they are unlikely to be adopted in the near future. On the other hand, cryptographic libraries, common targets of cache attacks, typically use tricks (e.g., the scatter-gather technique) to thwart those attacks. However, those tricks require code-rewriting, and sometimes, a constant-time version requires significant changes compared to the more efficient implementations. For example, bit-sliced AES implementations [89, 90] rely on a circuit implementing the AES S-box, which diverges from table-based implemen-
tations significantly. These tricks are subtle to implement, and they are highly tailored for cryptographic algorithms; it is still challenging to protect other applications, or even a different implementation of a cryptographic algorithm.

Work by Gruss et al. [1] offers a promising application-level solution requiring few code changes. The mechanism utilizes hardware transactional memory (HTM) to ensure that all sensitive cache lines are “locked” into the cache when sensitive code blocks are executed. However, the approach relies on HTM, which is still absent in many processors (i.e. AMD processors). To avoid frequent transaction failures, the protected memory is limited to the size of CPU’s caches; for example, the write set of HTM is limited to the size of the L1 cache (typically several KB) [1]. Moreover, the overhead on memory-intensive applications can go as high as 30X, due to more frequent transaction abotions.

To address these pressing issues, in this chapter we present a novel application-level defense mechanism called Ghost Thread to mitigate cache-based side channels at the user-space level. Compared with existing defense mechanisms, Ghost Thread works on any commodity OS and hardware that supports concurrency. Moreover, it protects vulnerable applications without any code change except for a few library calls. Ghost Thread serves as a flexible library that protects accesses to the sensitive memory region by injecting random accesses to the same region. By doing this, an adversary now has to distinguish random behavior added by Ghost Thread from actual program behavior to launch a successful attack. While the overall idea seems simple, the insight behind Ghost Thread is that the overhead of using a concurrent thread to randomly pollute the cache will have limited impact on the total execution time of the protected application, particularly when the machine is not under heavy load. Meanwhile, Ghost Thread offers strong security assurances: we demonstrate that the number of samples needed to launch a successful attack grows exponentially with the number of injected accesses. Moreover,
Ghost Thread defends against various kinds of existing attacks, including attacks on both symmetric and asymmetric ciphers.

### 4.2 Attack Model

We consider the scenario where an adversary and victim share the same physical hardware, a common scenario today due to the prevalence of cloud computing. Furthermore, the adversary is able to be co-scheduled on the shared hardware with the victim. The adversary is capable of setting the CPU cache into a known state (e.g., flushing the L3 cache) before the victim process runs and query the cache state after. Hence, the attacks sketched in Section 6.3 are feasible. We assume that the shared cache between adversary and victim process (e.g., LLC cache) is also shared by Ghost Thread, but Ghost Thread works on any level of cache. By default, we refer to LLC when a concrete context is needed.

### 4.3 System Design

Ghost Thread works by inserting phony cache accesses to interfere with an adversary’s ability to learn the actual cache accesses from the victim. With Ghost Thread, an adversary now has to determine if the status of the targeting cache line was caused by the legitimate behavior of the victim process or a noise access made by Ghost Thread. Randomly adding data to the CPU cache can significantly limit an adversary’s ability to learn useful information. In this section, we discuss the design of Ghost Thread; we provide a formal analysis highlighting the added protection in Section 4.4.
4.3.1 Why a User-Level Library

While injecting random memory accesses should be effective at thwarting cache attacks, where it is implemented impacts the amount of incurred performance overhead, as well as the compatibility of the approach. One way is to rewrite the victim code to inline those random memory accesses. However, this would incur a significant performance overhead. For instance, we observed over 60% overhead on execution time when we
inlined 16 memory accesses in each round of encryption (160 total accesses) to protect accesses to the S-Box used by AES. One key insight of Ghost Thread is that we can take advantage of multiple cores in modern systems to inject phony cache accesses in a more efficient way. This results in virtually no overhead on systems that are not being fully utilized.

Another potential software-level solution we considered was to add noise using an operating system, in a similar manner as [54]. While this option may have resulted in more consistent results in some cases thanks to more control over process scheduling, it requires substantial modification of an OS. Ghost Thread is instead designed as a user-space library, allowing it to be easily integrated into an existing application on an unmodified OS and hardware.

### 4.3.2 System Overview

Figure 6.3 illustrates how a user interacts with Ghost Thread to protect a vulnerable application. In the first step, the user identifies what memory regions in the application need protection against cache attacks. These are called *sensitive memory regions*; if an attacker can directly observe what locations in these regions are accessed by the application, secret information can be inferred. In the second step, the user modifies the application’s source code with calls to the Ghost Thread library. Typically, she would insert calls to create protection threads before code regions that access sensitive data; these code regions are shown in orange in Figure 6.3. Depending on the security requirements of the application, there can be as many protection threads as needed; they can protect the same memory region or each can protect a separate memory region. For each protection thread, the user can decide to keep it running (shown in green), pause it

---

1We show in Section 4.3.5 how program locations and memory regions can be automatically identified to help users accurately use Ghost Thread’s interface.
Figure 4.2: An example showing how a user can modify an application to provide protection. The highlighted lines of code would be added by the user. Green indicates necessary code modification, yellow indicates code added to reduce overhead.

(Shown in yellow), unpause it, or stop it.

To provide a more concrete example of how Ghost Thread can be used, Figure 4.2 shows a typical use case using AES encryption as an example. In the example, the lines in green indicate necessary code changes to get protection from Ghost Thread. The lines in yellow indicate optional features that reduce Ghost Thread impact on the application’s performance. This program starts a protection thread to protect an encryption routine’s S-Box (Substitution box). This is performed by providing the base address of the S-Box and the size.

To facilitate better performance, the thread is paused after the calling of AES_decrypt. This frees the system’s processor that was running the protection thread. After performing some work on the plaintext and prior to performing the next AES encryption, the protection is unpause. After the application is finished using AES routines, the protection thread can be stopped to free all resources associated with that thread.

4.3.3 Library Interface

We first introduce the interface of Ghost Thread and its basic functionality. We then describe advanced features in Section 4.3.4.

```c
int tid = GT_add(&sbox, 256, sizeof(int));
plaintext = AES_decrypt(ciphertext, key);
GT_pause(tid);
process(plaintext);
GT_unpause(tid);
ciphertext = AES_encrypt(plaintext, key);
GT_stop(tid)
```
Creating a Ghost Thread  In order to create a protection thread, a user should call the following function provided by Ghost Thread:

```c
u32 GT_add(void *base_addr, u32 num_elmts,
        u32 element_size,
        bool priority, u32 freq)
```

This function takes five parameters and returns a thread ID used internally to manage threads by Ghost Thread. The first three determine the sensitive memory region to be protected. Ghost Thread uses the base address of the memory region, the number of elements, and the elements’ size to determine the sensitive memory region. The other two parameters (priority and freq) will be discussed shortly. An example use of `GT_add` is shown in Figure 4.2. By default, the frequency will be set to max ensuring optimal security. The priority of the thread is set to the maximum available to the application; for example if run as the root user on Linux the thread would have real-time priority by default.

Each protection thread in Ghost Thread is given a contiguous range of memory locations to protect. This design decision facilitates fast injection of noise memory accesses as it benefits from hardware prefetchers, which bring in more than one protected cache lines for each injected noise access.

Pausing and Unpausing Protection  It is often the case that the majority of an application does not need protection against side-channel attacks. As shown in Figure 4.2, there is likely going to be time when protection is unnecessary. To accommodate this, Ghost Thread can be paused and unpaused throughout an application’s execution. Ghost Thread offers the following two functions to pause and unpause ghost threads:

```c
GT_pause(u32 tid)
GT_unpause(u32 tid)
```
Each function takes the thread ID of the protection thread to be paused/unpaused. Semantically, pausing a protection thread puts it to sleep. This allows the OS to schedule other tasks without having to schedule the protection thread. With the function call, Ghost Thread looks up the thread internally and puts the desired thread to sleep, as illustrated in Figure 4.2 on line 3. Similarly, the unpausing function wakes up the thread with thread ID $tid$ and resumes injecting noise.

In order to ensure that noise is currently being injected by a protection thread, the unpausing function also checks if the protection thread has completed all of its initialization and is actively adding noise to the cache. Doing so ensures that sensitive memory accesses after the unpausing function are properly protected by Ghost Thread.

**Stopping Protection**  When there is no longer a need to protect a specific memory region, a user should call the following function provided by Ghost Thread:

```
GT_stop(u32 tid)
```

This function allows the thread to terminate and return its resources to the OS. An example can be found on line 7 of Figure 4.2 after the program has completed all of its accesses to the S-Box.

To summarize, Ghost Thread’s main functionality is composed of 4 methods: GT_add, GT_pause, GT_unpause, and GT_stop. To reap the protection benefits of Ghost Thread, a user only needs to add one line of code to their existing application for each region that needs protection. The remaining features are optional, but can significantly enhance application performance by freeing system resources.
4.3.4 Tuning Ghost Thread

For advanced developers, Ghost Thread provides many features that allow them to tune the protection for desired levels of security and performance. The remainder of this section will highlight these features and explain how and why they should be used.

4.3.4.1 Multiple Protection Threads

For some applications, a single protection thread may not sufficiently address their security requirements. For instance, when an application has a large sensitive memory region, a single thread may not be able to inject sufficient noise memory accesses. As a second example, an application may have multiple non-contiguous sensitive memory regions. In both situations, Ghost Thread allows users to create multiple protection threads, which inject noise accesses independently and concurrently.

Multiple protection threads are simply created by multiple calls to the $\text{GT}\_\text{add}$ function. To protect multiple non-contiguous memory regions, multiple protection threads should be created with non-overlapping memory regions. Additional threads with the same memory regions can also be created to increase the injection rate of noise; this provides additional security if required, as shown in Section 4.5.

4.3.4.2 Ensuring Thread Scheduling

Ghost Thread is implemented as a user-space library without any modification to the OS including its scheduler. Modern OSes use schedulers that attempt to fairly allocate CPU time to all processes waiting to run. This means the threads that Ghost Thread uses to protect an application may not be scheduled when the protected application is running. The situation becomes more likely when a system is more utilized since the scheduler becomes more relied upon to allocate CPU time.
To overcome this issue, Ghost Thread provides an option to set the priority of ghost threads. This is the fourth parameter, `priority`, of the `GT_add` function. By default, Ghost Thread sets the priority of a protection thread to be real time. In this case, the priority parameter to `GT_add` is set to `true`. Line 1 of Figure 4.2 shows how the priority of a protection thread is set to the default value of `true`. When a real-time protection thread is created, Ghost Thread requests the OS to create a thread with corresponding scheduling priority.

Most modern OSes support some notion of real-time scheduling. On Linux, real-time tasks can be preempted only when a task with higher priority is ready to run. By giving a ghost thread the maximum priority (99 in Linux), no other tasks can preempt it. By further setting the CPU core affinity of the thread to a specific core, Ghost Thread prevents the Linux scheduler from migrating the real-time thread to another core. After these steps, protection threads using real-time priority are scheduled on specific cores until they voluntarily give up their CPU time. This guarantees that ghost threads are running at the same time as the application they need to protect.

We empirically verified that real-time protection threads are always scheduled. We used the Linux Performance (Perf) Tools to monitor what tasks were scheduled on each core. Perf Tools sample what is running on every processor at fixed units of time. In our experiments, we set the priority of a thread to be real time and checked the result of the Perf Tools. We found that once a real-time thread is scheduled, it was never descheduled until it terminated.

Optionally, a ghost thread can be set to regular priority by setting the priority parameter of `GT_add` to `false`. Although changing the priority of a thread to be regular priority weakens the protection, it could still be desirable in the scenarios where the root privilege is unavailable, or when the user is willing to sacrifice security for better performance. Moreover, adding multiple protection threads with regular priority can
Figure 4.3: The number of threads required to obtain 99.9%, 99%, 95%, or 90% overlap with a protected thread, when various background tasks are running at the same time on a 6 core machine. These values assume that the scheduler assigns every task with the same probability of being scheduled.

potentially offer similar level of protection as one real-time ghost thread.

Assuming a fair scheduler that schedules every thread equally, we can compute how likely at least one of the protection threads runs concurrently with an application that has just one thread. Figure 4.3 shows how many threads are required to obtain either 99.9%, 99%, 95%, or 90% overlap with the application thread, when the system is under various amounts of strain on a machine with 6 cores. The results show that on a system with a light load, just one protection thread is typically sufficient, even if the ghost thread is not real-time. As more background tasks are competing for the processors, the number of protection threads needed for a target overlap increases. The higher the percentage of overlap, the faster the number of threads required increases. But still, to offer 90% overlap, only a few regular priority threads are needed with heavy system load. We show in Section 4.5 how we can use these percentages to compute more accurately the protection added by Ghost Thread.
In sum, ensuring at least one protection thread is actively running when the protected application is running is a critical aspect of Ghost Thread. Ghost Thread offers two possible solutions to this problem. The ideal (and default) solution is to assign the real-time priority to the protection threads to ensure they are always running. Since this may not always be possible due to requiring root privileges, we also provide some guidance for users to obtain a target percentage of overlap by introducing multiple protection threads to ensure at least one of them is running when the application is running.

4.3.4.3 Varying Noise Injection Rate

By default, a protection thread in Ghost Thread injects memory accesses as quickly as possible in a tight loop. To achieve better flexibility between security and performance, Ghost Thread allows a user to configure the rate at which noise is injected.

Ghost Thread provides an option of setting the noise injection rate. This is set as the fifth parameter freq of the GT_add function. An example is shown at line 1 of Figure 4.2, where freq is set to the default value 0. When freq is of a special value 0 (the default value), the thread injects noise memory accesses at the maximum rate; the maximum frequency we observed on our machine is about 470 million noise accesses per second. For other target frequency numbers, Ghost Thread calculates appropriate n (the length of a sequence of noise memory accesses in an iteration) and t (the length of sleep). In order to ensure consistent noise, our implementation always sets n to be a constant (usually 1,000,000). Based on that gap, Ghost Thread computes the amount of sleep (t) per iteration needed in order to achieve the desired frequency. With computed n and t, a ghost thread (1) injects a sequence of n noise memory accesses, and (2) puts the thread to sleep for time t.
4.3.4.4 Memory Access Randomization

An important goal of Ghost Thread is that its protection threads inject noise accesses randomly to sensitive memory regions. Currently, Ghost Thread supports four different methods for obtaining randomness. Each of these methods can be selected by passing compiler options when building the Ghost Thread library.

The first method is to precompute the random numbers before the Ghost Thread’s protection threads begin injecting noise. This can either be performed at run time during thread initialization, using one of our supported random number generators, or added by the user using their preferred random number generator. We also allow these numbers to be added at compile time. Adding the numbers at compile time can be significantly faster since there would be no overhead incurred during the execution of the application.

The three other supported methods generate random numbers as needed using different random number generators. Generating random numbers along with noise injection can be more versatile since it can be done on demand and takes up significantly less memory compared to precomputing. This versatility comes at a cost to security, since it takes longer to generate random numbers compared to looking up from a precomputed random-number table. Table 4.1 shows the security impact of random number generation techniques; for each method, it shows the maximum number of memory accesses per window, where a window is 2,000 CPU cycles.

<table>
<thead>
<tr>
<th>Noise/Window</th>
<th>Precomputed</th>
<th>rand</th>
<th>PCG</th>
<th>RDRAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>467</td>
<td>32</td>
<td>124</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The average number of memory accesses that a ghost thread can inject in a window of 2,000 clock cycles.

The first option is the rand function in glibc, we used version 2.19. It uses a linear congruential generator (LCG) to produce numbers. This technique is not a secure method for generating random numbers but is considered to be fast [91].
that using rand allows for about 32 memory accesses on average in a typical observation window of cache-based side channel attack (around 2000 clock cycles).

The second option is a permuted congruential generator (PCG) [91]. This technique is not considered secure either but is significantly more challenging to break than numbers generated by an LCG algorithm [91]. We see from Table 4.1 that the PCG algorithm is able to obtain significantly more accesses than glibc’s rand function, but is still only about 25% as effective as using a precomputed list of numbers.

The last option is Intel’s RDRAND instruction [92]. Intel suggests that RDRAND is a hardware random number generator that is secure enough to be used in cryptographic applications [92]. Some results indicated that RDRAND did not provide a perfect secure solution to random number generation, but were unable to provide a practical attack against it [93]. RDRAND is able to generate the most secure random numbers among the three options. However, we see that this security comes at the cost of taking a significantly longer time to generate each random number, as shown in Table 4.1.

In sum, we suggest using a secure random number generator to create a table prior to using Ghost Thread (the default setting of Ghost Thread). This provides the best security and performance. With a strong attacker model where the adversary may observe the cache status in every 2,000 clock cycles (i.e., when the window size is about 2,000 cycles), the security of other random number generators could be insufficient due to the few number of noise memory accesses being injected.

### 4.3.5 Usability of Ghost Thread

While the default configurations offer the best security of Ghost Thread, the advanced configurations such as priority, noise frequency and randomness mechanism are designed for more capable users to tailor Ghost Thread to their needs.
To better facilitate a non-expert user to adopt Ghost Thread, Ghost Thread uses a static taint analysis to mark the sensitive memory regions as well as insert pausing and unpau sing statements. The taint analysis only requires users mark the sensitive data in their application; it then automatically identifies two vulnerable code patterns: tainted branches and tainted memory accesses, following the constant-time programming principle [80].

When those patterns are detected, the taint analysis reports the corresponding source code line numbers, which directly lead to the insertion of pausing and unpau sing statements. Tainted memory accesses also directly lead to memory regions to be protected. The only subtle case is tainted branches, which to be sound, require any memory region being accessed under the branches to be protected. Ghost Thread currently requires manual inspection to identify memory regions under tainted branches to be protected; in our experience, such manually work is typically straightforward.

We use an implementation of AES from libgcrypt 1.8.3 as a concrete example. With private key marked as tainted, the taint analysis marks all tainted memory accesses to the S-box; the marked code corresponds to the following pseudo code:

\[
ciphertext = s\_box[key \oplus plaintext];
\]

Thus, the taint analysis automatically identifies both the code region (the encryption routine that performs the sensitive accesses) and memory regions (the S-box table) to be protected.

\[2\text{In the future, we might also leverage existing static analysis tools [13–15, 22, 71] to do so in a more precise way.}\]
4.4 Security Analysis

Injecting noise typically does not eliminate a cache side channel. In this section, we show both in theory and practice that Ghost Thread effectively makes realistic side-channel attacks impractical. We break our analysis up into two components: (1) How much harder does Ghost Thread make the adversary’s task? (2) Does Ghost Thread thwart existing side-channel attacks?

4.4.1 Normalized Samples

We measure the attack difficulty with Ghost Thread by how many more samples are needed to successfully launch a cache attack on a vulnerable program protected by Ghost Thread. To accomplish this, we measure normalized samples, the ratio of samples needed to achieve the same level of confidence of retrieving the secret correctly.

Existing attacks typically observe the existence of accesses to sensitive memory regions to reveal private keys. This is done via exploiting the access time of a cache line being probed: a cache line being accessed in a window likely introduces a smaller latency compared with the case where it is absent. Intuitively, Ghost Thread works by shifting the access time when an access is absent to the case where it is accessed in a window. In this section, we follow an analytical model used in prior work [57, 94–96] to quantify the number of samples needed to launch a successful attack.

We first abstract the execution time under the condition of at least one access and no access in a window as $\mu_1$ and $\mu_2$ respectively:

$$\mu_1 = P_1 t_{hit} + (1 - P_1) t_{miss}$$

$$\mu_2 = P_2 t_{hit} + (1 - P_2) t_{miss}$$
where $t_{hit}$ and $t_{miss}$ are the (expected) probing times when the cache line being probed is present and absent in the cache respectively; $P_1$ (resp. $P_2$) are the probability that a cache line remains in the cache at the probing time when the cache line is accessed (resp. absent) in a window. Following prior analytical models [95, 96], the number of measurements required for a successful attack (i.e., one that distinguishes samples from either $\mu_1$ or $\mu_2$ with high confidence) can be estimated as

$$N \approx \frac{2Z^2_\alpha \sigma^2}{(P_1 - P_2)^2(t_{miss} - t_{hit})^2}$$

where $Z_\alpha$ is the quantile of the standard normal distribution for $\alpha$ (i.e., the desired success rate of an attack) and $\sigma$ is the variance of probing time.

**Samples needed without Ghost Thread** Due to background noise in a window, there is a small chance that an accessed cache line is evicted or an unaccessed one is cached. To simplify the model, we ignore such noise and estimate $P_1$ as 1 and $P_2$ as 0. Note that this is justified since it makes our estimation more conservative (i.e., it is in favor of the attack). Hence, we have $N_{w/o} = \frac{2Z^2_\alpha \sigma^2}{(t_{miss} - t_{hit})^2}$.

**Samples needed with Ghost Thread** With Ghost Thread that injects $\delta$ random accesses in one window, $P_1$ remains 1, but $P_2$ significantly changes compared with the previous case: there is a $1 - (\frac{M-1}{M})^\delta$ chance that the cache line is brought into the cache even though it is absent from the original program. Hence, we have: $N_{w/} = \frac{2Z^2_\alpha \sigma^2}{(\frac{M-1}{M})^{2\delta}(t_{miss} - t_{hit})^2}$.

**Normalized Samples needed with Ghost Thread** With Ghost Thread, it is more meaningful to measure $\frac{N_{w/}}{N_{w/o}}$ for the added security to the protected program. We call
Table 4.2: Normalized samples with various amounts of noise accesses for AES and RSA.

<table>
<thead>
<tr>
<th></th>
<th>( \alpha )</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>467</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES (M=16)</td>
<td>( \delta )</td>
<td>4.03E5</td>
<td>1.63E11</td>
<td>6.56E16</td>
<td>2.64E22</td>
<td>1.51E26</td>
</tr>
<tr>
<td>RSA (M=64)</td>
<td>( \delta )</td>
<td>2.33E1</td>
<td>5.44E2</td>
<td>1.27E4</td>
<td>2.96E5</td>
<td>2.44E6</td>
</tr>
</tbody>
</table>

Figure 4.2 shows the normalized samples needed for table-based implementation of AES (with 16 sensitive cache lines) and RSA (with 64 sensitive cache lines). We emphasize that the noise added by Ghost Thread significantly improves the security of the vulnerable programs: the number of samples needed for a successful attack grows exponentially. In a conservative setting of windows with a size of 2,000 cycles for asynchronous attacks, Ghost Thread injects 467 noise cache accesses (Table 4.1). Hence, both attacks quickly become infeasible: over 100,000X more samples are needed to launch a successful attack. Even for a synchronous attack on AES with a small window size of 520 cycles [33], Ghost Thread can inject over 100 noise cache accesses, requiring over 100,000X more samples to launch the attack on AES. This is for just a single Ghost Thread active, multiple threads can be used to increase security even further.

Security against cross-processor attacks  We note that Ghost Thread requires that the cache being observed by the adversary be shared by the protected application and the protection threads spawned by Ghost Thread for this analysis to be applicable. In the common scenario when the LLC is being observed, the protection thread and protected application simply need to be running on the same processor since the LLC is usually
shared among all cores. For cross-processor attacks, such as those shown in [97], we can apply the same analysis described in this section when either the processor interconnects all have the same memory access latency, or that Ghost Thread’s protection thread is always scheduled on the same processor as the protected application. This is the case because cross-processor attacks apply the same principles of an intra-processor attack, but also leverage the high-speed processor interconnects between processors to detect the memory access time differences between cache hits and misses.

4.4.2 Empirical Study on Cryptography Libraries

Previously, we have shown the theoretical attack difficulty with Ghost Thread. Next, we use Ghost Thread to protect two representative vulnerable cryptography libraries against real cache attacks. We show that Ghost Thread completely breaks those attacks even in the face of strong attack models. The first attack, following the attack in [5], is a synchronous attack targeting the first round of AES in two commonly used implementations (mbedTLS and libgcrypt). The second attack, following [27], is an asynchronous attack targeting the square and multiply modular exponentiation routine of RSA found in GnuPG 1.4.23. To briefly summarize our findings, we show that Ghost Thread can successfully prevent this attack as well, even though it targets the instruction cache.

4.4.2.1 AES Attack

We follow the attack that targets the first round of AES in [5]. The pseudo code of the attack for revealing one key byte is shown in Figure 4.4. Note that this is a synchronous attack where the attacker is able to collect the cache state right after the first round of encryption; doing so makes the attack feasible even if only a very small number of samples are collected. Nevertheless, we show that Ghost Thread successfully thwarts
count = [0 for x in range(256)]
for i in range(numPlaintexts):
    plainTexts[i] = genRandPlaintext()
    usedCaches[i] =
        observeEncrypt(key,plainTexts[i])
for keyByteGuess in range(256):
    for i in range(numPlaintexts):
        if guessCacheLine(keyByteGuess,plainTexts[i])
            in usedCaches[i]:
            count[keyByteGuess]++
    return maxIdx(count)

def guessCacheLine(guess,plaintext):
    return (guess ^ plaintext) / 16

Figure 4.4: Pseudo code of the AES attack.

such attacks.

In this attack, the attacker first encrypts a number of known plaintexts (numPlaintexts) and collects the resulting cache states right after the first round (usedCacheLines). After that, the attacker enumerates over 256 possible values of a certain key byte, and sees which values best “matches” the resulting cache states.

To do that, recall that the index into the S-Box used by AES in the first round is \( p_i \oplus k_i \) where \( p_i \) is the \( i \)th byte of plaintext and \( k_i \) is the \( i \)th byte of the key. Hence, for each sample \( i \), the attacker can successfully compute the index by using the known plaintext plainTexts[i] and the guessed key value. Hence, the key value that resulted in the most correct cache line guesses (stored in array count) is the key byte used in the encryption.

We performed this attack on three different machines running different Intel chips (i3, i5, and i7). For brevity, we present the results on the Intel i7 chip since all 3 experiments are very similar. The result for the unprotected program is shown at the top of Figure 4.5. Here, each blue dot corresponds to the percentage of correct cache line guesses for the corresponding guessed key byte (the x-axis); the red triangle identifies
the actual value of the key byte used in these experiments (which is set to be 46). Without Ghost Thread, the real value of the key byte is easily identified among 16 guessed values (sharing the same cache line) that stand out.

In order to see how effective Ghost Thread is, we ran the attack again with the AES program protected. The results are shown at the bottom of Figure 4.5. With Ghost Thread, the real value of the key byte can no longer be identified. This is due to the fact that Ghost Thread injects hundreds of extra memory accesses, making all cache lines being cached in the experiments.
(a) Flush+Reload attack against RSA in GnuPG 1.4.23. The dots under 150 clock cycles indicates a cache hit; those dots clearly leak the sequence of corresponding function calls, which reveals the entire RSA privacy key.

(b) Flush+Reload attack against RSA in GnuPG 1.4.23 protected by Ghost Thread. The dots under 150 clock cycles indicates a cache hit; those dots no longer leak the sequence of corresponding function calls.

Figure 4.6: RSA Attack without and with Ghost Thread.

4.4.2.2 RSA Attack

Modern processors generally have a unified L2 and L3 cache containing both data and instructions. We take advantage of this fact and show that Ghost Thread can also protect programs from attacks probing the instruction cache.

To demonstrate this, we perform an asynchronous Flush+Reload attack as described in [27]. This attack attempts to infer the order of functions that are executed in a modular exponentiation routine in RSA. In particular, the functions of interest are mul (multiplication), sqr (square) and mod (modulus). When an adversary learns the order of those functions during execution, they can infer all the bits of the private key in RSA [27].

In a nutshell, the attack works by monitoring three instruction cache lines, each associated with one of the three functions of interest. Consistent with [27], we find that

---

The attack targeting the first round of AES cannot distinguish the exact key value since 16 key values are stored on the same cache line. However, this attack is much easier to launch compared to the complete attack in [5]; thwarting this version also prevents real attacks with more system noise and more bits to learn.
a window of 2,000 cycles works well to allow enough time to flush the cache, wait for the victim to access the cache, and record cache accessing time. The recorded access times of each function’s selected cache line are shown in Figure 4.6a. We note that under 150 clock cycles (a threshold when a memory address is cached vs. in RAM on our machine), there is a clear sequence of cache line accesses caused by corresponding function calls. Following this sequence for the entire modular exponentiation routine would reveal the entire bits of a private key in RSA.

We applied Ghost Thread to the same program and created one protection thread for each function of interest. Since the adversary can target any cache line associated with any instruction within each function to attack, we protected all instruction addresses within each of the functions. Figure 4.6b shows that after Ghost Thread was applied, the attacker was no longer able to infer the sequence of function calls. This is the case because all but 3 of the memory accesses (in this sample) are cached, meaning that it is not possible to accurately infer the sequence of the three functions. This example demonstrates Ghost Thread’s effectiveness and versatility at protecting a moderate sized chunk of memory (in this case, 2KB of instructions) against cache-based side channel attacks.

### 4.5 Evaluation

In this section, we answer the following questions: (1) How much overhead does Ghost Thread have on a protected application? (2) What is the maximum overhead an application can incur when being protected by Ghost Thread? (3) What are the normalized samples needed for a successful attack after applying Ghost Thread (with various settings) to an application? (4) What is the impact of using Ghost Thread on other applications running on the machine?
To evaluate Ghost Thread, we use a machine with 32 GB of RAM on an Intel Core i7-5820K CPU at 3.3 GHZ using Ubuntu 14.04. For consistent results, we disabled hyper-threading and turbo boost.

4.5.1 Application Overhead

A key aspect of our evaluation is understanding how much overhead Ghost Thread adds to applications. To explore this, we applied Ghost Thread to an Apache web server.

Setup In our experiments, we analyze the throughput (i.e. the number of requests per second) of an Apache web server with various numbers of concurrent users. We use an HTTP server benchmarking tool called Siege [98] to load a 2KB static web page using HTTPS for differing numbers of concurrent users; each configuration is repeated 100 times to reduce noise. We follow the experimental setup in [99]. There are two security-sensitive components of the Apache web server: an RSA routine is invoked to establish connections, and an AES routine is used to encrypt the transmitted data.

We apply Ghost Thread to protect the T-table implementation of the AES routine, requiring about 4KB of memory to be protected. Ghost Thread is also applied to the precomputed table (1KB) used by the RSA routine, which was shown to be vulnerable to cache attacks [100]. We use the taint analysis described in Section 4.3.5 to determine appropriate points to add the instrumentation.

Results We measured the throughput overheads with various concurrent users ranging from 1 to 150 where each user requests a webpage of size 3KB. The results are shown in Figure 4.7 with two possible configurations: using real-time ghost threads, or normal-priority threads. The results demonstrate that in either case, Ghost Thread adds little to no overhead to the Apache web server regardless of the priority Ghost Thread used.
Figure 4.7: The throughput overheads of concurrent users accessing a webpage hosted on an Apache server with both the AES and RSA routines protected by Ghost Thread.

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloak [1]</td>
<td>-0.8%</td>
<td>248%</td>
</tr>
<tr>
<td>Ghost Thread</td>
<td>-1.25%</td>
<td>20.1%</td>
</tr>
</tbody>
</table>

Table 4.3: Overheads of Cloak [1] and one single Ghost Thread.

This is because Ghost Thread only needs to be active during critical sections of the application where secret data is being processed.

When Apache serves webpages, it spends 0.19% of the execution time on AES encryption, 0.23% on RSA encryption, for a combination of 0.42% of execution time that requires protection. This is consistent with our belief that programs used in practice likely require Ghost Thread to be active for small portions of their execution.

Comparison with Cloak [1] A comparison between state-of-the-art user-space mitigation mechanism Cloak [1] and a single non-real-time Ghost Thread is shown in Table 4.3. Since our system does not support TSX, we use the results reported in [1]
for comparison. Although the experimental environments are not completely identical, we note that both best cases reflect light system workloads. Compared to Cloak [1], Ghost Thread preforms comparably. Cloak reports a 0.8% speed up when protecting AES from side channel attacks due to a higher cache hit rate and few memory accesses. We observe the same in Figure 4.7: Ghost Thread also has near zero overhead due to a higher cache hit rate and limited required resources of Ghost Thread since the region that requires protection in Apache is not large.

Figure 4.8: The latency overheads of Ghost Thread on an AES implementation from libgcrypt 1.8.3 with various configurations. The boxes show the additional normalized samples needed.
Figure 4.9: The latency overheads of Ghost Thread on an RSA implementation from GnuPG 1.4.23 with various configurations. The boxes show the additional normalized samples needed.

### 4.5.2 Worst Case Analysis

To protect a vulnerable application, Ghost Thread typically does not need to be active throughout the entire execution of an application. However, it is also useful to measure the maximum overhead an application can incur: both to understand the limit of Ghost Thread as well as how parameters such as thread priority and noise rate affect performance and security. Next, we conducted experiments on AES and RSA where Ghost Thread is active 100% of the time.

**Setup** We ran AES using an implementation in libgcrypt 1.8.3 with 128-bit keys protecting the S-boxes (1 KB). We also applied Ghost Thread to the precomputed tables...
(4 KB) in the modular exponentiation routine of GnuPG 1.4.23 used by the RSA algorithm. We again used the taint analysis presented in Section 4.3.5 to identify where to insert Ghost Thread instrumentation. We used the corresponding AES and RSA implementations without Ghost Thread protection as the baseline. To get consistent results, we recorded the execution time of encrypting 5 million messages.

To fairly evaluate Ghost Thread under various system loads, we considered the percent increase in execution time of an AES application with and without Ghost Thread while other background processes are competing for resources. For the latter, we used two programs from SPEC CPU 2006: bzip2 and gobmk. We selected these two programs since bzip2 represents a memory-intensive program and gobmk represents a CPU-intensive program. Another scenario we used in our evaluation is when there is a mix of different kinds of programs on the system. To do this, we ran all of the integer benchmarks from SPEC CPU 2006 written in C.

The various background applications resulted in similar results, within measurement error. Thus we show only the results using bzip2 as background programs hereafter. We presume this is the case since Ghost Thread only protects a small region of memory: the background application is unlikely to change the victim program’s cache hit rate significantly.

4.5.2.1 Performance Results

The latency overheads added by Ghost Thread to AES and RSA respectively with various system loads and Ghost Thread configurations are shown in Figure 4.8 and Figure 4.9 (ignore the boxes for now). The results indicate that adding memory accesses to protect against cache side channels does not add noticeable overheads when a system is being under utilized (i.e., having fewer tasks than cores.).

We note that when Ghost Thread is active 100% of the time and all cores are oc-
cupied, the performance overhead is non-negligible both for AES and RSA. This is expected given that Ghost Thread competes for computation resources. However, we note that the worst-case overhead ranges between 13% to 35%, depending on the configuration of Ghost Thread. Compared Cloak [1], this overhead is reasonable since their worst case overhead was shown to be up to 3078% for memory intensive tasks. Moreover, as we show in Section 4.5.1, the overhead could be much smaller in more common scenarios where only part of an application requires protection.

Figure 4.8 further shows the impact of different configurations of Ghost Thread: a normal-priority protection thread, a real-time priority protection thread, and two normal-priority protection threads. All protection threads inject noise to the memory region that holds the S-Box for AES or the precomputed tables used in modular exponentiation for RSA.

When the priority of the thread is set to real time, the execution time is similar to a thread with normal priority until the system becomes fully utilized. This is due to the fact that the OS is unable to schedule anything on the core the real time thread gets assigned to. This causes a larger amount of overhead (about 20%) compared to the case of using a normal-priority thread (about 10%). We also observe that creating two normal-priority threads incurs the largest latency overhead in both cases.

**Comparison with Cloak [1]** While the overhead of Ghost Thread is non-negligible in the worst-case scenario, we note that it is still reasonable compared with the state-of-the-art of user-level defence mechanism: Cloak was shown to have a worst case overhead of 248% on average in a heavily loaded system [1]. As shown in Table 4.3, the worst-case overhead of Ghost Thread (20.1%) is still reasonable and in fact, an order of magnitude better.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time w/o GT</th>
<th>Time w/ RT GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES [101]</td>
<td>0.064s</td>
<td>1E17 years</td>
</tr>
<tr>
<td>AES (Cross-VM) [102]</td>
<td>4.5s</td>
<td>1E19 years</td>
</tr>
<tr>
<td>RSA [100]</td>
<td>30 min</td>
<td>3170 years</td>
</tr>
</tbody>
</table>

Table 4.4: A summary of how long it takes to collect sufficient number of samples on our test machine to launch the corresponding cache attacks without and with Ghost Thread.

### 4.5.2.2 Security Results

While Ghost Thread is able to obtain negligible (when the system is under loaded) to modest (when the system is fully loaded) runtime overheads, it provides strong security for all system loads. We find that Ghost Thread is able to make side-channel attacks infeasible to launch.

We use normalized samples presented in Section 4.4 as a metric for measuring security. During experiments, we recorded the number of injected cache accesses per window and the percentage of the time when the protected program and Ghost Thread execute simultaneously. To handle non-real-time protection, we extend the analysis in Section 4.4 in the following way. For protection threads that are not real time, $P_2$ in the analysis is multiplied by the percentage of time Ghost Thread and the program co-execute. The order of magnitude of the normalized samples (e.g., 1E26 represents $10^{26}$) is shown in the boxes near data points in Figure 4.8 and Figure 4.9.

The results indicate that while Ghost Thread cannot completely eliminate cache side channels, it is infeasible to attack protected applications when we use real-time ghost threads, the default configuration of Ghost Thread. This is true regardless of the system load: it will require $10^{26} \times$ samples to perform the same attack on AES. For RSA, $10^6 \times$ samples are needed to perform the same attack. To better connect normalized samples to attack difficulty (in terms of computation time), we collected the time to generate sufficient number of samples to successfully launch state-of-the-art attacks, as summarized in Table 4.4, under column “Time w/o GT”. In order to compute the time needed
to launch the same attacks with Ghost Thread, we use normalized samples to compute the corresponding time (assuming the same computation resource), shown in Table 4.4, under column “Time w/ RT GT”. We see that for AES attacks, it will take over 1E17 years to complete, that is longer than the current age of the universe. For a more realistic attack scenario spanning across VM boundaries, we see the expected computational time is even worse, requiring at least 1E19 years of time to compile enough samples. Although the normalized samples is much smaller in the RSA case, due to the longer attack time on RSA, it still requires over 3,000 years to obtain enough samples.

When one normal-priority ghost thread is used, the overall security is also affected by system load: when the system is under utilized, we observe the same level of protection as real-time ghost threads. But the normalized samples drop when the system is fully utilized. This is due to the fact that a normal-priority thread injects less noise due to scheduling, but still requires over 1 million times more samples (resp. 100 times more samples) to have the same success rate in the AES case (resp. RSA case). For AES, it takes approximately 3,000 years to collect a sufficient number of samples. While normal-priority ghost threads offer weaker security compared with real-time threads, they may be preferable when the reduced security is acceptable considering the gained performance.

Ghost Thread also provides the capability to add multiple threads to improve security. By using multiple protection threads more noise can be added per window and also increases the likelihood at least one protection thread is concurrently executing with the protected application. Our data indicates that adding one additional protection thread requires $10^{50} \times (10^{12} \times$ for RSA) more samples compared to not having protection. This is an enormous amount of samples. Even when the system is loaded the additional protection thread performs between a single default priority thread Ghost Thread and a real-time thread requiring $10^{14} \times (10^{4} \times$ for RSA) more samples. This many more sam-
Figure 4.10: How adjusting the frequency impacts the execution time and leakage of an libgcrypt’s AES implementation. Due to space constraints, we only show results for up to 9 background processes.

amples will take over 200,000 (0.6 for RSA) years to collect using the same method used to compute the data in Table 4.4.

4.5.2.3 Adjusting Noise Frequency

Next, we evaluate the impact of noise injection rate on performance and security. The motivation is that for users who can tolerate weaker security, Ghost Thread can inject fewer noise accesses to reduce the performance cost. In the other direction, users can
strengthen security by adding additional protection threads.

**Setup**  This experiment is configured the same as in Section 4.5.2 with the addition of samples showing the performance and security of AES protected with Ghost Thread configured for various noise injection frequencies. Each of the Ghost Threads used have normal priority.

**Results**  Figure 4.10 shows the overheads of four different thread frequencies along with the normalized sample security metric. The metric shows the magnitude, so a result of $1E5$ means it requires $10^5 \times$ samples to launch a cache-based side-channel attack. The result indicates that lowering the number of memory accesses tends to lower the overhead of Ghost Thread. For instance, with 50 million accesses per second by Ghost Thread, the overhead with 6 background processes is 11% better than when the frequency is set to max (around 471 million accesses per second).

However, lowering the frequency also can significantly reduce the security improvement. When the protection thread is set to its maximum frequency, we see gains of up to 26 orders of magnitude. This security improvement decreases over 20 orders of magnitude when the frequency is lowered and the system is being under utilized. We see a decrease of up to 10 when the system is fully utilized. We also note that the rate of 150 Million offers a reasonable balance between security and performance in this case.

4.5.3 **System Impact**

Another concern to users is the impact of Ghost Thread on other programs running at the same time. There are two main reasons Ghost Thread could cause an increase in execution time for other applications. The first is increased contention for the cache, and the second is contention for CPU cores.
Figure 4.11: Percent overhead on bzip2 when a program protected by Ghost Thread is running concurrently.

Figure 4.12: The percent overhead on gobmk from the SPECCPU 2006 benchmark when a program protected by Ghost Thread is running concurrently.
To evaluate the impact on other applications, we ran bzip2 and gobmk from SPEC2006 while running AES encryption in a loop with and without Ghost Thread. Ghost Thread is active for 100% of the duration of the execution of bzip2 and gobmk to demonstrate the worst-case overhead to the system. We selected those two programs since bzip2 represents a memory-intensive program and gobmk represents a CPU-intensive program.

The results with bzip2 and gobmk is shown in Figures 4.11 and 4.12 respectively. The data indicates that the overhead on other applications is again not significant until all of the CPU cores are being used. After all of the cores are being utilized, the overhead increases to its peak of about 16%, and then slowly declines as more processes are added. This experiment also shows that there is not a significant impact caused by Ghost Thread’s added memory accesses since on an under-loaded system, there is no noticeable increase to the overhead. This is likely due to the fact Ghost Thread only accesses a small region of memory addresses to provide protection.

Note that since Ghost Thread is protecting the encryption the entire time, this experiment shows the worst-case overhead of Ghost Thread. As shown at the beginning of this section, applying Ghost Thread to applications in practice only requires protection for a fraction of the program’s total lifetime. This will significantly reduce its impact on other applications running concurrently.

4.6 Discussion

The experiments in Section 4.5 were conducted on a machine with a processor containing 6 cores. We note that since Ghost Thread utilizes additional CPU resources, the overhead added is proportional to the number of CPU cores available on a system. Thus, less powerful machines with less cores will see a higher overhead; a more powerful machine with more cores (e.g., cloud servers, a very important target of cache attacks) will
see less overhead.

We note that although Ghost Thread is implemented as a shared library, the only data flowing to Ghost Thread from the application is the memory region to be protected along with the frequency and priority of the thread. Since the memory accesses made to the memory region by Ghost Thread are random, an adversary cannot gain any additional information by virtue of Ghost Thread being a shared library that is not already assumed to be public (e.g., the memory location of the S-Boxes for AES).

4.7 Summary

In this chapter, we introduced a novel application-level defense mechanism called Ghost Thread to mitigate cache side channels. Compared with existing defense mechanisms, Ghost Thread is applicable to virtually any platform that supports multi-threading. The following is a brief summary of the highlights of this chapter:

- We described a new user-space cache-based side channel mitigation called Ghost Thread. It offers low overhead and strong security assurance in practice without any changes to existing hardware or the operating system. Ghost Thread achieves this by randomly injecting phony cache accesses to a sensitive memory region, making it difficult for an adversary to distinguish real memory accesses from those memory accesses injected by Ghost Thread.

- We presented a security analysis that quantifies the security Ghost Thread can provide. This is performed by estimating the number of samples needed to launch a successful attack. We also present empirical security experiments showing that Ghost Thread can thwart known cache attacks against common crypto algorithms.
• Ghost Thread is implemented as an application-level library and evaluates the runtime overhead of Ghost Thread under different system loads, different Ghost Thread configurations, and different tradeoffs between security and performance. We show that Ghost Thread has negligible overhead when a system is under utilized, or in real-world situations where not all code requires protection. In the worst-case scenario where a system is fully utilized and the entire application is protected, the overhead is still acceptable.
Chapter 5

Cache-Aware Symbolic Execution

5.1 Motivation

Detecting cache-based side channels in software is a challenging task. These side channels occur because of subtle dependencies created between the state of the CPU cache and sensitive data being processed by the program. Due to the proprietary design of CPU caches as well as their inherent complexities, accurately modeling them is both challenging and computationally expensive.

Existing tools, such as CacheAudit [13, 14] and CacheD [15], have attempted to address this problem by proposing automatic mechanisms to identify cache-based side channels in software. While those systems have successfully identified side channels in real-world programs, they still face a couple of significant limitations.

First, although some of the work based on symbolic execution (e.g., [15]) naturally offers counterexamples that represent program executions that exhibit the identified side channels, they only explore one or multiple dynamic execution paths and suffer from the problem of code coverage. That is, they are unable to detect side channels in unexecuted code, and side channels due to branches conditioned on confidential data. Some
systems [13, 14] use abstract interpretation, which covers all program paths. But they do not show developers where the side channels are and why they are side channels. More importantly, such missing information could be used to construct mitigations that fix the identified side channels.

Second, an adequate cache model is important for language-based analysis. To detect cache-based side channels, one approach is to directly check the existence or absence of data in the cache via architecture-dependent cache models (e.g., [13]). But the unpleasant consequence is that the security guarantee offered by those systems becomes architecture-dependent, and reasoning on concrete cache models is typically costly. Other systems (e.g., [14, 15]) check that the entire trace of accessed memory addresses is secret-independent, without using a concrete cache model. However, doing so can be too conservative. For example, AES implementations with preloading still exhibit key-dependent memory accesses, but they are secure since all key-dependent memory accesses will result in a cache hit with preloading.

To address these shortcomings of prior works, we propose CaSym (Cache-Aware Symbolic Execution). CaSym uses symbolic execution to soundly analyze programs for cache-based side channels and pinpoint their occurrence in software when detected. Furthermore, CaSym is capable of being configured with multiple cache models, allowing a variety of real world scenarios to be more accurately modeled. The rest of this chapter presents the details of CaSym.

5.2 Threat model

In this chapter, we consider an attacker who shares the same hardware platform with the victim, a common scenario in the era of cloud computing. The attacker has no
void Example() {
    // computing RK[0..3] from key
    res = res * res;
    res = res % mod;
    if(bit_set_at_i(key[0],i)) {
        res = base * res;
        res = res % mod;
    }
}

Figure 5.1: Example program composed of snippets of real code which demonstrate the two kinds of side channels.

direct access to the memory bus. Hence, he cannot directly observe the memory access traces from CPU. However, we assume the attacker can probe the shared data cache state, in order to detect whether some victim’s data is in cache or not. This model captures most cache-based side channel attacks in the literature, such as an attacker who observes cache accesses by measuring the latency of the victim program (e.g., cache reuse attacks [26,103–105] and evict-and-time attack [5]), or the latency of the attacker’s program (e.g., prime-and-probe attacks [5,25,104,106,107]).

Based on at which point of the victim’s program the attacker may probe the shared cache, there are two kinds of attackers (here, we follow the terminology used in [13]¹):

- access-based: when an attacker can probe the cache (i.e., determine if data are in cache or not) only upon the termination of the victim program.

- trace-based: when an attacker can probe the cache after each program point in the victim program.
5.3 System Overview

Fig. 6.3 depicts the system flow of CaSym. It takes the source code of the input program and uses LLVM’s front end, Clang, to convert the program into LLVM IR code. CaSym then performs cache analysis on the IR code to build a cache formula (verification condition) that represents how the program manipulates the cache state. The formula is fed to an SMT solver. A satisfiable answer of the formula means a side channel. In this case, CaSym uses the solution from the solver to localize the error and report it; this information can then be used for mitigation. An unsatisfiable cache formula means no cache-based side channel exists in the input program.

CaSym’s cache analysis is based on symbolic execution. At a high level, the symbolic execution takes an input program and outputs a symbolic state that models how the program relates the program’s initial program and cache states to its final program and cache states.

To discuss the process more formally, we introduce some notation used throughout this chapter. Assume the input program has $n$ program variables, $X_1$ to $X_n$, as well as some arrays with statically known sizes (e.g., $A[16]$ and $B[1024]$). We use small-case letters $x_1$ to $x_n$ for logical variables that represent the symbolic initial values of $X_1$ to $X_n$. Therefore, at the beginning of the program, we have $X_1 = x_1 \land \ldots \land X_n = x_n$.

---

1The work [13] also considers a time-based attacker, who may only observe the overall execution time of victim program. We do not consider this model since it is weaker than cache-probing attacks we consider in this chapter.

2For scalability, CaSym does not track symbolic values in arrays; more discussion on this later.
At the end of the program, the final values in \(X_1\) to \(X_n\) may have changed and are represented as symbolic expressions that may contain occurrences of \(x_1\) to \(x_n\). As an abbreviation, we write \(\pi\) for \(x_1, \ldots, x_n\) and \(\overline{X}\) for \(X_1, \ldots, X_n\).

We use \(c\) to represent symbolically the initial cache state and \(C\) for an implicit variable that tracks the current symbolic cache state; initially, \(C = c\) and at the end of the program the cache state \(C\) is a symbolic expression that may contain occurrences of \(c\) and \(x_1\) to \(x_n\).

According to the input program’s semantics, CaSym’s symbolic execution then builds a formula that represents the final program and cache states, using the initial program and cache states \((\pi, c)\):

\[
\sigma(\pi, c, \overline{X}, C)
\]

Assume at the beginning of the program, \(K_0\) is a set of secret variables whose values should be protected from side-channel attacks.\(^3\) A variable not in \(K_0\) is considered a public input whose value does not need protection. Informally, a program is free of side channels if \(K_0\) has no influence on the final cache state \(C\). Following the definition of noninterference [108], we formalize the verification condition for cache-based side channels as follows.

Given formula \(\sigma\) and \(K_0\), CaSym issues the following verification condition to an SMT solver:

\[
\begin{align*}
\text{VC}(\sigma, K_0) \equiv & \exists \pi, \pi', \overline{X}, \overline{X}', c, c', C, C', \\
& \neg(\forall X \in K_0, x = x') \land (\forall X \not\in K_0, x = x') \\
& \land c = c' \land \sigma(\pi, c, \overline{X}, C) \land \sigma(\pi', c', \overline{X}', C') \\
& \land \neg \text{eq}_C(C, C')
\end{align*}
\]

(5.1)

For the special case when \(K_0 = \{X_1\}\), the formula requires \(x_1 \neq x'_1 \land x_2 = x'_2 \land \ldots\)

\(^3\)Note that our implementation also allows marking a fixed-size array as the secret (e.g., the \texttt{key} array in the example of Fig. 5.1).
\[ \ldots \land x_n = x'_n; \text{i.e., two different values for } X_1 \text{ and the same value for other variables.} \]

When equation (5.1) is satisfiable, then it is possible to run the program twice with two different secrets, the same public inputs, and the same initial cache state, and get two different final cache states \( C \) and \( C' \). That is, by observing the final cache state, an attacker can learn information about the secret.

Compared with previous work based on abstract interpretation \([13, 14]\), a benefit of checking the verification condition above is that a solution of equation (5.1) leads to two program executions that exhibit the existence of side channels: in this case, the SMT solver produces two different secret values, which serve as the witness of the identified side channel(s). CaSym then uses an error localization algorithm to diagnose the witness (i.e., two program executions) to pinpoint the problematic program points that cause the side channel(s) in the source code.

On the other hand, when the formula is unsatisfiable, we are assured that there is no cache-based side channel in the given environment: regardless of which secret values are chosen, the cache state does not change. We note that this property is essentially a cache-aware variant of noninterference \([108]\), which states that confidential data cannot influence public outputs (i.e., cache states).

### 5.4 Tracking Cache States

In this section, we discuss how CaSym performs symbolic execution to go from the input program to the formula \( \sigma(\overline{x}, c, \overline{X}, C) \) that relates the initial program and cache states to the final program and cache states. As in all symbolic-execution systems, CaSym’s symbolic execution is based on program paths. For each path, CaSym computes a symbolic state (including cache state) at the end of the path. For a loop-free program, CaSym considers every path in the program and combines the symbolic states of all paths into a
single formula. Moreover, for programs with loops, CaSym uses novel language statements and a transformation that soundly converts the loops into loop-free counterparts before performing symbolic execution.

Note that since CaSym considers all paths, its symbolic execution is similar to strongest postcondition calculation in Hoare Logic [109]. However, since CaSym soundly converts loops into loop-free counterparts before symbolic execution, verification in CaSym does not require explicit loop invariants, which is required in traditional postcondition calculation in Hoare logic.

5.4.1 Loop-free programs

A loop-free program has a finite number of paths. For a path \( i \), symbolic execution computes a symbolic state \( \sigma_i(\overline{x}, c, \overline{X}, C) \) at the end of the path. For a finite number of paths, CaSym could combine the symbolic states of all paths through disjunction; however, this would not be scalable. Therefore, after discussing how CaSym performs symbolic execution on a single path, we present how CaSym shares parts among paths to get compact formulas through path merging.

Symbolic states A symbolic state contains (1) the symbolic values of program variables, (2) a symbolic cache state, and (3) a path condition, which is a conjunction of boolean tests performed on the path. Specifically, a symbolic state \( \sigma(\overline{x}, c, \overline{X}; C) \) is of the following form:

\[
X_1 = e_1 \land \ldots \land X_n = e_n \land C = ce \land \psi
\]

where \( e_i \) is a symbolic expression that represents the value in \( X_i \) and it may contain occurrences of \( x_1 \) to \( x_n \) (the initial symbolic values for \( X_1 \) to \( X_n \)). Similarly, \( ce \) is a
symbolic cache state that represents the current cache state; \( e \) may contain occurrences of \( c \) as well as \( x_1 \) to \( x_n \). Path condition \( \psi \) may contain occurrences of \( x_1 \) to \( x_n \). As an abbreviation, we write \( \overline{X} = \overline{e} \) for \( X_1 = e_1 \land \ldots \land X_n = e_n \). The initial symbolic state is

\[
\text{initial}(\overline{x}, c, \overline{X}, C) \triangleq \overline{X} = \overline{x} \land C = c \land \text{True}
\]  

We note that even though our goal is to track cache states, for precision it is important to also track program states (path conditions and symbolic values of variables). Since a program has many paths and each path can lead to a different cache state, the path condition in a symbolic state tells under what condition the program produces the associated cache state. Furthermore, tracking symbolic values of variables is important for computing accurate cache states. For instance, when the program accesses \( A[X] \), knowing variable \( X \)'s symbolic value is important to model what element of the array is being accessed.

**Interface to cache models** To track how the input program affects the cache state, CaSym needs a cache model that specifies how the cache is affected by memory accesses. A cache implementation makes various choices about cache capacity, cache block size, associativity, and the cache-replacement policy. To accommodate cache diversity, CaSym’s symbolic execution is parameterized over a cache model so that different cache models can be plugged into the system. This set up also enables abstract cache models, which abstract away certain cache-implementation details and provide generality (we introduce two abstract cache models in Section 5.5).

CaSym’ symbolic execution interacts with a cache model through a well-defined interface. We postpone the discussion of how CaSym’s cache models implement the interface to Section 5.5. For now, it is sufficient to understand a cache model’s interface,
listed as follows:

(i) An empty cache state, written as $\text{empty}_C$.

(ii) A cache-update function $\text{upd}_C(l, ce)$, which takes a symbolic memory location $l$ (discussed soon) and a symbolic cache state $ce$ and returns a new symbolic cache state for the result of accessing $l$ under $ce$.

(iii) An equality test $\text{eq}_C(ce_1, ce_2)$. It holds if and only if the two symbolic cache states $ce_1$ and $ce_2$ are equal according to the cache model.

Two kinds of symbolic memory locations are used. The symbolic memory location for variable $X$ is written as $M_X$. The symbolic location for array $A$ at symbolic expression $e$ is written as $MA[e]$. As an example, for the program “$X_2 = X_1 & 0xFF; X_3 = A[X_2]$”, the symbolic memory location for the array access in the second statement is $MA[X_1 & 0xFF]$.

**Symbolic execution over a path** To formally present symbolic execution over a program path, we introduce a small path language in Fig. 6.2. A statement in the language represents a list of commands in a program path. In the language, we use $n$ for constant numbers, $X$ for program variables, $A$ for fixed-size arrays, $\otimes$ for a binary arithmetic operator, and $\odot$ for a binary comparison operator. We use “$A[X] := E$” for an array assignment. “$B \rightarrow S$” is a statement guarded by boolean condition $B$; it is the single-guard variant of guarded commands in Dijkstra’s guarded command language [110]. An if-statement “IF $B$ THEN $S_1$ ELSE $S_2$” can be split into two paths: one has “$B \rightarrow S_1$” and the other has “$\neg B \rightarrow S_2$”.

Fig. 5.3 formalizes CaSym’s symbolic execution over the path language. Its way of tracking program states (variables’ symbolic values and path conditions) is standard in
Syntax for a path language and symbolic execution over a path.

symbolic execution. It uses auxiliary functions $E_{\sigma}$ and $B_{\sigma}$ to compute the symbolic values of arithmetic expressions and boolean expressions, respectively; we omit the standard definition of $B_{\sigma}$.

In addition, cache states are also tracked during symbolic execution. We use the following notation in Fig. 5.3 for tracking cache states. The cache-update function is lifted to a list of symbolic memory locations: $\text{upd}_C(L, ce)$ returns the new cache state after accessing the list of locations in $L$. We use $\text{locs}(E, \sigma)$ for the list of symbolic memory locations in program expression $E$ under symbolic state $\sigma$, and similarly $\text{locs}(B, \sigma)$ for the list of symbolic memory locations in boolean expression $B$. We omit their straightforward definitions. As an example, if in $\sigma$ we have $X = 2x$, then $\text{locs}(X + A[X], \sigma) = [MX, MA[2x]]$. Fig. 5.3 presents how cache states are tracked for each kind of statements. As an example, showing how cache states are tracked,
take $X_i = E$, which accesses the memory locations in $E$ and the memory location of
$X_i$; therefore, it updates the symbolic cache state with those locations. Other cases are
similar.

We next present symbolic execution for a toy example “$X_2 = X_1 \& 0xFF; X_3 = A[X_2]$”,
starting from the initial symbolic state.

$$\{ X_1 = x_1 \land X_2 = x_2 \land X_3 = x_3 \land C = c \land \text{True} \}$$

$X_2 = X_1 \& 0xFF$

$$\{ X_1 = x_1 \land X_2 = x_1 \& 0xFF \land X_3 = x_3 \land C = \text{upd}_C([MX_1, MX_2], c) \land \text{True} \}$$

$X_3 = A[X_2]$

$$\{ X_1 = x_1 \land X_2 = x_1 \& 0xFF \land X_3 = u \land C = \text{upd}_C([MA[x_1 \& 0xFF], MX_3], \text{upd}_C([MX_1, MX_2], c)) \land \text{True} \}$$

The end symbolic cache state $\sigma$ contains a symbolic memory location $MA[x_1 \& 0xFF]$.
Feeding $VC(\sigma, \{ X_1 \})$ to an SMT solver would produce two different values for $x_1$,
resulting in two different cache states. This is a side channel caused by key-dependent
memory accesses.

As a note, for scalability CaSym’s symbolic execution does not track array contents
symbolically. This is why $[A[X]]_\sigma$ produces a fresh unconstrained variable, which
implements an approximation (reading from an array returns arbitrary values). On the
other hand, when accessing $A[X]$, CaSym uses $X$’s symbolic value to capture which
location of the array is accessed and uses that knowledge to update the symbolic cache
state accurately.

CaSym employs a coarse-grained taint tracking for arrays. This means we can use
the arbitrary values stored in the arrays and treat them as public values (when not tainted)
or sensitive values (when tainted). Therefore, two symbolic execution traces use the
\begin{align*}
\{ X_1 = x_1 \land X_2 = x_2 \land C = c \land \text{True} \} \\
\text{if } (X_1 > 0) \{ \\
\{ X_1 = x_1 \land X_2 = x_2 \land C = \text{upd}_C([M_x_1], c) \land x_1 > 0 \} \\
X_2 = X_2 + 1; \\
\{ X_{1a} = x_1 \land X_{2a} = x_2 + 1 \land C_a = \text{upd}_C([M_x_1, M_x_2], c) \land (\psi_a = x_1 > 0) \land \psi_a \} \\
\} \\
\text{else} \{ \\
\{ X_1 = x_1 \land X_2 = x_2 \land C = \text{upd}_C([M_x_1], c) \land x_1 \leq 0 \} \\
X_2 = X_2 + 2; \\
\{ X_{1b} = x_1 \land X_{2b} = x_2 + 2 \land C_b = \text{upd}_C([M_x_1, M_x_2], c) \land (\psi_b = x_1 \leq 0) \land \psi_b \} \\
\} \\
\{( (\psi_a \land X_{1c} = X_{1a} \land X_{2c} = X_{2a} \land C_c = C_a) \lor \}
(\psi_b \land X_{1c} = X_{1b} \land X_{2c} = X_{2b} \land C_c = C_b)) \land 
(\psi_c = \psi_a \lor \psi_c = \psi_b) \land \psi_c 
\}
\end{align*}

Figure 5.4: Sample program illustrating how path merging is handled using the infinite cache model.

same arbitrary value from a public array (i.e. these values cannot contribute to a difference in the cache state). This helps reduce false positives which would occur if we considered all array values to be sensitive.

**Path merging**  Simply performing symbolic execution over every path and combining the symbolic states of all paths through disjunction at the end is not scalable as it would generate large formulas. As an optimization, CaSym performs path merging to generate formulas that share parts among program paths. In particular, when multiple paths converge at a point, it merges their symbolic states by introducing new logical variables and equations as illustrated by the example in Fig. 5.4. Consider two paths with path conditions \( \psi_a \) and \( \psi_b \), the symbolic values of variables \( \overline{X}_a \) and \( \overline{X}_b \), and the symbolic cache states \( C_a \) and \( C_b \). At the merge point, CaSym introduces new logical variables \( \psi \),
\( X \), and \( C \), and adds the following equalities:

\[
\psi = \psi_a \lor \psi = \psi_b 
\]  
(5.3)

\[
(\psi_a \land X = X_a \land C = C_a) \lor (\psi_b \land X = X_b \land C = C_b) 
\]  
(5.4)

All equations reflect that either one of the paths could be taken, but each case in equation (5.4) is further guarded by the corresponding path condition for precision. The equation at the end of Fig. 5.4 follows equations (5.3) and (5.4).

The benefit of path merging is that further symbolic execution beyond the merge point can just use the newly introduced logical variables (\( \psi \), \( X \), and \( C \) in the equations above); so all the paths beyond the merge point share the logical variables and equations, resulting in compact logical formulas. We also note that CaSym’s implementation takes a control-flow graph as an input and treats every node with more than one adjacent predecessor in the graph as a merge point; as a result, new logical variables are introduced and path merging is performed before every node in the graph with this characteristic.

We note that our path merging is similar to those proposed in [111, 112] (though they do not consider merging cache states). We could further optimize the formulas by applying techniques in [113, 114], which selectively use path merging when it is adventurous to do so; but we leave that as future work.

### 5.4.2 Compositional reasoning

The cache-aware symbolic execution sketched above only handles programs with a finite number of execution paths. However, practical software usually have an unbounded or a large number of execution paths, making symbolic execution infeasible or inefficient.
To tackle such a challenge, we introduce two novel statements to enable compositional reasoning

\[ \text{Stmt} \ S ::= \ldots \mid \text{reset} \mid \text{check } K \]

The “reset” statement resets the current symbolic state to an arbitrary initial symbolic state; “check \( K \)” directs CaSym to issue a verification condition to a solver based on the current symbolic state, assuming only the secret-variable set \( K \) carries confidential data at the last reset. The rules for performing symbolic execution over them are shown below: The definitions of \( \text{initial}(x', c', X, C) \) and \( V\text{C}(\sigma, K) \) were given earlier in equations (5.1) and (5.2). Note that we use \( x' \) and \( c' \) to distinguish them from the initial state of symbolic execution.

The introduced “reset” and “check \( K \)” statements have several benefits:

**Flexibility** The check and reset statements allow CaSym to flexibly decide where to reset to the initial state and where to check for cache-based side channels. For example, by turning \( S \) into “reset; \( S \); check \( K_0 \)”, we tell CaSym to perform symbolic execution from the initial symbolic state and perform the side-channel check at the end, assuming that the initial secret variables are in \( K_0 \) and an attacker observes the cache state only at the end of the program. As another example, when “\( S = S_1; S_2 \)” and the attacker can observe the cache state in the middle and at the end, we can perform symbolic execution over “reset; \( S_1 \); check \( K_0 \); \( S_2 \); check \( K_0 \)”, which triggers two side-channel checks. In the most extreme case, a check can be inserted at every control-flow point in the program, corresponding to what a trace-based attacker can observe (discussed in Section 5.2).

**Scalability** The new statements also enable compositional, scalable reasoning. Suppose performing symbolic execution over the entire program \( S \) produces a large formula
at the end. Feeding the formula to an SMT solver may not be feasible given the amount of time that is needed to solve the formula. One way of reducing the time pressure is to break $S$ into two parts and check them individually. Suppose $S = S_1; S_2$, we can then turn it into

$$\text{reset}; S_1; \text{check } K_0; \text{reset}; S_2; \text{check } K_1$$

The first check verifies that $S_1$ is free of cache-based side channels; that is, running $S_1$ twice with two different secrets and the same initial cache states results in the same cache states. After this check, we can reset the symbolic state and perform symbolic execution on $S_2$ and the check after $S_2$ verifies that running $S_2$ twice with different secrets and the same initial cache states results in the same cache states. This is a rely-guarantee reasoning since, when checking $S_2$, it relies on the assumption that the initial cache states are the same and the assumption is discharged by the verification on $S_1$.

The compositional reasoning is more scalable than checking $S_1; S_2$ as a whole, since the reset in the middle throws away the symbolic state. However, it may cause false positives when checking $S_2$ due to the loss of information.

One complication in the above process is that the two check statements are with respect to two separate secret-variable sets: the first check assumes $K_0$ is the secret-variable set at the beginning, while the second check assumes $K_1$ is the secret-variable set at the point between $S_1$ and $S_2$. The two sets might be different; for instance, $K_0$ might be $\{X_1\}$, and, if $S_1$ copies $X_1$ to $X_2$, then the set of secret variables after $S_1$ becomes $\{X_1, X_2\}$. In general, the set of secret variables may change due to secret propagation in a program. To soundly estimate the set of secret variables, CaSym has a static taint tracking component, which takes initial secret variables and outputs the set of secret variables at each program location. This was implemented by a standard flow-sensitive dataflow analysis in LLVM. With the result of this analysis, CaSym knows the
secret-variable set at each location, including $K_1$.

### 5.4.3 Transforming loops

Symbolic-execution systems for bug finding only explore a limited number of paths. Hence, they do not guarantee a coverage of all paths for programs with loops. With the help of the new statements introduced in Section 5.4.2, CaSym transforms programs with loops into loop-free programs. Specifically, the transformation works as follows:

$$S_1; (\text{WHILE } B \text{ DO } S); S_2 \Rightarrow S_1; \text{check } K_0; \text{reset};$$

$$(\text{IF } B \text{ THEN } (S; \text{check } K_1) \text{ ELSE SKIP}); \text{reset};$$

$$\neg B \rightarrow S_2$$

This transformation is sound (i.e., the original program is side-channel free whenever the transformed one is) since the loop-free program enforces the following invariant in the original program: any two executions of $S$ starting from the same initial cache state results in the same final cache state. From the Hoare logic point of view, this is the cache-state loop invariant checked by CaSym. Note that this is performed with respect to $K_1$, the set of secret variables right before the loop body $S$ (this is determined by tracking how values of secret variables propagate in the original program, as discussed before).

To see why the invariant is enforced, the first check makes sure that the initial cache states are identical before the loop (i.e., $S_1$ is side-channel free). After that, the symbolic state is reset. Hence, statement $(\text{check } K_1)$ ensures that there is no side channel for the
loop body starting from any memory and cache state. After checking the loop body, the symbolic state is reset again so that the verification of $S_2$ assumes nothing after the if statement, which semantically represents the memory and cache state after zero or one loop iteration. After that, $\neg B$ can be assumed when checking $S_2$.

In theory, the transformation may cause some false positives. For example, the transformation assumes nothing on the initial memory and cache state before each iteration, which may cause false positives. But in practice, we have found only one false positive due to the transformation in database systems (Section 5.8). Moreover, when a loop has a constant number of iterations, we can also unroll it for better precision.

### 5.5 Cache Models

CaSym takes a cache model and identifies potential side channels based on it. In principal, it can take any cache model with sufficient abstractions in place: the empty cache state, the cache-update function, and the equality-testing function. In this section, we introduce two novel abstract cache models: the infinite cache model and the age model. We also discuss how to support more concrete models, such as the LRU model, used in previous work.

#### 5.5.1 Abstract vs. concrete cache models

Concrete cache models (e.g., LRU, FIFO, PLRU models used in [13]) accurately model details such as the replacement policy of the expected architecture that a program will be executed on. While such detailed models allow accurate reasoning about the cache state (i.e., existence or absence of data in the cache), one downside is that the verified programs are secure only on those expected architectures. For example, a crypto implementation that is side-channel-free on cache with LRU might have side channel on
cache with FIFO. Moreover, reasoning over a concrete model likely will cause scalability issues for static program analysis.

Another approach is to use a higher-level abstraction, such as the entire trace of memory accesses [14, 15]. Doing so is architecture-independent and sufficient since in all realistic architectures, cache state is determined by the trace of memory accesses. However, this approach may be too conservative, since the footprint of secret dependent memory accesses might be erased by later accesses before an attacker probes the cache.

We propose two novel cache models that offer good balance between precision and generality. The infinite cache model represents an optimistic view of the cache: if there is a side channel under this model, then the side channel likely will exist in realistic cache models (i.e., they are high-priority side channels that may show up on most architectures); the age model represents a pessimistic view of the cache: if there is a side channel under the this model, then the side channel likely will exist in some realistic cache model (i.e., they are low-priority side channels that may show up on some architectures). Empirical results suggest that the infinite cache model and the age model achieves a good balance between analysis scalability and precision (Section 5.8).

### 5.5.2 Infinite cache model

This is an idealized cache model with an infinite size and associativity, so that it never evicts any data that is already being cached. This is clearly idealized, but it is also interesting since it represents an optimistic view of cache: if there is a side channel under the infinite cache model, the side channel likely will exist in other more realistic cache models. Moreover, it is the (conceptual) model that cryptography software writers have in mind when they apply software countermeasures to cache-based side channels. One example is preloading in cryptography software, which we detail in Section 5.6.2.1.
Furthermore, empirical results suggest that the infinite cache model offers a significant speedup with few false negatives on both crypto systems and database systems, compared with more conservative and realistic cache models.

In the infinite cache model, a cache state is represented as a set of symbolic memory addresses for program variables and array elements.

- The empty cache is the empty set: $\text{empty}_C = \{\}.$

- The cache-update function is implemented as set union: $\text{upd}_C(l, ce) = \{l\} \cup ce.$

- The cache-equality testing becomes set equality:

$$\text{eq}_C(ce_1, ce_2) = \forall l, l \in ce_1 \leftrightarrow l \in ce_2$$

To see why this model is more optimistic than other more realistic models, we note that $ce_1$ and $ce_2$ are different only if there is some address $l$ that is accessed in one execution but not in the other, starting with different confidential data. Except for a fully-associative cache, that implies if the compiler maps $l$ to some cache set and maps all other addresses to other cache sets, most cache replacement policies will result in a difference in the cache set that $l$ gets mapped to. Hence, this model gives a “lower bound” on side channels among various cache models.

### 5.5.3 Age model

Unlike the optimistic infinite model, the age model is on the pessimistic end: for each symbolic memory location, it tracks the distance to its most recent access, called the age. The recently accessed location has age zero, while the second recently accessed location has age one, and so on. In this model, a cache state is a map from symbolic memory locations to their ages:
• The empty cache maps all memory locations to infinity: \( \text{empty}_C = \lambda l. \infty \).

• The cache-update function marks the current location’s age to be zero and increments other locations’ ages by one: \( \text{upd}_C(l, ce) = \lambda l'. \text{if } l' = l \text{ then } 0 \text{ else } ce(l') + 1 \)

• The cache-equality tests equality of ages: \( \text{eq}_C(ce_1, ce_2) = \forall l, ce_1(l) = ce_2(l) \).

The age model is the opposite of the infinite cache model: while the infinite cache model may miss (less crucial) side channels that only manifest themselves for some particular caches, the age model captures all potential side channels for most caches.

**Property 1.** If there is no cache-based side channel on the age model, then there is no cache-based side channel for any cache replacement policy that replaces cache lines based on the most recent accesses, such as LRU.

To see why, we note that the final cache expression \( ce \) tracks the sequence of the last access to each memory address. For any cache replacement policy that depends only on the latest usage of memory addresses, such as LRU, it implies that the final cache state can be expressed as a function of ages. Hence, \( \text{eq}_C(ce_1, ce_2) \) implies the same cache state under those policies.

For a trace-based attacker, a stronger result holds:

**Property 2.** For a trace-based attacker, no cache-based side channel on the age model implies no cache-based side channel for any deterministic cache replacement policy (i.e., a replacement policy that can be expressed as a function of memory address traces), such as FIFO and LRU.

The reason is that there is a one-to-one mapping between a sequence of ages (for all symbolic locations), and a sequence of memory locations being accessed. Consider a
sequence of ages \( A = \{ ce_1, ce_2, \ldots, ce_n \} \) as well as a sequence of memory locations being accessed, say \( T = \{ t_1, t_2, \ldots, t_n \} \). Then, we can construct \( A \) from \( T \) as follows:

\[
ce_i = \lambda l. \text{if } t_i = l \text{ then } 0 \text{ else } ce_{i-1}(l) + 1
\]

Also, we construct \( T \) from \( A \) as follows: \( t_i = l \) iff \( ce_i(l) = 0 \) (note that exactly one \( l \) in \( ce \) is 0 at any program point).

### 5.5.4 More concrete models

While our infinite cache and age models are capable of detecting side channels, we show how to enrich a cache model in CaSym if more cache details (such as cache line size, associativity and replacement policy) are needed for precision reasons.

**Cache line size** To model cache lines, we simply take the index into the array and compute the cache-line-granularity location being accessed. More specifically, suppose that an integer array \( A \) is aligned and location \( A[X] \) is being accessed and \( x \) is the symbolic value of \( X \). We simply use \( (x/LINESIZE) \) as the location \( l \) in the \( \text{upd}_C(l, ce) \) interface to the cache models above, where \( LINESIZE = 64/4 = 16 \) assuming 4-bytes integer and 64-bytes cache line.\(^4\) Note that cache line size only affects array accesses, since the memory layout for other variables are unknown at the IR level.

**Cache associativity** To model cache associativity, we model the cache state \( ce \) as a collection of non-overlapping cache sets (i.e., \( ce = [c_1, c_2, \ldots, c_W] \)). The empty cache and equality test on \( ce \) is simply the lifted definition of those on each cache state. For the update function, let \( \text{way} \) be a function that maps an array index to the corresponding

\(^4\)The computation assumes row-major layout for arrays. Column-major layout can be handled in similar way.
cache set (the definition of way depends on cache configuration), then the following formula illustrates how the new cache state would be computed when an array is accessed.

$$\text{upd}_C(\text{MA}[X], ce) = \text{upd}_C(\text{MA}[X], ce[\text{way}(X)])$$

**LRU replacement**  In the LRU model, the cache state is still modeled as a map from symbolic memory locations to their ages. The empty cache and the cache-update function remain the same as the age model. The cache-equality test, however, is changed to

$$\text{eq}_C(ce_1, ce_2) = \forall l, ce_1(l) < n \leftrightarrow ce_2(l) < n.$$  

to reflect the fact that if $l$ is in the final cache state or not. Here we assume $n$ is the cache size.

### 5.6 Localizing and Mitigating Side Channels

As discussed so far, satisfiable constraints at a certain program point suggest potential side channels. Although such a binary decision helps to some extent, one novel feature of CaSym is the ability to help programmers localize the cause of the identified side channels as well as to mitigate them.

#### 5.6.1 Localizing side channels

To localize and explain the causes of the identified side channels, we leverage the key information generated by the SMT solver: a model of constraints. A model consists of the concrete values for each constraint variable in the inequality test $\neg(\text{eq}_C(ce_1, ce_2))$. According to the way that the formulas are built, a model consists of two sets of values
(one used by $ce_1$ and one used by $ce_2$) that will lead to different cache states. We refer to those two value sets as $M_1$ and $M_2$ respectively. Our localization algorithm proceeds in two steps.

**Compute shared path** We first use $M_1$ and $M_2$ to reconstruct two execution paths taken according to the model reported by the solver. This is possible since CaSym keeps track of the path condition for each basic block. Given $M_1$ and $M_2$, we can tell which basic block is in a path by checking the validity of its path condition. Based on that, we compute the *shared* blocks between them. Finally, we traverse the control flow graph in topological order to recover a path of those *shared* blocks. We call the path the *shared path*, denoted by $SP$.

**Discover divergence** In the second step, we first find all symbolic addresses which are different in the final cache states. That is, we use $M_1$ and $M_2$ to find addresses such that they make a difference between $ce_1$ and $ce_2$. Intuitively, these are the problematic memory addresses that we need to localize the error cause for each of them.

For each problematic memory address $l$, the localization algorithm reports the first point in $SP$, say an instruction $c$, so that the abstract cache states of $l$ are different for the next point in $SP$, but are identical for the previous point in $SP$. Such an instruction is reported as the root cause of the side channel at $l$. Note that when multiple addresses may cause side channels, our algorithm reports multiple instructions in a program, following the same procedure for each $l$ that causes a difference between $ce_1$ and $ce_2$.

**Example** Fig. 5.1 shows a simple example with two side channels. For the final cache state, an SMT solver reports a model where $RK[3] = 256, key[0] = 255$ in value set $M_1$ and $RK[3] = 0, key[0] = 0$ in value set $M_2$. Based on the model and tracked path
conditions in the control flow graph, the localization algorithm constructs a shared path consisting of the blocks in grey, shown in Fig. 5.5.

In this example, the final cache state differs for several symbolic addresses, including $\text{Sbox}[0]$, $\text{base}$ and so on. For the address $\text{Sbox}[0]$, the first point in the shared path that $ce$ differs between the two execution paths is after the assignment $\text{RK}[4]=\text{RK}[0] \lor \text{Sbox}\{(\text{RK}[3] >> 8) \& 0xFF\}$, which is the correct location to blame for the cache difference of address $\text{Sbox}[0]$. For the address $\text{base}$, the first point in the shared path that $ce$ differs between the two execution paths is after the branch condition $\text{bit\_set\_at\_i(key[0],i)}$. This is the correct location to blame as well, since the secret dependent branch caused cache difference. For other problematic addresses, the localization algorithm points to those two problematic instructions as well in this example.

### 5.6.2 Fixing side channels

The localized causes of side channels enable a programmer or a compiler to fix the identified side channels. We explore two commonly used techniques for cache-based side channel mitigation in this section and show how the localized error causes facilitate error fixing.

#### 5.6.2.1 Preloading

Preloading eliminates cache-based side channels by loading certain memory addresses before the vulnerable instructions [5, 115]. It is typically used in AES implementations, where all SBox tables fit in cache. In AES, those tables only contain public data, but indexes used to access the tables are key-dependent, which enables an attacker to infer the key based on the footprints of the AES implementation on cache [5, 26, 106]. This
Figure 5.5: Shared path computed from the model generated by the SMT solver for the code in Fig. 5.1.

is similar to the code in Example 5.1. To mitigate such attacks, AES implementations preload the entire lookup tables into the data cache before the actual encryption/decryption. That is, they insert code that accesses every table entry to ensure all table data are in the cache before encryption/decryption starts. Hence, even if there are key-dependent table look-ups, they will not affect the cache state as long as all table entries are already in the cache initially and they are not evicted during encryption/decryption.

5.6.2.2 Pinning

Pinning prevents cache misses on the data that is explicitly “pinned” in a program. For instance, this feature can be implemented in a customized cache, where a cache entry
with the “pin” bit set is never evicted [65]; it can also be implemented on some commodity hardware with Hardware Transactional Memory (HTM) [86]. Compared with preloading, pinning provides extra assurance that pinned data will not be evicted until it is explicitly “unpinned” in the program. Previous work has shown that pinning can be use to defend against cache-based side channels [65, 86].

5.6.2.3 Fixing side channels

To support preloading and pinning, CaSym introduces special instructions in the form of \texttt{PRELOAD} \textit{l} and \texttt{PIN} \textit{l}, which semantically preload/pin the corresponding symbolic addresses into cache (when \textit{l} is an array, the instruction preloads/pins all elements in the array).

The localized root causes of side channels makes it straightforward to insert needed preloading/pinning instructions to remove side channels: if the vulnerable point for memory address \textit{l} is an instruction \textit{c}, then preload/pin \textit{l} before \textit{c} will remove the counterexample found by the SMT solver. For example, preloading/pinning the entire \texttt{SBox} table right before its vulnerable point at line 3 in Fig. 5.1 as well as preloading \texttt{base}, \texttt{res}, \texttt{mod} right before their shared vulnerable point at line 6 in Fig. 5.1 (found by the localization algorithm in Section 5.6.1) will remove the side channels in this program.

Although fixing side channels seems easy with the help of CaSym, we emphasize that finding \textit{where and what} to preload/pin is nontrivial without CaSym, since identifying what data to preload can be difficult. For example, the crucial data in AES is the lookup table, which only stores public information. Moreover, for preloading, fetching the data too early may cause the data to be evicted before the vulnerable instructions, which undermines the effect of preloading.
5.7 Implementation

CaSym is implemented inside LLVM [116] as a compiler pass that performs cache analysis and error localization. It analyzes LLVM IR code and performs symbolic execution to build a cache formula. We use the Z3 SMT solver [117] to check the satisfiability of the cache formula, but any SMT solver with theories for bit-vectors and arrays could suffice.

The compiler pass of CaSym sends to the Z3 solver (in the same process as the compiler pass) the cache formula as an in-memory object. If the formula is satisfiable (meaning there is a side channel), the solver generates a model containing the assignments for the formula’s variables. Using this model, CaSym localizes the vulnerable LLVM IR instructions. CaSym then uses the debugging information to report the corresponding line numbers in the source program to the user.

5.8 Evaluation

We evaluated CaSym on a set of crypto and database benchmarks. All experiments were run on Ubuntu 14.04 in a virtual machine with 16 GB of RAM and an Intel i7-5820K CPU at 3.30 GHZ. During evaluation, we were mostly interested in answering the following questions:

1. how effective is CaSym in identifying cache-based side channels and how accurate are the results?

2. how efficient is CaSym and whether it can generate useful results within a reasonable amount of time?

3. how do different cache models compare when identifying side channels?
4. how well does CaSym’s error localization perform?

5. whether CaSym can validate the result after applying prefetching or pinning to fix a side channel?

For benchmarks, we collected realistic crypto implementations from popular libraries, including Libgcrypt 1.8.1, mbed TLS 2.6.0, and glibc 2.26. These benchmarks can be roughly divided into two categories: encryption using symmetric ciphers and modular exponentiation using asymmetric ciphers. In order to evaluate CaSym on other less scrutinized codebases, we also analyzed functions from the PostgreSQL 10.2, which is a popular database back end.

**Symmetric cipher benchmarks** We include six benchmarks: AES gcry, AES mbed (the 128-bit-key AES encryption in Libgcrypt and mbed TLS respectively), triple-DES gcry, triple-DES mbed (the triple-DES encryption in Libgcrypt and mbed TLS respectively), DES glibc (the DES encryption in the glibc library) and UFC glibc (the ultra fast encryption algorithm in glibc).

**Asymmetric cipher benchmarks** Given a base \( b \), an exponent \( e \), and a modulus \( m \), modular exponentiation computes \( b^e \mod m \). The majority of asymmetric encryption such as RSA and ElGamal performs modular exponentiation.

Computing modular exponentiation directly would be rather costly in both time and space. Libgcrypt implemented three versions of efficient modular exponentiation, which we call sqr-alwys-mul gcry, sqr-mul gcry, and LR-mod-expo gcry. The first two versions implement the square-and-multiply method; the main difference between the two is that the square and multiply algorithm only performs the multiplication when the bit being processed is set, while the square-and-always-multiply algorithm performs the multiplication regardless of whether the current bit is set. The final version implements the
Table 5.1: Evaluation results for access and trace-based attackers for crypto benchmarks. For access-based results, \(\bigcirc\) means a side channel identified and \(\checkmark\) means no side channel identified; it also reports the amount of time in seconds for CaSym to perform side-channel checking. For trace-based results, the TP column identifies the number of true positives found. Similarly FP is the number of false positives. The third column for each models depicts the amount of time (in seconds) for each test.

Database benchmarks The crypto benchmarks are typically well scrutinized for side channels. To see how CaSym performs on other codebases, we consider PostgreSQL, a popular database used by many applications. For this database system, we treat the primary keys (identifiers for records in a database) as sensitive, since they are commonly account numbers, social security numbers, etc.

The entire PostgreSQL contains over 1 million lines of code. We narrowed down the scope to a set of functions that process likely sensitive data. In particular, we investigated the binary tree implementation under the `/src/backend/nbtree` directory. Under this directory, there are 16 functions that use the primary key of a record. Of those 16,
Table 5.2: Evaluation results for access and trace based attackers for the PostgreSQL database.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Access</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Infinite</td>
<td>Age</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>TP</td>
</tr>
<tr>
<td>advance array keys</td>
<td>99</td>
<td>✓ 0.26</td>
</tr>
<tr>
<td>binsrch</td>
<td>126</td>
<td>✓ 0.06</td>
</tr>
<tr>
<td>compare</td>
<td>174</td>
<td>✓ 1.24</td>
</tr>
<tr>
<td>find xtreme element</td>
<td>117</td>
<td>✓ 0.38</td>
</tr>
<tr>
<td>heap key test</td>
<td>118</td>
<td>✓ 0.07</td>
</tr>
<tr>
<td>is equal</td>
<td>89</td>
<td>✓ 0.12</td>
</tr>
<tr>
<td>mark array keys</td>
<td>34</td>
<td>✓ 0.01</td>
</tr>
<tr>
<td>sort array elements</td>
<td>144</td>
<td>✓ 0.81</td>
</tr>
<tr>
<td>start array keys</td>
<td>42</td>
<td>✓ 0.01</td>
</tr>
</tbody>
</table>

7 pass the key onto other functions without processing it. Therefore, we evaluate on the remaining 9 functions which actually process the primary keys. The functions are: advance array keys, binsrch, compare, find xtreme element, heap key test, is equal, mark array keys, sort array keys, and start array keys.

### 5.8.1 Evaluation for access-based attackers

Recall that an access-based attacker observes only the final cache state of the victim program. Therefore, CaSym performs the verification-condition check only at the end of each benchmark. We evaluated benchmarks based on the two abstract cache models: the infinite model and the age model (we will compare the abstract cache models with concrete cache models in Section 5.8.4).

**Cryptography Benchmarks**  Table 5.1 presents the results on cryptography benchmarks. We note that we have removed preloading that is present in the original AES and triple-DES code; we will report separately the results when it is present. UFC and DES
in glibc do not use preloading.

From the table, we observe that CaSym can finish side-channel checking rather quickly for most of the cases: all checks under the infinite cache model finish under 1 second; those under the age mode are slower, but all finish in about 2 minutes. Moreover, we note that the optimistic infinite cache model gives exactly the same result as the more conservative age model.

The side channels reported for the two AES, two triple-DES and one DES implementations are due to key-dependent array accesses (we discuss examples in the next section). Side channels are also reported for the three modular-exponentiation algorithms, since they contain secret-dependent branches (examples in the next subsection). These are previously known side channels and CaSym confirms their presence.

We note that DES-glibc contains a side channel that was newly found by CaSym; it results from key-dependent memory accesses to a lookup table that was not preloaded (key is the private key):

\[
t = r^\text{key}[1 + 1]; \\
l = \text{des_SPtrans}[1][(t)\&0x3f];
\]

UFC-glibc is the only benchmark that is side channel free with respect to the cache without extra security mechanisms. This is because it does not use any precomputed table and is virtually straight line code. It illustrates how different techniques can avoid side channels.

**Database Benchmarks**  Table 5.2 presents the results on the 9 functions we analyzed for an access based attacker. Consistent with the results on cryptography benchmarks, analysis based on the infinite cache model is more efficient: on average, the age model takes over 100 times longer. Also, most of the functions that we test are potentially vulnerable to side channel attacks (examples in the next subsection).
Interestingly, there are two functions (binsrch & find xtreme element) in which the infinite cache model missed two positives reported by the age model. The cause is a key dependent branch which accesses different locations but they were previously used, and thus already in the cache. This example demonstrates the difference of those two abstract models: the infinite cache model is optimistic (that is, it optimistically assumes that the loaded memory locations were not evicted before the sensitive branch), while the age model is pessimistic (that is, it pessimistically assumes that the loaded memory locations were evicted before the sensitive branch).

### 5.8.2 Evaluation for trace-based attackers

Recall that a trace-based attacker can observe the intermediate cache states of the victim program. For each benchmark, we ran CaSym to perform a check on the symbolic state after every statement (following a topological order of the program’s control-flow graph); CaSym then stopped at the first point where a side-channel was found. Since CaSym’s error reporting includes the source line number where the side channel is, we then went to that line and fixed the problematic statement, as described in Section 5.6.2. Then we applied CaSym on the fixed program to find the next side channel. Through this iterative process, we were able to find a set of independent side channels in each benchmark.

**Cryptography Benchmarks**  Table 5.1 reports the number of side channels identified on the cryptography benchmarks. Similar to the results for access-based attacks, we found that the infinite cache model is more efficient than the age model, while they provide very close results (we discuss the only difference in sqr-alwys-mul shortly).

We inspected the error-reporting results and manually confirmed that for all cases, CaSym localized the side channels to the right lines and also confirmed that all reported
side channels are true side channels. It was a surprising result considering that CaSym’s symbolic execution approximates a program’s behavior, for example, when the program reads from arrays; also our loop transformation could also introduce false positives.

Next, we take a closer look at the positives. The side channels CaSym found in symmetric ciphers are due to the sbox tables being indexed by key-dependent variables in all AES/DES implementations. Below is a representative example from the AES Libgcrypt benchmark where \( \text{encT} \) is the encryption table and the array index \( sa[0] \) is derived from the key.

\[
\text{sa}[0] = \text{sb}[0] \oplus \text{key}[0][0];
\]

\[
\ldots \text{encT}[\text{sa}[0] \gg (0 \times 8)] \ldots
\]

The reason why the number of side channels for symmetric ciphers are high is because they contain multiple lines of code following the same sbox table access pattern as above.

For the three modular exponentiation algorithms implemented in Libgcrypt, CaSym found multiple side channels. The side channels are due to either array accesses indexed by the exponent bits or branches whose outcome depends on a key. A similar situation happens for the left-to-right algorithm (LR-mod-expo gcry). A simplified code snippet that depicts the issue can be found in Listing 5.1.

```c
if (c >= W) // c is tainted by
  the key
  c0 = 0;
else {
  e0 = (e >> (BITS_PER_MPI_LIMB - c));
  j += c - W;
}
```

Listing 5.1: LR-mod-expo Example

Interestingly, we notice that CacheD [15] reports no side channels in Libgcrypt 1.7.3.
The reason is two fold. First, two algorithms (sqr-alwys-mul gcry and sqr-mul gcry) are not used in the default configuration of Libgcrypt. Since CacheD [15] only explores side channels exhibited in an execution trace, side channels in those algorithms are missed. Second, CacheD [15] does not detect side channels due to secret-dependent branches (e.g., the side channels in the code snippet above), since it detects cache difference only for executions that follow the same control flow. The example in Listing 5.1 confirms that CaSym avoids the coverage issue of CacheD.

To assess false negatives, we treat the trace-based age model result as the ground truth, since under this model, no positive implies no cache-based side channel on most realistic caches (Property 2). Based on the results, the only case where the infinite cache model had a false negative was for the square and always multiply implementation. The relevant code snippet can be found in Listing 5.2.

```c
res = res * res; res = res % mod;
temp = res * base; temp = temp % mod;
if (((1 << 31) & expo) != 0)
{
    res = temp;
}
```

Listing 5.2: A False Negative of the Infinite Cache Model

**Database Benchmarks** The results on database benchmarks are summarized in Table 5.2 on the right. Both of our models perform well in the trace based attacker analysis. Again, analysis based on the infinite cache model is more efficient than the age model, while they provide very close results. In our evaluation, CaSym identified potential vulnerabilities in most benchmarks.
Among all positives, we do find one false positive in a function that sorts array elements. The false positive presented in Table 5.2 is due to an if-statement where one branch has a reset (due to our loop transformation) and the other one does not. Due to the information lost from issuing a reset, CaSym reports a side channel but in fact there is not one at this point. This causes two different scopes to be compared at the merge point, which causes a side channel reported. The false positive reported by CaSym is presented in Listing 5.3.

```c
if (nelems <= 1) {
    return nelems;
}

elemtype = skey->sk_subtype;
if (elemtype == InvalidOid) {
    elemtype = rd_opcintype[skey->sk_attno - 1];
}
...
last_non_dup = 0;
for (i = 1; i < nelems; i++) {
    ...
} // reset after loop
return last_non_dup + 1;
```

Listing 5.3: A False Positive in PostgreSQL

```c
for (i = 1; i <= keysz; i++) {
    datum = index_getattr(itup, attno, itupdesc, &isNull);
    result = DatumGetInt32(FunctionCall2Coll(
        &scankey->sk_func,
        scankey->sk_collation, datum,
        scankey->sk_argument));
    if (result != 0) // result is tainted
        return false; // early exit
```
Table 5.3: Evaluation results for symmetric ciphers with table preloading and pinning for trace-based attackers.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Preloading</th>
<th>Pinning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Infinite</td>
<td>Age</td>
</tr>
<tr>
<td></td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>AES gcry</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AES mbed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>triple-DES gcry</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>triple-DES mbed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DES glibc</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Listing 5.4: A True Postive in PostgreSQL

Based on the infinite cache model, our analysis was able to detect 20 unique newly found locations in the source code which could leak information to an attacker about a database key. These 20 reported locations contain both secret-dependent array accesses and secret-dependent branches. The former usually happens when a variable is tainted by the primary key and then used as an index in an array. The code snippet shown in Listing 5.4 shows a common pattern of a key-dependent branch in PostgreSQL.

The age model was able to detect even more positives in the benchmarks, finding 5 new potential vulnerabilities. This model found more locations since it considers temporal differences of when memory locations can be placed into the cache. Even though this model is more conservative, we see that it does not introduce any new false positives in our experiments.

5.8.3 Fixing side channels

We next discuss our experience of fixing the side channels that were discovered in the previous experiments. In symmetric ciphers, one common strategy to avoid side channels resulting from key-dependent table lookups is to preload their sbox tables. To
support that, our LLVM implementation supports a special `PRELOAD` attribute, which can be used by programmers to annotate their source code to specify what variables or arrays should be preloaded at the start of execution. A preloaded variable/array means that it is initially in the cache. We used this attribute to annotate our symmetric-cipher benchmarks and reevaluated them using CaSym. Table 5.3 presents the results of performing preloading in symmetric ciphers for trace-based attackers. We did not evaluate preloading/pinning for asymmetric cipher benchmarks since other techniques (e.g. scatter/gather) are used to secure them [14]. Also, we did not include the UFC implementation since it is already side channel free.

As expected, Table 5.3 shows that preloading is sufficient to eliminate the side channels in the infinite cache model: after the sbox tables are preloaded, they always stay in the cache and the following key-dependent table accesses will not change the cache state. For the age model, side channels still exist since the age model tracks the ordering of memory accesses; the preloading at the beginning will not change the ordering of memory accesses in the following execution.

As we have discussed, another strategy for fixing side channels is to pin some data in the cache. We also implemented a special attribute for programmers to specify what variables/arrays should be pinned into the cache. Table 5.3 presents the results of performing pinning of tables in symmetric ciphers for trace-based attackers. It shows that all side channels disappear with this fix, across all cache models. When cache entries are pinned to the cache, they are never evicted; therefore, CaSym does not update the ages of pinned cache entries. The ages at the beginning are the ages at the end for the pinned entries, preventing CaSym from reporting them as causes of side channels.
<table>
<thead>
<tr>
<th>Benchmarks w/o Preloading</th>
<th>Access</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>AES gcry</td>
<td>timeout</td>
<td>64</td>
</tr>
<tr>
<td>AES mbed</td>
<td>timeout</td>
<td>17</td>
</tr>
<tr>
<td>triple-DES gcry</td>
<td>1654</td>
<td>128</td>
</tr>
<tr>
<td>triple-DES mbed</td>
<td>8531</td>
<td>48</td>
</tr>
<tr>
<td>DES glibc</td>
<td>1044</td>
<td>2</td>
</tr>
<tr>
<td>UFC glibc</td>
<td>✓ 0.09</td>
<td>0</td>
</tr>
<tr>
<td>sqr-alwys-mul gcry</td>
<td>2.64</td>
<td>3</td>
</tr>
<tr>
<td>sqr-mul gcry</td>
<td>2.19</td>
<td>4</td>
</tr>
<tr>
<td>LR-mod-expo gcry</td>
<td>23.45</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.4: Evaluation results using the LRU model; the table shows results for an access based attacker on the left and a trace based attacker on the right.

### 5.8.4 Abstract vs. Concrete Cache Models

In order to show the effectiveness of the infinite cache model and the age model, we implemented the following features as described in Section 5.5.4: the LRU replacement policy, cache lines, and cache associativity.

**LRU replacement policy** Table 5.4 shows the evaluation results on our cryptography benchmarks, based on the LRU cache model with 2k cache slots. We note that the LRU model takes a significant amount of time to finish (the SMT solver even timed out with a limit of 3 hours for three tests). Despite the significantly longer execution time, for all tests that finish, we note that it reports exactly the same results as the age model, and very similar results as the infinite cache model. This result demonstrates that our abstract cache models offer better balance between precision and efficiency, when compared to more concrete cache models.

**Cache line size and cache associativity** We also implemented cache models with various cache line sizes and cache associativity, as discussed in Section 5.5.4. We tested
our benchmarks using common cache associativity values and cache line sizes. The results reported were the same as without specifying associativity or cache line size\(^5\).

### 5.9 Limitations

As mentioned before, the symbolic execution of CaSym does not track array contents. That is, reading from an array returns an arbitrary value. This in theory can cause false positives, but has not caused problems during our experiments. The implementation also does not support dynamic allocation and deallocation of memory. Array sizes must be statically declared. Our support of pointers is also limited. We require that a pointer variable must be initialized to the base address of some array. Pointer arithmetic on the pointer is allowed, but the pointer can only reference locations inside the array it was initialized to for the entire lifetime of the pointer. This reflects how pointers are used in crypto applications, but it is not the case for general C/C++ applications. Finally, CaSym inlines all functions before performing symbolic evaluation. We plan to gradually lift these restrictions so that CaSym can support more general applications beyond crypto applications.

### 5.10 Summary

In this chapter, we presented a novel technique for detecting and mitigating cache-based side channels called CaSym. Next is a brief summary of the contents of this chapter:

- We presented CaSym, which takes a program in a compiler IR (specifically, LLVM IR) as input and performs symbolic execution to track both program and cache

\(^5\)This does not mean different associativity or cache line sizes will never impact the result; it just means that typical values for associativity and cache line sizes have no impact on the benchmarks we evaluated.
states. The symbolic state produced by CaSym’s symbolic execution is used to construct a formula fed to an SMT solver. A satisfying solution to the formula produces public values as well as two sensitive values that demonstrate the existence of a cache-based side channel: those values trigger two program executions that result in different cache states.

- We introduced novel cache models that define two operations (initialize and update) and an equality test. Unlike other existing models, they hide implementation details and provide good balance between precision and generality. The infinite cache model represents an optimistic view of cache: if there is a side channel under the this model, then the side channel likely will exist in other more realistic cache models. The age model represents a pessimistic view of the cache: if there is a side channel under this model, then the side channel likely will exist in some realistic cache models. Compared with concrete cache models (e.g., LRU), the infinite cache model and the age model offer significantly better scalability and comparable precision.

- From the counterexample (i.e., values that trigger two program executions that result in different cache states), CaSym utilizes the solver solution to localize the causes of the identified side channels and reports the problematic program points. From those problematic program points, mitigation mechanisms, such as preloading and pinning, can then be applied to eliminate those side channels in a straightforward manner.

- We have applied CaSym on a set of crypto benchmarks using both symmetric and asymmetric ciphers as well as database benchmarks. The experiments confirmed that CaSym can identify known side channels in these benchmarks, report precise information about where the side channels are and why they are side channels,
as well as verify that some benchmarks are side-channel free (based on realistic
attack and cache models). We also present new vulnerabilities, which to the best
of our knowledge have not been found before in the glibc library code and the
PostgreSQL database.
Detecting Cache Side Channels with Speculative Execution

6.1 Motivation

Much work has been done to automatically detect cache side channels as discussed in Chapter 5. However, most of them ([13–15, 22, 71, 80]) fall short in soundness in a speculative world: they only detect conventional cache side channels (i.e., in the absence of speculative executions). To detect code vulnerable to Spectre attacks, a recent tool oo7 [16] searches for vulnerable code patterns – however, no rigorous definition of side-channel security is provided; as a result, the patterns might still miss vulnerabilities. More recently, formal security definitions [17, 19, 81] catching cache side channels in a speculative world have been developed. However, they all fall short in flexibility and precision: they use the program counter security model [50], which disallows any branching on sensitive data. Note that while the program counter security model is commonly used to formalize timing channel free code (i.e., to ensure that the code runs in constant time), it is also too restrictive for cache side-channel free code (i.e., to ensure...
that the code leaks no information via cache). Such restriction motivated the development of static analysis with more permissive attack models and more precise cache models [13–15, 22, 71], which allow permissive and efficient countermeasures against cache attacks (e.g., aligning branches of a sensitive conditional into a single line of the instruction cache, balancing memory accesses between branches, preloading lookup tables, etc.), have been adopted as options in popular cryptographic implementations such as libgcrypt and OpenSSL [14].

In this chapter, we develop a static program analysis that automatically detects cache side channels in the source code, with four important goals:

- **Soundness**: the analysis catches all cache side channels in a speculative world.
- **Flexibility**: the analysis reasons about security against a wide range of cache attacks, e.g., asynchronous and synchronous attacks.
- **Precision**: the analysis embodies abstract cache models to reason about cache status.
- **Efficiency**: the analysis is applicable to real-world applications.

In this chapter, we first show that with speculative execution, it is *subtle* to define cache side channel security against a wide range of adversaries. In particular, the idea of combining analysis tools for conventional cache side channels and tools for speculative execution side channels (i.e., tools enforcing security definitions in [17, 19]) do not necessarily cover all possible cache side channels on hardware with speculative execution. We construct simple code patterns to demonstrate that cache side channels might be missed by *either* a conventional cache side channel detector [13–15, 22, 71] that does *not* assume the program counter security model [50], *or* a speculative side channel detector built on existing security definitions (e.g., [17, 19]). We also build covert channels
on the code patterns to demonstrate their feasibility. These patterns demonstrate that the ignored attack surface in previous security definitions is realistic on commodity hardware. Although we have not detected these patterns in the wild in our experiments, we emphasize that our contribution is to identify limitations of existing security definitions, and show that an attacker can utilize them to bypass state-of-the-art tools for detecting cache side channels.

Second, we propose a security definition for side-channel security against all forms of cache side channels (including conventional, Spectre, and the new Spectre instances introduced in this chapter). Our security definition can be soundly applied to a variety of attack and cache models. Then, we propose SpecSafe to detect violations of the security definition in reasonably sized applications. At the core of SpecSafe is speculative-aware symbolic execution. For better scalability, we also use SpecSafe in tandem with taint analysis for large programs. The taint analysis component helps filter out code regions that cannot be vulnerable to cache attacks (e.g., code region where no secret information is used). The symbolic execution then performs fine-grained analysis on the remaining code to identify vulnerable code regions. In order to show the effectiveness of SpecSafe, we apply it to instances of the Spectre vulnerability by Kocher [2], our newly discovered vulnerabilities, along with multiple crypto routines from libgcrypt 1.8.5.

### 6.2 Threat Model

Similar to past chapters, we assume an adversary who is co-located on the same physical machine as the victim. The adversary cannot directly observe the victim’s program counter and memory bus; however, the adversary shares a cache with the victim, can observe the state of the shared cache, and can control the victim’s branch predictors. Moreover, we consider two kinds of attackers (following terminology in [13,22]):
• access-based: an attacker can learn the shared cache state only after the victim program terminates; this models synchronous cache attacks [5].

• trace-based: an attacker can learn the shared cache state after each program point in the victim program; this models asynchronous cache attacks [25–27].

The focus of this chapter is side-channel leakage caused by the cache footprint left by program execution (speculatively or not). We follow [14] to allow a wide range of attack models such as a block/page attacker who can monitor memory accesses at the granularity levels of cache lines/memory pages respectively, and attackers who share data and/or instruction cache with the victim program.

Among all known variants of Spectre attacks, we focus on Spectre Variant 1 attacks [11], also classified as Spectre-PHT attacks in [35], which utilize speculation due to conditional branches. Hence, our threat model not only covers “bounds check bypass” variants as described by Intel [119], but also other attacks exploiting conditional branch misprediction, including the new instances we present in Section 6.3.1.

However, speculative execution caused by indirect branches [11], return instructions [24, 43] and speculative stores [23] are outside the scope of this chapter; we leave a static analysis of those Spectre variants as future work.

6.3 New Speculative Side Channels

```
if (idx < b_size) {
    temp = A[B[idx]*512]
}
```

Listing 6.1: Canonical Spectre Example

In this section, we first present examples for new instances of Spectre Variant 1 attacks and show such attacks are possible in practice. What makes the new attacks
if (key == 0) {
    temp = A[key*512];
} else {
    temp = A[key*0];
}

if (key%2 == 0) {
    temp = A[((key+1)%2)*512];
} else {
    temp = A[(key%2)*512];
}

if (key == 0) {
    d = a;
} else {
    ... {
        ...; b = 0;
    }
    else c = 0;
}

(a) (b) (c)

Figure 6.1: Cache side channels that elude existing security notions. key is a one-bit secret in all the examples.

especially interesting is that, as we show in Section 6.5, existing security definitions of cache side channels, with or without speculative execution, fail to catch them; our new security definition in Section 6.5, on the other hand, captures all avenues of cache-based leakage with or without speculative execution.

6.3.1 Attack Code Snippets

We present three code snippets that demonstrate new kinds of speculative cache attack, which we call conditional branch speculative side channels, abbreviated as CB-channels.

To clearly show the leakage in each code snippet, we use ”begin...rollback” to enclose memory addresses accessed by instructions that are speculatively executed, but are rolled back due to misprediction. Consider the code in Listing 6.1. With idx ≥ b_size and the predictor predicting the true branch to be taken, the following memory accesses are generated:¹ begin B + idx, A + B[idx] * 512 rollback

Due to the accessed memory address A + B[idx] * 512 (during speculative execution), the cache line being accessed reveals the value of B[idx]; hence, the program leaks the memory content at B[idx].

¹We provide the formal program semantics that emits traces and cache models that compute cache states from traces in Appendix A.1.
**Instance 1**  The first instance of CB-channels is shown in Figure 6.1a. In this example, \( key \) is a one-bit secret and we assume \( key \times 512 < A_{size} \) (i.e., there is no out-of-bound access). Although the code always accesses \( A[0] \) without speculation, it emits the following execution subtrace when \( key = 1 \) and the predictor predicts the true branch to be taken: \( \text{begin } A + key \times 512 \text{ rollback.} \)

Therefore, the value of \( key \) directly affects the cache line being accessed; the value of \( key \) can be revealed via a cache attack.

**Instance 2**  The second instance, shown in Figure 6.1b, is more subtle than Instance 1. In this example, only \( key \) is confidential. Depending on the value of \( key \), the following access subtraces will be generated when the predictor predicts the true branch to be taken:

\[
A + 512 \quad \text{when } key \% 2 = 0
\]

\[
\text{begin } A \text{ rollback, } A + 512 \quad \text{when } key \% 2 \neq 0
\]

In other words, the cache line corresponding to address \( A \) is accessed \textit{if and only if} the last bit of \( key \) is 1; note that since the hardware does not roll back microarchitectural effects, including cache effect, a cache attack can reveal at least one bit of key. We note that in this example no information is leaked from memory accesses during speculation, but \textit{whether speculation happens or not} leaks information. Leaking data solely based upon whether speculation occurs or not, to the best of our knowledge, has \textit{not been previously explored} as a source of leakage. Technically, the new attacks can be classified as instances of Spectre Variant 1 attacks since they use conditional branches. We note that while this sample code leaks one bit at a time, if it were in a loop, it could leak multiple sensitive bits.
Instance 3  The last instance, shown in Figure 6.1c, is similar to the example in Figure 6.1b since it also leaks data based on whether speculative execution occurs. However, the reason for the leakage is a bit different: in this instance, the leakage occurs as a result of whether or not the variable $a$ is cached or not. This is because speculation will likely occur only when variable $a$ is uncached. Depending on the value of $key$, the following two traces will be generated when the predictor predicts the false branch at line 5:

$$\text{key, } a, d, \ldots, a, b \quad \text{when } key = 0 \text{ and } a = 0$$

$$\text{key, } \ldots, a, \text{begin c rollback}, b \quad \text{when } key \neq 0 \text{ and } a = 0$$

Note that when $key = 0$, speculative execution is unlikely to occur for any reasonable speculation depth since $a$ is cached at line 5 due to an earlier read in $\text{d=a}$; the second trace has $a$ uncached when reaching the second branch, and therefore there is speculative execution since the processor will need to wait for $a$ to be fetched from memory. Whether $key$ is 0 or not is revealed by two side channels in the code (1) whether the cache lines corresponding to $a$ and $d$ are accessed after the first branch, and (2) whether the cache line corresponding to variable $c$ is accessed or not. We note that the first one is a conventional side channel, but the second one only shows up with speculative execution.

Although all examples in Figure 6.1 contain key-dependent branches; however, they are not essential. Consider the following variation of Figure 6.1c:

```c
a = arr[key*512];
...
if (arr[0]) b = 0;
else c = 0;
```
In this case, we know that \( arr[0] \) is cached only if \( key \) is zero. As discussed previously, the presence or absence of speculation again leaks the value of \( key \) in this example.

### 6.3.2 Measuring Channel Throughput

For each of the samples in Figure 6.1, we construct an attack to reveal confidential data by mostly following the Spectre attack code presented in [11] as a template. The only subtlety is for Instance 3; we need to ensure enough time elapses to allow the branch condition’s data cached before line 5 while not allowing instructions to cause it to be evicted from the cache. We insert a sufficient number of \texttt{NOP} to tackle the issue.

Moreover, we demonstrate how to use the code snippets as gadgets when they are embedded in sensitive applications to exfiltrate sensitive data. This is interesting since, as we will show in Section 6.5, state-of-the-art analysis for speculative side channels cannot detect them. In particular, we embedded Instance 1 code into the AES encryption routine so that it reveals one bit of the encryption key per AES encryption.

### 6.4 Attack Setup

To reveal secrets from the samples in Figure 6.1, there are three technical challenges: (1) training the branch predictor to control branch mispredictions and consequently speculative execution; (2) detecting which memory was accessed; and (3) ensuring enough time elapses to allow the branch condition’s data cached for Instance 3.

For challenge 1, we mostly follow the Spectre attack code presented in [11] as a template. Ensuring speculative execution happens in a controlled manner requires carefully training the branch predictor. This is done by providing training data to cause a branch to be taken \( \sim 5 \) times in succession. This is followed by an attack phase, where an input
that causes the branch not to be taken is given and will cause speculative execution. We used a total of 30 invocations of the vulnerable code, including 25 training iterations and 5 attack iterations, before we attempt to observe what memory addresses were accessed by the vulnerable code.

For challenge 2, we set up an intra-process covert channel such that the sample code keeps running at the maximum speed to send one bit of the key in each iteration. At the same time, the receiver’s code synchronously observes the target cache lines using the Flush+Reload technique [27]. By adjusting the offset of the memory accesses in the vulnerable code, we are able to map the addresses in the cache to an exact index used in the victim program.

For challenge 3, we insert \texttt{NOP} instructions between the if statements in the example to allow the variable $a$ to be cached if it is accessed. The key is to observe here is whether or not speculation occurs now depends on the cachedness of variable $a$; thus, the last step here is to observe if speculative execution happens by checking if both variable $b$ and variable $c$ are cached.

**Results** Table 6.1 shows the throughput of each code snippet with various accuracy rates. Note that each of the code patterns presented in Figure 6.1 can leak thousands of bits per second with high accuracy. The results indicate that CB-channels can leak significant amounts of data in a short period of time, even though they do it bit by bit.

Table 6.1: The throughput (bits per second) of the new side channels in Figure 6.1 as well as AES-128 embedded with Instance 1 at different levels of accuracy.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>&gt;75%</th>
<th>&gt;95%</th>
<th>&gt;99%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance 1</td>
<td>64747</td>
<td>11181</td>
<td>2038</td>
</tr>
<tr>
<td>Instance 2</td>
<td>7040</td>
<td>2652</td>
<td>1630</td>
</tr>
<tr>
<td>Instance 3</td>
<td>5355</td>
<td>120</td>
<td>12</td>
</tr>
<tr>
<td>AES-128 (with Instance 1)</td>
<td>2456</td>
<td>1816</td>
<td>1604</td>
</tr>
</tbody>
</table>
Statements
\[ s ::= s_1; s_2 \mid \text{skip} \mid \text{fence} \mid X := E \mid A[X] := E \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \mid \text{while } B \text{ do } S \]

Expressions
\[ E ::= n \mid X \mid A[E] \mid E_1 \otimes E_2 \]

Boolean Expressions
\[ B ::= E_1 \odot E_2 \mid \neg B \mid B_1 \land B_2 \mid B_1 \lor B_2 \]

where \( \otimes \) represents binary arithmetic operations, and \( \odot \) represents binary comparison operators

Figure 6.2: Language Syntax

We also note that the throughput difference among the attacks largely depends on the number of operations in each code fragment.

6.5 Speculative-Aware non-interference

In this section, we first formalize attack and cache models to reason about cache side channels. Then, we show that except under the most conservative model, none of the existing security definitions for cache side channels can identify the new CB-channels in Figure 6.1. Then, we introduce speculative-aware non-interference (SANI), a new semantic definition of cache side-channel security against a variety of attack and cache models.

Speculative Program Semantics  To formalize SANI and compare it with previous security definitions, we formalize the syntax (shown in Figure 6.2) and speculative semantics of an imperative language. Most language features are standard, such as sequential composition, no-op, assignment, branch, and loop. The only non-standard command is \texttt{fence}, which is a special command that will temporarily disable speculation.

Unlike standard semantics without speculation, the semantics is parameterized on two features:
• **prediction oracle \( \mathcal{O} \):** a partial function that takes the id of a branch command and returns a queue of booleans that predict the outcomes of the branch.

• **speculative transaction’s length \( w \):** the size of the reorder buffer \([120]\).

Since the semantics mostly follows prior work \([17]\), we omit the evaluation rules (formalized in Figure A.1 in the Appendix).

Given a program \( s \), an oracle \( \mathcal{O} \) and transaction length \( w \), we use \( [s]_{(m,w)}^{\mathcal{O}} \) to denote the sequence of emitted events of \( s \) with initial memory \( m \). Each event tracks the memory locations being accessed and program instructions being executed; we use \( \text{pc}(i) \) to denote the latter. Finally, we use \( \tau \) to denote event traces, and assume a trace view model \( \text{view} \) (to be defined next), which intuitively specifies the side channel information visible to the attacker. Hence, two events traces \( \tau \) and \( \tau' \) are indistinguishable for an attacker with \( \text{view} \) if \( \text{view}(\tau) = \text{view}(\tau') \).

**Attacker View and Cache Model**

To reason about cache side channels, one crucial and challenging question is how to model attacker’s view of the cache. To do so, we first assume **no cache model** and define attacker’s view as done in \([14]\) – an attacker model can be abstracted as a projection of \( \tau \) that is visible to an attacker. For instance, the most conservative model is when the attacker can **directly** view all the memory locations being accessed and program instructions being executed (i.e., \( \text{view}(\tau) = \tau \)).

A **block-trace observer** \([14]\) models attackers that can monitor memory accesses at the granularity level of cache lines; it can be formalized (on a 64-bit machine with cache line size of 512 bits) as \( \tau \upharpoonright_{\text{mem}}^{63:8} \), a projection of \( \tau \) that only contains bits 8 to 63 of each memory access, excluding \( \text{pc}'s \). A hierarchy of memory trace observers can be formalized in a similar way \([14]\).

The definition above implicitly uses trace projection to estimate cache effects. For
better accuracy, we can tailor a more detailed cache model $\text{cacheM}(\tau)$ that computes the (abstract) cache state at the end of a subtrace $\tau$, which captures the attacker view at the corresponding program point. Hence, we define the attacker view with projection $\downarrow$ and cache model $\text{cacheM}$ on the entire trace $\tau$ as:

$$\text{view}(\tau) \triangleq \text{cacheM}(\emptyset), \text{cacheM}(\tau \downarrow_{i}), \cdots, \text{cacheM}(\tau \downarrow_{|\tau|})$$

where $\tau \downarrow_{i}$ represents the prefix of $\tau$ up to position $i$. Note that when $\text{cacheM}(\tau)$ returns the last element of $\tau$, a cache model is absent. As another example, a more realistic cache model (e.g., LRU model) might only return the last memory location on $\tau$ when it is not among the most recently used locations. We refer to [13, 22] for various cache models ranging from very intricate models including cache size and replacement policies to abstract cache models that provide a good balance between analysis scalability and precision [22].

In this chapter, we develop a general static analysis (Section 6.6) that can be parameterized on any attacker view defined as above.

However, for simplicity, we will use a concrete and representative attacker view, called block trace view, for the rest of this section:

$$\text{view}_{\text{BT}}(\tau) \triangleq \tau \downarrow_{63:8_{\text{mem}}}$$

This attacker view is particularly interesting since (1) most known cache-based attacks exploit observations at the granularity of cache lines, (2) most tools for detecting conventional cache side channels use this attacker view\(^2\), and (3) recall that we will show that conventional side channel detectors are unable to catch the new CB-channels in

---

\(^2\)We note that this is the attack model assumed by CacheD [15] and CacheS [71]; it is called “block-trace observer” in [14].
Figure 6.1; a tool based on more precise cache models [13, 22] will miss vulnerabilities if they are missed under $\text{view}_c$.

### 6.5.1 Limitations of Existing Definitions

#### 6.5.1.1 Conventional Cache Side Channels

Cache side-channel security can be formalized as a noninterference property [108]: confidential data does not affect the cache state. Let policy $P$ be a set of public variables.

We first define a low-equivalence relation on memory as follows:

**Definition 1 (Low Equivalence).** Given a policy $P$, two memories $m_1$ and $m_2$ are low-equivalent according to $P$, written as $m_1 \simeq_P m_2$ if and only if $\forall x \in P. m_1(x) = m_2(x)$.

To capture side channels without speculation, we note that given an oracle that perfectly predicts every single branch outcome and with $w = 0$, the program semantics coincides with standard semantics without speculation. Hence, we define the perfect oracle to be the one that correctly predicts every single branch outcome for the execution of program $s$ on memory $m$.\(^3\)

To simplify notation, we use $\llbracket s \rrbracket_{(m,w)}^{\text{perf}}$ to denote the execution of $s$ under $m$, transaction length $w$, and the corresponding perfect oracle. Cache side-channel security without speculation is then formalized as follows:

**Definition 2 (Cache noninterference (CNI)).** A program $s$ with policy $P$ satisfies CNI if for all $O$, $m_1 \simeq_P m_2$, we have

$$\text{view}(\llbracket s \rrbracket_{(m_1,0)}^{\text{perf}}) = \text{view}(\llbracket s \rrbracket_{(m_2,0)}^{\text{perf}})$$

3Note that since a program $s$ might take different control flows during different executions, the perfect oracle is parameterized with both $s$ and $m$.\(^3\)
that the address of A and the value of A’s elements within bound are public. Then, it is easy to show that with $m_1 \simeq_P m_2$ but $m_1(secret) = 0$ and $m_2(secret) = 1$, their memory trace projections are different between $m_1$ and $m_2$. That is, the program violates CNI, and hence, is vulnerable to traditional cache attacks with $\text{view}_{BT}$.

6.5.1.2 Speculative Side Channels

Definition 2 falls short for Spectre attacks since it completely ignores speculative execution. Recent work [17,19] proposed new security definitions to identify side channels that only exhibit themselves with speculative execution. In this chapter, we focus on the definition called *speculative noninterference (SNI)* [17] since both security definitions in [17] and [19] are similar, despite different ways of formalism [19].

Informally, SNI requires that, for every $m_1 \simeq_P m_2$, if their non-speculative traces have exactly the same event traces, including program counters, then their speculative traces will do the same. We present a version of SNI generalized with an attack view as follows and emphasize that the original SNI definition in [17] assumes $\text{view}_{\text{SNI}}(\tau) = \tau$.

**Definition 3 (SNI-GEN).** A program $s$ with policy $P$ satisfies SNI-GEN with transaction length $w$ if for all $m_1 \simeq_P m_2$,

$$
\text{view}(\llbracket s \rrbracket_{(m_1,0)}^{\text{perf}}) = \text{view}(\llbracket s \rrbracket_{(m_2,0)}^{\text{perf}}) \implies \text{view}(\llbracket s \rrbracket_{(m_1,w)}^{\text{mis}}) = \text{view}(\llbracket s \rrbracket_{(m_2,w)}^{\text{mis}})
$$

where $\llbracket s \rrbracket_{(m,w)}^{\text{mis}}$ is the execution of $s$ under $m$, transaction length $w$, and an oracle that always mispredicts (called the *always-mispredict speculative semantic* in [17]).

6.5.1.3 Limitations of Existing Security Definitions

SNI is defined to detect side channels that only show up with speculative execution. However, as we show next, information leakage in Figure 6.1 remains "undetected"
by both CNI and SNI. We will discuss instance 2 (Fig. 6.1b) in detail to identify the limitations; other variants are also vulnerable for similar reasons as instance 2.

With instance 2, consider two low-equivalent memories with \( m_1(key) = 0 \) and \( m_2(key) = 1 \), we have

\[
\begin{align*}
[s]_{\text{perf}}^{(m_1,0)} &= \text{key, pc(2), key, A + 512, temp} \\
[s]_{\text{perf}}^{(m_2,0)} &= \text{key, pc(5), key, A + 512, temp} \\
[s]_{\text{mis}}^{(m_1,w)} &= \text{key, begin pc(5), key, A, temp rollback, } \\
&\quad \text{pc(2), key, A + 512, temp} \\
[s]_{\text{mis}}^{(m_2,w)} &= \text{key, begin pc(2), key, A, temp rollback, } \\
&\quad \text{pc(5), key, A + 512, temp}
\end{align*}
\]

**Limitation 1:** SNI, which instantiates Definition 3 with \( \text{view}_{\text{SNI}}(\tau) = \tau \), assumes a (conservative) cache side channel detector that rejects any program with a secret-dependent branch. This is dangerous since the assumption is implicit in the definition, and state-of-the-art detectors for conventional cache side channels do not necessarily follow this assumption: as discussed earlier, most tools for detecting conventional cache side channels use the block trace view \( \text{view}_{\text{BT}} \), or an even more precise cache model.

To see why a mismatch in attacker views is problematic, we first point out that with the block trace view \( \text{view}_{\text{BT}} \) (which filters out PC values in traces), Definition 2 holds. Note that despite a secret-dependent branch in instance 2, there is no cache side channel in non-speculative execution, since the same sequence of memory addresses are accessed. Further, SNI holds since the assumption \( \text{view}_{\text{SNI}}([s]_{\text{perf}}^{(m_1,0)}) = \text{view}_{\text{SNI}}([s]_{\text{perf}}^{(m_2,0)}) \) is false due to \( [s]_{\text{perf}}^{(m_1,0)} \neq [s]_{\text{perf}}^{(m_2,0)} \). Hence, both CNI (with \( \text{view}_{\text{BT}} \) or a more precise cache model) and SNI (with \( \text{view}_{\text{SNI}} \)) fail to reject the vulnerable code.
of instance 2. But, as we showed in Section 6.3.1, the code leaks information via a cache side-channel only when speculative execution is enabled.

**Limitation 2** Instance 2 also shows a more fundamental limitation of SNI and its generalized form SNI-GEN: they both ignore leakage resulting from whether or not to roll back a prediction transaction.

To see that more clearly, we use view\textsubscript{BT} for both CNI and SNI-GEN, which avoids the mismatching attacker view issue discussed in Limitation 1. It is easy to check that the traces satisfy both Definitions 2 and 3 (with view\textsubscript{BT}) due to the same sequences of memory addresses being accessed under oracles perf and mis. Nevertheless, whenever the branch predictor \(O\) is trained to predict that the true branch should be taken, the program will produce the following two traces given \(m_1\) and \(m_2\) respectively:

\[
\begin{align*}
[s]_{O_{(m_1,1)}} &= \text{key,pc(2),key,A + 512,temp} \\
[s]_{O_{(m_2,1)}} &= \text{key,begin pc(2),key,A,temp rollback, pc(5),key,A + 512,temp}
\end{align*}
\]

Hence, the value of key is leaked by deciding whether memory location A is accessed during the execution.

With a more careful inspection, we observe that the issue here is that SNI only checks information leakage between two executions that both mispredict. However, the fact that misprediction happens might also leak information. Hence, SNI-GEN is insufficient under various attack models, such as view\textsubscript{BT}: with the same predictor, one execution (\([s]_{O_{(m_1,1)}}\)) might not have a misprediction while the other execution does (\([s]_{O_{(m_2,1)}}\)). As we demonstrated in Section 6.3.1, an attacker can reveal at least one bit of key from Instance 2.
6.5.2 Speculative Aware Noninterference

Due to the limitations of SNI-GEN, we introduce a new noninterference definition called Speculative Aware noninterference (SANI). The idea behind SANI is to compare the behavior of the program’s memory access trace with any possible outcomes of the branch predictor, without assuming the original program is side-channel free. More formally:

**Definition 4** (Speculative Aware noninterference (SANI)). A program $s$ with policy $P$ satisfies SANI w.r.t. attacker view $\text{view}$ and transaction length $w$ if $\forall \mathcal{O}$, $m_1 \simeq_P m_2$, we have

$$\text{view}(\llbracket s \rrbracket^\mathcal{O}_{(m_1,w)}) = \text{view}(\llbracket s \rrbracket^\mathcal{O}_{(m_2,w)})$$

To see how the new definition correctly identifies CB-channels, consider Instance 2 again. Given an oracle that predicts the true branch, and two low-equivalent memories with $m_1(key) = 0$ and $m_2(key) = 1$, we have

$$\llbracket s \rrbracket^\mathcal{O}_{(m_1,1)} = \text{key}, \text{pc}(2), \text{key}, A + 512, \text{temp}$$

$$\llbracket s \rrbracket^\mathcal{O}_{(m_2,1)} = \text{key}, \text{begin pc}(2), \text{key}, A, \text{temp rollback}, \text{pc}(5), \text{key}, A + 512, \text{temp}$$

Hence, SANI is violated since the cache line corresponding to address $A$ is only accessed in the second trace.

**Take Away** SNI [17] (i.e., Definition 3 instantiated with $\text{view}_{\text{SNI}}$) is not compatible with many tools for detecting conventional cache side channels. Under other (more precise) cache models, however, SNI-GEN is unsound since they ignore leakage resulting from whether or not to roll back a prediction transaction, as illustrated by Instance
2. On the other hand, the SANI definition (Definition 4) is applicable to a variety of cache models, including the ones used by state-of-the-art cache side channel detectors. To classify conventional vs. Spectre side channels, we can use CNI (Definition 2) vs. SANI. For the examples presented in Figure 6.1, we can correctly classify Instance 1 and Instance 2 as Spectre side channels since they both violate SANI but not CNI.

### 6.6 SpecSafe

In this section, we present a speculative-aware symbolic executor, SpecSafe, to enforce SANI. Compared with existing symbolic executors that are capable of detecting Spectre-like attacks (e.g., [17, 81]), SpecSafe adopts a novel code transformation, which benefits from the following property: if the transformed code is free of conventional side channels (i.e., satisfying CNI in Definition 2), then the original code satisfies SANI. The novel design allows us to reuse existing conventional side channel detectors on the transformed code to enforce SANI on the original code. Furthermore, the transformation soundly removes loops in the original code. Hence, SpecSafe avoids the code coverage limitation of other symbolic executors (i.e., the code is not verified until all paths are analyzed by those tools). Although soundly removing loops is only possible with the
cost of some precision loss, we show that SpecSafe is still precise enough for real-world applications in Section 6.7.

SpecSafe’s workflow is shown in Figure 6.3. SpecSafe performs code transformation on LLVM IR code and then symbolic execution on the transformed code. The symbolic execution reasons about architectural states (e.g. variable values), micro-architectural states (e.g. cache states), and speculative behavior. Security conditions needed for SANI are formalized as logical formulas, which are sent to an SMT solver (Z3). If the SMT solver determines the formula is satisfiable, it will also report line numbers identifying the vulnerabilities, making it easy to patch the program and restart verification. If the SMT solver reports the formulas are unsatisfiable, it is safe to conclude there is no cache-based leakage in the program under conventional or speculative execution.

6.6.1 Program transformation

In order to model the speculative behavior of a program, we introduce a program transformation that embeds the speculative behavior into the transformed program. The syntax of transformed programs includes two new primitives:

- capture(id): an instruction that records the current architectural state (i.e., memory and registers) of the program so that it can be later retrieved with id.

- rollback(id): an instruction that resets the program’s architectural state (but not the microarchitectural state such as cache) to the recorded state with id.

To soundly remove loops, the transformed program also includes two security-related primitives [22]:

- reset: an instruction that sets current architectural and microarchitectural state to an arbitrary value.
Figure 6.4: Transformation rules for composition, if and while with transaction length parameter $w$. The helper function $[S]_w$ statically extracts $w$ instructions from $s$, unless a fence is encountered during that. Furthermore, $s_*$ is used to denote the sub-program right after the sub-program being analyzed (i.e., the extra instructions being padded in case of misprediction).

- **check $K$:** an instruction that checks cache side channels assuming only the secret-variable set $K$ carries confidential data at the last reset.

The most interesting rules are listed in Figure 6.4; for all other instructions, the transformed program is the same as the original one.

**Composition**  A sequential composition $s_1; s_2$ is transformed to $s_1; \text{check } K; s_2$, where $K$ is a set of sensitive variables at the last reset, provided by a sound taint analysis (described in Section 6.7). The inserted check ensures that the attacker view right after $s_1$ reveals no confidential data.

**If transformation**  Intuitively, programs during ordinary execution have two possible control flows when it encounters a branch; with speculation, there are four possible
control flows due to the two extra branches on a fresh variable $pred_{id}$, which is true if and only if the predictor predicts the true branch. For a misprediction, speculatively executed code up to transaction length $w$ (i.e., $[s'_2; s'_w], [s'_1; s'_w]$) are explicitly instrumented into the resulting code, followed by $rollback(id)$, which resets the architectural state (but not the microarchitectural state) to the state before the branch.

Consider the example from Figure 6.1b. Following the transformation rule in Figure 6.4, the source code is transformed into a speculation-aware form, assuming $w = 1$:

```plaintext
capture(1);
if (key%2 == 0) {
    if (!pred_1) {
        temp := A[(key%2)*512];
        rollback(1);
    }
    temp := A[((key+1)%2)*512];
} else {
    if (pred_1) {
        temp := A[((key+1)%2)*512];
        rollback(1);
    }
    temp = A[(key%2)*512];
}
```

Hence, the transformation faithfully models four possible control flows of the example, assuming $E_1=A[(key%2)*512]$ and $E_2=A[((key+1)%2)*512]$:

```plaintext
capture(1);temp:=E1;rollback(1);temp:=E2;
capture(1);temp:=E2;
```
capture(1); temp := E2; rollback(1); temp := E1;
capture(1); temp := E1;

In practice, the program will speculate up to a certain depth \( w \), specified as a parameter of SpecSafe. The transformation statically inserts \([s]_w\), which denotes the first \( w \) instruction of \( s \). Note that with a fence command, the remaining commands are cut off in the construction of \([s]_w\).\(^4\)

**Loop transformation**  The loop transformation is motivated by the transformation presented in [22]. However, the latter faces two limitations when being applied to speculative execution. First, it does not consider speculative behavior. Second, it assumes that the loop condition is not sensitive. In comparison, our novel loop transformation is both speculative-aware and allows sensitive loop conditions. Moreover, we formally prove that our transformation soundly captures both conventional and speculative side channels.

The transformation, presented in Figure 6.4, performs the following major tasks. First, it conservatively resets the initial state to over-approximate both architectural and microarchitectural states right before each loop iteration. Second, it introduces two new variables \( \text{pred}_id \) and \( \text{pred}'_id \) to model the outcome of the branch predictor. Unlike in the if-statement case, two such variables are needed since the loop condition is checked multiple times during execution. Hence, each iteration might encounter different prediction outcomes. Third, under a mis-prediction, speculatively executed instructions are instrumented and then rolled back. Note that when \( b \) is false but \( \text{pred}'_id = \text{true} \), the instrumented commands are \([s'; (\text{while } b \text{ do } s'); s'_s]_w\), which essentially unrolls the loop to extract the first \( w \) instructions.

\(^4\)Although we formalize SANI and symbolic execution on a C-like source language for simplicity, SpecSafe is built on LLVM IR. The length of IR code is not the same as the length of micro-ops, but the gap can be reduced by a cost model that maps from IR instructions to the number of micro-ops.
The last component of the transformation: check $K_0$; reset; first ensures that there is no cache side channel after each loop iteration, forming a loop invariant, and then resets the final architectural and microarchitectural states to an arbitrary state as an over-approximation when checking the code after the loop.

**Transformation Soundness** The novel program transformation in Figure 6.4 satisfies the following property: for any terminating program, oracle $O$ and transaction length $w$, if the transformed code satisfies CNI (Definition 2), then the original code satisfies SANI (Definition 4). More formally:

**Theorem 1** (Transformation Soundness).

\[
\forall s, s', w, O. s \text{ terminates } \land s \rightarrow s' \implies \\
\forall m_1 \simeq_P m_2. \text{view}(\llbracket s \rrbracket_{\text{perf}}^{m_1, 0}) = \text{view}(\llbracket s \rrbracket_{\text{perf}}^{m_2, 0}) \implies \\
\forall m_1 \simeq_P m_2. \text{view}(\llbracket s \rrbracket_{O}^{m_1, w}) = \text{view}(\llbracket s \rrbracket_{O}^{m_2, w})
\]

**Proof.** By induction on the structure of the original program $s$. In most cases, we can show that the transformed program approximates (in conventional semantics) speculative execution. The tricky case is for the while loop: the inserted check $K_0$ and reset intuitively enforces the following loop invariant: any two executions of the loop starting from the same attacker view results in the same final attacker view. The full proof can be found in Appendix A.2.

**Handling speculation-related instructions.** SpecSafe is built using CaSym as a foundation, and thus uses the symbolic execution rules presented in Chapter 5 for conventional execution. To capture speculative behavior, Figure 6.5 formalizes SpecSafe’s symbolic execution over the new speculation-related primitives. 

\[
\text{fence has no effects in }
\]
\[
\begin{align*}
\left[ SE \right]_{\text{fence}} \sigma & := \sigma \\
\left[ SE \right]_{\text{capture}(id)} \sigma & := \sigma, \text{mems}[id] = \bar{e} \\
\left[ SE \right]_{\text{rollback}(id)} \sigma & := \bar{X} = \text{mems}[id] \land C = c \land \Psi
\end{align*}
\]

Figure 6.5: Speculative-aware symbolic execution of SpecSafe assuming that the input \( \sigma \) is \( \bar{X} = \bar{e} \land C = c \land \Psi \). The symbolic execution also maintains a global list of symbolic memory snapshots \( \text{mems} \), indexed by transaction identifiers.

symbolic execution since it only affects the number of instructions being speculatively executed (i.e., during the construction of \([S]_w\) in Figure 6.4). \text{capture}(id) saves the current architectural state (i.e., the symbolic state for variables) with \( id \) and \text{rollback}(id) recovers the architectural state from the stored states. We use a global list \( \text{mems} \) to store and retrieve such memory snapshots. Note that \text{rollback} only rolls back the architectural state but not the cache state and path condition.

6.7 Implementation

SpecSafe is comprised of two core components: a symbolic executor and an SMT solver. We built the symbolic executor on top of earlier work presented in Chapter 5 to include reasoning about speculative execution. The implementation was done as an analysis pass in LLVM version 3.7. Besides reasoning about speculative execution, SpecSafe also extends CaSym with a more robust reasoning of pointers\(^5\). SpecSafe uses Z3 as the back-end for reasoning about the generated verification conditions, using the theories for arrays and bitvectors to process the generated SMT2 formulas.

By default, SpecSafe instantiates the configurable transaction length parameter \( w \) to be 200 since this has been used in [17] and approximates the maximum speculation

\(^5\)SpecSafe is implemented as an analysis on LLVM IR code. Ideally, an analysis on lower-level code (e.g., binary) offers stronger security guarantee since it is immune to code transformations by later compiler stages [82]. However, we note that even binary analysis faces similar limitation due to hardware features such as out-of-order execution. Moreover, this is largely an implementation limitation, since our methodology can generally be applied to any level of abstraction, from the source code to binary.
depth.

**Improving Scalability**  One limitation of symbolic execution is that it does not scale to large programs (e.g., the entire libgcrypt library). Hence, in order to analyze large programs, we use a static taint analysis to provide a structured way to find intriguing functions where symbolic execution is subsequently applied. The taint analysis is implemented as an LLVM analysis pass, also using LLVM version 3.7 [121]. It is context-, field-, and flow-sensitive. The taint analysis looks for three patterns in programs that encompass both conventional and speculative cache side channels.

For Spectre attacks, we follow patterns described in oo7 [16] in our taint analysis. In particular, the pattern consists of three components: 1) a conditional branch that uses attacker-controllable data, 2) attacker can control index to an array within the speculation window of the attacker-controllable branch, and 3) the data read from the array in component 2 is used as an index for another array also within the speculation window of the attacker controllable branch. Here, attacker controllable data means the data is public and can be either directly or indirectly set by the attacker.

By detecting these three patterns in programs, it allows us to identify interesting code snippets to analyze with the more-precise symbolic execution. We note that although Spectre patterns (in particular, patterns from [16]) used in our taint analysis will find the most serious vulnerabilities, there may still be vulnerable instances where the pattern does not apply in the large programs that we evaluated on. Since we use taint analysis to filter out functions before passing them to symbolic execution, some avenues of leakage may be missed. However, this is not a limitation of the symbolic execution, since more general patterns (e.g., any function that uses confidential data) can be applied to improve code coverage.
Table 6.2: Results of SpecSafe on micro-benchmarks of both vulnerable code as well as their corresponding protected versions. Spectre examples includes all 15 from [2]. The numbers in the table are SpecSafe’s execution times. ✗ means no side channel is found. ✠ means side channels are detected.

<table>
<thead>
<tr>
<th>Spectre Examples</th>
<th>Vulnerable</th>
<th>LFENCE</th>
<th>SLH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 6.1a</td>
<td>41.9s ✠</td>
<td>0.06s ✠</td>
<td>0.02s ✠</td>
</tr>
<tr>
<td>Figure 6.1b</td>
<td>0.20s ✠</td>
<td>0.02s ✠</td>
<td>0.59s ✠</td>
</tr>
<tr>
<td>Figure 6.1c</td>
<td>0.63s ✠</td>
<td>0.61s ✠</td>
<td>0.87s ✠</td>
</tr>
</tbody>
</table>

6.8 Evaluation

We selected a variety of benchmarks to evaluate SpecSafe. These benchmarks include (1) 15 well-known samples vulnerable to Spectre Variant 1 attacks [2]; (2) the 3 samples of CB-channels from Figure 6.1; (3) a set of common ciphers from libgcrypt version 1.8.5. Our evaluation of SpecSafe on these benchmarks was run on Ubuntu 14.04 in a virtual machine with 30GB of RAM and 4 cores of an Intel i7-5820K CPU. We use the trace-based attack model and age cache model presented in CaSym [22] in our evaluation to instantiate our cache model due to their conservativeness. In our experiments, we answer the following questions:

1. Can SpecSafe detect cache side channels that are missed by other state-of-the-art tools?

2. How effective can SpecSafe identify vulnerabilities?

3. Does SpecSafe verify the absence of cache side channels manifesting from the speculation of branch conditions?

4. Can SpecSafe scale to programs of moderate sizes, by using taint analysis combined with symbolic execution?
Table 6.3: Comparison with existing tools for detecting speculative side channels. A ✓ indicates the tool correctly judges the security/insecurity of the code (i.e. marking it secure/insecure if it is secure/insecure). A ✗ indicates the tool either has a false negative (when the code is vulnerable) or a false positive (when the code is secure). n/a applies when the tool fails to detect a vulnerability in the original code and thus cannot reason about mitigations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SpecSafe</th>
<th>Spectector [17]</th>
<th>oo7 [16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Spectre Examples</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Figure 6.1a, 6.1b, 6.1c</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>LFENCE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SLH</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

6.8.1 Micro-benchmarks

The second column of Table 6.2 presents the results of applying SpecSafe on two sets of micro-benchmarks, including 15 well-known samples that are vulnerable to Spectre Variant 1 attacks [2] and the three examples in Figure 6.1. Each of these samples is about 10 to 20 lines of C code. SpecSafe successfully detects all vulnerabilities in those micro-benchmarks, within a reasonable amount of time.

Once a vulnerability is detected, we use the locations of the failed checks reported by SpecSafe to place appropriate protection mechanisms such as `lfence` or `SLH` to fix the reported vulnerabilities. After applying a protection mechanism, we reran SpecSafe, which was able to verify that the samples with `lfence` and `SLH` defenses are free of cache side channels.

Interestingly, two of the new side channels proposed cannot be patched using SLH. This is because the leakage in Figure 6.1b and Figure 6.1c happens due to the presence
<table>
<thead>
<tr>
<th>Module</th>
<th>IR LoC</th>
<th>Functions</th>
<th>Time (sec.)</th>
<th>Sensitive Access</th>
<th>Sensitive Branch</th>
<th>Spectre V1</th>
<th>Vulnerable Candidates</th>
<th>SANI</th>
<th>Time (sec.)</th>
<th>CNI</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>6375</td>
<td>49</td>
<td>293</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>58.4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>DES</td>
<td>8619</td>
<td>44</td>
<td>618</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>422</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>arcfour</td>
<td>536</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0.75</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>blowfish</td>
<td>3535</td>
<td>31</td>
<td>215</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>37.5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>camellia</td>
<td>14436</td>
<td>14</td>
<td>4344</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cast5</td>
<td>4458</td>
<td>27</td>
<td>455</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3.03</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>chacha20</td>
<td>3500</td>
<td>20</td>
<td>88</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>37.1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>idea</td>
<td>1650</td>
<td>16</td>
<td>61</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3.09</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>md4</td>
<td>1272</td>
<td>12</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>19.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>md5</td>
<td>2252</td>
<td>12</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>19.0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>salsa</td>
<td>1756</td>
<td>20</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2.12</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>sha256</td>
<td>6086</td>
<td>22</td>
<td>424</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>26.7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sha512</td>
<td>2743</td>
<td>26</td>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>31.8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tiger</td>
<td>2309</td>
<td>17</td>
<td>59</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>39.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>twofish</td>
<td>8238</td>
<td>28</td>
<td>334</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>9.79</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>67765</td>
<td>346</td>
<td>7095</td>
<td>16</td>
<td>13</td>
<td>8</td>
<td>29</td>
<td>24</td>
<td>709.88</td>
<td>23</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.4: Evaluation results on libgcrypt ciphers. Here, the first three columns indicate the number of lines of LLVM IR are in each module, the number of functions found in the module, the time to analyze the module. The First column under taint analysis indicates how long the taint analysis ran. The next three columns indicate the number of functions found containing those code patterns described in Section 6.7. The last column under taint analysis is the union of the previous three columns indicating the number of potentially vulnerable functions. The CNI and Spec columns indicate the number of functions the symbolic execution found containing those kinds of vulnerabilities. The SANI column contains all functions found that violate our SANI definition, along with the total analysis time for symbolic execution.
of a misprediction. Since mitigation techniques such as SLH do not prevent mispredictions from occurring, it cannot protect the new side channels in Figure 6.1b and Figure 6.1c.

We note that although we have not observed the code patterns presented in Figure 6.1 in the wild, we have 1) demonstrated that they are not detectable by existing tools under trace-based attacks, and 2) show that existing defenses that rely on creating data dependencies and masking (e.g. SLH) might fail to mitigate them.

We also compared SpecSafe with other state-of-the-art systems geared towards finding leakage caused by speculative execution (Table 6.3). We note that both Spectector and oo7 cannot detect the three new instances of speculative leakage discovered in this work. Furthermore, since they cannot detect the vulnerabilities in the vulnerable code in the first place, their reports on the code with defense are not meaningful, hence, reported as n/a in the table.

### 6.8.2 Security Benchmarks

Next, we evaluate SpecSafe on libgcrypt ciphers. For each cipher, we treated the encryption key as sensitive data. Since the sizes of these ciphers were relatively large, symbolic execution was not scalable to handle them directly; therefore, we had to apply taint analysis (discussed earlier) to filter functions in those programs. The pattern of Spectre attacks [16] requires us to specify what data is considered attacker controllable. We include plaintexts, ciphertexts, and the size of the data to process as attacker-controllable, since these inputs are commonly provided to encryption routines by users, including a malicious user.

**Taint Results** The first part of Table 6.4 presents the results gathered by applying our taint analysis to the various ciphers in libgcrypt. For each cipher, the first column tells
the cipher being analyzed. The next column, IR LOC, is the number of lines of IR code in the cipher, as produced by Clang. The functions column tells the number of functions in the cipher, and the time column tells how long it took the taint analysis to analyze the cipher in seconds. Each of the following columns tells the number of functions that were identified by the taint analysis in the specified category.

The *Sensitive Access* category contains functions marked because they have a memory address indexed by sensitive data. *Sensitive Branch* tells that there is a conditional branch that depends on sensitive data. The *Spectre V1* column tells the number of functions that match the pattern specified by oo7 [16]. The last column, *Vulnerable Candidates*, is computed by taking the union of the functions from the previous three columns; it tells all of the interesting functions to be further investigated by symbolic execution.

Overall, we find that the taint analysis can take at most an hour to finish when running on moderately sized modules, but is often much faster. This is an improvement over applying symbolic execution to all of these functions, which would likely take many hours if it terminates at all. Even though the taint analysis is rather coarse-grained, we find that it is often able to rule out about 90% of the functions in each cipher.

One source of imprecision comes from the points-to analysis used by the taint analysis: it tends to collapse all of the fields, making it field-insensitive. We found that almost all of the sensitive branches identified by the taint analysis were due to the collapsing of the encryption context and subsequent checks on benign fields of the context (e.g., `if (ctx->prefetch_dec_fn)`). This limitation is typically propagated onto symbolic execution; thus, some of the extraneous functions identified were manually pruned from the list of sensitive functions. In total, about 10 functions were manually pruned this way; almost all of these were found in the AES cipher due to checking the encryption context for certain features. The task of pruning is rather simple because the taint analysis reports the line number associated with the detected vulnerability and the issue is
obvious.

The identified 29 interesting functions are then further analyzed by symbolic execution.

Categorizing Side Channels SpecSafe by design detects any SANI cache side channels. We present all functions violating our SANI definition under the column "SANI" in Table 6.4, along with the total analysis time for symbolic execution under the column Time. To classify the results into conventional ones and speculative ones, we note that SpecSafe can be easily modified to detect conventional side channels only. In particular, for cache non-interference (CNI), we modify SpecSafe to add a constraint for each branch command as follows: $\text{ite}(\text{cond}_i, \text{pred}_i = \text{true}, \text{pred}_i = \text{false})$ where $\text{cond}_i$ is the branch condition being speculated. This effectively creates a perfect oracle, which allows us to analyze a program that never mispredicts. With the modification, we identified the conventional side channels (column “CNI”) as well as the remaining ones (column “Spec”) that only exhibit themselves with speculative execution.

Conventional Cache Side Channels Interestingly, we find that for conventional cache-based side channels, SpecSafe and taint analysis find the same vulnerable program points. The taint analysis finds 23 functions in total that may be vulnerable to cache side channels, the union of columns Sensitive Access and Sensitive Branch, and the symbolic execution finds 23 functions violating the CNI definition. This is mainly for two reasons. First, as discussed earlier, the symbolic execution relies on the taint analysis to identify interesting functions to be further analyzed in the first place. Second, the additional precision of symbolic execution does not help in the identified functions. This, however, is not the case when evaluating defenses, such as preloading and pinning [22].
A New Conventional Cache Side Channel  In our evaluation, we find that SpecSafe
detects a previously unreported, to the best of our knowledge, potential vulnerability in
the triple DES routine of libgcrypt. This routine finds weak encryption keys, with the
following code snippet:

```c
is_weak_key ( const byte *key ) {
...
while(left <= right) {
    middle = (left + right) / 2;
    // this branch causes the leak
    if (memcmp(key, weak_keys[middle]))
        return -1;
    if ( cmp_result > 0 )
        left = middle + 1;
    else
        right = middle - 1;
}
}
```

It is clear from the identified if statement that this routine will terminate early depending
on the key’s value, thus resulting in a different cache state. Furthermore, the index to
the table weak_keys is dependent on the value of the key since it uses the result of the
memcmp function which depends on the key.

Checking for weak keys is common in encryption algorithms such as DES, 3DES,
RC4, Blowfish, IDEA, etc. The NIST suggests that weak keys be avoided, at least for
3DES [122], since they can significantly weaken security, but we find that checking for
the keys in naive manners such as the code snippet above also pose security risks by
potentially leaking bits of secure keys.
Speculative Execution Results  Side channels that only show up with speculative execution are summarized in the last column of Table 6.4. We detect two true positives that can potentially be used to perform a Spectre style side-channel attack. These side channels, found in salsa and chacha20, are similar. The following is a simplified code snippet highlighting the potential leakage:

```c
if (n > length) // begin speculation
...
// input is attacker controllable
idx = input;
...
buf2 = buf + BLOCK_SIZE + idx;
...
addr = *buf2; // loads secret into addr
...
value = *addr; // use secret addr to load
```

In this example, we see that within the depth of speculation of the conditional branch, we have a potential out of bounds read via `addr=*buf2;` since what `buf2` points to may be controlled by the adversary, an attack can load sensitive data into `addr`. Furthermore, the sensitive data in `addr` is used as an address for a subsequent load on line `value=*addr`. As a result, a cache attack can be used to discover the sensitive data in `addr`.

We investigated if this code pattern could be used to construct a Spectre attack. To this end, we constructed a test for this snippet similar to what we described in Section 6.3.2. Our test shows that this code snippet is capable of leaking information via speculative execution at a rate of over 75k bits per second with over 99% accuracy.
Unlike the case on conventional side channels, symbolic execution was able to remove 6 speculative cache side channels reported by taint analysis. The result highlights the benefit of using precise symbolic execution, compared with pattern matching methods, such as oo7 [16].

**Take Away**  To summarize our findings: 1) built on the new security definition SANI, SpecSafe is able to detect speculative cache side channels that are missed by Spectector [17] and oo7 [16], the state-of-the-art tools for detecting speculative side channels; 2) SpecSafe has shown great promise in identifying both conventional and speculative cache side channels; 3) with the help of taint analysis, SpecSafe can work on programs of roughly 3,000 source lines or about 30,000 lines of IR code in about an hour which is reasonable considering how long it would take using only symbolic execution; 4) a coarse-grained taint analysis does a great job zeroing in on problematic functions to be further refined by symbolic execution, although the symbolic execution is still, in general, more precise, especially for intricate cases.

### 6.9 Summary

In this chapter, we presented SpecSafe. SpecSafe utilizes symbolic execution to perform program analysis to detect cache-based side channels under both speculative and non-speculative conditions. To summarize, our contributions in this chapter include:

- By analyzing and identifying the limitations of existing security definitions against cache side channels, we proposed and showed the practicality of a new kind of speculative cache attack that belongs to the Spectre Variant 1 attack family, but is more stealthy than existing ones since cannot be detected by existing cache side channel security definitions.
Motivated by the new vulnerabilities that elude existing security definitions, we proposed SANI (Speculative Aware Noninterference), a security definition that captures conventional cache attacks, Spectre attacks, and beyond. It can also be soundly applied to different attack models (e.g., access-based, trace-based) and cache models (e.g., LRU, age-based).

The core of SpecSafe is a symbolic execution engine that reasons about program variables, cache state, and speculative execution at the same time. To improve scalability, SpecSafe uses taint analysis to filter functions; only interesting ones are sent for symbolic execution. After symbolic execution, SpecSafe emits verification conditions for the analyzed program to satisfy SANI and feed them to an SMT solver; satisfiable verification conditions imply a violation of SANI in the source code.

We analyzed programs known to be vulnerable to Spectre-like attacks to demonstrate that SpecSafe is capable of detecting all of the vulnerabilities. We also showed that SpecSafe is the only tool that can detect the new side channels discovered in this work. Lastly, we applied SpecSafe to a variety of crypto benchmarks from libgcrypt to analyze code used in practice. We found two previously unknown possible side channels, which to the best of our knowledge, were not found before in libgcrypt ciphers.
Chapter 7

A Systematic Comparison of User-Space Spectre Mitigations

7.1 Motivation

Modern processors are designed to achieve high performance as the priority. As a result, manufacturers have introduced many optimizations that allow computation to proceed even when data is unavailable. For example, a branch prediction unit allows the predicted branch to execute even before the branch condition is fully computed. In case of a misprediction, the transient execution of the mispredicted instructions is rolled back. However, their footprints in microarchitecture features such as CPU cache are not erased. Hence, attackers can use those footprints to uncover portions of the execution context that can contain sensitive information [11, 12, 23, 24, 36–39, 43, 123].

Beginning with the initial discovery of Spectre [11] and Meltdown [12], new attacks leveraging transient execution to construct covert and side channels have frequently sprung up over the past few years [23, 24, 36–39, 43, 123]. These attacks manifest due to optimizations present on nearly every modern CPU, including from vendors such as
Intel, AMD, and ARM. However, since transient execution is a feature, rather than a bug, of modern architectures, the root cause of transient execution attacks is difficult for manufacturers to address; they are generally still an issue even two years after their initial discovery. While manufacturers grapple with how to retain performance while maintaining security, software developers are left to mitigate the threat with existing infrastructure.

In the wake of these new discoveries, developers have been left scrambling to find solutions that do not severely jeopardize performance. Further exacerbating this is the lack of support available on commodity hardware to mitigate some of these vulnerabilities in an efficient manner. Since speculative execution is a feature allowing CPUs to obtain significantly more performance, it is unlikely to see manufacturers completely address the issue in the near future. Given that reality, software defenses that harden programs against transient execution attacks have become appealing to developers. Unfortunately, the landscape of software defenses is not obvious to navigate and different defenses come with a variety of subtle trade-offs.

In order to address these pressing concerns, it is important to have a study that surveys the landscape of user-level defenses, and more importantly, compares defenses to highlight each defense’s strengths and weaknesses. One important aspect for comparison is to understand a defense’s performance impact. Past research papers have provided performance evaluation on some of the defenses, but the problem is that each is evaluated using a particular set up (benchmarks, running environments, etc.), which makes it hard to compare performance numbers in a scientific way. Therefore, it is imperative for a study to compare the performance impact of the defenses on a shared platform. Lastly, the ultimate goal of such a study should be to provide software developers with accurate guidance about how to select the best mitigation for their circumstances.
7.2 Attack Models

In this section, we present a general attack model that can be instantiated to capture a variety of side-channel attack scenarios seen in practice that target the CPU cache. The attack models are composed of three parts: what the adversary can observe, when they can make their observation, and where they can observe. Next, we describe in more detail each component.

7.2.1 Memory Trace Granularity

The “what” component of the attack model describes what kinds of events (emitted by program execution) are visible to the attacker. If we focus on traces of memory-related events, there exist multiple levels of granularity that an attacker can observe, depending on the attack’s capability. The finest grained model would give an adversary the ability to observe every event on the memory bus, including its memory address, the value it retrieves from or writes to memory, and its timing. A more practical attack (e.g., CacheBleed [100]) works at the cache bank level of granularity. Most cache side channel attacks work at the cache line granularity, meaning that the attacker can only observe which cache line is accessed. This is commonly seen in Flush+Reload [27] attacks and Prime+Probe [25] attacks. At an even more coarse-grained level, the attacker may only learn what page is being accessed.

Note that, memory accesses of transient execution should also be contained in the trace since they are not erased and are hence observable to the adversary by probing shared caches.
7.2.2 Attack Interaction Granularity

The “when” component of the attack model describes when an attacker can make observations during the victim program’s execution. In this spectrum, the finest-grain attacker can observe and set the microarchitectural state before and after every victim’s instruction. Next in the spectrum is an attacker that can observe and set the microarchitectural state before and after a block of victim instructions (e.g., a function) is executed. At the most coarse-grain level, an attacker is only able to observe and set the microarchitectural state before and after the entire victim application is executed.

We note that while it might appear unlikely for an attacker to gain fine-grain observations, past work [26] has shown that an unprivileged adversary can manipulate the Linux scheduler to improve the adversaries ability to interact with their victims. Furthermore, the adversaries can obtain even greater control over the scheduler if they are privileged users as a result of their control over interrupts and scheduling [124, 125].

7.2.3 Attack Location

The last component of our threat model specifies where the attacker can make observations. Since we focus on cache side channels, an attacker may either observe the data cache, or the instruction cache. We note that although in the context of cache side channels, it might also be helpful to distinguish which cache level (e.g., L1, LLC) is observable, such information is not critical for the purpose of analyzing user-level defenses for transient execution attacks.
<table>
<thead>
<tr>
<th>Spectre Variant</th>
<th>Barrier</th>
<th>PDC</th>
<th>Data Dependency</th>
<th>SCFC</th>
<th>Partitioning Mask</th>
<th>Sandbox</th>
<th>Obfuscation</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 (Spectre-PHT)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>⊘</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>V2 (Spectre-BTB)</td>
<td>⊘</td>
<td>●</td>
<td>⊘</td>
<td>●</td>
<td>⊘</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>V4 (Spectre-STL)</td>
<td>●</td>
<td>⊘</td>
<td>⊘</td>
<td>●</td>
<td>⊘</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multi-architecture</th>
<th>Future resilience</th>
<th>Non-Probabilistic</th>
<th>All memory granularity (§7.2.1)</th>
<th>All interaction granularity (§7.2.2)</th>
<th>Data cache safe (§7.2.3)</th>
<th>Instruction cache safe (§7.2.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 7.1: Summary of when certain classes of defenses are effective against different Spectre variants as well as the characteristics of mitigation classes. We use ● to indicate that all defenses in the class apply, ⊘ to indicate a mitigation first proposed in this work, G # to indicate some of the defenses apply, and ⊘ to indicate none of the defenses apply. Note that Spectre Variant 3 (Meltdown) and Spectre RSB are not listed since they are beyond the scope of user-level defenses.
7.3 Defenses

In this section, we present state-of-the-art user-space defenses that developers can use to protect their software from transient execution attacks. As discussed earlier, both Meltdown and Spectre RSB are out of scope and hence, we focus on other Spectre variants.

**Robustness Attributes** For each defense that we introduce next, we also evaluate its robustness through a set of quality attributes. Table 7.1 summarizes the main findings that we will discuss in detail. The robustness attributes include: *multi-architecture* specifies whether a defense is applicable to multiple computer architectures; *future resilience* specifies whether a defense is applicable even in some future implementation of an architecture; *non-probabilistic* means that the defense eliminates attacks, whereas a probabilistic defense makes attacks more difficult but still possible.

**Attack Model Attributes** We also use a set of attributes related to attack models: *all memory granularity* specifies whether a defense is effective against attack models of varying granularity of memory traces (cache bank level, cache line level, etc.); *all interaction granularity* specifies whether a defense is effective against attack models of varying granularity of attacker interaction (instruction level, function level, etc.); *data cache safe/instruction cache safe* specifies whether a defense is effective against an attacker who can observe data/instruction cache accesses.

7.3.1 Halt Speculation with Execution Barriers

The root cause of transient execution attacks is speculative execution. One (extreme) potential defense is to turn off speculative execution entirely. However, doing so is simply impossible since speculative execution is a critical feature of modern architectures...
idx = input;
if (idx < arry_size)
{
    BARRIER;
    t = arry[idx];
    ...
    res = arry2[t];
}
BARRIER;
...

(a) Immediate Barrier

Figure 7.1: Examples of Protection by Execution Barrier.

of high performance. Even if an on-off switch is offered by hardware, the performance overhead would be prohibitive without speculative execution.

An alternative is to halt speculative execution before transient instructions that potentially leak information. Since the discovery of Spectre attacks [11], halting speculative execution has been an important user-level defense mechanism. In this chapter, we use the term “execution barrier” for an instruction that halts the CPU pipeline, and hence, prevents transient instructions following it from being executed.

What instructions are execution barriers The specific execution barrier to be used depends on the target architecture. For example, the lfence instruction is the most commonly used in practice for x86, although other serializing instructions can be used too, such as cpuid on AMD and Intel processors [126, 127]. lfence is typically used since it takes less time to execute than other instructions; it requires only the reorder buffer (ROB) to be exhausted, while other instructions, like cpuid, require full serialization and wait for both the reorder buffer and store buffers to be completely

---

1Memory fence mfence might also achieve similar effects, however, it does not serialize the instruction stream as desired: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf, sec. 4 pg 22.
For ARM processors, a new speculative condition barrier \texttt{csdb} is recommended [128, 129]. This barrier is exclusive to ARM processors and functions differently from other barriers. The use of \texttt{csdb} is restricted to conditional select and conditional mov instructions to halt instructions that depend on a conditional select/mov. Since this approach is not always applicable, ARM also suggests using the following instruction sequence: \texttt{dsb sys} followed immediately by \texttt{isb} [128, 129]. This instruction sequence behaves similarly to \texttt{lfence} in the sense that no instructions can be executed until these instructions retire. Since it has stronger serializing properties than \texttt{lfence}, ARM suggests using the \texttt{csdb} when possible.

**Placement of execution barriers** With correct placement of execution barriers, vulnerable transient instructions will not be executed since they will be rolled back once all pending data are resolved when the pipeline resumes. However, since execution barriers only halt instructions, but not disable speculative execution, they must be placed correctly to ensure security\textsuperscript{2}.

For Spectre Variant 1 attacks, vulnerable transient instructions are the ones following branches. Hence, in the most naive implementation, a \textit{barrier} can be placed at the beginning of the successor basic blocks following each branch, as shown in Figure 7.1a.

In this example, the barriers immediately terminate speculative execution at lines 4 and 9 regardless of which branch is taken, thus thwarting Spectre variant 1 attacks.

The naive implementation halts all instructions after branch, and hence, undermines the benefits of branch prediction. For more efficient defense, the barrier can be deferred until the first instruction that might leak data, as shown Figure 7.1b. Deferring the

\textsuperscript{2}Kocher compiles interesting code examples demonstrating what can go wrong with incorrectly inserted barriers at https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.
barrier allows the beginning of the basic block to execute speculatively from lines 3 to 6, while still halting the vulnerable instruction where sensitive data (t in the example) is used to access memory. We note that a compiler might further reorder non-dependent instructions before the barrier as an optimization for performance.

For Spectre Variant 2 attacks, however, execution barriers are not applicable since the adversary can lead speculative execution to begin at an arbitrary point in the victim’s address space. Even when all possible targets are available and being defended by barriers, an adversary can cause speculative execution to begin immediately after the barrier.

For Spectre variant 4 attacks, barriers can be placed between the vulnerable store and load instructions to prevent potential exploits. Intel notes that for performance, this should be used sparingly, and only when an attack that leaks sensitive data is deemed realistic [42]. Recently, Intel released a microcode update that disables speculative loads until store addresses are known, thus thwarting the attack [42]. When enabled, this feature mitigates Spectre V4 without requiring code modification.

Quality attributes Barriers are one of the more comprehensive defenses shown in Table 7.1. Since barriers terminate speculative execution, they are capable of being used to mitigate both Spectre V1 and V4. As discussed earlier, it is not considered a practical approach for V2.

As shown in Table 7.1, barriers are one of the best mechanisms with respect to different attack models as well as being a general and future resilient mechanism. Hardware manufacturers suggest using barriers, specifically dispatch serializing barriers, since by their nature they must prevent speculative execution. This feature makes them both applicable on virtually every architecture with serializing instructions and likely to retain their behavior for the foreseeable future. Furthermore, barriers are applicable against
almost all configurations in our proposed spectrum of attack models. This is because when properly placed, execution barriers eliminate transient execution, and hence, any leakage from it.

However, one potential issue is that barriers do not delay instruction fetch and decoding stages in the CPU pipeline. Hence, in theory, barriers may not prevent leakage via the instruction cache, since the adversary might learn that a transient instruction is being fetched [130]. However, we note that the potential threat has not been demonstrated in practice yet, and its bandwidth is likely to be small, if possible.

7.3.2 Halt Speculation with Data Dependence

An alternative approach of halting speculative execution is to create data dependency: modern processors execute instructions that are data-dependent sequentially. In particular, for Spectre Variant 1 attack, it is sufficient to create data dependency between the branch condition and potentially vulnerable memory accesses. This approach is widely accepted across hardware vendors (AMD [127], ARM [128, 129], Intel [126]),
and has been adopted by popular compilers such as GCC [131] and Clang [132]. The most widely used implementation of this approach is called Speculative Load Hardening (SLH), but others have also been introduced [133].

**How to create data dependency?** For Spectre Variant 1 attacks, we note that a processor serially executes data-dependent instructions [133]. We use the code snippet\(^3\) in Figure 7.2a to illustrate how to create a data dependency using the flags [133]. While we use x86 assembly in the example, the techniques in this subsection also apply to other architectures such as AMD and ARM.

In Figure 7.2a, a data dependency between flags and the potentially out-of-bounds index \(t\) is created by using the LAHF instruction, which loads the values of all flag registers into the \(ah\) register (i.e., bits 8-15 of the \(rax\) register). Since LAHF uses the \(rax\) register, the code first saves \(rax\) to the stack at line 2. The \(rax\) register is then used to create a data dependency on register \(r15\) and is restored afterward. Lastly, \(t\) is xor-ed with \(r15\) twice to propagate the dependency to the potentially out-of-bounds access at line 11 without changing the value of \(t\).

The previous transformation introduces significant overhead due to pushing/popping data onto/off the stack. A more efficient approach is known as Speculative Load Hardening (SLH) [132,134]. SLH is a two faceted approach that 1) creates a data dependency on the branch condition and 2) masks potentially secret data on a misspeculation. To introduce SLH, we consider the example in Figure 7.2b. SLH creates a data dependency using a conditional move (CMOV) instruction at line 4, which uses the condition flag register. As a result, CMOV which is dependent on the condition flag register is halted until the instruction that sets the flag register commits the result. Moreover, the potentially out-of-bounds access at line 11 is made dependent on the CMOV to prevent exploitation.

\(^3\)The pseudo-code mixes assembly and source code syntax for readability.
Note that the construction above assumes that CMOV is an *unpredicted* branchless conditional update; otherwise, it will not halt the out-of-bounds access at line 11. This assumption holds on current x86 CPUs, but is not guaranteed in the future. To mitigate this, a mask is used, so that it stores all 1’s on a correct speculation and all 0’s on a misspeculation. The array index is masked at line 10 to rule out any out-of-bounds memory access during a speculative execution.

Lastly, another approach is to create dependency on the arguments of the vulnerable conditional branch as opposed to the conditional flag register [133]. This approach is shown in Figure 7.2c, where line 2 creates a dependency between condition argument idx and register r15. This dependency is then propagated onto the potentially dangerous memory access at line 11 using two XOR operations. It is noted in [133] that the ordering of the loads with respect to the comparison are not guaranteed because in this approach the loads are not data dependent on the result of the comparison as they are in SLH, thus may still be vulnerable due to the processor reordering the loads to be after the comparison. They show that some workloads can perform significantly better using this approach over SLH and on average using the Phoenix benchmark is slightly better in terms of performance overhead than SLH.

**Quality attributes** Although execution barrier and data dependency both halt potentially vulnerable transient instructions, the key difference is that execution barrier halts *all* following instructions, while data dependency still allows independent instructions (e.g., line 7) to execute in an out-of-order fashion. As we show via empirical study (Section 7.4), defenses based data dependency are typically more efficient than those based on execution barriers.

The data dependency defense mitigates only Spectre V1 attacks. This is because they attempt to propagate a data dependency from the condition used for a jump to some
memory access to delay its execution. It is not applicable to Spectre V2, which does not require a conditional jump, nor Spectre V4 since the defense does not prevent a store instruction from being speculatively executed. This information is summarized in Table 7.1.

Creating data dependencies is one of the most robust defenses across multiple architectures as described in white papers from prominent hardware manufacturers [126–129]. Unfortunately, the reliance on data dependencies to prevent speculative execution on future architectures may be in peril since Intel states it may produce chips that do speculate in the presence of data dependencies [126] and could introduce value prediction, making these defenses undesirable for programs that require future resilience.

Table 7.1 states that with respect to different attack model configurations, the data dependency defense performs well. The attacker can start and stop the mitigated program at the instruction granularity and even see the memory access pattern of the victim at the address granularity, but cannot uncover the sensitive information. However, the data dependency defense falls short if the attacker can inspect the instruction cache [35].

7.3.3 Prevent Speculation with Preemptive Data Caching (PDC)

We observe that transient execution is triggered by unavailable data. For example, when data needed by a conditional jump or an indirect branch is not in the cache, the CPU has to wait for it to be fetched from memory to decide the outcome of the jump/branch; the data fetching lasts for a significant number of cycles, sometimes over hundreds of cycles, causing the CPU to execute instructions speculatively. Hence, instead of halting instructions, a defense can also preload necessary data into the cache before it is needed to avoid speculative execution, which we call Preemptive Data Caching (PDC). PDC has been suggested by AMD [127] to mitigate Spectre V2 attacks. In this chapter, we
idx = input;
...
read(idx); // cache idx
...
MOV *idx_addr, rax
CMP rax, arry_size
JMPGE if_else
if_then:
    t = arry[idx];
    ...
    res = arry2[t];
if_else:
    ...

(a) PDC with Load.
(b) PDC with Barrier.

Figure 7.3: Examples of PDC defenses.

further propose that PDC can be used in a broader context to mitigate Spectre V1 as well.

Ideally, the necessary data should be prefetched sufficiently far in advance such that when it is needed, the data is available, but also close enough so that the data is not evicted from the cache prior to the use. Consider the example in Figure 7.3a, where read(idx) represents an instruction that reads from idx. The dilemma is that read(idx) needs to be placed at the right place so that idx is cached at line 6. However, we note that it is very challenging to assure this for multiple reasons, including: where the data is located (i.e. in L1, L2, L3, RAM, etc.), the hardware configuration, as well as system load. Hence, PDC with Load cannot guarantee the elimination of speculative executions; it only provides probabilistic assurance by making speculative execution less frequent.

Another approach is to add an execution barrier right before the branching instruction, to ensure that the data is ready when the latter is executed, as shown in Figure 7.3b. With an execution barrier right before the jump, it ensures that all previous memory
accesses have retired, guaranteeing all data needed to compute the branch condition are known prior to executing the jump. Although this approach also uses a barrier, as in the execution barrier defense we discussed earlier, it does so not by halting speculative execution, but instead by preventing speculative execution from occurring. It provides a sufficient defense against an adversary that can interact with the victim on a per instruction basis. PDC with Barrier is more costly than PDC with Load due to the barrier, however, it eliminates mispredicted branches, while attacks are still possible under PDC with Load.

PDC is also applicable to Spectre Variant 2 attacks. AMD suggests using barriers before indirect jumps/calls to ensure data is in registers/caches to avoid speculation [127]. Some branch instructions may load from memory directly in indirect jumps. In such cases, a compiler can transform them into separate loads and insert a barrier right before the transformed indirect jump to an address stored in a register.

**Quality attributes** As discussed earlier and shown in Table 7.1, PDC with barrier prevents Spectre V1 and V2 attacks. However, this approach is not applicable to Spectre V4 since it does not prevent the speculation, nor does it prevent the loaded address from being cached.

Table 7.1 shows how PDC stacks up with respect to robustness and attack models. PDC relies on data caching and memory barriers. These hardware features are universal and can be found in many different processors, including Intel, AMD, ARM, etc. Furthermore, we believe these features should be future proof as they are used for many other purposes, such as performance optimization and data synchronization.

Looking at different attack models, the success of PDC depends on how small of an interaction window the attacker has. PDC that uses only preloading can be successful when an attacker cannot modify the cache state between when the data is preloaded and
when it is used. A stronger attacker that can flush the cache between the preload and the use breaks the defense and can cause speculative execution. To alleviate this concern, as discussed earlier, memory barriers can be used together with PDC. Further, PDC can protect against an attacker who observes at the memory address granularity, making it safe for any level of memory granularity. Lastly, since the goal of PDC is to prevent speculation on jump/call instructions, they also do not leak data through the instruction cache as long as speculative execution is prevented by ensuring the target is known.

7.3.4 Speculative Control-Flow Capturing

Another class of mitigation that can deal with Spectre V2 attacks is what we call ”Speculative Control-Flow Capturing”. The most notable in this group is the well known retpoline mitigation [135]. Recently, there have been spin-offs of retpoline called randpolin [136], and JumpSwitches [137], which apply the core ideas of retpoline with the goal to improve performance.

We first take a close look at how retpoline transforms a simple \texttt{jmp *eax} instruction into the code in Listing 7.1. The key idea for retpoline is to replace indirect jumps with a sequence of instructions that take advantage of the predictable behavior of the implementation of the Return Stack Buffer (RSB). This is performed by (1) replacing the indirect jump with a direct call to a new target, at line 1; (2) at the new target, replacing the return address with the original target of the indirect jump before returning; (3) following the direct call at line 1, placing a dummy loop.

In the example, the execution of \texttt{call new\_target} pushes \texttt{spec\_spin} into RSB. In normal execution, control continues to \texttt{*eax}. When there is speculation (due to the return instruction at line 9), the control flow transfers to \texttt{spec\_spin} because that is the first entry in RSB; at \texttt{spec\_spin}, a dummy loop executes to wait for speculative
call new_target;
spec_spin:
   // speculative execution begins
   PAUSE;
   JMP spec_spin;
new_target:
   // replace return address with eax
   MOV eax, (rsp);
   RET;

Listing 7.1: Retpoline Example

LEA randpoline_base + random * 4, rcx;
MOV *rax, rbx;
JMP rcx;

Listing 7.2: Randpoline Example

to be over.

To ensure that RSB does not underflow, a technique called RSB-stuffing is used. This technique fills the RSB with benign code locations to ensure speculation causes the control to transfer to benign locations. Coincidentally, this technique not only ensures that retpoline cannot encounter an underflow condition, but also mitigates Spectre-RSB [24,43] since it prevents the attackers from controlling locations inside RSB.

In an attempt to improve performance and also improve compatibility with existing and future architectures, Intel developed randpoline [136]. The goal of randpoline is not to confront the root issue, speculative execution, but allow it to happen in a random manner. Intel describes this approach as a kind of “ASLR” for branch prediction [138]. The transformation used by randpoline to transform JMP *rax is shown in Listing 7.2. In this example, the memory pointed to by randpoline_base contains many copies of JMP rbx. Since line 2 moves the originally intended target into rbx, a JMP rbx in the table causes the program to jump to *rax. This defense works by making it more difficult for the adversary to train the BTB predictor: to redirect the jump target to a desired address, the adversary now needs to train the BTB predictor for all JMP
rbx in the memory region starting at randpoline_base, since an actual jump uses a random copy of JMP rbx in that memory region.

Another proposed improvement over retpoline is called JumpSwitches [137], which transforms an indirect branch into a series of conditional branches that perform direct branches to the most likely targets based on some dynamic profiling. For example, JMP *eax is transformed to if (*eax = 0x1234) then JMP 0x1234 else JMP *eax, assuming 0x1234 is the most likely target. This approach can transfer only some of the likely targets into direct branches and still has to include an indirect branch as the fallback case. This fallback case needs to be protected using other techniques such as retpoline and randpoline. Compared to retpoline and randpoline, Jump Switches provide an efficiency boost since direct branches are faster than indirect branches.

**Quality attributes** Examining Table 7.1, we see that Speculative Control-Flow Capturing defenses, listed under the SCFC column, mitigate Spectre V2. These approaches are not applicable to Spectre V1, which uses conditional jumps. In fact, approaches like JumpSwitches may introduce even more Spectre V1 vulnerabilities due to the additional conditional jumps introduced. Lastly, Spectre V4 requires the speculation between stores and loads to be mitigated, which these techniques cannot do.

Retpoline is known not to work on all of the processors; for instance, it does not provide full protection for Intel Skylake processors but works on earlier models [139]; thus we do not consider retpoline nor JumpSwitches, which uses retpoline as a fallback, to be generally applicable to different architectures. Randpoline on the other hand is a promising approach that may not provide complete protection, but should be applicable to many architectures. Due to the predictable nature of RSB on modern processors and no suggestion that major vendors plan on changing this implementation soon, these defenses are likely to be a successful mitigation for the foreseeable future. Randpoline
only makes attack more difficult, making it a probabilistic approach. With respect to the spectrum of attack models presented in Section 7.2, we find that each technique either prevents the speculative execution from happening (retpoline/JumpSwitches), or interferes with the attacker’s ability to execute specific gadgets, making them generally immune to even the strongest attacker.

### 7.3.5 Partitioning

The core idea behind partitioning is to make sensitive data unreachable to the adversary. This can be accomplished in multiple manners. With respect to mitigating Spectre attacks, there are two main approaches seen in the literature.

The most common technique is commonly referred to as *bounds clipping* and has been adopted in the Linux kernel to provide protection without paying huge performance penalties. The idea is to use a mask to constrain the index used in an array access so that it stays within the bounds of the array even during speculative execution; as a result, it can mitigate typical Spectre V1 attacks, which require an out-of-bound array access to retrieve sensitive data. A similar idea has also been proposed by Ojogbo et al. [140] to automatically apply bounds checks via masking using a compiler to mitigate Spectre attacks.

Recent work has also suggested using both new and existing hardware capabilities to create a *sandbox* between sensitive and insensitive data [141–143]. One such mitigation called Ghostbusting [141] proposes that the virtual address space be partitioned. They accomplish this using the Intel MPK technology to create up to 16 different partitions. The key idea is that a memory access to sensitive data from a partition without sufficient privilege results in a page fault, causing the page data to *not* be accessed speculatively and thus preventing potentially sensitive data from being brought into the cache. An-
other approach proposed by Reis et al. [143] is called site isolation. The key idea is to move sensitive data to a separate process, making it impossible for an adversary to access them speculatively or not. This technique leverages process isolation in operating systems.

**Quality attributes** In Table 7.1, we divide partitioning into two parts: *Mask* and *Sandbox*. Bounds clipping (masking) can be used to mitigate typical Spectre V1 attacks, which rely on out-of-bounds memory accesses. Since not all Spectre V1 attacks fundamentally rely on out-of-bounds accesses, it cannot mitigate all of Spectre V1 attacks. Other approaches such as Ghostbusting [141] and site isolation [143] can mitigate Spectre V1 attacks because they make sensitive data inaccessible, as shown under the Sandbox column of Table 7.1. Although not discussed in the presentation of Ghostbusting [141], we believe that if partitioned correctly using Intel MPK or similar technology can be effective at preventing Spectre V1, V2, and V4 attacks. Site isolation can also mitigate variants 1, 2, and 4, provided that the sensitive data is correctly separated [142].

When considering other characteristics of partitioning, we see that in general partitioning is applicable across architectures since all of them offer methods to either apply masks, enforce separation through processes, or separation based on memory domains similar to Intel’s MPK. Due to the uncertainty of processors introducing value prediction, techniques that use bounds clipping may not function properly on future processor designs [126].

Partitioning performs well across the spectrum of attack models. This is because partitioning prevents dangerous memory accesses from accessing sensitive data. Due to this, it does not matter the memory granularity, interaction granularity, or what cache is being observed.
7.3.6 Obfuscation

Some recent mitigations propose obscuring data to make it either difficult for the attacker to find the sensitive data or difficult to decipher the sensitive data. In particular, Palit et al. propose using encryption to hide sensitive data [144]. They implement a compiler pass that requires the sensitive data to annotated and develop novel methods to prevent an attacker from obtaining the encryption keys. Hence, their approach prevents the attacker from uncovering the data since it remains encrypted even if it is loaded speculatively. Sianipar et al. propose a moving target defense that moves secret data around in the address space [145]. Their goal is to prevent the attacker from learning where the data resides in memory and thus preventing them from recovering it by accessing it during speculative execution. Note that the moving target defense does not prevent or limit the speculative execution from occurring, but instead make extracting sensitive data difficult.

Quality attributes Techniques such as the one proposed by Sianipar et al. [145] move data throughout the address space and are immune to all variants of Spectre as long as the location the data is being moved to cannot be predicted. Techniques that encrypt sensitive data in memory, such as what is proposed by Palit et al. [144], are immune to Spectre V1 and V4 by virtue of the data being encrypted. Spectre V2 was not explicitly discussed in [144], however as long as the encryption routine cannot be bypassed and the key remains secret, it is immune to all variants.

For quality attributes, since the mechanism used by Palit et al. [144] requires specialized registers to hold the encryption keys, it is not a great candidate to be applied on multiple architectures. Both approaches use technology that is likely to exist in processors for the foreseeable future. The approach used in Sianipar et al. [145] relies on randomness to be effective: if the location where the data is being moved can be pre-
dicted, the defense is thwarted. Since all approaches presented make the sensitive data inaccessible, the spectrum of attacker capabilities presented in Section 7.2 is irrelevant since they cannot learn information about the sensitive data as long as the encryption key cannot be recovered or the addresses of sensitive data cannot be predicted.

### 7.4 Performance Impact

We have discussed a set of defenses against Spectre attacks. In addition to robustness and attack models, another crucial factor in choosing the proper defense is the performance impact. However, there is no comprehensive performance evaluation of defenses: each research paper uses its own evaluation platform and benchmark programs. Therefore, one important goal of this study is to offer an apples-to-apples comparison whenever possible. In particular, for many defenses we have either taken existing implementations or implemented our own version in LLVM 11, and evaluated them based on common benchmarks. For the defenses that we are unable to implement in LLVM 11 for various reasons, we take the performance numbers from related papers as a reference.

#### 7.4.1 Experimental Set Up

LLVM 11 already provides defense mechanisms including Execution Barriers, SLH, and Retpoline, which we reuse in the evaluation. For PDC, we implement three versions of PDC with Barrier: one that mitigates only Spectre V1 (PDC V1), one that mitigates only Spectre V2 (PDC V2), and one that mitigates both (PDC ALL) as described in Section 7.3.3.

For benchmark programs, we include two server applications (nginx and thttpd)

\footnote{The barrier implementation they use places a \texttt{lfence} at the beginning of each successor block following a conditional branch.}
and the C/C++ programs from the SPECint 2017 benchmarks. These benchmarks are selected to show the impact of defenses on a wide range of CPU intensive tasks (i.e., SPECint) and programs that process confidential information (i.e., the server programs). SPECint 2017 is configured to use the rate configuration, which evaluates the overall throughput. To evaluate the server benchmarks, nginx and thttpd, we use an http benchmarking program called wrk [146]. It is configured to use 12 concurrent threads and a total of 400 connections and runs for 30 seconds for each mitigation and the baseline. Wrk records the requests handled per second as well as the rate that data is transferred. Due to the inherent noise involved with establishing connections and transmitting the data, we run each test 5 times and report the arithmetic mean. All of the experiments were run on a virtual machine on a VMWare Workstation 15 with Ubuntu 18.04. The physical machine has 32GB of RAM and an Intel i7-5820K processor. The virtual machine was allocated 20GB of RAM and 4 of the 12 logical cores of the processor.

For other defenses in Section 7.3, we use results reported in related papers for various reasons. First, although Intel has provided a sample implementation of randpoline [147] as a plugin for GCC, we were unable to get it to function properly on our benchmarks. Second, we were unable to find an implementation of JumpSwitches [137] as well as partitioning and obfuscation methods. Finally, the approach of argument dependencies was evaluated in [133], from which we take the evaluation results. Figure 7.4 provides the average overheads reported from external sources.

### 7.4.2 Performance Evaluation

The results of our experiments are presented in Figures 7.5, 7.6, and 7.7. All figures report the normalized execution time, which is the execution time of a program after mitigation divided by the execution time of the baseline program without mitigation.
Figure 7.4: Average overheads reported from external sources that provide Spectre mitigations.

Figure 7.5: Normalized execution time w.r.t. an unmitigated baseline to mitigate Spectre V1 for SPECint 2017, nginx and thttpd.

Therefore, the *performance overhead* is the normalized execution time minus 100%. For instance, a normalized execution time of 105% implies an overhead of 5%. The last group of bars in each figure shows the geometric means of each mitigation among all benchmarks. Note we further present normalized binary size increase in Section 7.5.

**Spectre V1 mitigation** The experiments show that defenses for only Spectre V1 generally incur a significant amount of overhead ranging from 60% to 213%. Figure 7.5 shows that on average, the best performing mitigation on our benchmarks is SLH, with a mean performance overhead of 70%. We see from Figure 7.5 that SLH can have as low as just 4% overhead on nginx and thttpd, which indicates it can have a limited impact on
some applications. On the other end of the spectrum, the worst performing mitigation uses barriers at the beginning of every successor basic block of a branch. This mitigation incurred an average overhead of 213% across all benchmarks. This result is not surprising since SLH still allows speculative execution to some extent, while execution
barriers halt all instructions beyond the barriers.

Compared with execution barriers, PDC with Barrier improves performance overhead by 18% on average. This is likely because the total number of barriers being executed is fewer since PDC with Barrier requires only one barrier for a branch, while the execution barrier approach requires one barrier for each successor block following a branch.

Since bounds clipping is challenging to implement, we use the results presented in Baggy Bounds Checking [148] since it was tested on a variety of CPU intensive benchmarks. The results in Figure 7.4 tell that Baggy Bounds Checking on average performs better than SLH by about 11%, making it the most lightweight Spectre V1 defense that does not require the user to specify sensitive data; however, as mentioned in Section 7.3, it is specific to out-of-bounds access leakage caused by Spectre V1. Further, this approach requires identifying bounds information through instrumenting the program or through changing the memory allocator. Based on the results from [133], the argument dependency approach has a slightly lower overhead than SLH, which is consistent with the empirical results presented in [133].

Notably, for the server applications, nginx and thttpd, the overheads of all of the V1 defenses are significantly lower compared to other benchmarks, ranging from 4% to just under 20% overhead. This is likely because these applications are not as CPU intensive as those in the SPECint suite. This result highlights that, for non-CPU bound tasks that handle sensitive information, applying these defenses may not be very detrimental to performance.

**Spectre V2 mitigations** Figure 7.6 shows that on average, retpoline incurs an overhead of 18% while PDC incurs an overhead of 22%. Retpoline has better performance on 5 out of the 11 benchmarks we analyzed while PDC performs better on 6. This result is
interesting because it suggests that PDC is a promising alternative defense mechanism when retpoline has poor performance.

To compare Randpoline with Retpoline, we collect the results for Randpoline from [138], as shown in Figure 7.4. We observe that Randpoline improves the performance of Retpoline by approximately 15%, making it the best Spectre V2 mitigation with respect to performance. This observation is consistent with the comparison reported in [138], based on different benchmarks.

In general, Spectre V2 defenses have lower overhead than those for Spectre V1. This is largely because there are significantly more locations that require protection to mitigate V1 compared to V2, without additional static analysis to remove superfluous instrumentation.

**Spectre V1 and V2 mitigation** Figure 7.7 shows the overhead for protecting against both Spectre V1 and V2. Note that PDC is the only defense applicable to both. However, we also combine retpoline with barriers (retpoline+barrier) and retpoline and SLH (retpoline+SLH), as potential defenses for a comparison. The results indicate that mitigating both V1 and V2 can lead to significant performance penalties. Combining SLH and retpoline yields the best performance, but still averaging 183% overhead across all benchmarks. On the other hand, using retpoline with a barrier results in an overhead of 224% on average. Lastly, using PDC ALL results in an average overhead of 193%. For the server benchmarks, we again observe that they are not nearly as impacted by the mitigation as those in the SPECint suite.

**Partitioning and Obfuscation Methods** It would be unfair to directly compare partitioning and obfuscation methods with other ones since they both assume that all secrets in memory can be identified. Among these methods, both the moving target defense in [145] and encrypted memory in [144] can mitigate Spectre V1, V2 and V4. As
shown in Figure 7.4, both of these have relatively low-performance overhead when com-
pared with defenses that involve barriers and control flow modification. Furthermore,
approaches that apply site isolation [142] see a slight performance increase of approximately 10% largely due to using more CPU resources by splitting up tasks. Due to the
use of extra processes, the main downside of site isolation is the use of additional mem-
ory of approximately 30% [140], instead of an execution time penalty. Additionally,
Ghostbusting [141] is a promising approach to mitigate variants 1, 2, and 4 according
to the results on toy examples, although it is unclear how well it would perform on real
programs.

**Spectre V4 mitigation** While none of the defense mechanisms is specifically designed
for Spectre V4 attacks, some of them are still applicable. For example, one naive but
secure defense would insert an execution barrier before each load. However, the per-
formance would obviously be unacceptable. Some of the partitioning [141, 142] and
obfuscation [144, 145] techniques can be more promising. Since these techniques re-
duce their instrumentation burden by requiring the specification of sensitive data, they
provide protection while keeping a low overhead.

### 7.5 Evaluation of binary size increase

Figures 7.8, 7.9, and 7.10 plot the normalized binary size for benchmarks we evaluate.
Interestingly, we find that SLH has the highest overhead with respect to binary size
while maintaining the lowest overhead with respect to execution time, compared to other
defenses. In Figure 7.8, we see that SH can in some cases increase the binary size by
nearly 50%. Other defenses evaluated range from between no overhead and about 10%
overhead with respect to their binary size.
Figure 7.8: Normalized binary size w.r.t. an unmitigated baseline to mitigate Spectre variant 1 for SPECint 2017, nginx, and thttpd.

Figure 7.9: Normalized binary size w.r.t. an unmitigated baseline to mitigate Spectre variant 2 for SPECint 2017, nginx, and thttpd.

7.6 Developer Guidance and Discussion

The ultimate goal of this study is to provide developers with accurate guidance to choose the most appropriate Spectre defenses. For that, we produce Figure 7.11, which provides a two-dimensional map that summarizes results from Sections 7.3 and 7.4. In the figure, the y-axis is the performance impact of a defense. The performance impact of defenses are taken from data of Section 7.4. For defenses we implement, we take the geometric mean of the normalized execution time from all benchmark programs; for techniques we do not implement, we take numbers from Figure 7.4.

The points in the x-axis of Figure 7.11 are triples based on the results of Table 7.1.
Figure 7.10: Normalized binary size w.r.t. an unmitigated baseline to mitigate Spectre variants 1 and 2 for SPECint 2017, nginx, and thttpd.

Figure 7.11: Developer guidance, summarizing the results presented in this chapter. In this figure, the orange points require manual effort to identify sensitive data while green points do not.

The first element of the triple is the Spectre variant being mitigated. The second element is an attack model score, $A$. It is calculated as the sum of the attack model rows (the last four rows) in Table 7.1 that the mitigation applies to. The final element of the triple is the robustness score, $R$. It is computed by summing up the other quality attributes in the table. To use Figure 7.11, a developer can identify the performance and quality constraints of her application, and use the figure as a lookup map to find appropriate defenses.

From this data, we observe a trade-off between the robustness of a defense and the
overhead. Currently, the lowest overhead Spectre V1 defenses (i.e., bounds clipping and data dependencies) that do not require the user to specify sensitive data are less general. It makes some of the more robust solutions such as PDC and Barriers more appealing.

Furthermore, the defenses that are the most detrimental to performance are also the most likely to work on both current and future processors. Due to their high cost, they are generally not suitable to be applied everywhere since, as we see from the experiments, they tend to add high overheads to the program. This highlights the need for mechanisms to selectively apply appropriate defenses given the context of the threat.

Another aspect to consider is whether a mechanism requires first identifying sensitive data, as is the case for partitioning and obfuscation. If so, sensitive data must be either identified manually or through some kind of analysis. Moreover, during program execution that sensitive data may taint other data either implicitly or explicitly; a precise analysis that tracks all tainted data, while having a small amount of data to isolate, is technically challenging and may further require complex mechanisms such as data declassification.

During our evaluation, we find that there is a wide variety of tools to address Spectre vulnerabilities in software. Approaches that conservatively harden every branch, indirect jump, and memory load tend to incur higher overheads. Hence, emerging tools that apply pattern matching or static analysis to identify where to apply a mitigation as performed in MSVC [2], SpecFuzz [18], oo7 [16], and Blade [83] can dramatically improve performance. For instance, in oo7 [16], it is reported that the overhead of applying mitigation everywhere for Spectre V1 is 430% on average, while when selectively applying mitigation the overhead drops to only 5.9%. However, many of these tools utilize unsound static analysis ([2, 16, 18]) and thus leave some vulnerabilities unmitigated. All things considered, the most promising approach is to selectively apply mitigation using sound static analysis to reduce the overhead and avoid frivolous applications of
mitigation. By selectively applying mitigation like Blade [83], the overhead of even
the most costly approaches (e.g., barriers) can be drastically reduced, while retaining
quality including being future resilient and applicable across many CPUs.

7.7 Summary

In this work, we set out to perform a study of user-space software defenses against
transient execution attacks. In addition to surveying the landscape of this space, this
study makes the following systematization:

- We describe a set of quality attributes that characterize robustness and attack mod-
els used in common defenses. These attributes enable an easily digestible and
qualitative comparison amongst state-of-the-art defenses.

- In order to understand the impact on performance, for many defenses, we either
take existing implementations or implement our own on a common compiler plat-
form (LLVM 11) and evaluate their performance using a common set of bench-
mark programs.

- Based on our quality attributes and performance evaluation, we develop a roadmap
(Figure 7.11), which can be used by developers to decide what defenses are desir-
able with their constraints on performance and quality.
Future Work

In this section, we sketch interesting future directions that can enhance the contributions presented in this dissertation.

8.1 Program Analysis

In this dissertation, we presented a program analysis technique in Chapters 5 and 6 that is capable of detecting both conventional and speculative side-channels that leak data via the CPU cache. These techniques are narrowly scoped to capture only Spectre V1 vulnerabilities. Recalling from Section 2.3.2, there are multiple Spectre variants that need to be considered to ensure a program is protected.

To this end, one obvious solution is to extend SpecSafe further to also include semantics in our symbolic execution to model features such as data dependency that may be used by Spectre V4 attacks. In general, a fine grained modeling of memory leads to significant performance issues when using symbolic execution. Due to this limitation, we believe a composition of other program analysis techniques could be the best solution for providing a comprehensive analysis to capture all variants.
To expand on this idea, one interesting approach is applying a fine-grained points-to analysis to identify Spectre V4 leakage. The idea is to use an analysis that can precisely identify an alias statically between a store and load address. By statically detecting potentially problematic locations, this would allow mitigations to be applied in a more targeted manner.

Aside from using points-to analysis for identifying Spectre V4 vulnerabilities, another interesting approach we think is worth investigating is deciding if a load can actually target sensitive data even during speculative execution. This approach requires developing a points-to analysis that also has rules for speculative execution. As we saw in Chapter 7, mitigations that focus on mitigating the sensitive data instead of generally mitigating the speculative execution have significantly better overhead. The goal of this approach is to only apply mitigations to loads that can potentially target sensitive data during speculative execution, allowing multiple Spectre variants to be detected and mitigated efficiently.

### 8.2 Mitigations

While detecting potentially problematic areas in programs is important, it is only half of the issue. Mechanisms to mitigate the detected vulnerabilities are also necessary. In Chapter 4, we presented Ghost Thread, which is a user-space mitigation that can mitigate cache-based side channels on virtually all commodity hardware and highlights the value of defenses that can easily be applied on any platform.

Another possible mitigation for Spectre attacks that is applicable across many systems involves moving sensitive data such that it cannot be targeted by loads that may be vulnerable during speculative execution. Recall that the core issue with speculative executing is the ability for a CPU to access sensitive data that is accessed by instructions
that are not yet retired. Thus, the key insight here is to move the data sufficiently far such that it cannot be referenced accidentally during speculative execution.

To be a little bit more concrete. If we look back to Chapter 2 at Figure 2.2, the dangerous memory access occurs at:

\[ res = arry2[t] \]

This code becomes benign if \( arry2[t] \) cannot target sensitive data. Thus, moving sensitive data sufficiently far from the base address of \( arry2 \), maybe in the other half of the virtual address space, would ensure that no matter the value of \( t \) the program does not leak sensitive data.

Since there may not always exist a region of the virtual address space that the data can be moved to, another option is to create a region by splitting the virtual memory in half and use a mask to prevent unauthorized memory accesses. This would work by applying the mask to prevent instructions from making a memory access that could target sensitive data. For example, if the sensitive data were stored in the high virtual addresses, the mask would force the left most bit to be 0 preventing it from targeting the higher addresses where sensitive data resides. This approach could be successful against multiple Spectre variants and should introduce low overhead since the mask can be carried out with a simple bit-wise operation.

Another interesting approach to mitigate Spectre V1 attacks that we have not seen in the literature is to use the compiler to re-order instructions such that dangerous memory accesses are outside the speculation window. Recall that processors can only speculatively execute a finite number of instructions before their reorder buffer is filled. Using tools such as SpecSafe, the compiler can identify problematic branches and possibly reorder the instructions to force the load that leaks data to occur outside the speculation window. Since this approach may not always be possible, the compiler could still use other, higher overhead, mitigations such as barriers to protect the branch against Spectre
V1 attacks.
In this dissertation, we presented methodologies to soundly detect cache-based side channel attacks. Furthermore, we thoroughly investigated techniques to mitigate cache-based side channels and also proposed a new solution.

In order to capture cache-based side channel attacks, we developed a novel symbolic execution method that is instantiated in a tool called CaSym. Compared to existing work, we showed that providing more realistic cache models provides a great balance between performance and efficiency with respect the complexity of the formulas. Moreover, we showed that our technique is practical by applying it to multiple benchmarks and finding previously unknown vulnerabilities.

With the advent of Spectre vulnerabilities, we extended CaSym in a tool called SpecSafe. During our investigation, we discovered new Spectre attack patterns that existing techniques would be unable to properly classify. Moreover, we combine SpecSafe with a taint analysis pass to further improve scalability while retaining soundness. During our evaluation, we discovered vulnerabilities due both to conventional and speculative execution as a result of the extended semantics of SpecSafe and the large code coverage of the taint analysis pass.
Due to the rapidly developing landscape of Spectre mitigations, we systemically investigated existing user-space mitigations to provide the most comprehensive comparison available. This resulted in first developing a spectrum of attack models to depict different attack scenarios. Furthermore, we proposed multiple useful traits that each mitigation possess in order to highlight the key differences among mitigations. Lastly, we provided the most comprehensive performance comparison in the literature.

Lastly, we proposed a new cache-based side channel mitigation called Ghost Thread. This approach works using resources available to virtually every user-space program making it readily available. Furthermore, since the only hardware technology it requires is support for multiple threads, it is readily applicable on virtually every modern CPU. We show that Ghost Thread provides reasonable overhead and strong security against common attack models by evaluating it on multiple commonly used encryption routines.

In summary, we developed new program analysis techniques capable of soundly detecting both conventional cache-based side channels and cache-based side channels arising from the CPU’s use of speculative execution. Furthermore, we produced the most comprehensive and systematic evaluation of user-space Spectre mitigations. Finally, we created a user-space mitigation to efficiently mitigate cache-based side channel attacks and demonstrated that in real world settings it has little overhead.
Appendix A

Formalization

A.1 Formalizing Program Semantics with Speculation

Language Syntax  In order to formally discuss cache side-channel security, we define a simple imperative language in Figure 6.2. This language models important features of the C language, such as sequential composition, assignments, branches, and loops. skip represents a no-op, which becomes handy to model single-sided branches, for instance. The only non-standard command is fence, which is a special command that will temporarily disable speculation. For expressions, the language uses $n$ as a numerical constant, $X$ as a program variable, $A[E]$ an array with a base location of $A$ at offset $E$. Lastly, the language allows both arithmetic operations using $\otimes$ and Boolean operations using $\odot$ on two expressions.

Language Semantics  Recall that to model speculative execution, we introduce two parameters: a prediction oracle, $\mathcal{O}$, as well as the speculative transaction’s length $w$ in the language semantics. Moreover, $\mathcal{O}$ is a partial function that takes the id of a branch command and returns a queue of predicted branch outcomes (i.e., a queue of truth values). A queue is used in the oracle so that the semantics can have different
\[\tilde{\mu}' = \text{zero}(\mu) \quad \text{enabled}(\mu) \quad (\text{FENCE})\]
\[\langle m, \mu, \text{fence} \rangle \rightarrow \langle m, \mu', \text{skip} \rangle\]

\[\tilde{\mu} \neq \emptyset \quad \text{enabled}(\mu) \quad (\text{SKIP})\]
\[\langle m, \mu, \text{skip} \rangle \rightarrow \langle m, \text{zero}(\mu), \text{skip} \rangle\]

\[\tilde{\mu}' = \text{decr}(\mu) \quad \text{enabled}(\mu) \quad (\text{ASGN})\]
\[\langle m, \tilde{\mu}, x := e \rangle \xrightarrow{\text{loc}(e) \cdot x} \langle [x/v]m, \mu', \text{skip} \rangle\]

\[\text{eval}(e, m) = v \quad \mu' = \text{decr}(\mu) \quad \text{enabled}(\mu) \quad (\text{ASGN})\]
\[\langle m, \mu, x := e \rangle \xrightarrow{\text{loc}(e) \cdot x} \langle [x/v]m, \mu', \text{skip} \rangle\]

\[\text{eval}(b, m) = p \quad \text{enabled}(\mu') \quad (\text{COMMIT})\]
\[\langle m, \langle p, m', 0, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \cdot \mu', c \rangle \xrightarrow{\text{commit}} \langle m, \mu', c \rangle\]

\[\text{eval}(b, m) = v \neq p \quad \text{enabled}(\mu') \quad (\text{ROLLBACK})\]
\[\langle m, \langle p, m', 0, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \cdot \mu', c \rangle \xrightarrow{\text{rollback}} \langle m', \mu', c' \rangle\]

Figure A.1: Speculative-Aware semantics given an oracle \(\mathcal{O}\) and a transaction length \(w\).
outcomes for the same branch at different execution points.

We assume that each command in the source code has a unique id (such as the line number), denoted by $\eta$. To get the next predicted outcome of the branch command with id $\eta$, we use $O(\eta).pop().$

We define a configuration as a tuple $\langle m, \vec{\mu}, S \rangle$ that consists of a memory $m$, a sequence of (potentially nested) speculative states $\vec{\mu}$ as well as $S$, the remaining program to be executed. As standard, memory $m$ is formalized as a map from variables/array elements to their values. A speculative state $\mu = \langle p, m, r, S \rangle$ represents a speculative transaction [17], which consists of the predicted boolean outcome $p$, the memory $m$ prior to the start of the transaction, the remaining number of instructions to be executed in the transaction $r$, and the commands $S$ to be resumed in case of a rollback. Moreover, we use the following helper functions to update and check the remaining number of instructions in all transactions:

- $\text{decr}(\vec{\mu})$: decrease the remaining instructions by one for each $\mu \in \vec{\mu}$.
- $\text{zero}(\vec{\mu})$: set the remaining instructions to zero for each $\mu \in \vec{\mu}$.
- $\text{enabled}(\vec{\mu})$: returns true if $r \neq 0$ for any $\mu \in \vec{\mu}$ or when $\vec{\mu} = \emptyset$.

Given an oracle $O$ and a transaction length $w$, each evaluation rule (summarized in Figure A.1) has the form of $\langle m, \vec{\mu}, S \rangle \xrightarrow{e} \langle m', \vec{\mu}', S' \rangle$ where event $e$ tracks the memory locations being accessed and program instructions being executed; we use $pc(\eta)$ to denote the latter. Note that the given $O$ and $w$, the semantics is deterministic: rules COMMIT and ROLLBACK apply when one transaction ended, while the other rules require $\text{enabled}(\vec{\mu})$; rule COMMIT requires $\text{eval}(b, m) = p$ and ROLLBACK requires $\text{eval}(b, m) \neq p$.

Given a program $S$, an oracle $O$ and transaction length $w$, we use $[S]^{O}_{(m,w)}$ to denote the sequence of emitted events starting from the initial configuration $\langle m, \emptyset, S \rangle$ according
to the semantics given \( O \) and \( w \). For example, let \( S \) be the code in Listing 6.1, \( O(1) = \text{true}, w=1 \) and \( m(idx) \geq m(b_{size}) \), we have the following evaluation steps according to the semantics:

\[
\begin{align*}
\langle m, \emptyset, S \rangle \\
\xrightarrow{\text{begin } pc(2)} \langle m, \langle \text{true}, m, 1, S \rangle, \text{temp}=A[B[idx]*512] \rangle \\
\xrightarrow{idx,B+idx,A+B[idx]*512, A+idx*512} \langle m', \langle \text{true}, m, 0, S \rangle, \text{skip} \rangle \\
\xrightarrow{\text{rollback}} \langle m, \emptyset, \text{skip} \rangle
\end{align*}
\]

Hence, \( [S]_O^{(m,w)} \) generates an event trace \( \text{begin } pc(2), idx, B + idx, A + B[idx] * 512, \text{rollback} \) for the concrete execution.

We note that our interpretation of \( O \) and \( w \) over-approximates the ones in [17], which assumes a partial function that takes as input a program, a branching history and a branch instruction, and that returns as output the predicted branch outcome and transaction length. Our formulation considers all possible outcomes of the semantics in [17].

### A.2 Soundness Proof

**Proof of Theorem 1 (Transformation Soundness)** For any terminating program \( s \), if it transformed program is side channel free under CNI, then \( s \) is side channel free under SANI. That is:

\[
\forall s, \overline{s}, w, O. s \text{ terminates } \land s \rightarrow \overline{s} \implies \\
\forall m_1 \simeq_P m_2. \text{view}([s]_{(m_1,0)}^{\text{perf}}) = \text{view}([\overline{s}]_{(m_2,0)}^{\text{perf}}) \implies
\]


∀m_1 \simeq_P m_2. \text{view}(\langle s \rangle_{(m_1, w)}) = \text{view}(\langle s \rangle_{(m_2, w)})

**Proof.** Note that the transformed program executes on *extended* memory with added variables in the form of \text{pred}_q. Hence, we use notation \( m \uplus (x : v) \) to denote the extended memory of \( m \) (without variable \( x \)) so that \( m \uplus (x : v)(y) = m(y) \) if \( y \neq x \) and \( m \uplus (x : v)(x) = v \).

Moreover, for conciseness, we abuse the abstract cache model notation so that we use

\[
\text{view}(\langle s_1; s_2 \rangle_{(m, w)}) = \text{cacheM}(\langle s_1 \rangle_{(m, w)}) \cdot \text{cacheM}(\langle s_2 \rangle_{(m, w)})
\]

to represent a concatenation of the abstract cache states after executing \( s_1 \) and \( s_1; s_2 \) respectively, instead of the more precise but bulky notation

\[
\text{view}(\langle s_1; s_2 \rangle_{(m, w)}) = \text{cacheM}(\langle s_1 \rangle_{(m, w)}) \cdot \text{cacheM}(\langle s_1; s_2 \rangle_{(m, w)})
\]

Finally, we use \( s_* \) to denote the sub-program right after the sub-program being analyzed (i.e., the extra instructions being padded in case of of misprediction).

We prove the theorem using structural induction on \( s \).

- **skip** and **fence**: trivial since they do not emit any events and hence, their views are both empty.
- **x := e**: since the transformed program is identical to the original, the conventional and speculative semantics yield the same events. Hence, the conclusion is correct.
- **s_1; s_2**: we have the following traces for a program under speculative semantics where \( i \in \{1, 2\} \)

\[
\langle m_i, \mu, s_1; s_2 \rangle \xrightarrow{\ast} \langle m_i', \mu_i, s_2 \rangle
\]

Hence, we have that for
\[ i \in \{1, 2\}, \text{view}(\llbracket s_1; s_2 \rrbracket_{(m_i, w)}) = \text{CacheM}(\llbracket s_1 \rrbracket_{(m_i, w)}) \cdot \text{CacheM}(\llbracket s_2 \rrbracket_{(m_i, w)}). \]

On the other hand, the transformed program yields the following traces under the conventional semantics:

\[
\langle m_i, \mu, s_1; \text{check } P_1; s_2 \rangle \xrightarrow{*} \langle m'_i, \mu_i, \text{check } P_1; s_2 \rangle \rightarrow \langle m'_i, \mu_i, s_2 \rangle
\]

which gives that for \( i \in \{1, 2\}, \)

\[
\text{view}(\llbracket s_1; s_2 \rrbracket) = \text{CacheM}(\llbracket s_1 \rrbracket_{(m_i, 0)}) \cdot \text{CacheM}(\llbracket s_2 \rrbracket_{(m'_i, 0)})
\]

In the trace, check \( P_1 \) ensures that \( \text{CacheM}(\llbracket s_1 \rrbracket_{(m_1, 0)}) = \text{CacheM}(\llbracket s_1 \rrbracket_{(m_2, 0)}) \). By assumption, \( \text{view}(\llbracket s_1; s_2 \rrbracket_{(m_1, 0)}) = \text{view}(\llbracket s_1; s_2 \rrbracket_{(m_2, 0)}) \), allowing us to know \( \text{CacheM}(\llbracket s_2 \rrbracket_{(m'_i, 0)}) = \text{CacheM}(\llbracket s_2 \rrbracket_{(m'_2, 0)}) \) must be true. Due to the soundness assumption of the taint analysis, we have that \( m'_1, m'_2 \) are low equivalent regarding set of public variables right before \( s_2 \). Hence, by induction hypothesis on \( s_1 \) and \( s_2 \), we have

\[
\text{CacheM}(\llbracket s_1 \rrbracket_{(m_1, w)}) = \text{CacheM}(\llbracket s_1 \rrbracket_{(m_2, w)})
\]

\[
\text{CacheM}(\llbracket s_2 \rrbracket_{(m'_1, w)}) = \text{CacheM}(\llbracket s_2 \rrbracket_{(m'_2, w)})
\]

Hence, we conclude with

\[
\text{view}(\llbracket s_1; s_2 \rrbracket_{(m_1, w)}) = \text{view}(\llbracket s_1; s_2 \rrbracket_{(m_2, w)})
\]

- if \( B \) then \( s_1 \) else \( s_2 \): Let \( p = O(\eta) \cdot \text{pop}() \) where \( \eta \) is the command id. Let \( c \) be the initial cache state, and \( m_1, m_2 \) be low equivalent initial memories. There are 8 cases for if statements:
Case 1: \(\langle B, m_1 \rangle \downarrow true, \langle B, m_2 \rangle \downarrow true, p = true\):

In this case, the IF condition \(B\) evaluates to true for both memories \(m_1\) and \(m_2\), and the speculation oracle predicts \(p\) to be true, resulting in a correct speculation in both traces. This results in the following next steps and views under the speculative semantics:

\[
\langle m_1, \mu, if \begin{pc}(s_1) \rangle \xrightarrow{begin-pc(s_1)} \langle m_1, \mu, s_1 \rangle \quad \text{and} \quad \text{view}(\lbrack if \begin{pc}(s_1) \rbrack = \text{view}(\lbrack s_1 \rbrack)
\]

\[
\langle m_2, \mu, if \begin{pc}(s_1) \rangle \xrightarrow{begin-pc(s_1)} \langle m_2, \mu, s_1 \rangle \quad \text{and} \quad \text{view}(\lbrack if \begin{pc}(s_1) \rbrack = \text{view}(\lbrack s_1 \rbrack)
\]

On the other hand, with \(m_1 = m_1 \uplus (pred_\eta : true)\) and \(m_2 = m_2 \uplus (pred_\eta : true)\), the transformed program, say \(ifcmd\), generates the following next steps and views under the conventional semantics where \(i \in \{1, 2\}\):

\[
\langle m_i, \mu, ifcmd \rangle \xrightarrow{*} \langle m_i, \mu, s_1 \rangle \quad \text{and} \quad \text{view}(\lbrack ifcmd \rbrack = \text{view}(\lbrack s_1 \rbrack)
\]

Hence, if the views agree on the transformed programs, we have \(\text{view}(\lbrack s_1 \rbrack) = \text{view}(\lbrack s_1 \rbrack)\). By induction hypothesis on statement \(s_1\), we get that \(\text{view}(\lbrack s_1 \rbrack) = \text{view}(\lbrack s_1 \rbrack)\). Hence, the views must agree on the transformed programs under speculative execution. That is,

\[
\text{view}(\lbrack if \begin{pc}(s_1) \rbrack = \text{view}(\lbrack if \begin{pc}(s_1) \rbrack)
\]

Case 2: \(\langle B, m_1 \rangle \downarrow true, \langle B, m_2 \rangle \downarrow true, p = false\):

In this case, the IF condition is true, but the speculation oracle predicts \(p\) to be
false, resulting in a mis-speculation for each trace. This results in the following next steps and views under speculative semantics:

\[
\langle m_1, \mu, \text{if } B \text{ then } s_1 \text{ else } s_2 \rangle \xrightarrow{\text{begin-}pc(s_2)} \langle m_1, \mu_1, s_2; s_* \rangle \\
\rightarrow \ldots \\
\xrightarrow{\text{rollback}} \langle m_1, \mu'_1, s_1 \rangle
\]

\[
\langle m_2, \mu, \text{if } B \text{ then } s_1 \text{ else } s_2 \rangle \xrightarrow{\text{begin-}pc(s_2)} \langle m_2, \mu_2, s_2; s_* \rangle \\
\rightarrow \ldots \\
\xrightarrow{\text{rollback}} \langle m_2, \mu'_2, s_1 \rangle
\]

Hence, we have that

\[
\text{view}(\lfloor \text{if } B \text{ then } s_1 \text{ else } s_2 \rfloor^C_{(m_1,w)}) = \\
\text{cacheM}(\lfloor [s_2; s_*]_w^C_{(m_1,w)} \rfloor) \cdot \text{cacheM}(\lfloor s_1^C_{(m_1,w)} \rfloor)
\]

as well as

\[
\text{view}(\lfloor \text{if } B \text{ then } s_1 \text{ else } s_2 \rfloor^C_{(m_2,w)}) = \\
\text{cacheM}(\lfloor [s_2; s_*]_w^C_{(m_2,w)} \rfloor) \cdot \text{cacheM}(\lfloor s_1^C_{(m_2,w)} \rfloor)
\]

On the other hand, the following are the next steps and views of the transformed program under conventional semantics with \( m_1 = m_1 \uplus (\text{pred}_\eta : \text{false}) \) and \( m_2 = m_2 \uplus (\text{pred}_\eta : \text{false}) \), where \( i \in \{1, 2\} \):
\[ \langle m_i, \mu, \text{ifcmd} \rangle \overset{*}{\to} \langle m_i, \mu_i, [s_2; s_\ast]_w; \text{rollback}(\eta); s_1 \rangle \overset{*}{\to} \langle m'_i, \mu'_i, \text{rollback}(\eta); s_1 \rangle \overset{*}{\to} \langle m_i, \mu_i, s_1 \rangle \]

Hence, we have for \( i \in \{1, 2\} \), \( \text{view}([\text{ifcmd}]_{(m_i,0)}^{\text{perf}}) = \text{cacheM}([\{s_2; s_\ast\}_w]_{(m_i,0)}^{\text{perf}}) \cdot \text{cacheM}([s_1]_{(m_i,0)}^{\text{perf}}) \).

Moreover, with \( m'_1 = m_1 \uplus (\text{pred}_\eta : \text{true}) \) and \( m'_2 = m_2 \uplus (\text{pred}_\eta : \text{true}) \), we also have the following evaluation traces on the transformed program, where \( i \in \{1, 2\} \):

\[ \langle m'_i, \mu, \text{ifcmd} \rangle \overset{*}{\to} \langle m'_i, \mu_i, s_1 \rangle \]

Due to the assumption that the transformed program is side channel free, we have \( \text{cacheM}([s_1]_{(m_i,0)}^{\text{perf}}) = \text{cacheM}([s_1]_{(m'_i,0)}^{\text{perf}}) \). Since \( m'_i \) and \( m_i \) only differ on the value of \( \text{pred}_\eta \), which is not used in \( s_1 \) by construction, we have \( \text{view}([s_1]_{(m_i,0)}^{\text{perf}}) = \text{view}([s_1]_{(m'_i,0)}^{\text{perf}}) \) as well. Hence, it must be true that

\[ \text{view}([s_2; s_\ast]_{(m_i,0)}^{\text{perf}}) = \text{view}([s_2; s_\ast]_{(m'_i,0)}^{\text{perf}}) \]

since we have already derived that,

\[ \text{view}([s_2; s_\ast]_{(m_i,0)}^{\text{perf}}) \cdot \text{view}([s_1]_{(m_i,0)}^{\text{perf}}) = \]
Hence, based on the induction hypothesis, we have
\[ \text{cacheM}([s_1]_{O(m_1,w)}) = \text{cacheM}([s_1]_{O(m_2,w)}) \] and
\[ \text{cacheM}([s_2; s_*]_{O(m_1,w)}) = \text{cacheM}([s_2; s_*]_{O(m_2,w)}) \]. Therefore, we have
\[ \text{view}([\text{if } B \text{ then } s_1 \text{ else } s_2]_{O(m_1,w)}) = \text{view}([\text{if } B \text{ then } s_1 \text{ else } s_2]_{O(m_2,w)}) \]

- **Case 3**: \( \langle B, m_1 \rangle \Downarrow \text{false}, \langle B, m_2 \rangle \Downarrow \text{false}, p = \text{true} \):
  Symmetric to Case 2.

- **Case 4**: \( \langle B, m_1 \rangle \Downarrow \text{false}, \langle B, m_2 \rangle \Downarrow \text{false}, p = \text{false} \):
  Symmetric to Case 1.

- **Case 5**: \( \langle B, m_1 \rangle \Downarrow \text{false}, \langle B, m_2 \rangle \Downarrow \text{true}, p = \text{false} \):
  In this case, the branch condition \( B \) is \text{false} in one trace and \text{true} in the other.
  The speculation oracle predicts prediction \( p \) to be false, resulting in a mis-speculation with \( m_2 \). This results in the following next steps and views under speculative semantics:
  \[ \langle m_1, \mu, \text{if } B \text{ then } s_1 \text{ else } s_2 \rangle \xrightarrow{\text{begin-\text{pc}(s_2)}} \langle m_1, \mu_1, s_2 \rangle \xrightarrow{\ldots} \] and
  \[ \langle m_2, \mu, \text{if } B \text{ then } s_1 \text{ else } s_2 \rangle \xrightarrow{\text{begin-\text{pc}(s_2)}} \langle m_2, \mu_2, s_2, s_* \rangle \xrightarrow{\ldots} \] and
  \[ \xrightarrow{\text{rollback}} \langle m_2, \mu_1', s_1 \rangle \]
Hence, we have:

\[
\text{view}([\text{if } B \text{ then } s_1 \text{ else } s_2]_{(m_1, w)}) = \text{cacheM}(pc(s_2)) \cdot \text{cacheM}([s_2]_{(m_1, w)}) \quad \text{and}
\]

\[
\text{view}([\text{if } B \text{ then } s_1 \text{ else } s_2]_{(m_2, w)}) = \text{cacheM}(pc(s_2)) \cdot \text{cacheM}([s_2; s_*]_{(m_2, w)}) \cdot \\
\text{cacheM}([s_1]_{(m_2, w)}).
\]

On the other hand, we have the following next steps and views under conventional semantics, with \( m_1' = m_1 \uplus (\text{pred}_\eta : \text{false}) \) and \( m_2' = m_2 \uplus (\text{pred}_\eta : \text{false}) \):

\[
\langle m_1, \mu, \text{ifcmd} \rangle \xrightarrow{*} \langle m_1, \mu_1, s_2 \rangle
\]

which gives \( \text{view}([\text{ifcmd}]^\text{perf}_{(m_1, 0)}) = \text{view}([s_2]_{(m_1, 0)}) \) and

\[
\langle m_2, \mu, \text{ifcmd} \rangle \xrightarrow{*} \\
\langle m_2, \mu_2, [s_2; s_*]_{w}; \text{rollback}(\eta); s_1 \rangle \xrightarrow{*} \\
\langle m_2', \mu'_2, \text{rollback}(\eta); s_1 \rangle \rightarrow \\
\langle m_2, \mu_2, s_1 \rangle
\]

which gives us:

\[
\text{view}([\text{ifcmd}]^\text{perf}_{(m_2, 0)}) = \text{cacheM}([s_2; s_*]_{w})^\text{perf}_{(m_2, 0)} \cdot \text{cacheM}([s_1]_{w})^\text{perf}_{(m_2, 0)}
\]

Since \( \text{view}([\text{ifcmd}]^\text{perf}_{(m_1, 0)}) = \text{view}([\text{ifcmd}]^\text{perf}_{(m_2, 0)}) \) by assumption and we also have \( \text{view}([s_2]_{w})^\text{perf}_{(m_1, 0)} \) is a prefix of \( \text{view}([s_2; s_*]_{w}; s_1]_{w})^\text{perf}_{(m_2, 0)} \), we have that

\[
\text{cacheM}([s_2]_{w})^\text{perf}_{(m_1, 0)} = \text{cacheM}([s_2]_{w})^\text{perf}_{(m_2, 0)} \quad \text{and}
\]

\[
\text{cacheM}([s_*]_{w}; s_1]_{w})^\text{perf}_{(m_2, 0)} = \emptyset. \quad \text{By induction hypothesis, view}([s_2]_{w})^\text{perf}_{(m_1, w)} = \\
\text{view}([s_2]_{w})^\text{perf}_{(m_2, w)} \quad \text{and}
\]
view([s_w; s_1]^{\text{O}}_{\langle m_2, w \rangle}) = \emptyset. Hence,

view([\text{ifcmd}]^{\text{O}}_{\langle m_1, w \rangle}) = view([\text{ifcmd}]^{\text{O}}_{\langle m_2, w \rangle})

- Case 6: \langle B, m_1 \rangle \Downarrow \text{true}, \langle B, m_2 \rangle \Downarrow \text{false}, p = \text{false}:
  Symmetric to Case 5.

- Case 7: \langle B, m_1 \rangle \Downarrow \text{false}, \langle B, m_2 \rangle \Downarrow \text{true}, p = \text{true}:
  Symmetric to Case 5.

- Case 8: \langle B, m_1 \rangle \Downarrow \text{true}, \langle B, m_2 \rangle \Downarrow \text{false}, p = \text{true}:
  Symmetric to Case 5.

Case (while B do s): Let \( p = \text{O}(\eta).\text{pop()} \) where \( \eta \) is the command id. Let \( m_1 \) and \( m_2 \) be low equivalent memories. Furthermore, we denote the transformed program as \( \text{whilecmd} \). Due to the check, reset commands inserted in the transformed code, we have that the following holds:

\[
\forall m_1 \simeq_p m_2. \text{view}(\{S\}^{\text{perf}}_{\langle m_1, 0 \rangle}) = \text{view}(\{S\}^{\text{perf}}_{\langle m_2, 0 \rangle})
\]

where \( S \) is defined in the transformation rule as capture(\eta);

\[
(\text{if } B \text{ then } (\text{if } \neg\text{pred}_\eta \text{ then } [s_w; \text{rollback}(\eta)]; s) \\
\text{else } (\text{if } \text{pred}_\eta \text{ then } [s; \text{while } B \text{ do } s]; s_w; \text{rollback}(\eta)));
\]

There are 8 cases,
Case 1: \( \langle B, m_1 \rangle \Downarrow true, \langle B, m_2 \rangle \Downarrow true, p = true \)

In this case, the condition for the while loop is true on entry in both memories and the prediction \( p \) is true. This results in the following traces under speculative semantics, where \( i \in \{1, 2\} \):

\[
\langle m_i, \mu, (\text{while } B \text{ do } s) \rangle \rightarrow \langle m_i, \mu, \text{if } B \text{ then } s; (\text{while } B \text{ do } s) \text{ else skip} \rangle \xrightarrow{\text{begin pc}(s)} \langle m_i, \mu_i, s; (\text{while } B \text{ do } s) \rangle \xrightarrow{*} \ldots
\]

Hence, we have for \( i \in \{1, 2\} \)

\[
\text{view}(\llbracket \text{while } B \text{ do } s \rrbracket^O_{(m_i, w)}) = \text{cacheM}(\text{pc}(s)) \cdot \text{cacheM}(\llbracket s \rrbracket^O_{(m_i, w)}) \cdot \text{cacheM}(\llbracket (\text{while } B \text{ do } s) \rrbracket^O_{(m'_i, w)})
\]

On the other hand, we consider execution traces on the transformed program, denoted as \( \text{whilecmd} \). As discussed in the beginning of the Loop case, we have

\[
\forall m_1 \simeq_p m_2. \text{view}(\llbracket S \rrbracket^\text{perf}_{(m_1, 0)}) = \text{view}(\llbracket S \rrbracket^\text{perf}_{(m_2, 0)})
\]

In this case, we consider the execution of \( S \) with \( m_1' = m_1 \cup (\text{pred}_\eta : \text{true}) \) and \( m_2' = m_2 \cup (\text{pred}_\eta : \text{true}) \) under the conventional semantics, where \( i \in \{1, 2\} \):

\[
\langle m_i, \mu, S \rangle \xrightarrow{\text{begin pc}(\neg \text{pred}_\eta), \text{pc}(s)} \langle m_i, \mu, s_i \rangle
\]

Hence, we have that \( \text{view}(\llbracket S \rrbracket^\text{perf}_{(m_1, 0)}) = \text{view}(\llbracket S \rrbracket^\text{perf}_{(m_2, 0)}) \). By induction hypothesis
on $s$, we have $\text{view}(\llbracket s \rrbracket_{(m_1,w)}) = \text{view}(\llbracket s \rrbracket_{(m_2,w)})$. Due to the soundness of taint analysis, we know that $m'_1$ and $m'_2$ are low equivalent regarding secret-dependent variables before the loop. Therefore, by induction on the length of the evaluation, we get that

$$\text{view}(\llbracket \text{while } B \text{ do } s \rrbracket_{(m'_1,w)}) = \text{view}(\llbracket \text{while } B \text{ do } s \rrbracket_{(m'_2,w)})$$

Therefore, we conclude that

$$\text{view}(\llbracket \text{while } B \text{ do } s \rrbracket_{(m_1,w)}) = \text{view}(\llbracket \text{while } B \text{ do } s \rrbracket_{(m_2,w)})$$

• Case 2: $\langle B,m_1 \rangle \Downarrow \text{true}, \langle B,m_2 \rangle \Downarrow \text{false}, p = \text{true}$

In this case, the condition for the while loop is true on entry for $m_1$ and false on entry to $m_2$ and the prediction $p$ is true. This results in the following traces under speculative semantics:

$$\langle m_1, \mu, (\text{while } B \text{ do } s) \rangle \rightarrow$$

$$\langle m_1, \mu, \text{if } B \text{ then } s; (\text{while } B \text{ do } s) \text{ else skip} \rangle \xrightarrow{\text{begin}	ext{-}pc(s)}$$

$$\langle m_1, \mu_1, s; (\text{while } B \text{ do } s) \rangle \xrightarrow{*} \ldots$$

which gives us

$$\text{view}(\llbracket \text{while } B \text{ do } s \rrbracket_{(m_1,w)}) = \text{cacheM}(pc(s)).$$
cacheM([s]_{(m_1,w)}) \cdot cacheM([\text{while } B \text{ do } s])_{(m_1,w)}

and

\langle m_2, \mu, (\text{while } B \text{ do } s) \rangle \rightarrow \\
\langle m_2, \mu, \text{if } B \text{ then } s; (\text{while } B \text{ do } s) \text{ else } \text{skip} \rangle \\
\langle m_2, \mu, s; (\text{while } B \text{ do } s); s \rangle \rightarrow \ldots \text{rollback} \\
\langle m_2, \mu, \text{skip} \rangle

which gives us

\text{view}([\text{while } B \text{ do } s]_{(m_2,w)}) = \\
\text{cacheM}(pc(s)) \cdot \text{cacheM}([s; (\text{while } B \text{ do } s); s]_{w})_{(m_2,w)}

On the other hand, recall that we have the following on the transformed program:

$$\forall m_1 \approx m_2. \text{view}([S]_{(m_1,0)}) = \text{view}([S]_{(m_2,0)})$$

With $m_1 = m_1 \uplus (\text{pred}_\eta : \text{false}, \text{pred}_{\eta'} = \text{true})$ and $m_2 = m_2 \uplus (\text{pred}_\eta : \text{false}, \text{pred}_{\eta'} = \text{true})$, we can derive that

\text{cacheM}(pc(if \neg \text{pred}_\eta)) \cdot \text{cacheM}(pc(s)) \cdot \text{cacheM}([s]_{\text{perf}}_{(m_2,0)})

\begin{align*}
&= \text{view}([S]_{\text{perf}}_{(m_1,0)}) = \text{view}([S]_{\text{perf}}_{(m_2,0)}) \\
&= \text{cacheM}(pc(if \neg \text{pred}_\eta')) \cdot \text{cacheM}(pc([w])) \cdot \text{view}([s; \text{while } B \text{ do } s; s]_{w})_{\text{perf}}_{(m_2,0)}
\end{align*}

Hence, it must be true that cacheM ignores pc, and that $\text{cacheM}([S]_{\text{perf}}_{(m_1,0)}) =$
view([[s; while B do s; s*]_{w}^{perf}_{(m_2,0)}].

Following a similar derivation we can derive the following with \( m_1' = m_1 \cup (\text{pred}_{\eta} : \text{false}, \text{pred}_{\eta'} = \text{false}) \) and \( m_2' = m_2 \cup (\text{pred}_{\eta} : \text{false}, \text{pred}_{\eta'} = \text{false}) \)

\[
\text{view}([[s]^{perf}_{(m_1',0)}]) = \text{view}([[\text{skip}]^{perf}_{(m_2',0)}]) = \emptyset
\]

Note that \( m_1, m_1' \) only differ on the values of \( \text{pred}_{\eta} \) and \( \text{pred}_{\eta'} \), which are not used in \( s \) by construction. Hence, it must be true that the view of \( [[s]^{perf}_{(m_1,0)}] \) and \( [[s; while B do s; s*]_{w}^{perf}_{(m_2,0)}] \) are both empty, which implies that:

\[
\text{view}([[s]^{O}_{(m_1,w)}]) = \text{view}([[s; while B do s; s*]_{w}^{O}_{(m_2,w)}]) = \emptyset
\]

Note that due to the soundness of taint analysis, \( m_1' \) and \( m_2 \) must be low equivalent regarding secret-dependent variables before the loop since any modified variable in \( s \) is tainted due to the implicit flow from the sensitive branch condition. Therefore, by induction on the length of the evaluation, we get that

\[
\text{view}([[\text{while B do s}]^{O}_{(m_1,w)}]) = \text{view}([[\text{while B do s}]^{O}_{(m_2,w)}])
\]

- Case 3: \( \langle B, m_1 \rangle \downarrow \text{false}, \langle B, m_2 \rangle \downarrow \text{true}, p = \text{true} \)

  Symmetric to case 2.

- Case 4: \( \langle B, m_1 \rangle \downarrow \text{false}, \langle B, m_2 \rangle \downarrow \text{false}, p = \text{true} \)

  In this case, the condition for the while loop is false on entry in both memories and the prediction \( p \) is true. This results in the following traces under speculative
semantics, where $i \in \{1, 2\}$:

$$\langle m_i, \mu, (\text{while } B \text{ do } s) \rangle \rightarrow \langle m_i, \mu, \text{if } B \text{ then } s; (\text{while } B \text{ do } s) \text{ else skip} \rangle \xrightarrow{\begin{array}{c} \text{begin}\cdot \text{pc}(s) \end{array}}$$

$$\langle m_i, \mu, s; (\text{while } B \text{ do } s); s^* \rangle \rightarrow \ldots \xrightarrow{\text{rollback}}$$

$$\langle m_i, \mu_i, \text{skip} \rangle$$

Hence, we have:

$$\text{view}(\llbracket \text{while } B \text{ do } s \rrbracket_{(m_i,w)}) =$$

$$\text{cacheM}(\text{pc}(s)) \cdot$$

$$\text{cacheM}(\llbracket s; (\text{while } B \text{ do } s); s^* \rrbracket_{(m_i,w)})$$

with $i \in \{1, 2\}$.

Recall that the following holds:

$$\forall m_1 \simeq_P m_2. \text{view}(\llbracket S \rrbracket_{(m_1,0)}^\text{perf}) = \text{view}(\llbracket S \rrbracket_{(m_2,0)}^\text{perf})$$

In this case, we consider

$$m_1 = m_1 \uplus (\text{pred}_\eta : \text{true}, \text{pred}_{\eta'} = \text{true})$$

and

$$m_2 = m_2 \uplus (\text{pred}_\eta : \text{true}, \text{pred}_{\eta'} = \text{true})$$

, which yields the following traces for $S$ under conventional semantics, where
\[ i \in \{1, 2\}: \]
\[ \langle m_i, \mu, S \rangle \xrightarrow{\ast} \langle m_i, \mu, [s; \text{while } B \text{ do } s; s_{\ast}]_w; \text{rollback} \rangle \]

Hence, we have:

\[ \text{view} (\llbracket [s; \text{while } B \text{ do } s; s_{\ast}]_w \rrbracket^\text{perf} (m_1, 0)) = \]
\[ \text{view} (\llbracket S \rrbracket^\text{perf} (m_1, 0)) = \]
\[ \text{view} (\llbracket S \rrbracket^\text{perf} (m_2, 0)) = \]
\[ \text{view} (\llbracket [s; \text{while } B \text{ do } s; s_{\ast}]_w \rrbracket^\text{perf} (m_2, 0)) \]

By induction hypothesis on \([s; \text{while } B \text{ do } s; s_{\ast}]_w\), we have that:

\[ \text{view} (\llbracket [s; (\text{while } B \text{ do } s); s_{\ast}]_w \rrbracket^O (m_1, w)) = \]
\[ \text{view} (\llbracket [s; (\text{while } B \text{ do } s); s_{\ast}]_w \rrbracket^O (m_2, w)) \]

Due to the soundness of taint analysis, we know that \(m'_1\) and \(m'_2\) are low equivalent regarding secret-dependent variables before the loop. Therefore, by induction on the length of the evaluation, we get that:

\[ \text{view} (\llbracket [\text{while } B \text{ do } s]_w \rrbracket^O (m'_1, w)) = \text{view} (\llbracket [\text{while } B \text{ do } s]_w \rrbracket^O (m'_2, w)) \]

Therefore, we conclude that:

\[ \text{view} (\llbracket [\text{while } B \text{ do } s]_w \rrbracket^O (m_1, w)) = \text{view} (\llbracket [\text{while } B \text{ do } s]_w \rrbracket^O (m_2, w)) \]
Case 5: $\langle B, m_1 \rangle \downarrow true$, $\langle B, m_2 \rangle \downarrow true$, $p = false$

Similar to case 4.

Case 6: $\langle B, m_1 \rangle \downarrow false$, $\langle B, m_2 \rangle \downarrow true$, $p = false$

Similar to case 2.

Case 7: $\langle B, m_1 \rangle \downarrow true$, $\langle B, m_2 \rangle \downarrow false$, $p = false$

Similar to case 2.

Case 8: $\langle B, m_1 \rangle \downarrow false$, $\langle B, m_2 \rangle \downarrow false$, $p = false$

Both original program under speculative semantics and transformed program under conventional semantics will evaluate to \texttt{skip} with the same pc and no memory access. Hence, the conclusion is trivially true.
Bibliography


URL http://arxiv.org/abs/1807.05843


URL https://doi.org/10.1109/MSP.2009.165


URL http://dl.acm.org/citation.cfm?id=3241189.3241195

URL http://doi.acm.org/10.1145/2487726.2487728


URL http://dl.acm.org/citation.cfm?id=2831143.2831200

URL http://doi.acm.org/10.1145/2487726.2487728


[52] CARRUTH, C., “Speculative Load Hardening,” . URL https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6


URL https://doi.org/10.1109/MM.2016.85


URL http://dl.acm.org/citation.cfm?id=2337159.2337172


URL https://doi.org/10.1145/2897845.2897867


URL http://dblp.uni-trier.de/db/conf/ches/ches2016.html#YaromGH16


URL http://dl.acm.org/citation.cfm?id=977395.977673


URL https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00089

URL http://dl.acm.org/citation.cfm?id=3154768.3154779


EARNshaw, R., “Mitigation against unsafe data speculation (CVE-2017-5753),” https://lwn.net/Articles/759438/.


URL https://www.usenix.org/conference/atc19/presentation/amit


URL https://doi.org/10.1145/3368826.3377921


URL https://www.chromium.org/home/chromium-security/site-isolation

URL https://www.usenix.org/conference/usenixsecurity19/presentation/reis

URL https://doi.org/10.1145/3359789.3359815


Vita

Robert Brotzman

EDUCATION

Pennsylvania State University
Ph.D. candidate in Computer Science and Engineering, GPA: 4.0/4.0 May 2021

Kutztown University
B.S. in Computer Science, GPA: 4.0/4.0 May 2015

EXPERIENCE

Perspecta Labs
Research Scientist
Basking Ridge, NJ
December 2020 - Present

- Support a variety of research projects related to cyber security

Pennsylvania State University
Graduate Researcher
State College, PA
Fall 2016 - Fall 2020

- Studied Cache-based side channels culminating in the design and implementation of CaSym, SpecSafe, and Ghost Thread to detect and mitigate them.

PUBLICATIONS


- X. Ge, B. Niu, **R. Brotzman**, Y. Chen, H. Han, P. Godefroid, and W. Cui. “HyperFuzzer: An Efficient and Precise Hybrid Fuzzer for Hypervisors” under submission