

EXECUTION SECURITY IN THE SPECTRE ERA

A dissertation presented in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the field of

CYBERSECURITY

by

ANDREA MAMBRETTI

Committee Members

Engin Kirda, Northeastern University

Guevara Noubir, Northeastern University

Aanjhan Ranganathan, Northeastern University

Alessandro Sorniotti, IBM Research - Europe

Anil Kurmus, IBM Research - Europe

Vasileios Kemerlis, Brown University

NORTHEASTERN UNIVERSITY

KHOURY COLLEGE OF COMPUTER SCIENCES

BOSTON, MASSACHUSETTS

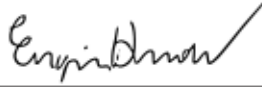
JANUARY 2022

Thesis Title: Execution Security in the Spectre Era


Author: Andrea Mambretti

PhD Program: Computer Science x Cybersecurity Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

Engin Kirda  01/26/2022
Thesis Advisor Date

Guevara Noubir  1/28/2022
Thesis Reader Date

Aanjhan Ranganathan  01/28/2022
Thesis Reader Date

Alessandro Sorniotti  01/26/2022
Thesis Reader Date

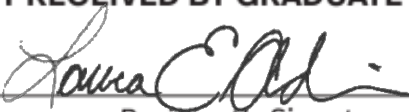
Anil Kurmus  26 Jan 2022
Thesis Reader Date

Vasileios Kemerlis  01/26/2022
Thesis Reader Date

KHOURY COLLEGE APPROVAL:

Amal Jackson 03/31/2022
Associate Dean for Graduate Programs Date

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

 31 March 2022
Recipient's Signature Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

*For my parents, Patrizia & Angelo.
For my sister Manu, and my brother Marco.*

Abstract

Since early 2018 with the release of new attacks such as Meltdown and Spectre the search for new attack surfaces left the software domain and reached the microarchitectural world. This new type of vulnerabilities exploits bugs, or performance optimizations within the CPU to carry out information disclosure even across privilege domains. This new class of vulnerabilities, referred broadly as *transient execution*, presents a unique challenge because of the lack of details in the microarchitectural realm and tools to study such behaviors. While high level views are available, internal CPU implementations are highly variable from the vendors and the CPU families and often covered by patents.

In this thesis, I provide research into understanding the impact of transient execution attacks in the field of system security. My contributions focus on two specific problems, improving the analysis of transient execution attacks, and understanding their impact on the security of modern systems—i.e., the effects of these attacks on the current existing threat models.

First, I provide a new debug-like technique to study transient execution attacks and reverse engineer the microarchitecture. Leveraging the CPU Performance Monitor Counters (PMCs), I show how it is possible to deterministically observe the side effects of transient execution. I integrate such principle in a new tool, SPECULATOR, that provides the infrastructure to easily build tests to shed light on the microarchitecture internals. Using SPECULATOR, I provide, as results, insights in the microarchitecture internal behaviors, the study of a new Spectre variant called *Split Spectre* and, two new side-channel gadgets—i.e., the Branch Target Buffer (BTB) and the instruction cache (i-cache)—that can be used as alternative to the common data cache.

Second, I provide insights on the effects of transient execution attacks in existing threat models. My effort towards solving this problem is twofold. On the one hand, I study a subset of the Spectre family of attacks, the SPEculative ARchitectural control flow hijacks (SPEAR) and their effect on current memory corruption mitigations—i.e., the Stack Smashing Protection (SSP), the Control Flow Integrity (CFI), and the stack protections in memory safe languages. I show how these mitigations, while mitigating memory corruption vulnerabilities, extend the attack surface in the context of transient execution attacks. My results indicate the need for such mitigations to be re-designed to include transient execution attacks in the threat model.

On the other hand, I present the first study of the transient execution vulnerability checkers. I provide insights in current methodologies—i.e., their strengths and weaknesses—, and show how these tools are not adequate to understand the security stance of a system against transient execution attacks. As a result, I propose a new hybrid tool, called GHOSTBUSTER that overcomes the issues of the state-of-the-art and provides results that are threat model aware.

Contents

List of Figures	xi
List of Tables	xiii
Acknowledgements	xiv
1 Introduction	1
2 Background	5
2.1 Microarchitecture	5
2.1.1 Pipeline	5
2.1.2 Cache	5
2.1.3 Branch Prediction	6
2.1.4 Out-of-order Execution	6
2.1.5 Speculative Execution	6
2.1.6 Multiprocessing and multithreading	7
2.2 Transient execution attacks	7
2.2.1 Fault-based attacks	7
2.2.2 Speculation-based attacks	7
2.2.3 Speculative Execution Attacks Phases	8
2.3 Privilege boundaries and attack impact	9
2.4 Defenses	9
2.4.1 Memory Fencing	9
2.4.2 Branchless masking	10
2.4.3 Retpoline	10
2.4.4 KPTI	10
2.4.5 Indirect Branch Restricted Speculation	10
2.4.6 Indirect Branch Predictor Barrier	10
2.4.7 Single thread indirect Branch Predictors	10
2.4.8 RSB filling	11
2.4.9 SSB mitigation	11
2.4.10 PTE inversion	11
2.4.11 VMC conditional	11

3	Related Work	13
3.1	Speculative Execution	13
3.2	Cache Side Channels	13
3.3	Speculative Execution Attacks	13
3.4	Mitigations	14
3.5	Safe Speculation Designs	15
4	Debugging Speculative Execution	17
4.1	SPECULATOR	18
4.1.1	Performance Monitor Capabilities	18
4.1.2	Objectives	19
4.1.3	Design and Implementation	19
4.1.4	Triggering Speculative Execution	21
4.1.5	Speculative Execution Markers	21
4.2	Using SPECULATOR: Dissecting the microarchitectural world	23
4.2.1	Return Stack Buffer Size	23
4.2.2	Nesting Speculative Execution	25
4.2.3	Speculative execution across system calls	26
4.2.4	Flushing the Cache	26
4.2.5	Speculation window size	27
4.2.6	Stopping Speculative Execution	30
4.2.7	Executable Page Permission	30
4.2.8	Memory Protection Extensions	31
4.2.9	Issued vs. Executed μ ops	31
4.3	Using SPECULATOR: Analyzing Attacks and Mitigations	32
4.3.1	SPLIT SPECTRE	32
4.3.2	BTI	33
4.3.3	Mitigations	33
4.3.4	Out-of-order execution bandwidth	34
4.4	SPLIT SPECTRE	35
4.4.1	The SplitSpectre Gadget	36
4.4.2	The Analysis	38
4.5	New microarchitectural side-channels	41
4.5.1	Icache Attack	42
4.5.2	Icache Discussion	42
4.5.3	Double BTI Attack	44
4.5.4	Practical considerations	47
4.6	Gadgets analysis	48
4.6.1	Icache Attack	48
4.6.2	Double BTI Attack	49
4.7	Mitigations	51

5	Impact of Spectre	53
5.1	Speculative execution attacks on memory safety mechanisms	54
5.1.1	SPEAR attacks	56
5.1.2	Speculation window and eviction	60
5.1.3	Speculative ROP	61
5.2	Case studies	61
5.2.1	Attacking stack canaries	62
5.2.2	Attacking CFI	68
5.2.3	Attacking memory safe languages	69
5.2.4	SPEAR attack against Rust bounds checking	73
5.3	Mitigations against SPEAR	75
5.3.1	Mitigations for SSP	75
5.3.2	Mitigations for the Go compiler	76
5.3.3	Mitigations for GCC VTV	78
5.4	Discussion on SPEAR	79
5.5	Testing Tools	81
5.5.1	Information gathering tools	81
5.5.2	Empirical tools	81
5.6	Methodology	82
5.6.1	Use Cases	83
5.6.2	Systems and Platforms	83
5.7	Testing Tools Analysis	84
5.7.1	Tools comparison	87
5.7.2	Analysis	89
5.8	Recommendations	90
5.8.1	Limit cache noise	91
5.8.2	Define the right use case and understand your threat model	92
5.8.3	Understanding information gathering tools results	92
5.8.4	Use a mixed approach	93
5.8.5	Static analysis	93
5.9	GhostBuster	94
6	Future Work	97
7	Papers	99
7.1	Related Publications	99
7.2	Other work	99
7.2.1	Lava: Large-scale automated vulnerability addition [1]	100
7.2.2	HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing [2]	100
7.2.3	Trellis: Privilege separation for multi-user applications made easy [3]	100
7.2.4	HONEYBUG: hypervisor-based approach for gathering attacker insights	101
8	Conclusions	103

List of Figures

4.1	The architecture of SPECULATOR. A template with the speculative execution trigger and a list of instructions to be speculatively executed are the input to the code generation. The code snippets are run repeatedly under supervision of the speculator monitor, which captures the event specified in the configuration file. Finally, the measurements are post-processed to present a final report on speculative execution behavior.	20
4.2	Flow chart of one of the experiment template used in SPECULATOR. The setup code brings the branch predictor in a specific state that will cause the later branch to mispredict and speculatively execute the code snippet consisting of the instructions. The speculative execution of the instructions is measured by the PMC infrastructure, which is triggered by the corresponding start/stop instructions indicated in the flow chart.	22
4.3	Return Stack Buffer test on Kabylake.	25
4.4	Return Stack Buffer test on AMD Ryzen.	25
4.5	Speculation window of a store-to-load forward failure, measured in executed <i>FNOPs</i> on Broadwell.	30
4.6	a) Speculative execution after an MPX bounds violation. b) Performance counter numbers for an increasing number of speculatively executed relative load instructions. The graph shows that the number of issued instructions corresponds to the number of executed instructions, justifying the use of such instructions as markers.	32
4.7	Reorder buffer size test results on Broadwell and Skylake. Since the marker instruction is no longer executed for a sufficiently large number of <i>NOPs</i> , the number of executed μ ops drops at the size of the reorder buffer.	35
4.8	RSB test on Broadwell. As in the AMD case, Broadwell is able to predict the location of my target even if the RSB is empty.	36
4.9	A comparison of regular Spectre v1 and SPLIT SPECTRE. While SPLIT SPECTRE only requires a simple array access, the speculation window needs to be sufficiently large to contain both the gadget and the second array access exercised by the attacker.	37
4.10	A conceptual view of a SPLIT SPECTRE attack instance with JavaScript.	38

4.11	An examination of the SPLITSPECTRE execution trace between the length check of <i>string.charCodeAt_impl()</i> and the second array access using SPECULATOR. The graph shows my results of the test on a Coffee Lake machine. It shows that, on average, I am not reaching the second array access in speculative execution. The small spikes in the graph are caused by mispredicted branches in the trace itself, which lead to nested speculative execution of fast-executing code paths.	39
4.12	Overview of Spectre v2, a SpCFH attack: the attacker performs BTI at first; the victim speculatively executes the injected gadget whose cache side effects are later measured by the attacker.	41
4.13	Description of the icache attack: the attacker performs BTI at first; the victim speculatively executes one of two functions depending on the content of a register; the attacker later times the execution of either function to learn one bit of the condition register.	43
4.14	Description of the Double BTI attack: the attacker performs BTI at first; the victim speculatively executes the “reverse” BTI Gadget that further trains the branch predictor with the value of a register or a memory location; the attacker later execute the same “reverse” BTI Gadget and based on the side effects of wrong prediction (e.g., executing an instruction marker to a given location) can guess the value of the register or memory location	45
4.15	<i>side-channel-receive</i> approach using data cache access pattern	47
4.16	Double BTI attack success rate on leaking a one byte of secret	50
5.1	Overview of speculative attack against memory safety mechanisms.	55
5.2	Overview of various Speculative control flow hijacking attacks.	56
5.3	Overhead computed as normalized runtime over SSP Disabled baseline.	77
5.4	Empirical CDF of the logarithm of the overhead percentage for the considered mitigations. Overhead data is gathered by running the full set of benchmarks of the Go runtime version 1.12.0.	79
5.5	GHOSTBUSTER’s overview. GHOSTBUSTER leverages SPECTRE-MELTDOWN-CHECKER and my modified version of SPECULATOR to assess the system security using both known methodologies, <i>gathering</i> and <i>empirical</i> . Then, it aggregates the results in a final report factoring in also the various use cases I identified to give a more accurate picture to the user. With solid circles I describe the major components of GHOSTBUSTER while with dotted circles I highlight operations performed.	91

List of Tables

4.1	The CPUs per architecture I use SPECULATOR on. While Haswell and Skylake are new designs – “ticks” in Intel nomenclature – Broadwell is a “tick”, a die-shrink of Haswell. Kaby and Coffee Lake are instead optimized versions of Skylake design within the same die size	24
4.2	Speculation window of a conditional branch depending on the type of instructions needed to resolve the branch as well as the placement of the value involved in the condition, measured in cycles.	29
4.3	Speculation window of an indirect control flow transfer, measured in cycles. The speculation window size depends on where the target of the indirect control flow transfer is stored.	29
4.4	Success rates for the SPLIT SPECTRE attack on JavaScript. I perform 100 runs, each run trying to leak a string of 10 consecutive characters. I provide numbers on both the highest and the second highest scoring characters.	40
4.5	icache attack experiment with a gadget from <i>libhttp-parser.so</i> : each row displays the success rate in guessing the value of the victim’s secret. The success rate is computed as the rate between samples displaying an icache hit (resp. miss) when the value of the victim’s secret is 0 (resp. 1). An icache hit is defined as an execution of the icache gadget timed below a pre-determined threshold.	49
4.6	Default STIBP settings in the kernel used by the distributions tested in my evaluation	51
5.1	Success rate (in percentage) computed over 1000 iterations for architectural or speculative overwrites of backward and forward edges performed on various architectures families.	59
5.2	ROP gadgets used for building Spectre v1 chain with their corresponding occurrences. The search space is a subset of <i>libc</i> , <i>libpng</i> , <i>libz</i> and <i>ld</i> executable pages, obtained by filtering out pages unmapped in the victim’s address space and pages without a valid TLB mapping.	65
5.3	CPU families the tools have been tested with the corresponding kernel version . .	83
5.4	Major pitfalls and limitations observed for each tool. I indicate with ✓ that the pitfall is present, whereas I leave blank otherwise.	85

5.5	Classification of the result types for each of the attacks with respect to the use cases described in Section 5.6.1. Empirical tools do not focus on specific use cases but rather on the existence of the attack vector. The table reflects this by referring the use case as synthetic. Results are reported as: ✓ if the tool reports information about a certain attack within the use case considered; ⇒ if the information can be inferred but it is not directly reported; ✗ where either no information can be inferred from the tool; finally the cell is left blank where the attack cannot be performed or it is not feasible under the specific use case.	86
5.6	Tools version used in the experiments	88
5.7	List of the 17 cloud provider tested and their available configurations	89

Acknowledgments

Some people might think that a PhD dissertation like this one is simply the result of the author's hard work throughout several years of research. While this is partially true, there is always way more behind the scene. I strongly believe that my work is also the outcome of the environment I had the luck to live in. The environment mostly being all the great people I had the pleasure to meet and learn from in my life. Here, I would like to try to acknowledge each and everyone of these people.

First and foremost, I would like to thank my parents, my mom Patrizia and my dad Angelo that through their hard work and amazing guidance allowed me to pursue my dreams and achieve all the things I had without any pressure or any constraint. I would like also to thank my sister Emanuela, my brother Marco, my brother-in-law Dario, all my cousins, and all my grandparents Caterina, and the late Alice, Giovanni and Federico for always being there supporting me. A special thanks also goes to Anna, Carolina, Danilo, and Vittorio for being my second family since many years and that, despite everything, never really left my side.

Beyond my family, I had the extreme luck to find on my path great advisors that directed me all the way here. Above all, I would like to thank my PhD advisor Engin Kirda which accepted me in his group and since 2014 has been advising and supporting me, and showed me the ropes of how to be a good researcher. Also, thanks for the great Netflix suggestions!!! ;-).

During my years of studies in Milano, I had the pleasure to encounter on my path Federico Maggi, Stefano Zanero, Marco Domenico Santambrogio, Davide Basilio Bartolini, Filippo Sironi, Francesco Trovò and Alessandro Barenghi that put me in contact with the security and the research worlds, and pushed me to pursuit a PhD.

From my time in Boston, I would like to thank Michael Weissbacher, Patrick Carter, Collin Mulliner, Tobias Lauinger, Amin Kharraz, Abdelberi Chaabane, Kaan Onarlioglu, Walter Rweyemamu, Andrew Fasano, Arash Molavi, Konstantinos Athanasiou, Christina Dimovasili, Andreas ten Pas, Sammie Katt, Molly Ohman, Yorgos Zirdelis, Yorgos Efthymiadis, William Lee, Domien Schepers, Andrea Baisero, Ahmad Bashir, William Blair, Manuel Egele, Wil Robertson, Joshua Bundt, Tim Leek, Erik-Oliver Blass, Guevara Noubir, Piotr Sapieżyński, Luciana Kiffer, Dimitris Tsipras, Raul Quadra, Daianne Maia, Claudia Marino, and Claudia Tomsa. Everyone of you made me feel at home and enjoy each and everyone of the days I spent in the US.

This dissertation would not have been possible without my amazing friends, colleagues, and mentors from IBM Research Anil Kurmus, Alessandro Sorniotti, Matthias Neugschwandtner, and Alexandra Sandulescu. To many more papers together! Also, from the big IBM Research family I would like to thank Chrysa Stathakopoulou, Kaoutar Elkhyaoui, Angelo De Caro, Marcus Brandenburger, Marc Stoecklin, and Ellie Androulaki for making IBM Research such a special place.

From my time in Zürich, I would like to thank Teri Andos, Christina Haupt, Valentina Marchionni, Davide Basilio Bartolini (again), Carlo Ciaccia, Kostantinos Tsoukalas, Yashashwa Pandey. If there was someone that could have made my decision to move back to Europe easier, it was you.

My passion for computers goes back all the way to my first Commodore64 we first got in '92/93. During those very early years of my life, through my father's passion for technology and electronics, I had the pleasure to come in contact with two very special people, the late Daniele Rizzo and Davide Butti. The first two hackers I met in my life. From them I learned, directly and indirectly, more than they can ever imagine.

This dissertation and the rest of the work I generally do is filled with C code or assembly language. Most people try to avoid such languages because tedious and painful to use. At the contrary, I fell in love with them during my early programming experience in High School, and never wanted to leave them ever since. A big part of my love for low level languages and therefore low level system security comes from my excellent High School teacher Virginia Bacchini which with extreme clarity and passion start me with programming and her lessons stayed with me all the way until now.

Chapter 1

Introduction

In early 2018, the release of the Spectre [4, 5] and Meltdown [6] vulnerabilities deeply shook the system security world. These new transient execution vulnerabilities are differently rooted than the very well studied and understood memory corruption vulnerabilities. They exploit side effects caused by modern CPU performance optimizations (e.g., *out-of-order* and *speculative* execution) and not bugs within a code base. Transient execution vulnerabilities concern the confidentiality of a program and not its integrity. Generally, whenever the CPU guesses incorrectly during the application of one of these performance optimizations, it needs to roll-back the micro-architecture status and squash the transient instructions involved. The problem that transient execution attacks leverage is that the status of the micro-architecture is never fully restored to the point before the guess has taken place. This incomplete operation leaves some visible side effects that are function of the accessed data and the code that was mistakenly executed. An attacker using one of the many side/cover channels available can reconstruct possible secrets from those side-effects.

One of the most prominent transient execution attacks is Meltdown. It rely on the fact that due to *out-of-order* execution the CPU does not stop executing in case of a fault until the faulty instruction retires. This has been shown to allow information leakage between different privilege levels (e.g., user and kernel space) that the fault should have prevented. Meltdown and its variants are powerful and relatively easy to carry out. However, they are also easy to mitigate in software [7] (arguably with high overhead), or to fix in hardware in future CPUs by verifying the fault during speculation and not at instruction retirement. Intel has released fixes in its 9 series of CPUs while AMD was never vulnerable to this type of attack.

Another notorious transient execution attack is Spectre. Spectre and its variants do not target a bug within the CPU as Meltdown but code patterns that can cause speculative execution to mi-speculate towards sensitive code that can leak information through covert channels. Given the nature of Spectre-like attacks, they are hard to efficiently mitigate and therefore, as it has been for the buffer overflow, they will not go away anytime soon. Spectre attacks affect any CPU that supports speculative execution. Mitigations against Spectre attacks are available but due to their high overhead are generally not applied, or left disabled by default.

In general, studying such new vulnerabilities present several challenges. The lack of details of the CPU micro-architecture requires extensive and tedious reverse engineering of all the internal components and their behavior in the context of out-of-order and speculative execution. This process is made even more challenging due to the lack of tooling to directly control the micro-architecture internal components leaving the researchers with just the noisy covert channel option

to verify their findings. Finally, even within the same hardware manufacturer the behavior of each CPU changes across families due to internal design changes. This forces the reverse engineering effort to be repeated across several CPU models.

Transient execution attacks are able to break the process to process isolation that is part of the security guarantees the CPU provides at the operating system. Process to process isolation is one of the building blocks for modern defense mechanisms and its breach is not accounted in many existing threat models. The ability of Spectre and Meltdown to bypass the isolation has forced the security community to revise their security threat models to verify they still hold in the new Spectre era. However, even simply understanding if a system is actually protected against Spectre and Meltdown is rather complicated. Few tools are available but correctly understanding their results can be confusing and misleading for system operators. An interpretation mistake can cause either a false sense of security or, huge performance penalties even when there is no actual threat. Improvements are necessary in this area to enable system administrators and security researchers to obtain a better picture of the system status.

Thesis Statement. *It is fundamental to have a deterministic view on transient execution to achieve fast and precise micro-architecture reverse engineering that is the basis for exploring new attack techniques, and analyze and improve defense mechanisms.*

For the purpose of my thesis, I propose to enhance the state of the art in the following areas:

Speculative execution analysis: The first step towards understanding the extent of these attacks requires the ability to reliably observe the speculative execution side-effects within the micro-architecture. While for memory corruption vulnerabilities there are several tools like *gdb* that allow to investigate program crashes, nothing is currently available to analyze speculative execution and inside the micro-architecture. Related work rely on the very noisy covert channels to study these attacks and do not decouple any of the phases the attack is composed making the attacks hard to analyze in details. In essence, using covert channels adds another source of non-determinism besides the never fully controllable predictors the attack is leveraging. Attributing the cause of a failed attack observation to either problems in training the predictor, or in using covert channel is often impossible. Having a deterministic way of observing the side-effects of mispeculation greatly reduces the complexity on analyzing attacks by limiting the possible root causes of failure. In this area, I provide the first methodology that allows to observe the speculative execution side-effects and to dissect each phase of the attack independently. This enables a debug-like approach and helps to clarify the role of each of the phases of the attack in the overall success and feasibility. I integrate this technique in a novel tool that I then use to investigate several micro-architectural behavior as well as a new Spectre variant called Split Spectre and two new side channels gadgets to build the covert channel.

Understanding Spectre’s impact: In this dissertation, I investigate in which way the security threat models are impacted by the discovery of transient execution vulnerabilities. First, I analyze common defenses against memory corruption vulnerabilities (e.g., CFI or SSP) and show how, while defending from memory corruption vulnerabilities, they extend the attack surface in the transient execution context. I show how these defenses require a re-design

in the Spectre era. Furthermore, I provide the first comparison of the available transient execution vulnerability checkers and their underlining methodologies. I describe the pitfalls all the available tools have and I propose a new guideline for building checkers to prevent common mistakes. Finally, I introduce a new tool that builds upon related work to overcome the individually identified pitfalls.

Chapter 2

Background

2.1 Microarchitecture

The microarchitecture can be generally refer to as a specific implementation of an Instruction Set Architecture (ISA). The ISA can be viewed as an high level description of a computer architecture. For instance, *x86-64* is one of the most popular ISA used today and can be found under two different microarchitectural implementations, one provided by Intel and the other provided by AMD. Based on the ISA that is aiming to implement, a modern microarchitecture contains a certain number of registers, a caching hierarchy, one or several communication busses, a set of decoding and executions units related to the instructions that are expected to be executed, and a series of components and optimization designs that aim to boost its performance. Instances of such performance enhancer optimizations can be branch predictor units, out-of-order execution, or speculative execution.

Today's microarchitectures try to optimize every clock cycles by applying extensively optimizations. Most of the details of their exact implementation are kept secret to preserve the IP. However, most of the major components of the microarchitecture are well understood and I will provide a brief description in the following sections.

2.1.1 Pipeline

Pipelined CPUs divide the life of an instruction within the microarchitecture into stages. While an instruction moves from one stage to the next, it opens up the possibility for a new instruction to start using the stage left empty. This allows the CPU to maximize the use of each stage of the pipeline without waiting for a single instruction to cross all the stages. This type of microarchitectural design allows to achieve instruction-level parallelism in a single processor. The number of stages inside the pipeline hugely depends on the CPU designs. It can variate from 2 to more than 30 stages.

2.1.2 Cache

Another optimization that can be found in modern microarchitecture design is the cache. The cache is a very fast piece of memory that is placed closer to the execution units so that most-used data does not need to be retrieved always from main memory. In today's CPUs there are several

layers of caches some of which are shared across all cores on the system and others that are instead private of a specific core. Each cache hierarchy implementation differ in the number of levels, dimensions, addressing technique and policies under which they operate. In the context of cache side channels attacks, it is critical to understand how the cache hierarchy works under the microarchitecture that is target of the attack.

2.1.3 Branch Prediction

Among the most common instructions that can be found in most programs there are branch or jump instructions, which role is to transfer the control flow to an arbitrary location instead of the next instruction in the flow. To optimize the pipeline, the CPU needs to know ahead of time which will be the next instruction to fetch. While this is a relatively easy task when dealing with direct jump/branch instructions, it is a very complex operation when the CPU needs to predict indirect branches in which the target is computed at runtime and it might change at each execution. To solve this issue, modern microarchitectures employ branch predictor units that are complex machinery which, by looking at the history of a specific branch, try to predict the next possible target for a specific indirect branch. An example of a modern branch predictor unit is TAGE [8] that is believed to be the based of many today's predictors.

2.1.4 Out-of-order Execution

The idea behind out-of-order execution is to allow stream of instructions, that are data independent, to be executed outside the original order in which the program was written. This optimization kicks in whenever an instruction causes a stall while waiting for data from main memory. In many situations, it is possible that downstream some instructions are already ready to go into execution because they do not depend on the stalled instruction. The CPU starts executing those instructions in parallel to keep the pipeline utilized. However, after execution these most recent instructions will be waiting for the stalled instruction to retire to officially be committed at the architectural level. This of course is done by the CPU to maintain the semantic of the program.

2.1.5 Speculative Execution

Speculative execution is one of the most critical optimizations of modern CPUs. There are several situations in which the microarchitecture might require to stall due to events such as a slow branch resolution. While with out-of-order execution the pipeline is filled with an independent stream of instructions that are found downstream, during speculative execution the CPU tries to guess which is the outcome of the stalled instruction and based on that guess it will start to execute dependent instructions to the one that has stalled. By doing so, the instruction stream can continue in the hope that the guess done by the CPU by using one of the many predictors available is correct. If the guess was not correct, the instructions executed are squashed and the CPU will restart the execution with the correct result. In the case in which the guess is correct thought, the CPU has saved several CPU cycles by not waiting the resolution of the stalled instruction.

2.1.6 Multiprocessing and multithreading

In the early stages of computing, most of the performance was obtained by raising the clock frequency of the CPU. However, due to physical limitations such as heat dissipation, since the late 90s two new techniques are introduced to obtain instruction level parallelism: multiprocessing and multithreading. In the first, the microarchitecture is equipped with multiple cores that are able to run more processes at the same time. In the second, each core is able to run different threads of the same program in parallel which allows to explicitly exploit the parallelism of several operations that would be otherwise sequentially executed within the context of a single program. A key difference between multiprocessing and multithreading is that in the case of multithreading the two threads might share several internal structures of the same core while that is not the case for two processes running on different cores.

2.2 Transient execution attacks

Transient execution attacks exploit a new class of vulnerabilities, targeting a particular microarchitectural CPU design with specially crafted software. These attacks leverage known attack vectors such as side channels, but go much further by combining them with vulnerabilities at the microarchitectural level. Numerous variants of transient execution attacks have been disclosed since the beginning of 2018.

Transient execution attacks are commonly divided into two major families: *Speculation-based* and *Fault-based* [9]. The *Speculation-based* family includes the various Spectre variants that leverage speculative execution to achieve data exfiltration. I refer to this family of attacks as *Speculative Execution Attacks*. The *Fault-based* family instead, comprises of all the Meltdown variants that rely on bugs in the way the CPU handles faults and out-of-order execution to achieve similar results.

2.2.1 Fault-based attacks

The Meltdown family of attacks exploits bugs within the CPU. During a Meltdown attack, the attacker tries to perform operations speculatively that are not allowed due to privilege boundaries. Meltdown relies on the fact that faults are handled by the CPU only when the faulty instruction retires, leaving the out-of-order execution to continue across privilege boundaries before the fault is registered. This can allow an attacker to create a cache side channel with data retrieved from a higher privilege level through out-of-order execution. Each variant of Meltdown exploits a different fault type.

While these attacks are powerful and relatively easy to perform, they are easily fixable and meant to disappear in future iterations of the CPU, i.e., they are less interesting from the research point of view.

2.2.2 Speculation-based attacks

In modern CPUs, speculative execution is employed in several situations to boost performance and avoid bottlenecks in the execution pipeline. For instance, whenever a conditional branch is encountered in the instruction stream and one of the operands is not readily available, the CPU

speculates the result of the conditional branch and continue to execute further instructions. When the uncached operand is finally loaded from the main memory and the conditional branch can resolve, the CPU verifies the correctness of the guess and either commit and retires the speculated instructions, if the guess was correct, or rollback otherwise. Similarly, speculative execution is also triggered when the destination of an indirect call is not cached.

Spectre [4, 10] was the first to show that whenever a mis-speculation takes place, the CPU might execute code that accesses sensitive data leaving side-effects in the micro-architecture. Such side-effects, that before were considered not accessible from outside the micro-architecture, can actually be observed through a side-channel. A critical factor for the feasibility of these attacks is that the attacker can force such mis-speculation through training of the particular predictor unit that is involved in the attack.

Speculative execution attacks can be further classified according to where the training of a predictor occurs. In particular, I define the following relevant configurations: *i) same address-space (sAS)* where training occurs in the same address space as that of the victim process, or *ii) cross address-space with simultaneous multi-threading (cHT)* where training occurs in a separate process running on the same physical core, or *iii) cross address-space without simultaneous multi-threading (cAS)* where training occurs between two processes running interleaved on the same physical core. Note that the cHT setting is a setting where attacker training and victim speculation occur in temporal colocation (different logical core, same time), whereas in the cAS setting they occur in spatial colocation (same logical core, different time). These settings are important because several attacks and mitigations rely on them. Thus, considerations on the specific setting must be taken into account in the threat model when evaluating the security posture of a system.

2.2.3 Speculative Execution Attacks Phases

Speculative execution attacks can be decomposed into the following five distinct phases:

- I) **Prepare side channel:** In this phase, the CPU performs operations that will increase the chances of the attack succeeding. For instance, the attacker can prime caches to prepare for a prime-and-probe [11] cache side channel measurement, make sure important target data is flushed, or ensure that the attacking thread and victim thread are co-located.
- II) **Prepare speculative execution:** In this phase, the CPU executes code that will allow speculative execution to start. This is code that is typically executed within the context of the victim.
- III) **Speculative execution start:** In this phase, the CPU executes an instruction whose outcome decides the next instruction to be executed, such as a conditional branch instruction. Between the time window where this instruction is issued and when it is retired, modern CPUs guess the outcome of the branch to avoid stalling the pipeline, and execute code speculatively. This is known as speculative execution [12].
- IV) **Speculative execution, side channel send:** In this phase, the CPU executes (but does not retire) instructions that will result in a micro-architectural state change.

- V) **Side channel receive:** In this phase, the CPU executes instructions that transform the micro-architectural state change that occurred in the previous step into an architectural state change.

2.3 Privilege boundaries and attack impact

The core element that turns transient execution into an attack is the breach of a privilege boundary that is established through hardware isolation support by the CPU. These privilege boundaries typically aim to provide confidentiality and integrity of the data residing within the boundary (i.e., preventing data from being read or modified directly from outside the boundary). All accesses to such data are mediated by code running within the privilege boundary, and that code may only be invoked from a lower privilege through well-defined entry points.

In the case of currently known speculative execution attacks, the attacker's aim is limited to breach confidentiality of data residing beyond the privilege boundary by either accessing arbitrary data or leaking specific metadata, such as pointer values, of the running program. For instance, privilege boundaries that can be bypassed by some known speculative execution attacks are:

- kernel vs. user-mode code
- hardware enclave (SGX) vs. user-mode or kernel-mode code
- sandboxed code in the same process, for example JavaScript JIT code
- processes-to-process boundary
- remote node to local node boundary

I note that code at each speculative execution attack phase previously described (Section 2.2.3) can potentially be run either in the higher privileged mode (victim-provided code) or lower privileged one (attacker-provided code).

2.4 Defenses

Several mitigations have been developed to protect against transient execution attacks. Some of these mitigations repurpose existing instructions or sequence of instructions to block speculative execution in sensitive areas of the code. To this category belong memory fencing instructions (e.g., *lfence*), branchless masking [13], and Retpoline [14]. Others, instead, are new hardware features that hardware vendors introduced in new iterations of the CPU. Changes are also made at the operating system level, including the re-design of entire subsystems.

2.4.1 Memory Fencing

Through the application of serialization instructions, such as *lfence* on Intel, it is possible to force the CPU pipeline to wait for prior instructions to retire and, as a consequence, to block speculation and related Spectre attacks. Such a pipeline interruption is an expensive operation, therefore fencing instruction should be placed carefully either manually or at compile time only where really needed. For instance, the Linux kernel uses manually instrumented code, if the Spectre mitigations are enabled.

2.4.2 Branchless masking

Branchless masking is a mitigation against Spectre-PHT attacks to harden load instructions that are gated by a condition. The technique involves introducing a *data* dependency (usually called mask) on the condition through a set of instructions which set the mask to zero in case the condition is false (and to unsigned negative one otherwise). The mask is then used to zero out pointers or array indices before performing the load when invalid. Masking is for example used in the Linux kernel [15] to block Spectre-PHT whenever a value coming from userspace is used as an index for an array access. It is also available as a compiler option (Speculative Load Hardening (SLH) [16]) to instrument conditional branches with control-flow dependent pointer masking.

2.4.3 Retpoline

As an answer to Spectre-BTB, the Retpoline [17] compile-time mitigation replaces indirect branches with *ret* instructions to prevent branch poisoning. This method ensures that return instructions always speculate into an endless loop through the RSB.

2.4.4 KPTI

KPTI mitigates Meltdown-US and fortifies KASLR. KPTI is based on KAISER [9] (short for *Kernel Address Isolation to have Side-channels Efficiently Removed*). If KPTI is enabled, whenever user-space code is running, Linux ensures that only the kernel memory pages required to enter and exit syscalls, interrupts and exceptions are mapped. With no other pages mapped, KPTI prevents the use of kernel virtual addresses from user-space because they cannot be correctly translated.

2.4.5 Indirect Branch Restricted Speculation

Indirect Branch Restricted Speculation (IBRS) [18] prevents indirect branch predictors executed in privileged code from being trained by less privileged code (i.e kernel-space cannot be influenced by user-space). This also includes the case of attacks from another logical core on the same physical core (cHT).

2.4.6 Indirect Branch Predictor Barrier

Indirect Branch Predictor Barrier (IBPB) [19] prevents code that executes before it from affecting branch prediction for code that executes after. When enabled, an IBPB barrier runs across user mode or guest mode context switches. In this way, a different user cannot attack a process of another user running on the same machine. On Linux machines, IBPB can be conditionally or fully enabled: in the first case, the barrier is raised only when switching to processes that request it using *seccomp* or *prctl*.

2.4.7 Single thread indirect Branch Predictors

Single Thread Indirect Branch Predictors (STIBP) [20] splits the branch predictor across sibling threads of a core, removing the attack vector constituted by a process training the predictor of a co-located victim process. This prevents attacks like Spectre-BTB in the cHT setting.

2.4.8 RSB filling

RSB filling blocks Spectre-BTB and Spectre-RSB in the cAS setting. It flushes the RSB whenever the CPU switches across privilege levels. For instance when the CPU switches from usermode to kernel mode the RSB might contain poisoned entries introduced by the attacker which might affect its speculative control flow. The process of RSB filling removes any such entry.

2.4.9 SSB mitigation

When software-based mitigations are not feasible (such as inserting an *lfence* instruction between the store and the load instructions) for Spectre-STL, some CPUs support Speculate Store Bypass Disable that can be used to mitigate speculative store bypass. When SSBD is set, loads will not execute speculatively until the address of older stores are known.

2.4.10 PTE inversion

PTE inversion is used to block Meltdown-P attacks. When a page table entry points to a non present page, the upper address bits are inverted so that an access to such entry resolves to uncacheable memory access. Given that Meltdown-P can only exfiltrate data from L1 cache, forcing the address translation to return an uncacheable address closes the attack vector.

2.4.11 VMC conditional

VMC conditional cache flushing is adopted on virtualized environments. The mitigation flushes the L1 cache at every VMENTER instruction. This way secrets possibly stored in the cache are no longer accessible via the Foreshadow-VMM attack.

Chapter 3

Related Work

3.1 Speculative Execution

Optimizing CPU instruction throughput through speculative execution has been extensively analyzed and implemented in the 1990s [21, 12, 22]. For information about the microarchitecture of CPUs with respect to out-of-order and speculative execution, I mostly have to rely on the material provided by the CPU manufacturers [23, 24, 25]. Unfortunately this material often just provides software performance optimization related aspects, not providing details on how mechanisms such as the branch predictor work. Agner Fog’s work [26] sheds light on those details, providing detailed information backed by a substantial amount of experimental research on the microarchitectural aspects of CPUs. This information is leveraged in processor simulators such as *gem5* [27].

3.2 Cache Side Channels

Many speculative execution attacks variants rely on cache side channels to infer the memory contents accessed by speculative execution. Cache side channels have been extensively studied: First, Tromer et al. introduced both the “evict-and-time” and “prime-and-probe” techniques to efficiently perform a cache attack on AES [11]. Prime and probe is a popular technique, which was also used for certain speculative execution attacks variants. “Flush-and-reload” [28] is a technique that allows for higher precision and is used in NetSpectre. Recently, other techniques such as “flush-and-flush” [29] and “prime-and-abort” [30] were presented. Flush and flush leverages the fact that *clflush* executes faster in case of a cache hit. Prime and abort makes use of Intel’s transactional memory mechanism to detect when an eviction has happened without the need to probe the cache.

3.3 Speculative Execution Attacks

Speculative execution attacks comprise those leveraging microarchitectural components such as the Pattern History Table (PHT) for Spectre v1 [10], the Branch Target Buffer for Spectre v2, the Return Stack Buffer (RSB) for Ret2Spec [31] and Spectre returns [32]. Both BTB and RSB attacks

are cases of speculative control flow hijacks, i.e., they provide the ability for an attacker to steer speculative execution to an arbitrary location. Varied and powerful attacks leveraging the BTB for speculative control flow hijacks have been demonstrated, in combination with port contention-based, instruction cache-based, or BTB-based side channels [33, 34]. In Spectre v1.1 [35], Kiriansky and Waldspurger point out that speculative overwrites of backward edges lead to speculative control flow hijacks.

In practice, BTB gadgets are hard to find, thus attacks have only been shown to be practical if the gadget is injected (e.g., by loading attacker-controlled eBPF bytecode into the kernel). The idea of chaining speculative gadgets in a way similar to ROP was suggested shortly after the first publication of Spectre attacks. Some publications have referred to the same idea [35, 33], the former only briefly mentioning speculative ROP attacks but practical aspects are neither discussed nor experimented on.

Netspectre [36] introduces a victim data eviction technique based on coarse-grained cache eviction. The method, *Thrash+Reload*, is a remote variant of *Evict+Reload* [37]. The attacker starts a large file download from the victim via a network interface. On the victim’s side, this action results in victim data eviction with a probability which depends on the file size. *Thrash+Reload* applicability is limited to scenarios where cache thrashing does not compromise the attack.

3.4 Mitigations

Since the first speculative execution attacks have been disclosed in early 2018, different mitigations have been proposed to prevent each variant. Some mitigations are introduced at hardware level meanwhile others are software-based. Many of these mitigations target Spectre v2 type of attacks, meanwhile no software-transparent mitigation has been introduced for Spectre v1.

The available software-based Spectre v1 mitigations consist in either deploying a serializing instruction (e.g., *lfence*) around each sensitive bounds check or, alternatively, masking the index used for accessing arrays [13, 38, 35, 39].

While *lfence* is an effective mitigation, it incurs huge performance penalties if widely applied. Static analysis tools have been proposed to search for sensitive code patterns. One example is the Linux kernel where vulnerable code is instrumented on a case by case basis either through manual audit or automatic tools (e.g., *smatch* [40]) detection [41]. The drawback of current available tools is that they target Spectre v1 code patterns such as array-out-of-bounds cases only and therefore are not useful in the general memory corruption case (where an overwrite of a control-flow influencing value can occur for any other mispeculation). Due to the high overhead, big projects like JavaScript engines deployed alternative techniques against Spectre v1 such as diluting timing precision, disabling concurrent threads to prevent homebrew-timers and masking pointer accesses to prevent speculative out-of-bounds accesses [42, 43, 44].

For Spectre v2 instead, there are software and hardware mitigations. The software mitigation currently available is Retpoline [14]. This mitigation targets indirect calls and indirect jumps and prevents them from being speculatively executed by trapping speculation within a loop. As in the barrier cases for Spectre v1, Retpoline requires code modification and therefore each program has to be recompiled to enforce such mechanism. Linux has deployed Retpoline in the kernel as mitigation for Spectre v2.

On the hardware side, Intel published three major protections: *i*) Indirect Branch Restricted

Speculation (IBRS) [18], which prevents speculation of indirect branches using target values computed using lower privileged predictor modes, *ii*) Single Thread Indirect Branch Predictors (STIBP) [20], which prevents *Branch Target Buffer* (BTB) poisoning from sibling threads, and *iii*) Indirect Branch Predictor Barrier (IBPB) [19], which ensures that code before a barrier does not influence the behavior of the code after. IBRS and IBPB are meant to protect higher privileged code from lower privileged code. The only mitigation that provides protection within the same privilege level is STIBP, which is not enabled by default for performance reasons.

Furthermore, Intel announced as part of its Control Flow Enforcement (CET) extension, the future introduction of a new mitigation that will constrain the target of near indirect jumps and calls to only *ENDBRANCH* instructions. Based on the release specifications, these constraints should also apply during speculative execution. Therefore, this mitigation reduces the number of possible gadgets where speculative execution can be redirected to during branch target injection attacks. For SPEAR attacks, this mitigation applies for the forward edge overwrite case, where it should restrict possible speculative control flow hijack targets. For the backward edge case, Intel has implemented a shadow stack which, if adequately enforced during speculative execution, should stop all SPEAR backward edge overwrites.

3.5 Safe Speculation Designs

In addition to mitigations that aim to protect already existing systems, several new design proposals have been presented for future architectures to prevent speculative execution attacks.

A line of research concentrates on analyzing the data flow within the CPU pipeline and preventing unsafe operations from leaving observable effects upon misprediction. NDA [45] restricts speculative data propagation that follows an unresolved branch (potential control flow misprediction) or unresolved store address (potential memory dependence misprediction). STT [46] selectively forward secrets based on a speculative taint tracking system. Dolma [47] presents a lightweight speculative information flow scheme with secure performance optimizations. SPT [48] is based on the principles that a piece of data can be speculatively leaked if it was already leaked during non-speculative execution, otherwise it would delay the transmitter operation. GhostMinion [49] instead presents Strictness Ordering, a permissive constraint system that allows all the information flows where non-committing speculative operations cannot leak to those that do commit.

Another set of work, instead, proposes new cache designs. InvisiSpec [50] removes cache covert and side channels by confining Unsafe Speculative Loads (USL) into a speculative buffer until the USL is considered safe and the changes can be exposed to the cache hierarchy. In a similar fashion, CleanupSpec [51] prevents the cache side-effects, however, its strategy differs from InvisiSpec because it allows the USL to modify the cache. CleanupSpec applies an Undo operation only when misprediction is detected, therefore limiting performance overhead. Conditional Speculation [52] and Sakalis et al. [53] block during speculation memory accesses that do not hit the L1 cache, as the L1 accesses are safe. MIRAGE [54] instead introduces a randomized approach in selecting the eviction candidates to eradicate set-conflicts that lead to cache attacks. Finally, DAWG [55] proposes a mechanism to partition the caches into domains to provide isolation.

Chapter 4

Debugging Speculative Execution

A developer’s view of the CPU when writing a low-level program is defined by the CPU’s instruction set architecture (ISA). The ISA is a well-defined, stable interface the developer can use to access and change the architectural state of a CPU. The software is in full control over memory, registers, interrupts and I/O. At the same time, the CPU has a lower-level state of its own – the extra-architectural state of the microarchitecture, commonly referred to as the *microarchitectural state*. In general, the ISA provides no direct access to the CPU microarchitecture, allowing the microarchitecture to evolve independently while keeping the programming interface backward compatible. The microarchitecture of a CPU is subject to frequent changes between generations and models, and is different even among vendors of a given ISA. A CPU’s microarchitecture typically also implements security controls, such as process isolation.

Recent works [4, 5, 6] have shown how security controls can be bypassed by submitting carefully-crafted inputs at the level of the ISA interface. These attacks exploit undocumented behavior at the microarchitectural level, and have been discovered through reverse engineering and trial-and-error. The full breadth of this class of attacks is not entirely understood, owing to the fact that details about the microarchitectural level of modern commercial CPUs are not publicly available. The research community cannot provide complete answers to questions about the existence of new attacks and the effectiveness of defenses.

More precisely, I identify two important related requirements: 1. When developing new attacks, it is often required to analyze and debug parts of the proof-of-concept code easily. For memory corruption, this would be achieved with a debugger. An equivalent for speculative execution attacks, that inspects microarchitectural state directly, is needed. 2. When testing speculative execution mitigations, the current option is either to attempt a proof-of-concept attack, or to trust the CPU flags and kernel configuration that are provided. A more granular testing tool that directly inspects microarchitectural state would be beneficial to gain confidence in the mitigations being properly implemented and enabled.

In this work, I propose a tool, SPECULATOR, with these two requirements in mind. SPECULATOR records or infers microarchitectural behavior by using performance counters, supports incremental analysis (evolution of microarchitectural state over a code snippet), runs on both Intel and AMD CPUs, and enables concurrent execution (interaction of two threads in an SMT environment).

This work makes the following contributions:

- A new performance-counter-based method and tool, SPECULATOR, to aid in designing attacks and mitigations.

- Insights into microarchitectural behavior relevant to attacks and defenses: I successfully verify the return stack buffer size, that nested speculative execution works, that speculation does not span across system calls and that *clflush* has no effect during speculation. I also measure the window size for indirect branches, indirect control flow transfers and store to load forwards. Finally, I document the effects of page permissions, memory protection extension and special instructions (e.g. *lfence*) on speculative execution.
- Examples of using SPECULATOR against attacks and mitigations.
- An Icache attack proof-of-concept: uses the instruction cache as a side channel, as part of a BTI attack to leak one bit of information at a time from a victim program.
- A Double BTI attack proof-of-concept: uses the branch target buffer (BTB) as a side channel, as part of a BTI attack to leak one byte of information at a time.
- An Analysis of current branch target injection mitigations on Linux, showing that both attacks work on user space programs with default settings.

4.1 SPECULATOR

Speculative execution is not well-documented compared to other features of modern CPUs. Being part of the microarchitecture, its implementation details are hidden behind the ISA and subject to optimization, which manufacturers keep to themselves.

However, understanding the internals of speculative execution is key to comprehending the limits of Speculative Execution Attacks (SEAs), and to designing adequate mitigations and defenses against SEAs. For this reason, I have designed and implemented SPECULATOR, a tool whose purpose is to reverse-engineer the behavior of different CPUs in order to build a deeper understanding of speculative execution. SPECULATOR aggregates the relevant sources of information available to an observer of speculative execution, chief among them CPU performance counters and model-specific registers, so that the behavior of different code snippets can be observed from a speculative execution standpoint. In this section, I describe the design and implementation of SPECULATOR.

4.1.1 Performance Monitor Capabilities

Modern CPUs provide relevant information through the performance counter interface. This interface is offered by most manufacturers, and it exposes a set of registers (some fixed and some programmable) that can be used to retrieve information on various aspects of the execution. Through these registers, counters for events or duration related to microarchitectural state changes such as cache accesses, retired instructions, and mispredicted branches, are made available to the developer. Events are manufacturer- and architecture-specific. This interface was originally made available to provide a method for developers to improve the performance of their code. The interface is typically used as follows: through a setup step, developers can choose which events will be measured by programmable counters out of a wide set of supported ones. Measurements can be started and stopped programmatically in order to carefully control the events of which precise sequence of instructions is being measured. Setting up, starting, and stopping measurements often requires supervisor mode (ring 0 in x86 nomenclature) instructions, whereas accessing counters is usually available in user mode.

SPECULATOR builds on top of performance counters to observe the nature and effects of

speculative execution. One challenge with this approach is that the performance counters interface was not designed with this objective in mind. One of the contributions of this dissertation is the identification of effective ways of using the interface, and a useful set of counters to accurately infer the behavior of speculative execution.

4.1.2 Objectives

The main objective of SPECULATOR is to accurately measure microarchitectural state attributes associated to the speculative portion of the execution of user-supplied snippets of code. Accuracy refers to the degree with which the tool is capable of isolating the changes to the microarchitectural state caused by the snippet being analyzed from that of the tool itself and the rest of the system (e.g., the OS or other processes). An incomplete list of SPECULATOR observables are 1. which parts of the snippet are speculatively executed, 2. what causes speculative execution to start and stop, 3. what parameters affect the amount of speculative execution, 4. how do specific instructions affect the behavior of speculative execution, 5. which security boundaries are effective in the prevention of speculative execution, and 6. how consistently CPUs behave within the same architecture and across architectures and vendors. The creation of a new tool is justified because none of the existing ones, such as `perf_events` [56] or `Likwid` [57], provide the required information with sufficient accuracy.

More precisely, `perf_events` has two modes of operations, sampling and counting. During sampling, there is no way to have precise quantitative information about code execution, and therefore it is not suitable for my purpose. When evaluating `perf_events`' counting mode, I experienced for very small snippets a certain level of overhead (in the order of 500 μ ops). This overhead was caused by the `perf_event` design decision of integrating all its operations (e.g., start counters, stop counters) in the kernel. Since the test snippets are 20-30 instructions long on average, this overhead completely prevents inferring any kind of relevant behavior.

`Likwid` operates instead in user space just as SPECULATOR, instrumenting the counters through the MSR register. However, its design only allows system-wide measurements and does not provide the same flexibility of handling the counter as the snippet progresses in its execution.

I also considered other tools and libraries such as `Oprofile` [58], `Perfmon2` [59], `Perfctl` [60], and `PAPI` [61]. Unfortunately, all of these possess either the same issues of measure inaccuracy or lack of flexibility, or otherwise are outdated and unmaintained. Performance comparisons among some of these interfaces are provided by Zapanuks et al. [62] and Weaver [63].

Another SPECULATOR objective is to provide tooling for the generation and manipulation of code snippets. The ability to inspect individual snippets and snippet groups during speculative execution allows the user to focus on combinations of instructions that are relevant for specific use-cases. Additionally, support for multiple platforms enables the inference of generalizable facts about speculative execution.

4.1.3 Design and Implementation

Figure 4.1 describes the architecture of SPECULATOR and its three main components: a pre-processing unit, a runtime unit called the Monitor, and a post-processing unit.

The task of the pre-processing unit is to compile the provided input into the appropriate execution format, and to introduce the instrumentation required by the performance monitor

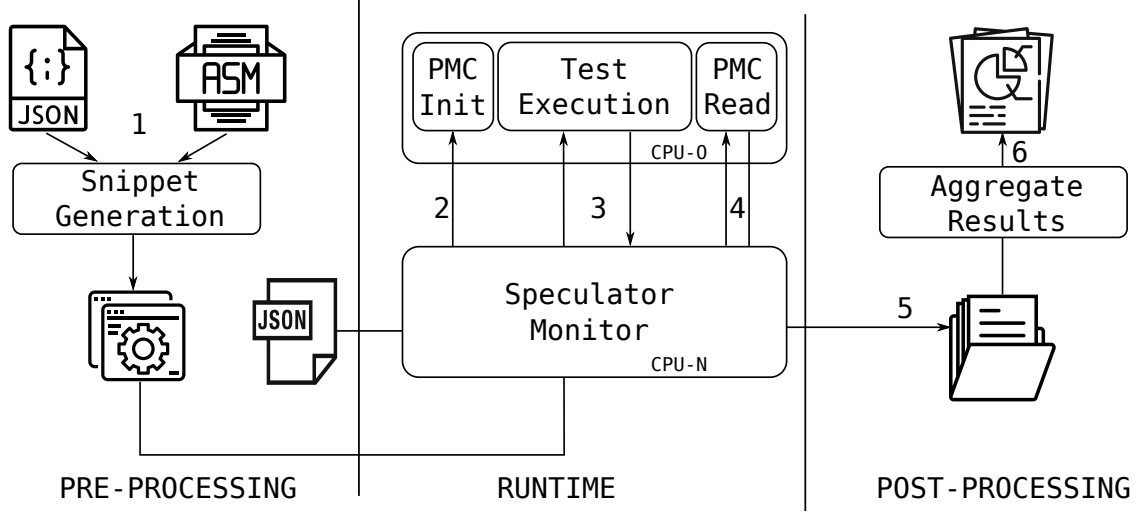


Figure 4.1: The architecture of SPECULATOR. A template with the speculative execution trigger and a list of instructions to be speculatively executed are the input to the code generation. The code snippets are run repeatedly under supervision of the speculator monitor, which captures the event specified in the configuration file. Finally, the measurements are post-processed to present a final report on speculative execution behavior.

interface to be able to observe the value of the selected set of hardware counters. Input can be provided as a snippet of C or assembly code, or as a template for the generation of code snippets. Code snippets are generated from templates in an incremental fashion, resulting in the output of multiple snippets with an increasing number of instructions taken from a pre-compiled JSON list. Each instruction is inserted by the SPECULATOR snippet generator in the specific location defined in the source template (Step 1 in Figure 4.1). The introduction of such “incremental” snippets is justified by the fact that the addition of a single assembly instruction may trigger optimizations that – while preserving the expected program semantics – alter the behavior of the CPU at a microarchitectural level and affect the nature of speculative execution. Having incremental snippets helps to verify when optimizations are triggered and take them into account during the analysis of the results.

After the generation of the executable (also referred to as the test application), the SPECULATOR runtime is invoked on each of the generated outputs (Step 3). To ensure that the Monitor does not perturb the measurements, the process executing the snippet and the monitor are pinned on different cores. The Monitor is responsible to configure the counters on the core used by the test application (Step 2). As previously mentioned, there are many programmable counters that can be used so I provide a configuration file that can be loaded into SPECULATOR to easily switch among them.

Once the Monitor has set up the environment, it loads and executes the snippet in a separate process, and waits for it to complete (Step 3). The test application prologue and epilogue will interact with the environment created by the Monitor, resetting, starting and stopping the counters as needed. The counters related to the core where the test application runs are stopped by the test application just before termination. When the test application terminates, the Monitor will be signaled by the Operating System. At this point, the Monitor can retrieve the values of the

counters from the core where the test application runs (Step 4) and store them in a result file (Step 5). The Monitor can be configured to run a specific test N times. In this case, the result file will contain the values of each run.

In some cases, it might be required to run two processes in an attacker and victim scenario. In this case, SPECULATOR is able to run two tests in a co-located manner to analyze the effects of a process influencing another, like in the case of Spectre v2 or DoubleBTI [33]. SPECULATOR collects different counters for the attacker and the victim. Under this configuration, SPECULATOR performs no synchronization between the two processes.

Once the tests results are collected from the Monitor, they are handed to the post-processing unit (Step 6). This unit aggregates the results from multiple runs by computing statistics (e.g., mean and standard deviation) and by removing clear outliers.

4.1.4 Triggering Speculative Execution

The SPECULATOR user supplies as input a code snippet to determine how the CPU behaves when speculative execution takes place. I note that in the absence of branch misprediction, instructions that are speculatively executed will eventually retire and there should be no undesired microarchitectural side-effects. The more interesting case for the SPECULATOR user is a snippet containing a branch, or other speculative execution trigger that the CPU does not predict accurately, leading to the speculative execution of instructions that will not retire. In this scenario, SPECULATOR helps the user detect which instructions the CPU executed and how they influenced the microarchitectural state.

To automate the generation of test cases, SPECULATOR provides the user with a series of templates that can be used to reproduce the various speculation triggers. For instance, SPECULATOR contains templates to study Branch Target Injection (BTI) cases including attacker and victim, or branch-based templates to study particular series of instructions, or templates that causes *ret* instructions to be speculated like in the Return Stack Buffer case, and so on.

An example using a common branch as trigger is described in Figure 4.2. The template is used as follows: the user supplies a snippet, expecting i) it to be speculatively executed, ii) that none of its instructions will retire, and iii) that SPECULATOR will report counters relating to its execution. To achieve this, the template prefixes the snippet supplied by the user with a branch instruction. The template begins with a setup step that trains the branch predictor not to take that branch. After the branch predictor is trained, the program state is set to require the branch to be taken to ensure that the snippet will be speculatively executed and that none of its instructions will retire. The template then starts the performance counters that were previously setup by the Monitor and executes the branch, after which it stops performance counters. In order to prolong or shorten the speculative execution of the user snippet, the condition variable of the branch can be placed in registers or memory. On the microarchitectural level, a variable placed in memory can also be cached in one of the levels of the cache hierarchy.

4.1.5 Speculative Execution Markers

In the context of SPECULATOR, I am mostly interested in determining the behavior of the CPU when instructions that are speculatively executed do not retire. A first natural question is whether non-retired instructions were speculatively executed at all and, if so, how many of them. An accurate

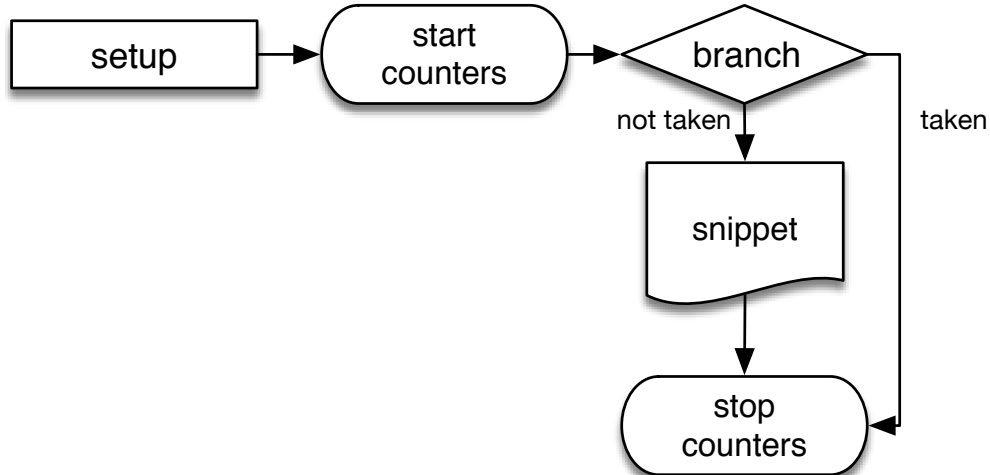


Figure 4.2: Flow chart of one of the experiment template used in SPECULATOR. The setup code brings the branch predictor in a specific state that will cause the later branch to mispredict and speculatively execute the code snippet consisting of the instructions. The speculative execution of the instructions is measured by the PMC infrastructure, which is triggered by the corresponding start/stop instructions indicated in the flow chart.

detection of these events is (perhaps surprisingly) not trivial. Indeed, the CPU strives to undo most observable architectural side-effects from non-retired speculatively executed instructions. However, as I know from the Spectre and Meltdown works [5, 6], not all side effects are undone. One possible approach to detect non-retired speculative execution would be to rely on the side-channels exploited in these works. This approach has several shortcomings: it is noisy, i.e., it has a relatively low single-run detection accuracy, it is costly to setup and read, and it requires otherwise unnecessary changes to program observables.

A more effective approach is based on markers of speculative execution, that is, special instructions or sequences thereof (which I will refer to as markers) that are detectable by performance counters even when they do not retire. The approach requires appending the marker to the snippet which is fed as input to SPECULATOR, and ensuring that there is no other occurrence of the marker in the snippet. If SPECULATOR detects the marker, the detection can be used as proof that the CPU executed the snippet.

The choice of which markers to use is manufacturer- and architecture-specific, given that not all CPUs expose the same set of counters. In general, the marker must cause a microarchitectural event that is detectable by a performance counter irrespective of its retired status. For example, counters that measure *issued* or *executed* instructions of a specific type irrespective of their retired status constitute a good marker. The selection of which counter to use on a given architecture requires manual inspection of the CPU architecture programmer’s manual. In what follows, I report my findings on the available markers for Intel processors:

`UOPS_EXECUTED.CORE/THREAD` counts the number of μ ops executed by the CPU. It can be used to report the exact number of μ ops that were executed out of the user-supplied snippet by subtracting the number of μ ops that retire in the template (the branch and the instrumentation to stop performance counters) from the output value of the counter. This counter is subject to μ -fusion of instructions and does not count instructions that do not require execution such as

NOP. An exception to that rule is *FNOP*, which is tracked by this counter as well.

UOPS_ISSUED.SINGLE_MUL belongs to a group of counters triggered only by a specific set of instructions. This counter is fired whenever a single-precision floating-point instruction that operates on the *XMM* register is issued. This means that such an operation can be inserted at the end of the user-supplied snippet to verify whether this counter is incremented or not. This counter has been dropped by Intel on most recent CPUs (e.g., Skylake) and therefore its usage is limited across platforms.

Similarly to *UOPS_ISSUED.SINGLE_MUL*, *UOPS_ISSUED.SLOW_LEA* is triggered by only a specific set of instructions. It counts *LEA* instructions with three source operands (e.g., *lea rax, [array+rax*2]*). Unfortunately, certain operations such as *clflush* are considered by the CPU as *SLOW_LEA* operations, so extra care must be taken to subtract any number of those present outside of the user-supplied snippet.

LD_BLOCKS.STORE_FORWARD is incremented for each store forward that result in a failure. An example of a sequence that triggers this kind of situation is shown in Listing 1.

```
1 mov DWORD[array], eax
2 mov DWORD[array+4], edx
3 movq xmm0, QWORD[array]
```

Listing 1: Failed store forward example

The following markers are available on the AMD Zen architecture: *DIV_OP_COUNT*, counting the number of executed *div* instructions. *NUMBER_OF_MOVE_ELIMINATION_AND_SCALAR_OP_OPTIMIZATION*, like *LD_BLOCKS.STORE_FORWARD*, does not track the execution of an instruction, but rather the effect of a certain instruction sequence. In this case, it tracks in how many cases move elimination was successful.

4.2 Using SPECULATOR: Dissecting the microarchitectural world

Using *SPECULATOR*, I explore the microarchitectural behavior of modern CPUs. Our goal is twofold: I aim to investigate several speculative execution properties, as well as test new PoC attacks and available mitigations in a deterministic manner using the speculative execution markers introduced in Section 4.1.5.

The results that I uncover are applicable to previously discovered and new attacks, and are also of independent interest. Since some of my findings are hardware-dependent, I also show the differences based on the underlying CPU architecture (Table 4.1).

4.2.1 Return Stack Buffer Size

The first set of experiments measures the size of the *Return Stack Buffer (RSB)*. The RSB is an internal buffer used by the CPU to predict where a *ret* instruction is returning to. Koruyeh et

Architecture	CPU	Design
Intel Haswell	i5-4300U	tock
Intel Broadwell	i5-5250U	tick
Intel Skylake	i7-6700K	tock
Intel Kaby Lake	i7-8650U	optimization
Intel Coffee Lake	i7-8559U	optimization
AMD Zen	Ryzen 1700	

Table 4.1: The CPUs per architecture I use SPECULATOR on. While Haswell and Skylake are new designs – “tocks” in Intel nomenclature – Broadwell is a “tick”, a die-shrink of Haswell. Kaby and Coffee Lake are instead optimized versions of Skylake design within the same die size

al [64] and Maisuradze et al [65] show how this buffer can be misused to perform speculative execution attacks. I start with the RSB since information on its size is available and can be used to validate the accuracy of SPECULATOR.

To perform the measurement, I design a test template similar to the one presented in [64]. The test performs a *call* to a victim function. Whenever the CPU executes a *call* instruction, it pushes the expected return address (the instruction after *call*) on the application stack (architecturally) and in the RSB (microarchitecturally). The victim’s code further changes its return address to an exit routine by manually overwriting the stack. This way, the code at the original return address is only speculatively executed since at the microarchitectural level, the first entry in the RSB is popped and execution (speculatively) continues at that address. In order to be able to detect whether speculative execution takes place, a marker is inserted at this target.

Based on the described template, I generate a series of snippets that, between the *call* and *ret*, have a call to a *filler* function that contains an increasing number of nested calls. For each of the nested calls, an entry is added to the RSB. When the nested call stack depth is bigger than the RSB size, the RSB loses the oldest entries. In this case, once the CPU speculates the last *ret*, it has nothing to pop from the RSB because the previous nested call/ret consumed all the available entries. In that case, I expect the CPU not to be able to speculatively execute my marker.

I report in Figure 4.3 and Figure 4.4 my results for Intel Kaby Lake and AMD Ryzen. For Intel Kaby Lake, I observe that the marker is observable up to 14 nested calls. To count the slots available in the RSB, I need to also consider the additional call to the *filler* function that contains the nested calls. This results in a total of 16 entries in the RSB, which matches the value reported by Intel for the Kaby Lake RSB size. Interestingly, after 15 nested calls the number of mispredicted branches increases almost linearly, by one for each nested call added. This indicates that a second predictor is used as fallback once the RSB cannot provide any more values.

Figure 4.4 shows the results for AMD Ryzen. After 30 nested calls, I observe the marker hit to transition between 1 and 0.25. As before I need to account for the call to *filler*. The result for the AMD Ryzen RSB size is 31. my result differs from the nominal value I expected from the manufacturer specification, which is 32. With further research into the optimization manual [66], I found that one entry is actually reserved for “pointer logic simplification”. Therefore, the observed 31 entries is correct. On AMD, after the Return Address Stack (RAS) (the RSB in AMD nomenclature) is emptied, I still observe a correct prediction 25% of the time and not 0 as seen for

Intel. This implies that the second predictor used can still predict correctly 25% of the time in this particular setup.

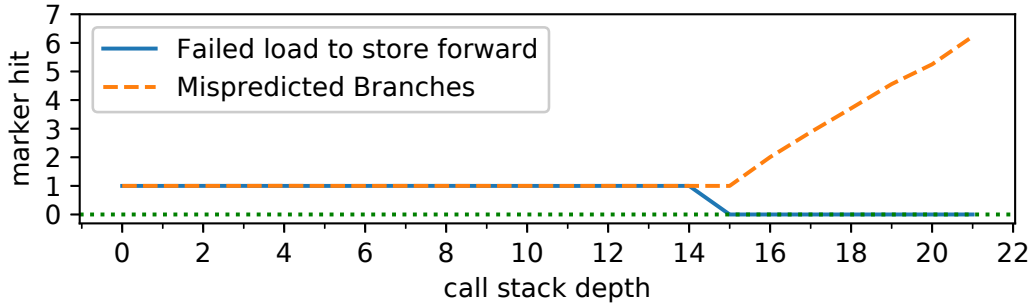


Figure 4.3: Return Stack Buffer test on Kabylake.

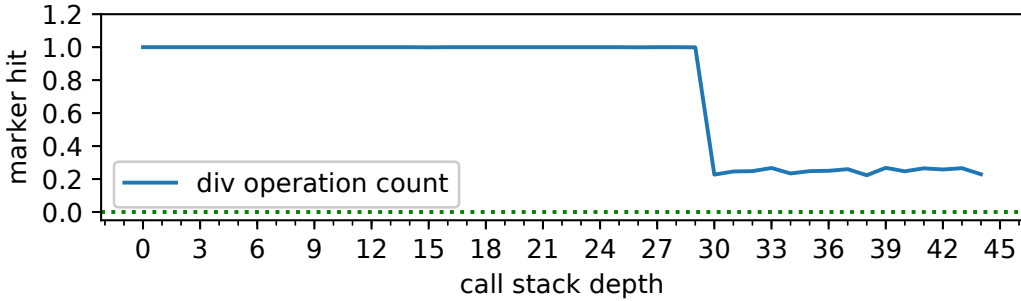


Figure 4.4: Return Stack Buffer test on AMD Ryzen.

4.2.2 Nesting Speculative Execution

An undocumented corner case that might affect the construction of attacks is when speculative execution encounters conditional branches in its path. The questions I try to answer with this experiment are “How is the speculative execution window affected by nested branches?”. And “What is the overall behavior of the CPU when nested branches are speculated?”.

I use SPECULATOR to evaluate the case of nested branches. This experiment has multiple potential outcomes: given two nested branches, an outer and an inner one, either *i*) the inner branch is not speculatively executed until the branch condition on the outer branch is resolved, or *ii*) speculative execution continues to the inner branch and beyond. In the second case, I am interested in the speculative execution behavior if the inner branch is resolved while the result of the outer one is still pending.

I design my experiment with three nested conditional branches, outermost to innermost, with the branch conditions being independent of one another. The conditions are set up with decreasing complexity, such that the outermost will take longest to resolve. I achieve this by involving an uncached value that is subject to multiple expensive operations (*divs*) in the outermost branch condition, a simple uncached value in the middle branch condition, and a cached value in the

innermost branch condition. As usual, I train the branch predictor for all branches in the setup phase such that it is going to mispredict all targets in the measurement phase. To evaluate which code paths are (speculatively) executed, I repeat the experiment multiple times with marker instructions placed in the opposite branch target paths.

I performed this experiment on both Broadwell and Skylake, yielding identical results: in both cases, nested speculative execution takes place, i.e., speculative execution continues along the trained branch targets for all branches. Second, if a nested branch condition is resolved before its parent branch and a misprediction has occurred, speculative execution picks up the opposite branch target. If a parent branch is resolved, all mispredicted code paths, including nested speculative execution, is canceled.

4.2.3 Speculative execution across system calls

An interesting case to analyze for new attacks is how speculative execution behaves in case the attack spans between multiple privilege boundaries. The question I try to answer here is: “Does speculative execution continue through instructions such as *syscall* and *vmcall*?”

I thus investigate whether speculative execution continues across the context switch from user- to kernel mode. To this end I design a simple test scenario, where the speculatively executed snippet issues a system call. For the system call itself I picked *sys_getppid* because of its low complexity – an execution only amounts to 47 instructions. I use the counter for executed μ ops and tune it to capture either just μ ops executed in user mode or kernel mode.

I performed the experiment on the Broadwell and Skylake microarchitectures with identical results:

- The number of μ ops executed in user mode corresponds to the instructions before the system call and does not increase with additional instructions added after the system call.
- The number of μ ops executed in kernel mode does not increase compared to a baseline measurement taken without speculative execution of the code snippet.

I conclude that a system call effectively stops speculative execution after the system call returns from kernel mode. I further conclude that a speculative execution attack across the system call boundary is not feasible on the tested Intel CPUs.

4.2.4 Flushing the Cache

The x86 instruction set provides a convenient, dedicated instruction to cause the CPU to flush the cache line indicated by a memory address from all caches, *clflush*. It is very useful in settings where an attacker can execute assembly instructions, as it allows easy eviction of data from the cache.

I use SPECULATOR to investigate how *clflush* behaves when executed speculatively. To this end, I create a snippet that first flushes the cache line corresponding to a value stored in memory and then loads the value. This is shown at line 4 and line 8 respectively in Listing 2. I perform two runs, one where the setup code warms up the cache by loading the value from memory (line 7) and one where the value is left uncached. In both tests, within the speculated sequence, I place a *clflush* followed by an *lfence* instruction to stop the speculation, making sure that the final load is not executed during speculation as well (line 15). I measure the execution cycles on both runs, which shows a difference of over 160 clock cycles between the two settings. This is a clear indication that

while *clflush* is speculatively executed, it does not affect the cache until retired. Thus, during a speculative execution attack, the attacker cannot extend the speculation window using providing in its code a *clflush* on the speculation starter variable.

Another result I draw from this experiment is that, to make sure *clflush* is effective, it needs to be combined with an instruction that stops speculative execution, such as *lfence*.

```
1      setup
2  .loop:
3      clflush[counter]
4      clflush[var]          ;var cache flush
5      lfence
6
7      mov eax, DWORD[var]    ;cached version
8      lfence                ;only
9
10     start_counter
11
12     cmp 12, DWORD[counter]
13     je .else
14
15     clflush[var]          ;var cache flush
16     lfence
17
18 .else:
19     mov eax, DWORD[var]    ;final load
20     lfence
21
22     stop_counter
23
24     inc DWORD[i]
25     cmp DWORD[i], 13
26     jl loop
```

Listing 2: Clflush test snippet structure

4.2.5 Speculation window size

The speculation window size is determined by the clock cycles that it takes until a speculation trigger is resolved. In this section, I provide my measurements of the speculation window for the different triggers used in the Spectre v1, v2, and v4 attacks. To measure clock cycles I use the facilities provided by the PMC of the respective platform: on Intel, a predefined counter tracks elapsed clock cycles according to the same settings as the configurable counters; on AMD, the APERF counter tracks elapsed clock cycles in general.

The theoretical upper limit of instructions that can be executed during speculative execution is given by the size of the reorder buffer, which I evaluated in Section 4.3.4. In practice, it is also limited by the execution ports and units available for executing those transactions. Thus, I

also investigate instruction sequences that do not lead to a bottleneck on those resources during speculative execution.

Conditional branches.

Conditional branches are the speculative execution triggers used in Spectre v1 to check for an out-of-bounds access to an array. The speculation window size depends on how fast the CPU determines that the actual branch target differs from the information provided by the branch target buffer. I place the conditional value that determines the actual branch target in different locations and involve it in additional computation to investigate how this affects the size of the speculation window. As a baseline, I measure how long the execution of the additional instructions takes. I then measure how long the execution of the instructions together with the conditional branch takes. The placement of the variable and the additional instructions on it affect the time it takes the conditional branch to retire. All measurements are performed a thousand times. Note that controlling the performance counters involves a system call. Since system calls stop speculation, I can only measure how long the retirement of an instruction sequence takes.

As described by Agner Fog in [67], the APERF interface offered by AMD Ryzen for clock cycles requires careful handling due to its scaling with the CPU frequency. Hence, since the measurement technique differs between the two CPU vendors for this particular test, results for Intel and AMD might not be directly comparable. However, while the post-processing of this dataset is different, the test methodology used to gather it is the same.

Table 4.2 shows the results of this experiment. I see that complex instructions such as *div*, which translates to multiple μ ops, widen the speculation window. The same is true for a cache miss, when the CPU needs to fetch the data from main memory.

At the same time, access to cached memory contributes little to the speculation window compared to a register access. Measuring a range from four to twelve cycles, the results for Broadwell and Skylake are in accordance with Intel's performance analysis guide [68] which states four cycles as the average for an access to L1 and ten cycles for L2.

On AMD, I see even less impact between register and cached accesses. In addition, adding a complex instruction on top of an access has a negligible effect on the speculation window size.

Indirect control flow transfer.

Indirect control flow transfers are the speculative execution triggers used in Spectre v2. The speculation window size depends on how fast the CPU determines that the target in the branch history buffer does not match the actual target. Table 4.3 shows the speculation window sizes depending on the location of the indirect branch target.

Store to load forwarding.

Modern CPU designs feature store and load queues, which capture the effects and dependencies of corresponding load and store operations before the data is even written to or read from the cache. This infrastructure allows for efficient store to load forwarding: if an instruction writes to a certain memory address and a following instruction reads from that very address, the CPU can leverage the result of the first instruction, which is written to the store queue, for executing the second instruction. This avoids unnecessarily stalling the execution of the second instruction

Conditional branch	Broadwell	Skylake	Zen
Register access	14	16	7
Access to cached memory	19	17	9
Access to uncached memory	144	280	321
Mul with register	19	19	2
Mul with cached memory	33	33	8
Mul with uncached memory	154	290	362
Div with register	35	41	17
Div with cached memory	34	39	30
Div with uncached memory	164	306	353

Table 4.2: Speculation window of a conditional branch depending on the type of instructions needed to resolve the branch as well as the placement of the value involved in the condition, measured in cycles.

Indirect branch target location	Broadwell	Skylake	Zen
Register	28	22	24
Cached memory	41	34	35
Uncached memory	154	303	301

Table 4.3: Speculation window of an indirect control flow transfer, measured in cycles. The speculation window size depends on where the target of the indirect control flow transfer is stored.

until the first is retired. In a recent attack, this behavior has been used for a “speculative buffer overflow” [69].

I am interested in the behavior a failed store to load forwarding causes. In this case, I deviate from my default SPECULATOR template and remove the branch instruction. Instead, I create a snippet with a data dependency that is not detected by the CPU in a combination with a sequence of store and load operations that triggers store-to-load forwarding.

Running the snippet in SPECULATOR reveals that store-to-load forwarding fails and the load instruction is in fact executed twice. This means that a failed store-to-load forwarding also creates a situation similar to speculative execution results being discarded because of a mispredicted branch, although it provides a significantly smaller speculation window.

Spectre v4 (a speculative store bypass) makes use of speculative execution through store-to-load forwarding. For this trigger I measure a speculation window of 55 cycles on average on Broadwell. I also measure the speculatively executed instructions using *FNOP*, which provides us with an upper bound for the speculation window in terms of instructions. I measure an average of 15 μ ops with a maximum of 23 μ ops (Figure 4.5).

Max speculation with optimized instruction sequence. During my experiments, I observed multiple situations in which the CPU back-end stalled. For instance, the CPU could stall due to exhaustion of execution units for a certain operation (e.g., *MOV*, *MUL*) or, for instance, data dependencies of multiple operations where one or more data loads caused cache misses. In a

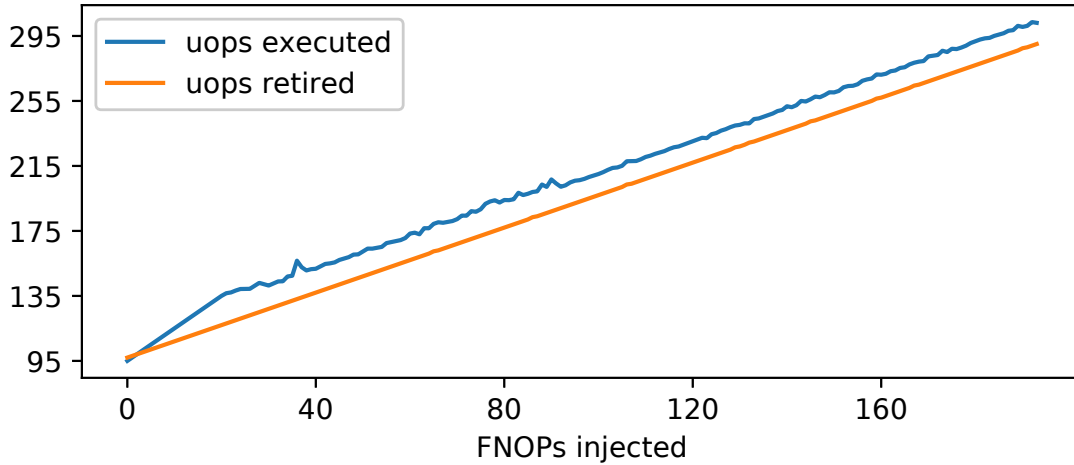


Figure 4.5: Speculation window of a store-to-load forward failure, measured in executed *FNOPs* on Broadwell.

hypothetical scenario, I wanted to verify how many non-*NOP* executed μ ops the CPU speculates within the maximum time window (e.g., access to uncached memory in combination with a *DIV* instruction). Based on the layout of the back-end of my Broadwell CPU under test, to the best of my abilities, I crafted an optimized sequence of instructions to account for the delay of each operation and the available execution unit. my tests show that the maximum number of non-trivial speculated instructions I could achieve was 160, with 187 being the maximum for *FNOP*.

4.2.6 Stopping Speculative Execution

Many instruction set architectures feature an instruction that stops speculative execution in the sense that no following instruction is speculatively executed. On x86 (and x86_64), one such instruction is *lfence*, short for “load fence”, the name reflecting its initial purpose of serializing all memory load operations issued prior to this instruction. In addition to this behavior, it also works as a barrier for speculative execution: the operational description in Intel’s manual [23] specifies that *lfence* waits on following instructions until preceding instructions complete.

I verify this behavior using SPECULATOR by creating a snippet with an *lfence* instruction followed by an increasing sequence of regular instructions. As expected, the counter for executed μ ops remains constant among the test runs irrespective of the number of instructions following *lfence*.

4.2.7 Executable Page Permission

Memory page permissions control access to memory regions at page-level granularity. As I have seen with Meltdown and Foreshadow, such permission checks might be lazily evaluated after an instruction is already executed, but before it is retired. Related work has so far focused on data read or write access to memory pages.

In this test I focus on whether execute permissions set by the NX bit are enforced. The NX bit is part of a hardware extension introduced by modern processors to mitigate stack-based code

injection exploits, among others. If the control flow of a program is diverted to a page without execute permissions, the processor will trap into the kernel to handle the fault. This raises the question of whether during speculative execution it is possible to execute instructions from a page without such a permission set.

Our corresponding experiment sets up a branch misprediction with a following control flow transfer to a non-executable memory region, to test whether instructions in it are (speculatively) executed. I ensure that the data from the page is in the L2 cache during speculative execution and the addresses are in the TLB. The result of the experiment is that the execute page table permission is honored during speculative execution by all architectures I examined. This is even true if an instruction spans over two pages: it will not be executed if the second page is set non-executable.

4.2.8 Memory Protection Extensions

Instead of performing bounds checks purely in software, Intel’s MPX instruction set extension [70] available on the Skylake platform provides hardware support for both efficiently keeping track of bounds information associated with pointers and corresponding spatial memory checks before dereferencing pointers. Pointer bounds information is stored in memory and loaded to dedicated registers before it can be used to check the upper bound using the *bndcu* and the lower bound using the *bndcl* instruction. If a bound check fails, a *#BR* exception is raised and the CPU traps into the kernel.

I used SPECULATOR to measure if and how much code following a bounds check instruction is speculatively executed. The setup executes the regular code path without the bounds violation for ten iterations and then fails on a *bndcu* twice. To measure the speculative execution window size, I first used an increasing run of *NOPs* in conjunction with a terminating slow *LEA* marker instruction. In this experiment, I measured that I speculatively execute the marker instruction for a sled of up to 122 *NOPs*. In my second experiment, I used *FNOP* instead of regular *NOP*, which is tracked by the *UOPS_EXECUTED* counter. As is shown in Figure 4.6a, in this case, the number of executed μ ops increases up to a sled of 22 *FNOPs* and remains constant beyond.

4.2.9 Issued vs. Executed μ ops

Some of the counters that I adopt as markers (e.g., *UOPS_ISSUED.SINGLE_MUL*, *UOPS_ISSUED.SLOW_LEA*) count the number of μ ops that are *issued*, as opposed to *executed*. Since issued μ ops are not necessarily executed, as is the case for the *NOP* instruction, I performed a dedicated experiment to verify whether they are also executed. I use the template introduced in Section 4.1.4 and generate tests where the code snippet just contains an increasing number of *RIP*-relative load instructions. As Figure 4.6b demonstrates, the number of executed μ ops increases at the same rate as the counter for slow load effective address instructions, which are load μ ops with three sources. This result confirms that the instruction is not only issued: its speculative execution does take place. I obtain similar results for other markers.

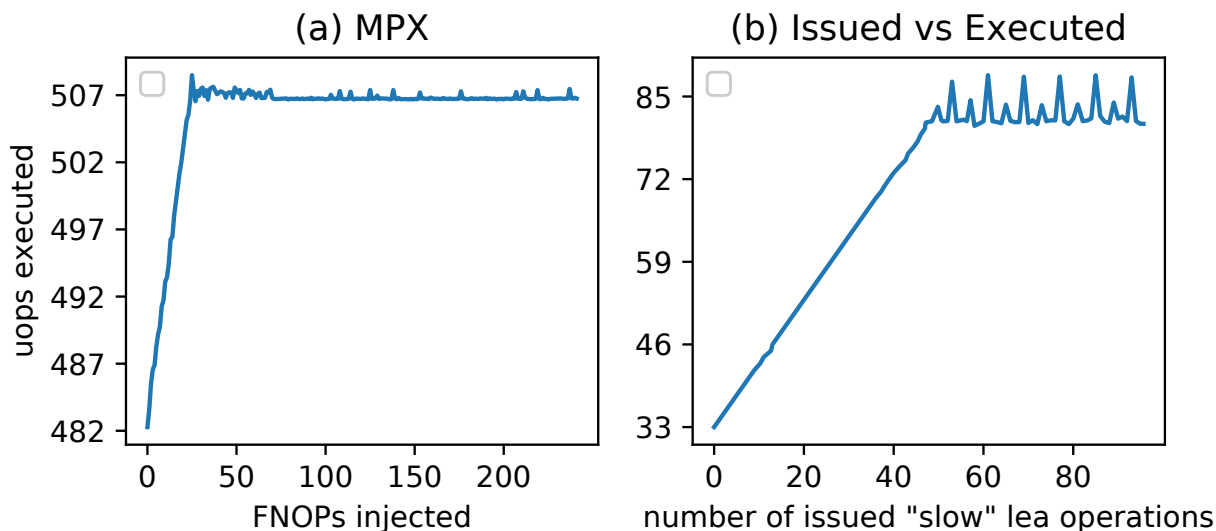


Figure 4.6: a) Speculative execution after an MPX bounds violation.
b) Performance counter numbers for an increasing number of speculatively executed relative load instructions. The graph shows that the number of issued instructions corresponds to the number of executed instructions, justifying the use of such instructions as markers.

4.3 Using SPECULATOR: Analyzing Attacks and Mitigations

I also use SPECULATOR to investigate new techniques to exploit speculative execution attacks. On one hand, I can use SPECULATOR to perform measurements on snippet of code to verify their behavior during speculation and verify that an attack might be feasible through those instructions (An example is described in the Section 4.4).

On the other hand, some of the attacks require two threads interacting with each other through a shared element such as the cache, the branch predictor or the RSB. For instance, during a Branch Target Injection (BTI) generally there is an attacker thread that trains the branch predictor which is shared between threads on the same physical core, and a victim thread that is condition by the attacker's training. I design SPECULATOR to support also an attack/victim scenario and used to analyze RSB and BTI (Section 4.3.2).

Even though speculative execution markers cannot be used in a real world attack since they require *root* access to the machine, they represent a valuable information source to verify the feasibility of a technique in a controlled and noise-free environment. Once an attack is proven to be working with speculative markers, it is easier to transition to methodology that do not require *root* access but that tend to be more noisy like cache side channels.

4.3.1 SPLIT SPECTRE

Here I try to mount a modified version of Spectre v1 I call SPLIT SPECTRE. Conceptually, I want to try to run a Spectre v1 attack with the attacker being able to provide the second of the two array accesses required. The aim is to lower the requirements for the attack, as gadgets for Spectre v1 are difficult to find in real software. I provide a detailed description of this attack in the Section 4.4,

Figure 4.9.

I implement SPLIT SPECTRE on SpiderMonkey 52.7.4, Firefox’s Javascript engine with standard configuration parameters. Although I found a real-world gadget corresponding to this attack easily (using *string.charCodeAtAt*), I were not able to make the exploit work. For a depiction of this attack, I refer to the Section 4.4, Figure 4.10.

To better understand the issues leading to the failed exploitation, I extracted the corresponding sequence of instructions from the trace of the attack and used them as a test inside SPECULATOR. The result of the experiment show that the speculation window is too short to perform both accesses. This fact is further confirmed when I run the attack using a shorter function that I manually provide in the Javascript engine: in this case, the attack is successful.

I can draw two important conclusions from the outcome of this experiment. First of all, SPECULATOR enables a systematic approach to the study of new attacks: *i*) formulate an hypothesis on a possible speculative execution attack; *ii*) identify a target and collect execution traces; *iii*) use the execution traces as part of a SPECULATOR snippet; *iv*) insert appropriate markers and gather results; *v*) repeat on all the desired architectures. Secondly, although no exploitation of SPLIT SPECTRE is known, the attack is theoretically feasible and there may be either architectures with a long enough speculation window to enable immediate exploitation on SpiderMonkey, or other exploitable targets with shorter gadgets.

4.3.2 BTI

One interesting scenario, I investigated with SPECULATOR, is the feasibility of BTI poisoning between co-located processes. I leverage the capability of SPECULATOR to run in attacker and victim mode. I design the victim process to perform an indirect *call* to a certain location *A*. Also, at a location *B*, I insert a marker instruction that is never executed by the victim process. I structure the attacker process with an indirect *call* aligned with the *call* in the victim process.

I run the test with attacker and victim on two co-located threads. I start the attacker before the victim to make sure that the indirect *call* in the attacker precede the one in the victim. Then, I start the victim and I observe the counter of the speculative execution marker at *B*. When the injection is successful, I observe the marker at *B* to be speculative executed by the victim. Our success rate is up to 82% over a thousand runs on Skylake and Kaby Lake and up to 55% on Coffe Lake and Broadwell. I report no success on AMD Ryzen.

4.3.3 Mitigations

Another interesting application of SPECULATOR is to test attack scenarios in the presence of mitigations. For instance, using the BTI poisoning test describes in Section 4.3.2, I test the current Spectre v2 mitigations available in the kernel. I focus on the following three: *STIBP* [20], *IBRS* [18] and *IBPB* [19]. These countermeasures require either microcode updates, or kernel updates or both. Our findings show that *BTI* between user space processes is mitigated only if *STIBP* is forced on all the applications or enabled conditionally by the use of *SECCOMP* or *prctl* from within the application.

It is worth noting that while these countermeasures are effective, the default settings in all the machines I analysed do not enable them, and very few application uses *SECCOMP*, and none *prctl*,

to enable request *STIBP* protection leaving them vulnerable to such attacks. I leave the complete analysis of the remaining security countermeasures implemented against SEAs to future work.

4.3.4 Out-of-order execution bandwidth

Speculative execution is no different in how it uses the resources available in both the front- and the back-end of a CPU compared to regular execution. On Intel platforms, instructions that have been fetched and decoded into μ ops by the front-end are entered in the reorder buffer of the back-end. This buffer contains all μ ops that are currently “in flight”, which means they are either ready for execution, are currently being executed, or have finished execution. The buffer’s name derives from the fact that on modern CPUs μ ops are executed *out-of-order*. This means they are dispatched to execution units based on their data flow dependencies, rather than the control flow of the program. After being executed, they remain in the reorder buffer until they are retired. Retirement of μ ops happens at an assembly-instruction granularity and *in-order*, honoring the control flow of the program. When μ ops are retired, the outcome of their computation is committed to the program’s state.

The size of the reorder buffer is a natural upper bound on the length of a sequence of instructions that can be speculatively executed. That is, the reorder buffer would hold the branch instruction that triggered speculative execution plus the instructions of the code path being speculatively executed. The branch instruction is the first one that is retired in-order, potentially causing all other μ ops in the buffer to be canceled in case of misprediction. If the branch instruction takes time to retire, e.g., because it depends on a compare that requires a slow memory access, chances are higher that the reorder buffer is filled with μ ops that are speculatively executed than for a branch that retires quickly. If the reorder buffer is full, the whole CPU back-end stalls.

A large reorder buffer is beneficial for attacks that exploit speculative execution because it lets a larger amount of instructions be speculatively executed, enhancing the capabilities of a speculative execution attacker. While the size of the reorder buffer is typically a known attribute of a CPU, I decided to empirically verify this number to show how precise measurements taken by SPECULATOR are. In my experiment, I use the *UOPS_EXECUTED.CORE* counter (see Section 4.1.5). Since the counter operates at the granularity of a core, I disable SMT to reduce the noise caused by Hyperthreads that are scheduled on the same core. I also use the *BR_MISP_RETIRED* counter, which counts the number of mispredicted, retired branch instructions.

When relying on the count of executed μ ops to measure the reorder buffer size, I need to keep in mind that the μ ops actually need to execute before the branch that triggered speculative execution is retired. This means I need instructions that execute quickly to achieve maximum throughput. Since “regular” instructions would easily saturate the available execution ports and units, I pick the *NOP* instruction. *NOP* is decoded into a single μ op, which occupies a single slot in the reorder buffer. It does not actually execute and thus neither requires an execution unit nor is it captured by the counter that measures executed μ ops. I thus put an arbitrary regular instruction as a marker at the end of the *NOP*-sled, increasing the latter in size for each test generated. When running this test with SPECULATOR, I expect to measure a constant amount of μ ops executed up to the point, where the *NOP*-sled takes up all slots in the reorder buffer and the terminating instruction is no longer speculatively executed. Indeed, the results match my expectation: as can be seen in Figure 4.7, the number of executed μ ops is constant up until 188 *NOP*s on Broadwell and 220 *NOP*s on Skylake. In addition to the *NOP*s I also need to account for the branch instruction,

taking up two slots in the reorder buffer as well as the marker instruction, taking up yet another two entries. In total, this is in line with the specifications published by Intel, which state a reorder buffer size of 192 entries for Broadwell and 224 entries for Skylake.

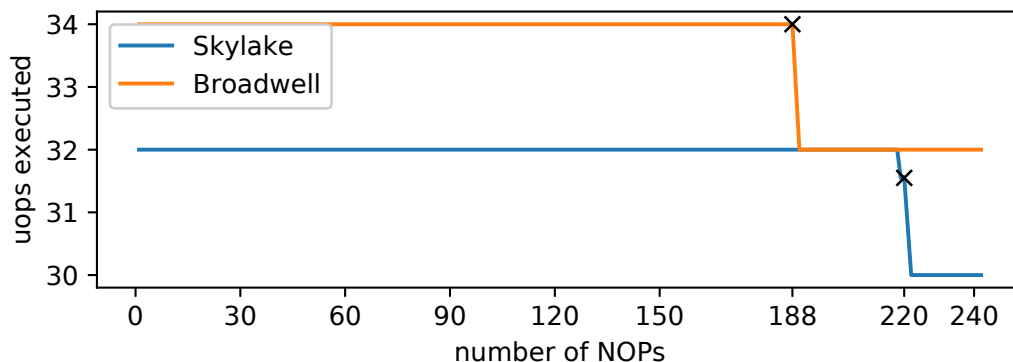


Figure 4.7: Reorder buffer size test results on Broadwell and Skylake. Since the marker instruction is no longer executed for a sufficiently large number of *NOPs*, the number of executed μ ops drops at the size of the reorder buffer.

Interestingly, the number of executed instructions differs for the architectures: it is 34 and 32 for Broadwell and 32 to 30 for Skylake, in spite of the code being exactly the same. Presumably, this is caused by extended μ op-fusion introduced as optimization on Skylake. Fused μ ops count as a single μ op.

AMD’s Zen platform has a construct similar to Intel’s reorder buffer: the retire queue. Every μ op that has entered the back-end and not been either retired or canceled takes a slot in this queue. my Ryzen CPU does not feature a counter for executed μ ops, so I can only provide a measurement based on my marker instruction in this case. The marker instruction, which takes up four μ ops in this case, is executed up until 186 *NOPs*. This is in line with the size of the retire queue, which is specified to have 192 entries ($= 186 + 2 + 4$). Interestingly, the speculation window seems to be halved when I switch off SMT: I recognize execution of the marker instruction only up to 91 *NOPs*.

Empty RSB behavior in pre-Skylake CPUs

During my RSB test, I run the test on all the Intel machines I have available that are listed in Table 4.1. Meanwhile the results related to the actual length of the RSB give us expected results, I notice that the behavior of the CPU after the RSB is emptied is different for machine pre-Skylake.

As Figure 4.8 shows, the CPU (in this case a Broadwell CPU) is still able to hit the expected return location around 25% of the time even though the RSB is actually empty. This behavior differs with the one of newer machines like Kaby Lake presented in Figure 4.3. This indicates that the re-design that happened in Skylake, and its optimizations, affected the second line of prediction in case of very deep call stack like the one I purposefully tested.

4.4 SPLIT SPECTRE

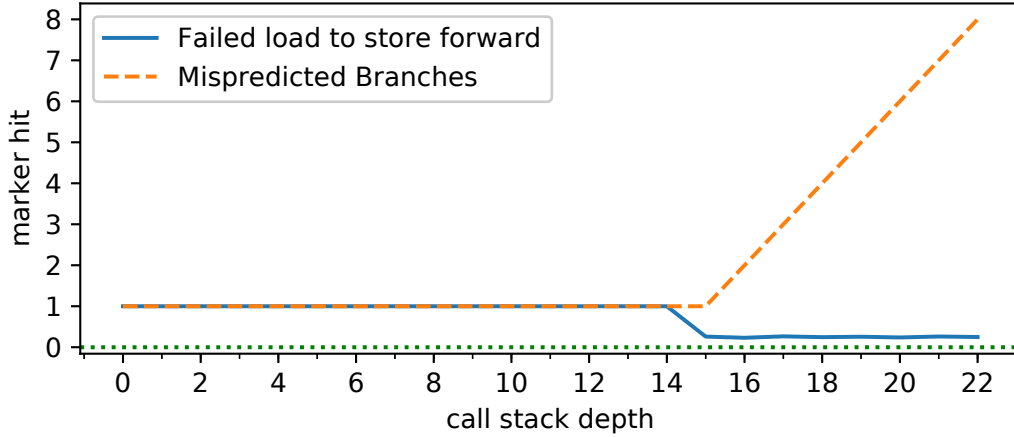


Figure 4.8: RSB test on Broadwell. As in the AMD case, Broadwell is able to predict the location of my target even if the RSB is empty.

4.4.1 The SplitSpectre Gadget

In Spectre v1, the victim code that is executed speculatively (“gadget”) consists of three components:

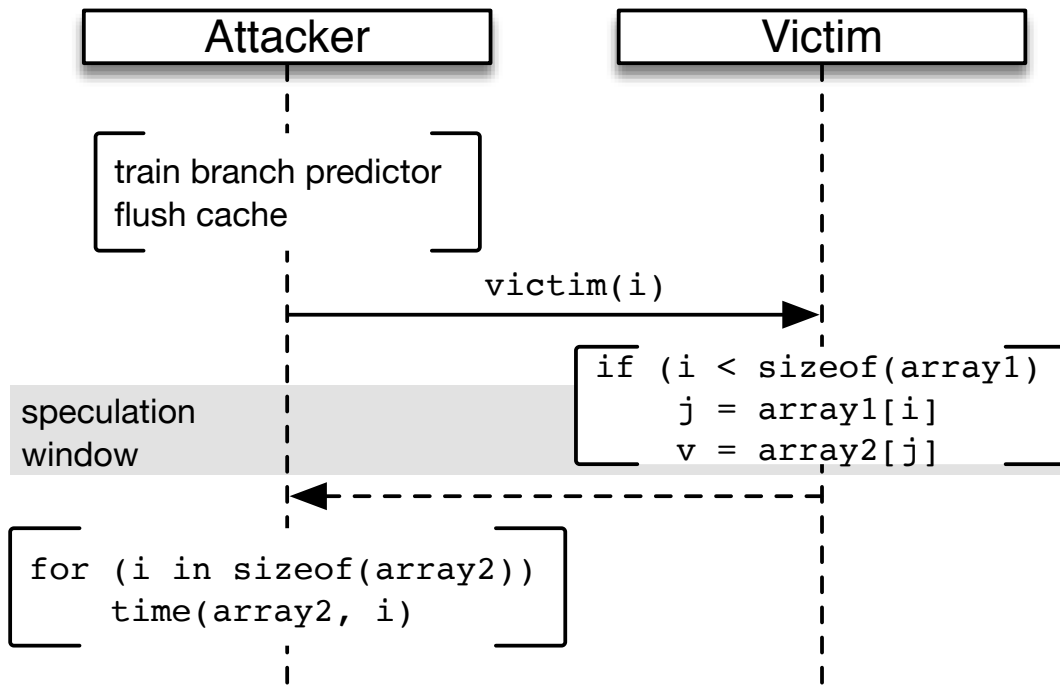
i) a conditional branch on a variable, typically a length check, *ii)* a first array access that uses the variable from the conditional branch as an offset, and *iii)* a second array access that uses the result of the first array access as an offset.

If the conditional branch triggers speculative execution of the following array accesses (phase ③ described in Section 2.2.3), the first array access may access an out-of-bounds memory region, revealing the contents of this region through a side channel (phase ④) by measuring the access time to the second array after executing the gadget (phase ⑤).

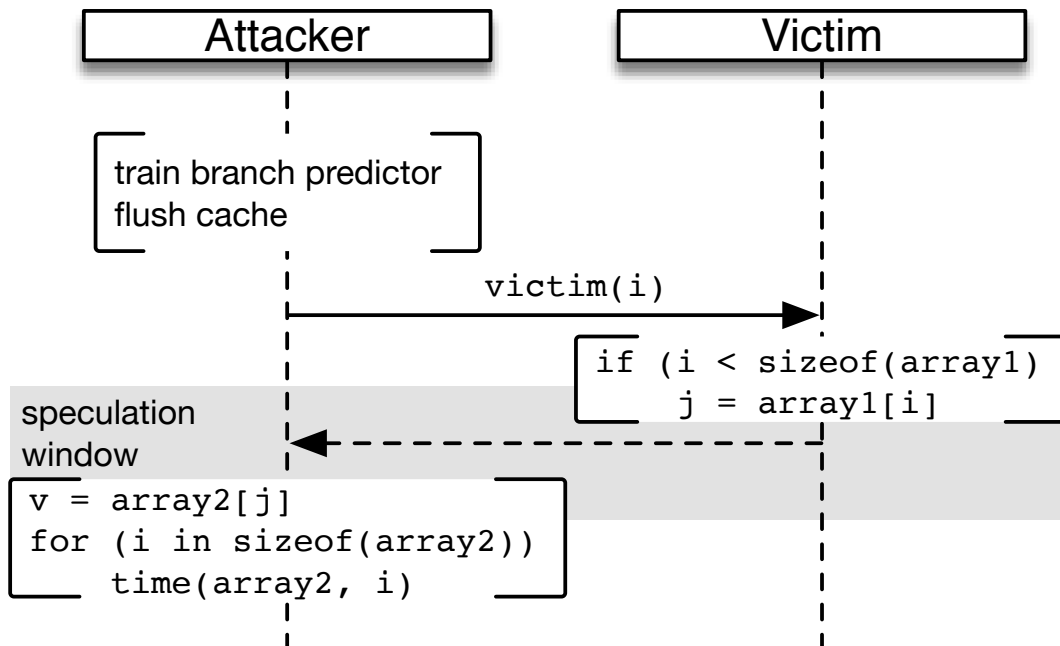
Although Spectre v1 is powerful and does not rely on SMT, it requires such a gadget to be present in the victim’s attack surface. Google Project Zero writes in their original blog post on Spectre v1 [4] that they could not identify such a vulnerable code pattern in the kernel, and instead relied on eBPF to place one there themselves.

In this point lies the idea of my Spectre v1 variant, SPLIT SPECTRE. As its name implies, it splits the Spectre v1 gadget into two parts: one consisting of the conditional branch and the array access (phase ③), and the other one consisting of the second array access that constitutes the sending part of the side channel (phase ④). This has the advantage that the second part, phase ④, can now be placed into the attacker-controlled code. It is more likely that an attacker finds such gadgets, thereby alleviating one of the main difficulties of performing a v1 attack. Furthermore, the attacker can choose to employ amplification of a v1 attack by reading multiple indices of the second array to deal with imprecise time sources.

Figure 4.9 compares the regular Spectre v1 with my split version. As shown in the figure, the speculation window needs to be sufficiently large such that it still covers the second part. I define the *speculation window* (short for speculative execution window) as the time interval between the event that triggers speculative execution, e.g., a branch condition, and the point in time when it is resolved and the speculatively executed instructions are either retired or rolled back. The speculation window is measured in cycles and determines how many instructions of a given sequence are speculatively executed. The number of instructions of a given sequence that



(a) Regular Spectre v1. The gadget requires two dependent array accesses in the victim's attack surface.



(b) Split Spectre v1. The second, dependent array access from a regular v1 gadget moves to the attacker code.

Figure 4.9: A comparison of regular Spectre v1 and SPLITSPECTRE. While SPLITSPECTRE only requires a simple array access, the speculation window needs to be sufficiently large to contain both the gadget and the second array access exercised by the attacker.

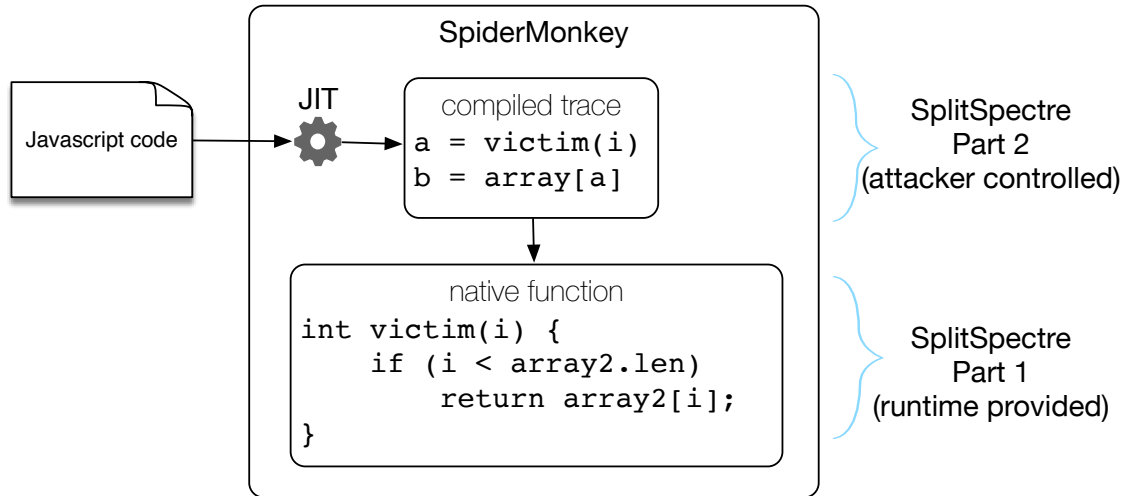


Figure 4.10: A conceptual view of a SPLIT SPECTRE attack instance with JavaScript.

can be speculatively executed at a given time also depends on the CPU’s microarchitecture. For example, some instructions are more “expensive” in the sense that they are split into a number of μ ops, and thus take a long time to execute. Also, the combination of instructions in a sequence affects how fast they execute: similar instructions might lead to congestion on the execution ports, as they require similar execution units.

The speculation window caps the maximum number of instructions executed between the two parts. Extending the length of the speculation window is an instrumental part in extending the capabilities of a speculative execution attacker and the reach of a SPLIT SPECTRE attack. In the course of the dissertation, I show how I use SPECULATOR to evaluate SPLIT SPECTRE and speculative execution aspects relevant to its feasibility.

4.4.2 The Analysis

I mounted a SPLIT SPECTRE attack in a real-world setting. I chose a browser-like setting, where untrusted JavaScript is executed in a trusted runtime environment, establishing a privilege boundary. Recall that a v1 gadget consists of a bounds check and two array accesses, the first one using the provided index and the second one using the content of the first array at that position as an index into the second array. In order to mount a regular Spectre v1 attack, I would require a complete Spectre v1 gadget available in the JavaScript engine. The intuition behind SPLIT SPECTRE permits us to relax this requirement and only require the first half of a V1 gadget, i.e., the bounds check and the first array access. The second half of this gadget is provided by attacker-controlled JavaScript code (Figure 4.10). The attack can only work if speculative execution spans across the privilege boundary from the bounds check in the runtime environment to the second array access in the attacker-controlled, unprivileged code.

I implemented SPLIT SPECTRE on SpiderMonkey 52.7.4 – Firefox’s JavaScript engine. I use the standard configuration parameters and conducted experiments on my Haswell, Coffee Lake, and Ryzen CPUs.

I start my experiments by introducing a built-in native JavaScript accessor function to SpiderMonkey’s source code that returns the content of a pre-allocated array at a given index. This

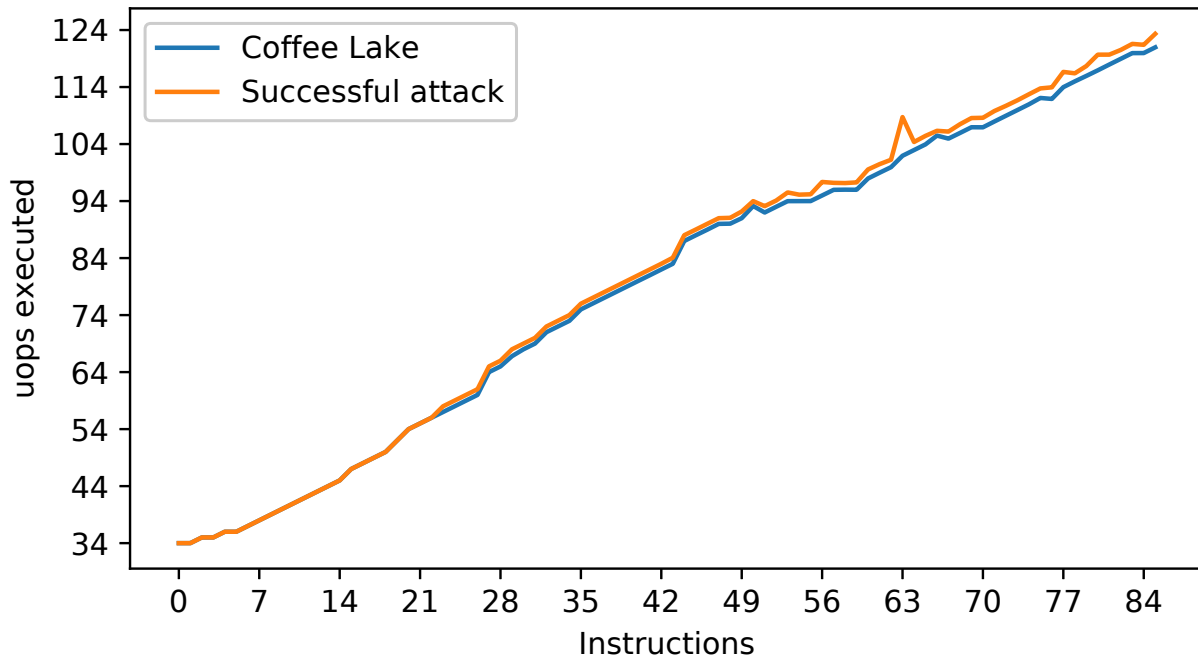


Figure 4.11: An examination of the SPLIT SPECTRE execution trace between the length check of *string.charCodeAt_impl()* and the second array access using SPECULATOR. The graph shows my results of the test on a Coffee Lake machine. It shows that, on average, I am not reaching the second array access in speculative execution. The small spikes in the graph are caused by mispredicted branches in the trace itself, which lead to nested speculative execution of fast-executing code paths.

function is the first part of the speculative execution gadget that needs to be part of the victim’s attack surface. To simplify the code, I explicitly flush the bounds of the array. my attacker code is an adapted regular V1 PoC code for JavaScript JIT engines, with just the first array access replaced by the call to the victim function. The time measurement is done using the SharedArrayBuffer technique, which reads the content of such a buffer while it is being incremented in the background by a web worker that is running in parallel.

The attack is successful. That is, on my Coffee Lake platform, I leak a string of ten characters with a success rate of over 86.2%, and I leak the full string with a success rate of 46% (i.e., see Table 4.4). Investigating the distance between the two parts of the speculation gadget, I measure the distance after 50 training runs of the JavaScript code that causes Spidermonkey’s tracing JIT to compile an optimized IonJIT trace implementing the JavaScript code in assembly. The distance between the bounds check and the second array access is 43 instructions, which is small enough for the attack to produce reliable results.

I proceed with my experiments by replacing my native built-in function with code already present in the SpiderMonkey source. my scan for a suitable gadget reveals the built-in *string.charCodeAt()* function, which returns the character code of a string at a given index and is implemented in native code. Internally, *string.charCodeAt()* calls *string.charCodeAt_impl()*, which includes the bounds check and actual access. Unfortunately, the speculation window is not large enough for

	Haswell	Coffee Lake
Runs	100	100
Only highest scoring char	76.6%	76.8%
1st and 2nd highest scoring char	80.7%	86.2%
Full string leaked	10%	46%

Table 4.4: Success rates for the SPLIT SPECTRE attack on JavaScript. I perform 100 runs, each run trying to leak a string of 10 consecutive characters. I provide numbers on both the highest and the second highest scoring characters.

the attack to work with *string.charCodeAt()*: After 50 training runs, the distance between the compare in *string.charCodeAt_impl()* and the dereference of the second array in the JIT trace is 90 instructions. An examination of the extracted execution trace with SPECULATOR shows that the number of speculatively executed μ ops is, on average, slightly lower than necessary for a successful attack (Figure 4.11). This means that in this scenario, the crucial load instruction is not always reached during speculative execution.

I also examine the execution trace on an AMD Ryzen CPU using a marker instruction, since the Zen performance counters do not feature a generic counter for executed instructions. I observe the marker instruction being executed for the full length of the trace. However, even here, the granularity of the time measurement is too coarse-grained to permit a successful read of the cache side channel. Amplifying the attack by adding multiple dependent array accesses would extend the trace so that it no longer fits into the speculation window.

I further optimize the attack by reducing the amount of code that is executed between the bounds check and the second access. This is achieved by implementing the second access and the call to the victim function in web assembly, which allows even more attacker control over the compiled JIT trace. However, using WebAssembly actually increases the number of instructions between the compare and the second access to 107. This is because the native call is not made directly from within the WebAssembly. Rather, additional JavaScript glue code is invoked.

JIT engine authors have already reacted with countermeasures [43, 42] in order to mitigate Spectre v1 in the context of browsers. These countermeasures mostly address sources for high-precision timers. Diluting the timing and disabling homebrew sources such as SharedArrayBuffers mitigate this version of JavaScript SPLIT SPECTRE. However, it remains to be seen if amplification of the attack’s timing properties make it feasible if only coarse-grained time sources are available.

On top of timing-related countermeasures, the V8 engine also masks addresses and array indices in JITted code before dereferences. While this mitigates a standard Spectre v1 attack, it does not help with SPLIT SPECTRE, where the bounds check is actually not exercised in JITted code, but the engine code itself.

my analysis lead us to conclude that the attack is viable, and that the ability to trigger it in practice depends on the identified microarchitectural properties of individual CPU families. I leave a comprehensive analysis of these properties for the various CPU architectures/models as an item of future work, which can be aided by SPECULATOR.

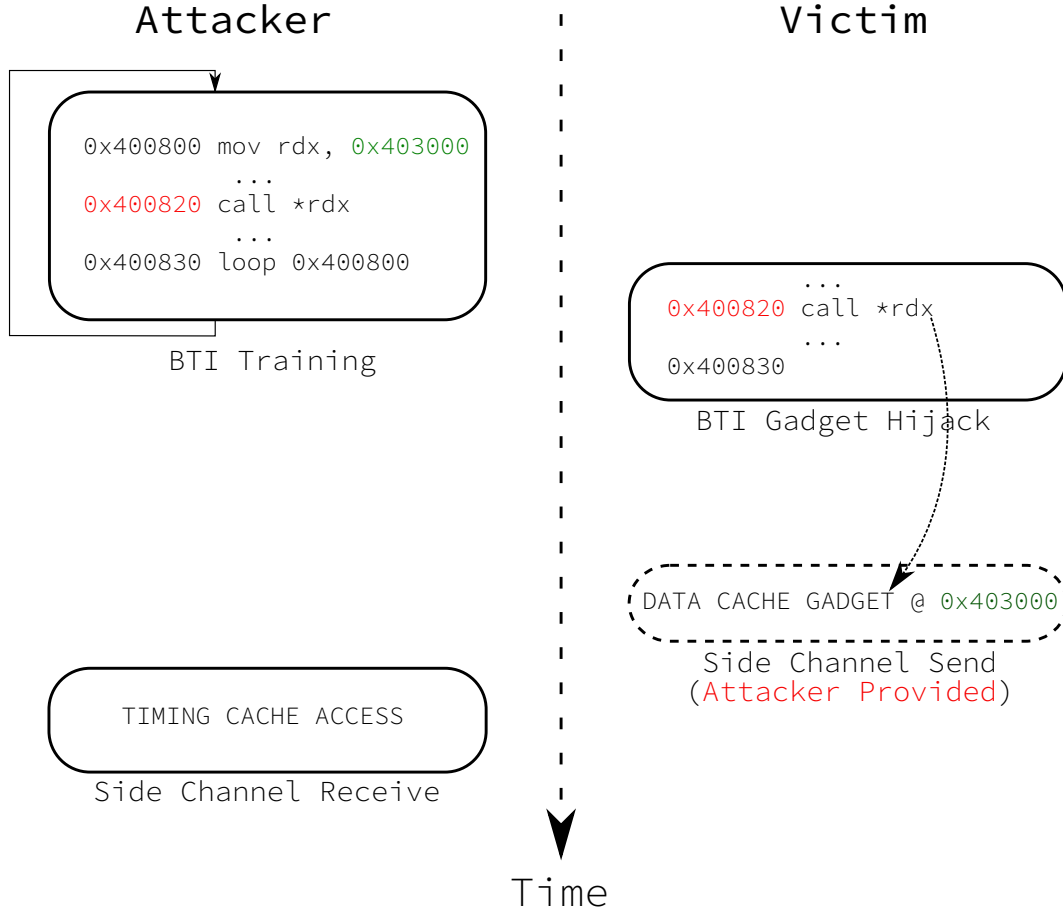


Figure 4.12: Overview of Spectre v2, a SpCFH attack: the attacker performs BTI at first; the victim speculatively executes the injected gadget whose cache side effects are later measured by the attacker.

4.5 New microarchitectural side-channels

Since the discovery of Spectre in 2018, a large number of SEAs have been presented. Most of these attacks rely on the Spectre V1 *side-channel-send* gadget (or *spadget*) presented in the original attack [5] which enables the information leak through a microarchitectural side channel. However, all the PoCs known to date require the ability to inject code or return into attacker-provided code (as in the Google Project Zero eBPF-based Spectre v2 exploit), showing that suitable spadgets have been hard to find. This motivates the research for new classes of spadgets.

In this dissertation, using the SPECULATOR and the technique described in Section 4.1, I investigated two new classes of spadgets. The first uses the instruction cache as a send and receive channel to leak a bit, dependent on a forced control flow in a spadget. The second uses BTI itself as a send and receive channel.

4.5.1 Icache Attack

The first contribution of this dissertation in discovering new side channel gadgets is the *icache attack*. Informally, this attack is based on the following observation: while the CPU strives to undo the effects of speculatively executed but not retired instructions, it does not hide effects on the instruction cache. As such, the instruction cache may be used to build a side channel between a gadget speculatively executed by a victim process and a gadget executed by an attacker process.

This attack makes use of speculative control flow hijack in order to redirect the victim to a gadget, henceforth referred to as the *icache gadget*. The icache gadget has the following characteristics: *i)* a compare-like instruction followed by a conditional jump; *ii)* target and fallthrough block of the jump leaving *measurable* and *distinct* side effects in the instruction cache; *iii)* the gadget is mapped by both the victim and the attacker. By measurable I mean that another process should be able to observe changes to the instruction cache left by the speculative execution of the gadget, for instance by attempting to execute either block (target or fallthrough) and measuring the speedup (or lack thereof) induced by the fact that the instructions of the block are present in the instruction cache. By distinct I mean that the effect left by speculative execution of one block should be different from those left by the other block. These two conditions constitute a *side-channel-send* operation over the information constituted by the condition of the jump. Clearly this information must be valuable from a security perspective: the condition may for instance depend on a compare instruction where the content of the register argument contains a secret for the victim. The last condition is required for the *side-channel-receive* operation, since cache line tagging in the instruction cache will not produce cache hits unless the cache lines have identical (physical) tags. Virtual indexing and ASLR also plays a role which will be discussed later in the section.

Figure 4.13 describes the attack. Attacker and victim are two co-located processes (either interleaved on the same hardware thread or running on different hardware threads in the same core). At first the attacker performs standard branch target injection by training an indirect jump to redirect the control flow to a specific address. The attacker chooses this address as that of the icache gadget. Whenever the attacker is successful, the control flow of the victim will be (speculatively) redirected to the icache gadget. In the figure, the gadget compares the content of *rax* to an immediate, and based on the result jumps to a block that performs a direct call – either to *fun1* or *fun2*. I assume that *rax* contains a secret, loaded before the indirect jump of the victim is executed. If BTI was successful, the attacker may later time the execution of either of the two functions to receive the leaked bit through the side channel. Note that the schedule of attacker and victim only needs to be loosely synchronised: the attacker’s BTI training needs to be scheduled before the victim’s targeted jump, and the attacker’s icache timing must be scheduled after the speculative control flow hijack takes place. The attacker is thus able to leak one bit for each successful round. By varying the icache gadget to point to gadgets that leak different bits of the secret, the attacker may be able to partially or entirely reconstruct the secret.

4.5.2 Icache Discussion

Anatomy of an icache gadget

As discussed, the icache gadget presents relatively few restrictions and it is thus expected to be widely available to an attacker. In particular, the requirement of a shared memory mapping is

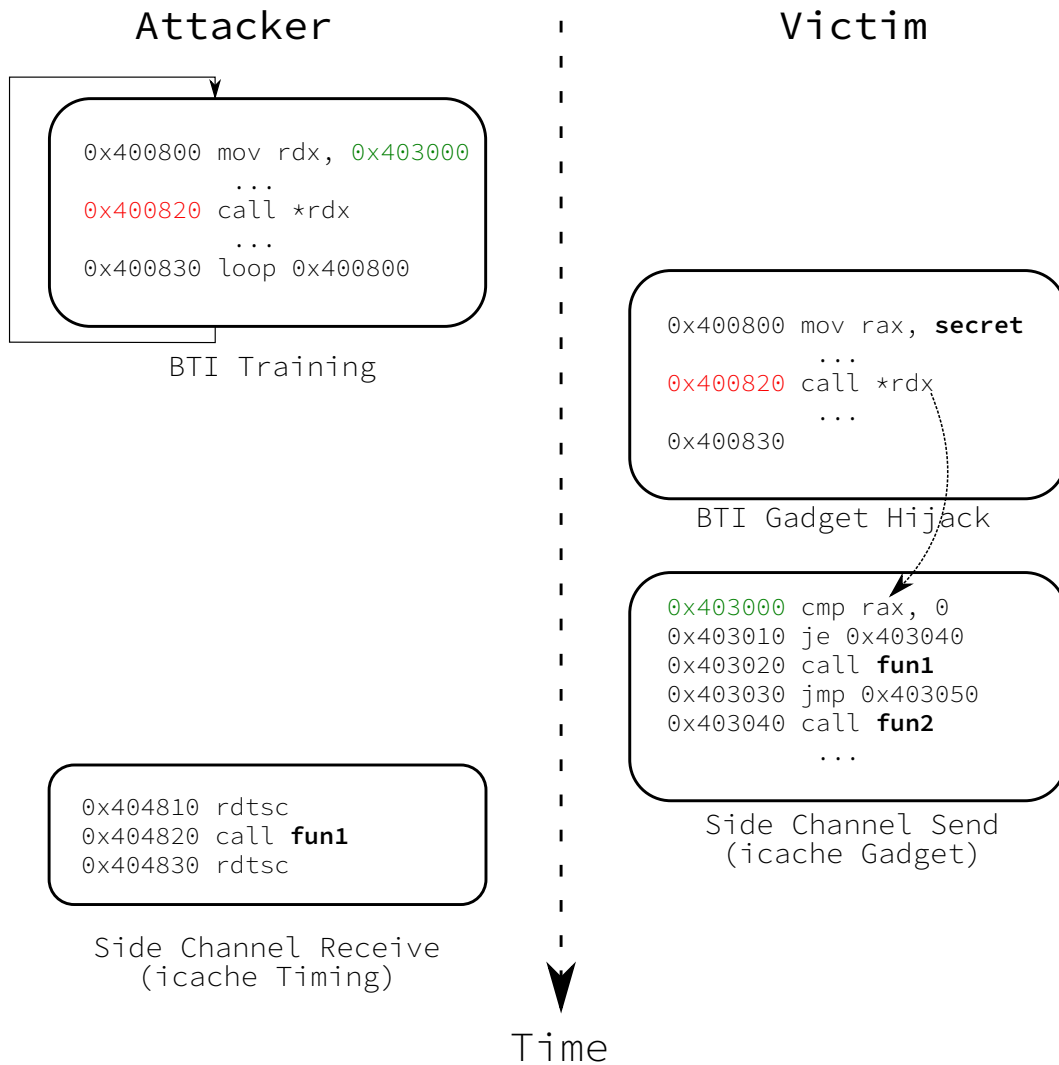


Figure 4.13: Description of the icache attack: the attacker performs BTI at first; the victim speculatively executes one of two functions depending on the content of a register; the attacker later times the execution of either function to learn one bit of the condition register.

satisfied in the (common) case of two processes (attacker and victim) using a common shared library, or the attacker mapping the executable of the victim. This ensures that instruction cache lines will have identical (physical) tags. Restrictions on virtual addressing will be discussed later in the section. The gadget shown in Figure 4.13 requires target and fallthrough of the conditional jump to contain a call to different functions. However, at its core, the gadget only requires that the icache-observable side effect be different depending on the outcome of the conditional jump. With this criterion I may eliminate gadgets whose size is a single cache line, or gadgets that will be prefetched in their entirety irrespective of the actual (speculated) control flow. No further restriction is imposed on the gadget. Finally, I stress that the icache gadget does *not* require the presence of the secret-dependent control flow antipattern in the victim code, e.g., as in previous icache-based attacks [71, 72]. While the icache gadget indeed performs a conditional jump based on the value of a secret, the secret is set by the victim in the completely unrelated BTI gadget.

ASLR

The presence of ASLR on most modern systems introduces an obstacle for the attacker; indeed, while the requirement on a shared mapping of the icache gadget ensures that cache lines will have identical (physical) tags, they must also have identical (virtual) indices. The attacker may either target a shared icache gadget that is not built as position-independent code (e.g., (rare) a shared library built without the *fPIC* or equivalent compiler option; or (more common) an executable built without the *fPIE* or equivalent compiler option), or utilise well-known means of discovering the ASLR offset [73, 74].

Alternative *side-channel-receive*

In the icache attack, the side channel is read by timing the execution of either the target or the fallthrough block of the jump in the icache gadget. An alternative to this approach is to perform a standard cache timing attack, by simply reading the code to probe it, instead of executing. Given that in my target platforms L1 data and instruction caches are separate, I did not try this experiment because the side channel would be noisier due to the smaller time difference between L2 cache and main memory.

4.5.3 Double BTI Attack

In this section I describe the second attack, called *Double BTI attack*. The Double BTI attack also exploits speculative control flow hijack, as first shown by the Spectre v2 PoC. The original Spectre v2 POC, depicted in Figure 4.12, requires the ability of the attacker to inject a gadget into the victim address space, namely, the data cache gadget used to perform the *side-channel-send* operation.

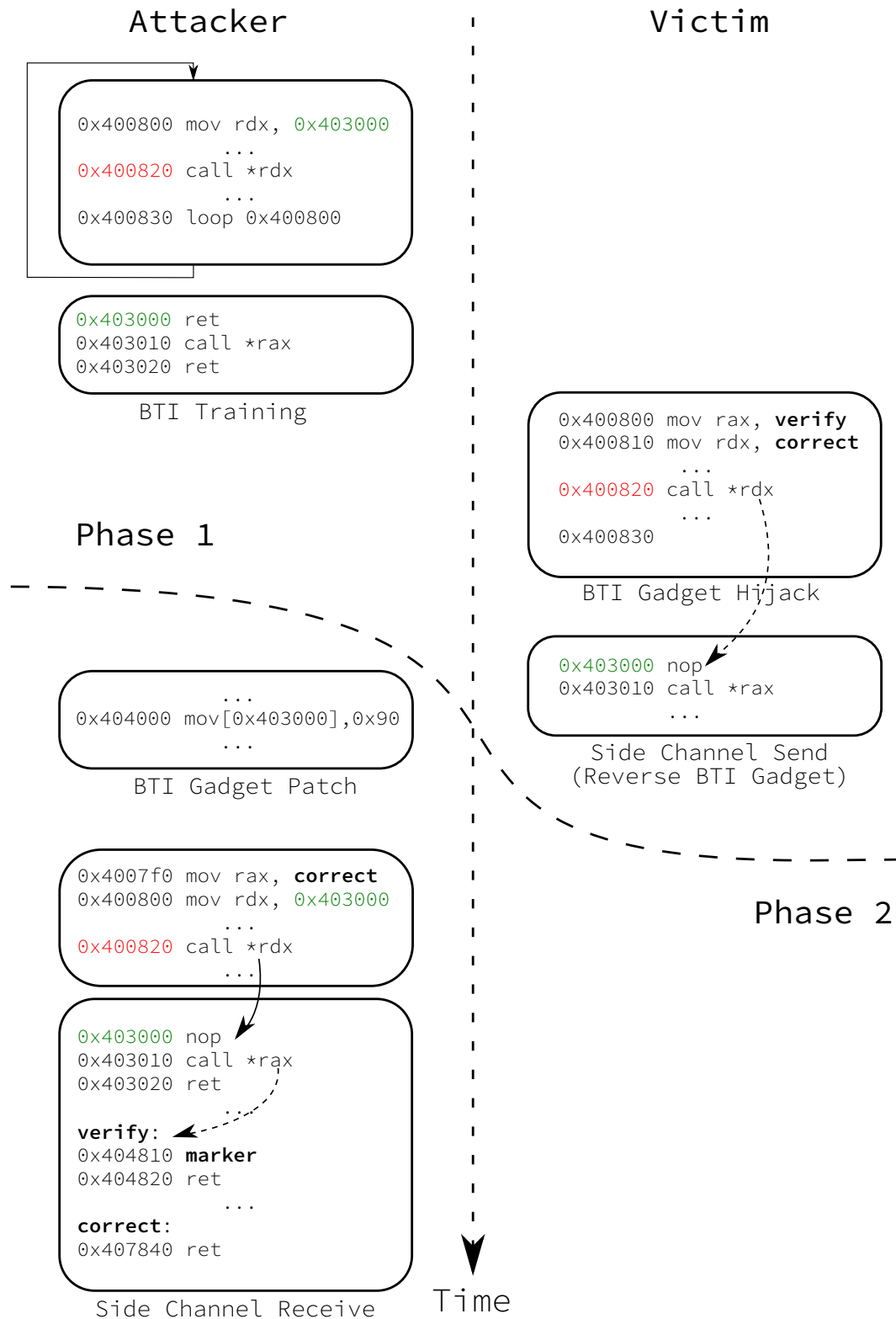


Figure 4.14: Description of the Double BTI attack: the attacker performs BTI at first; the victim speculatively executes the “reverse” BTI Gadget that further trains the branch predictor with the value of a register or a memory location; the attacker later execute the same “reverse” BTI Gadget and based on the side effects of wrong prediction (e.g., executing an instruction marker to a given location) can guess the value of the register or memory location

With the Double BTI attack I am able to lift this restriction, making speculative control flow hijack attacks far more pernicious. The intuition behind the attack is that the gadget implementing the *side-channel-send* operation may be instantiated as simply as by a second indirect call. Crucially, this indirect call will cause a second, “reverse” BTI, where this time the attacker is subjected to branch target injection. If the attacker is able to measure the effects of this second BTI and learn one or more bits of information about the injected target, the side channel is successfully read.

At a high level, the attack has 2 main phases: in the first phase, the attacker performs standard BTI and, whenever successful, causes the victim’s control flow to be (speculatively) hijacked to execute the reverse BTI gadget. This represents the *side-channel-send* operation. In the second phase, the attacker attempts to perform the *side-channel-receive* operation by observing the effects of the victim’s speculative execution. I can see the two phases in detail in Figure 4.14.

Phase 1

Phase 1 starts with the attacker training the BTB by repeatedly executing an indirect call whose target address is identical to the one of the reverse BTI gadget in the victim. The attacker can execute this either on the same thread or on a twin thread on the same physical core of the victim process. The gadget (identified in the figure as the BTI training gadget) the attacker calls into initially consists of a return instruction followed by a register-indirect call instruction that is never executed in this phase.

When the training is over and BTI is successful, I assume that the victim speculatively executes the reverse BTI gadget. The reverse BTI gadget is identical to the BTI training gadget in the attacker, save for the fact that it starts with a *nop*. The *nop* may be replaced in practice with any instruction that doesn’t disrupt the control flow and whose size still ensures that the indirect call in the victim’s reverse BTI gadget has the same address as the (so-far unexecuted) indirect call in the attacker’s BTI training gadget.

The reverse BTI gadget contains an indirect call which is speculatively executed. Crucially, my findings prove that the side effects caused on the BTB by its execution are not rolled back by the CPU. Further, I show that a single execution of the victim is sufficient to make this side effect persistent and observable. For these reasons, I can see the reverse BTI gadget as an implementation of the *side-channel-send* operation: if the information being sent depends on a secret of the victim, the attacker is later able to read it with a suitable *side-channel-receive* gadget in the next phase.

Phase 2

At this point phase 2 begins. In phase 2 the attacker “patches” its BTI training gadget by replacing the leading *ret* with a *nop*. This enables the attacker to perform the second indirect call without losing alignment with the victim and without requiring more complex gadgets to distinguish between training and measurement mode.

Subsequently, the attacker calls into the (now patched) BTI training gadget once more, finally executing the register-indirect call whose target was trained by the victim. If the victim’s training was successful, the attacker will not execute the code at the *correct* label but rather at the victim-trained *verify* label. This is because the CPU tries to predict the target of the call, and uses the history left from the victim execution. The attacker structures its address space to contain suitable speculative execution markers. Observing the side effects left by the marker corresponds to the

side-channel-receive operation.

4.5.4 Practical considerations

In my first proof-of-concept implementation, I instantiate the marker with a set of instructions that is measured by a specific Intel Performance Monitor Counter (PMC). The chosen event must be one that is triggered even if the responsible instructions do not retire. In practice I have chosen the failed store-to-load forward counter, which requires a sequence of 3 *mov* instructions. The performance counter related to the marker is incremented whenever the attack succeeds and the indirect call in phase 2 is speculatively redirected to the location trained by the victim. Clearly this technique is not applicable to a real-world setting since programming PMC counters requires root privileges.

I identify 2 realistic marker instances that implement a *side-channel-receive* operation. The first candidate uses instruction cache side effects. Assuming that the attacker knows the first 6 most significant bytes of *rax* and wants to discover the 7th, it would layout its address space by placing at each of the 256 possible addresses an icache-differentiable gadget. This gadget would in practice contain a suitable amount of nop padding to account for the content of the least significant byte and a call instruction to one of 256 different functions, followed by an *lfence* instruction to stop speculative execution. The attacker would speculatively execute one such gadget as the first part of the *side-channel-receive* operation, and then time the execution of all functions as second part of the *side-channel-receive* operation. If only one of the functions executes in less time than a pre-computed threshold, its ordinal number corresponds to the leaked byte. This approach suffers from a rapidly deteriorating signal quality, due to the noise induced in the instruction cache by the measuring process.

```
value0:
    mov rax, QWORD[array + 0 * 1024]
    ret
value1:
    mov rax, QWORD[array + 1 * 1024]
    ret
    ...
value255:
    mov rax, QWORD[array + 255 * 1024]
    ret
```

Figure 4.15: *side-channel-receive* approach using data cache access pattern

The second candidate uses data cache access as a measurable side effect. The setup is identical to the previous approach save for the fact that the 256 target functions each contain a different memory access (*load* operations on an array). When speculatively executed, this induces an effect in on the data cache, which can then be measured. This approach is described in Figure 4.15. With

this approach, the side channel signal maintains its quality throughout the measuring process and allows the attacker to extract a full byte from the *side-channel-receive* operation.

4.6 Gadgets analysis

4.6.1 Icache Attack

Experimental setup

I test the icache attack on an Intel Core i7-6700K CPU running Ubuntu 16.04.6 LTS, kernel version 4.15.0. The attacker and victim processes are co-located. The following system setup is in place: ASLR is off to ensure consistent virtual addresses for BTI training, scaling governor is set to *performance* for constant clock frequency. Clock frequency is set below turbo. On the speculative execution mitigation side, the default setup is in place – *spectre_v2* set to *auto* and *spectre_v2_user* set to *auto*.

The attacker process is timing the execution of target code that is shared between victim and attacker. The same icache (physical) tags allow the attacker to determine the exact path taken in the victim icache gadget. To enforce this behaviour, I test my attack on two different setups: in the first, the shared code resides in a POSIX shared memory region; in the second, the shared code is part of a shared library. For this second part, I test with both *libhttp-parser*, part of *nodejs* and *libcrypto*, part of *OpenSSL*.

Attacker and victim use lightweight synchronisation for higher BTI success rate. In practice, this synchronisation is not required as long as I can assume that the attacker is able to trigger the victim and can thus time its execution accordingly. To maximise the signal of the icache side channel I flush the cache lines that correspond to the target code area before each loop. Given that the shared gadget is dynamically mapped, the icache timing gadget in the attacker does not time a direct call but a register-indirect one.

Results and Discussion

The overall success rate of the experiment shown in Table 4.5 is above 80% for guessing either of the two secret bit values, which is well above the 50% random guess threshold. Therefore, the attack is successful. I compute the success rate per 100 runs to be the number of times the attacker correctly guesses the secret bit. I then compute average and 95%-confidence interval for the success rate by repeating this experiment 1000 times, and therefore collect a total number of 100k samples.

Table 4.5 shows results with a gadget chosen from *libhttp-parser.so*: in particular the chosen functions for *fun1* and *fun2* are 7 pages apart and are 29 and 870 bytes each. I obtain similar results for the other combinations (POSIX shared memory or different shared objects). The overall success rate is mostly dependent on how successful BTI is, with the BTI success rate itself varying from 70% to 90% over all my experiments. Each run collects one timing of the execution of the function *fun1* (with reference to Figure 4.13) corresponding to the function that the victim should speculatively execute in case of successful BTI and when the (secret) value of the condition register is 0. If the timing is below some threshold, the attacker guesses that the value of the secret is 0, and 1 otherwise. I determine the value of the threshold by timing the execution of *fun1* during a

Secret	Success Rate
0	80.84% \pm 1.37
1	97.29% \pm 0.11

Table 4.5: icache attack experiment with a gadget from *libhttp-parser.so*: each row displays the success rate in guessing the value of the victim’s secret. The success rate is computed as the rate between samples displaying an icache hit (resp. miss) when the value of the victim’s secret is 0 (resp. 1). An icache hit is defined as an execution of the icache gadget timed below a pre-determined threshold.

learning phase, building a distribution of timing samples for icache hits and setting a *hit threshold* as $ht = avg + 3 * \sigma$, where avg and σ are average and standard deviation of the distribution.

4.6.2 Double BTI Attack

Experimental setup

I tested my Double BTI attack on multiple Intel CPUs. On each machine, the attacker and the victim are co-located. In the PoC, the register (*rax*) that is the target of the indirect jump in the reverse BTI gadget is set as follows: the 3rd least significant byte is a secret value that the attacker wants to discover, prefixed by a (known) offset and suffixed by all zeroes. The prefix just ensures that the attacker can map its set of 256 markers at a non otherwise mapped location. In the PoC, the attacker uses Double BTI attack to learn the value of the secret byte. I use data cache timing markers as discussed in Section 4.5.4. During this experiment, the mitigations enabled against BTI are the default ones (see Section 4.7) enabled on a stock Ubuntu. In this attack, I do not employ any specific synchronisation between victim and attacker: the correct sequencing of the two processes is achieved simply by delaying the start of the victim by a suitable amount of time. I *clflush* the memory locations containing the indirect call targets to maximize the speculation window. With this setup, I measure the attack success rate over 1000 attempts to leak the unknown byte of *rax* by timing accesses to each of the 256 locations in the array that is filled by the corresponding markers (as described in Figure 4.15). I repeat this procedure 1000 for each 1000 attempts to create a statical distribution of the attack for each machine. The timing of the array is performed in non-linear order to avoid prefetching effects. The timing always reveals two different cases: either exactly one array location is below a pre-defined threshold (fixed at 80 clock cycles) or none is. The first case corresponds to a successful *side-channel-receive* operation.

Figure 4.16 shows the results of my experiments on different platforms. I can see that I have non-negligible successes on all platforms, with success rates peaking above 90% and, in average, never below 15%. Meanwhile, random guessing in this settings would result in a 1/256 probability of success. The quality of the side channel signal is excellent owing to the fact that the attacker performs both the initial (speculative) access followed in close succession by the timing of the array location accesses, yielding an extremely clean measurement environment.

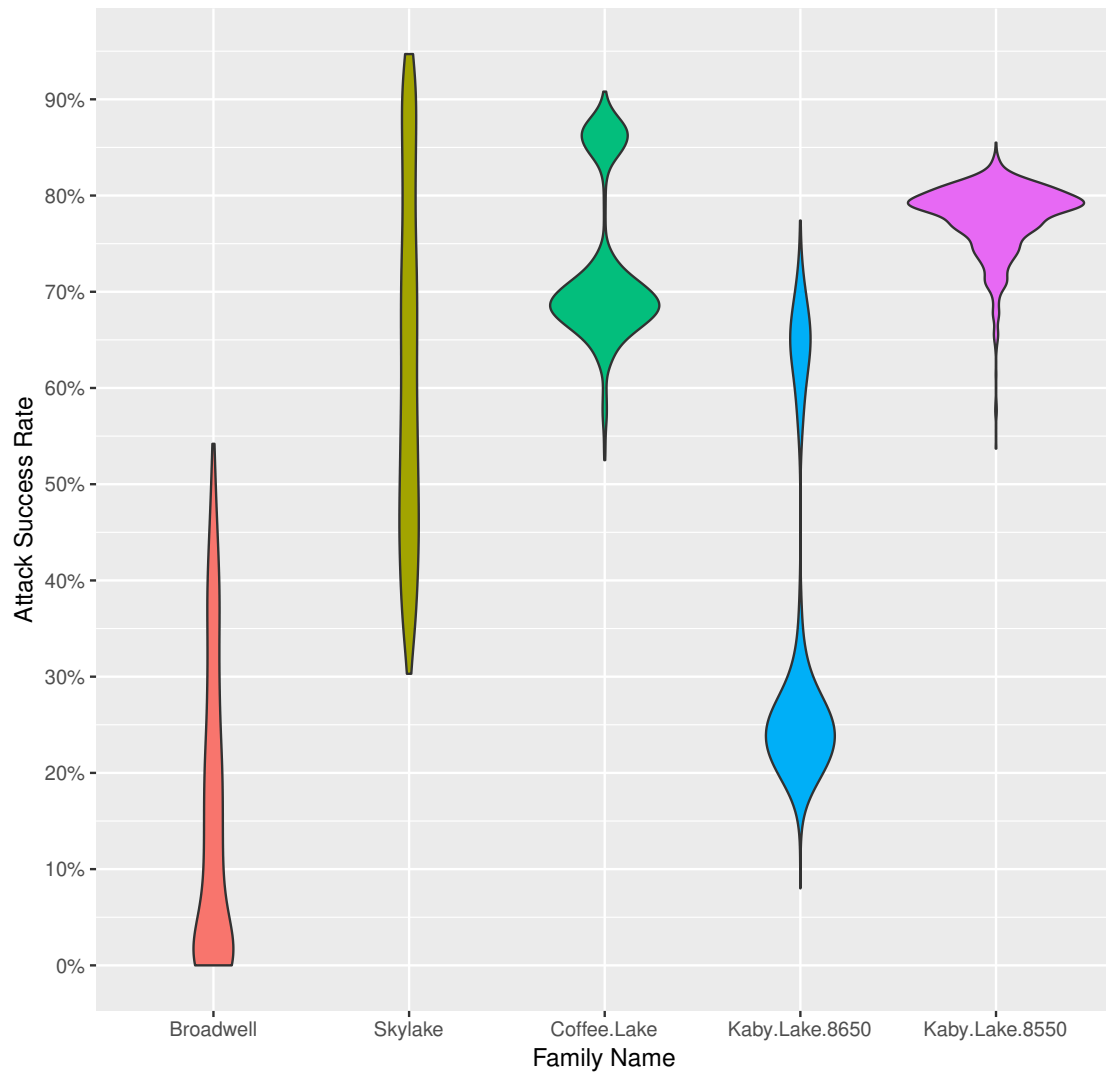


Figure 4.16: Double BTI attack success rate on leaking a one byte of secret

Distribution	Kernel	Generation Date	STIBP	Vulnerable?
Ubuntu 18.04.2 LTS	4.15.0-50-generic	May 6 18:46:08 UTC 2019	conditional	Yes
Ubuntu 18.04.2 LTS	4.18.0-18-generic	Apr 5 10:22:13 UTC 2019	conditional	Yes
Ubuntu 16.04.6 LTS	4.15.0-50-generic	May 8 15:55:19 UTC 2019	conditional	Yes
Ubuntu 18.04.2 LTS	4.19.0-041900-generic	Oct 22 22:11:45 UTC 2018	unsupported	Yes
Ubuntu 18.04.1 LTS	4.15.0-29-generic	Jul 17 15:39:52 UTC 2018	unsupported	Yes

Table 4.6: Default STIBP settings in the kernel used by the distributions tested in my evaluation

4.7 Mitigations

Both the icache and double BTI method presented here use BTI for speculative control flow hijacking. Therefore, BTI mitigations from Spectre v2 are applicable.

Mitigations are available at the hardware and software level to prevent BTI attacks. At the software level, compiling with retpoline [17] mitigates BTI by rewriting all indirect calls to avoid CPU prediction, through the use of a carefully crafted return sequence. At the hardware-level, Intel added Indirect Branch Restricted Speculation (*IBRS*), Indirect Branch Predictor Barrier (*IBPB*) and Single Thread Indirect Branch Predictors (*STIBP*). *IBRS* essentially flushes all branch predictor state when switching between user and kernel mode. *IBPB* essentially flushes all branch predictor state upon execution, even within a process. Finally, *STIBP* stops sibling SMT threads branch predictor from influencing the branch predictor decisions on other siblings threads on the same core.

I tested my attacks against the current implementation of BTI mitigations on the stock kernel 4.15.0 of my Coffee Lake machine. The kernel offers two switches to enable Spectre v2 protections. The first, *spectre_v2*, controls mitigations for protecting the kernel from userspace attacks, as well as functions as a master switch for enabling userspace protections. It can be set to *on*, *off* or *auto*. The option *on* and *off* forces respectively all the protection to be enabled or disabled. In my experiment, I left *spectre_v2* to *auto*, the default setting in recent Ubuntu distributions, to be able to enforce a finer grain control over the BTI mitigations and test functionality.

The second *spectre_v2_user* controls mitigations for userspace programs, and is gated by the previous setting. It can be set to *on*, *off*, *auto*, *prctl/ibpb* and *seccomp/ibpb*. As for the previous switch, *on* and *off* enable and disable all the protections. Meanwhile, *auto* defers the decision to enable or disable each protection and their mode based on additional configuration. Instead, both *prctl/ibpb* and *seccomp/ibpb* set *IBPB* always-on but leave conditional *STIBP* that has to be enabled on request by the process. For *seccomp* processes the restriction is enabled automatically.

Among those settings, my attacks are prevented if and only if *STIBP* is enabled (forced globally or the victim thread enables *STIBP* using *prctl*). Both attacks can also be prevented in software if the victim is compiled using retpoline. While non-SMT based BTI attacks can be mounted (i.e, attacker thread runs before and after victim threads, with two context switches), because of the enabled kernel mitigations flushing branch predictor state, these attacks do not apply.

Given the current performance penalties of enabling *STIBP*, this protection is set conditional by default or unsupported (as shown in Table 4.6) and therefore unless requested by the application, my attack is not mitigated. Furthermore, I verified that sensitive programs such as *passwd*, *sudo*

and *nginx* do not make use of the *prctl* interface to enable currently such protection. Given these default settings and the risks posed by BTI-related attacks, and in particular those presented in this dissertation, I recommend sensitive applications to enable STIBP through *prctl* when assuming local attackers.

Finally, other types of speculative control flow hijacks, i.e., return prediction based [32, 35] remain unaffected by these mitigations, and the two methods presented in this dissertation could be applied for those attacks as well.

Chapter 5

Impact of Spectre

Memory corruption vulnerabilities have plagued the computer security field for more than 30 years. Multiple ways of exploiting memory bugs have surfaced, requiring controls to be placed at different levels in the software stack: mechanisms such as stack canaries and control flow integrity have been designed and deployed as a mitigation in existing software, while new languages were designed with memory safety to close this class of bugs in new programs [75, 76].

Recently, the new transient execution attacks [77] class, and more specifically SEAs [10, 35, 31, 32, 36, 34, 33] have been the subject of intense scrutiny. The ensuing vulnerabilities appear difficult to mitigate without considerable performance trade-offs, leading to the conclusion that speculative execution attacks will remain a problem for the foreseeable future, and therefore a possibly fruitful area of research [78].

A natural question to ask is whether the advent of transient execution attacks has changed the security stance of modern computing systems against memory corruption attacks: does the security of memory safety mechanisms, such as stack smashing protection (SSP), control flow integrity (CFI), and those embedded in memory safe languages, hold in the post-Spectre threat model?

In this dissertation, I show that multiple memory safety mechanisms that would otherwise successfully prevent exploitation of vulnerabilities can be speculatively bypassed to perform arbitrary memory reads. Because these attacks require a combination of techniques, I show that they do not apply to all memory safety mechanisms and a careful, case-by-case analysis is necessary.

At a high level, these attacks work by overwriting, either architecturally or speculatively, a backwards or forward edge, followed by the use of speculative code reuse attacks to leak data. In all cases, this overwrite achieves a speculative control flow hijack, i.e., a redirection of the speculative control flow to an attacker-chosen arbitrary address. One case of such an attack is the *speculative buffer overflow* discovered by Kiriansky and Waldspurger [35], where a return address is speculatively overwritten.

I demonstrate that SSP, GCC’s vtable verification (VTV), and Go’s runtime memory safety checks are all vulnerable. In particular, I develop a practical attack against SSP, where the mitigations against a stack-based buffer overflow in *libpng* can be speculatively bypassed to read arbitrary bytes from the victim program. This attack additionally leverages a last level cache (LLC) eviction attack to extend the speculative execution window, and a speculative return-oriented programming (ROP) attack to achieve a Flush+Reload side channel by reusing 5 gadgets from the

victim program. Both components of the attack are not specific to SSP and generalize beyond my selected use case. My results demonstrate that, although such end-to-end attacks are not trivial to mount, they are realistic. For this reason, I evaluate countermeasures for each attack scenario, showing that mitigations are both effective and viable from a performance standpoint.

Other natural questions that arise while trying to understand the impact of these new attack is: how is it possible to verify if a system is vulnerable and understand which mitigations should be enabled given a threat model? Multiple community-developed tools are available to guide system administrators in the evaluation of their systems. I identified two main category of tools, the *empirical* and the *information gathering*. The empirical tools run PoC of the attacks on the machine under analysis and report if the attack has succeeded. The information gathering tools instead, collect information through the system (e.g., from the */proc* interface) about the available mitigations and software versions (e.g., microcode version). However, it is unclear to what extent can existing tools tell a system administrator whether a system is vulnerable to transient execution attacks. I provide the first analysis of such tools and shed light on their capabilities and limitations. Based on the experience gained, I proposed a hybrid tool, GHOSTBUSTER, that is meant to solve the pitfalls of the existing methodologies.

In this area, I make the following contributions:

- Demonstration of a practical attack against SSP-based buffer overflow mitigations, together with proof-of-concept attacks against GCC VTable Verification (VTV) and against Go's array bounds checks.
- Demonstration of speculation window lengthening leveraging LLC eviction of victim data.
- Practical speculative code reuse attack (ROP) to achieve side-channel send.
- Custom mitigations derived from *lfence*-based and masking-based approaches, withstanding the class of speculative architectural control flow hijacking attacks, together with a performance evaluation.
- I analyze transient execution vulnerability checkers, and report on identified shortcomings and pitfalls of these tools.
- I provide the first large-scale analysis for transient execution vulnerabilities on commercial cloud providers.
- I provide recommendations based on 6 different use cases to correctly test for transient execution vulnerabilities. As a result of my work, I provide a meta-tool, GHOSTBUSTER, that combines and enhances existing tools to avoid the pitfalls observed in the state-of-the-art.

5.1 Speculative execution attacks on memory safety mechanisms

In this part of my dissertation, I describe end-to-end speculative execution attacks on abstracted memory safety mechanisms. I begin with a high-level overview of the various components necessary to perform such an end-to-end attack. I then proceed to analyze the class of speculative

control flow hijacks which is at the heart of the attack; I refer to this general category of attacks as SPEculative ARchitectural control flow hijacks, or SPEAR, and detail them in Section 5.1.1. Furthermore, I analyze the eviction mechanism in Section 5.1.2, and the speculative ROP in Section 5.1.3.

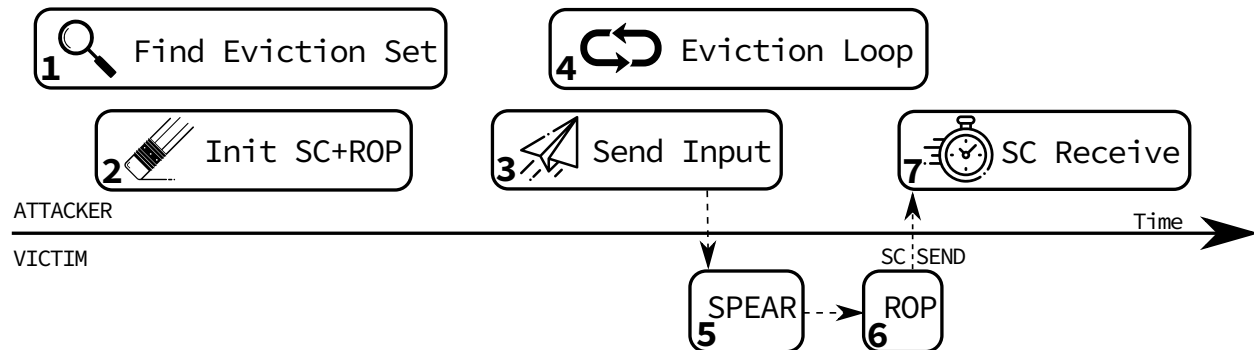


Figure 5.1: Overview of speculative attack against memory safety mechanisms.

Figure 5.1 shows an overview of the steps required to perform an end-to-end attack. The attack has a preparation phase (Steps 1 and 2), where eviction sets (to ensure the existence of a suitably long speculation window) are identified, memory used by the side channel is flushed and ROP gadgets are primed in the instruction cache. The attacker then submits an input to the victim in Step 3, crafted to trigger a violation of a memory safety property. I assume that traditional exploitation of the violation is prevented by a suitable memory safety mechanism. However, the attacker uses a speculative execution attack to bypass the mechanism by overwriting (architecturally or speculatively) control-flow data, and obtaining a speculative control flow hijack (Step 5). As a result, the victim is tricked into executing a side-channel send of attacker-chosen memory in Step 6: this is achieved with the ROP component, which reuses code snippets from the victim program, appropriately selected and primed in the initialization phase. The attacker can then execute the corresponding side-channel receive in Step 7. The success rate of the attack is increased by concurrently executing an eviction loop to lengthen the speculation window (Step 4) using the eviction sets found in Step 1.

Threat model: The general threat model for all attacks in this dissertation is a local unprivileged attacker, targeting a process holding a secret in memory. I do not assume that the attacker is able to inject code in the victim program’s address space. I assume the attacker has knowledge of the victim program code, as well as the virtual address of code at runtime as is the case for Go, or that they can recover this information if randomized, possibly by using microarchitectural side channels [79, 80, 73]. Finally, because I opt to use a speculative ROP payload, I assume a hyperthread-located attacker, thereby sharing the instruction cache with the victim program, which the attacker leverages during the ROP chain warm-up phase. The goal of the attacker is, as in all transient execution attacks, to leak secrets from the target program. Attacks based on the architectural overwrite of a backward or forward edge correspond to the case where an attacker can provoke a memory safety violation whose traditional exploitation is prevented by hardening mechanisms in place. This is demonstrated in the SSP and CFI use cases. In this case, I assume that the victim program can either be executed multiple times by the attacker or that the program automatically restarts, given that each attack run leaks a limited volume of information

and likely leads to abnormal program termination. This assumption remains realistic in practice because modern Linux distributions with *systemd* automatically restart services after abnormal termination.

Attacks based on the speculative overwrite of a forward edge correspond to a victim program with a memory safety check that the attacker can exercise and speculatively bypass. This is demonstrated in the Go use case. In this case, given that the overwrite of control flow data occurs speculatively, the attack does not lead to program termination, and so the attack does not require the ability to restart the victim.

5.1.1 SPEAR attacks

A SPEAR-vulnerable sequence is a code sequence that results in a speculative control flow hijack. A speculative control flow hijack allows an attacker to gain control of the target program’s speculatively-executed code. This is a powerful primitive: an attacker can follow such an attack with a speculative ROP sequence to speculatively execute code gadgets that access a secret and send it to the attacker via a side channel.

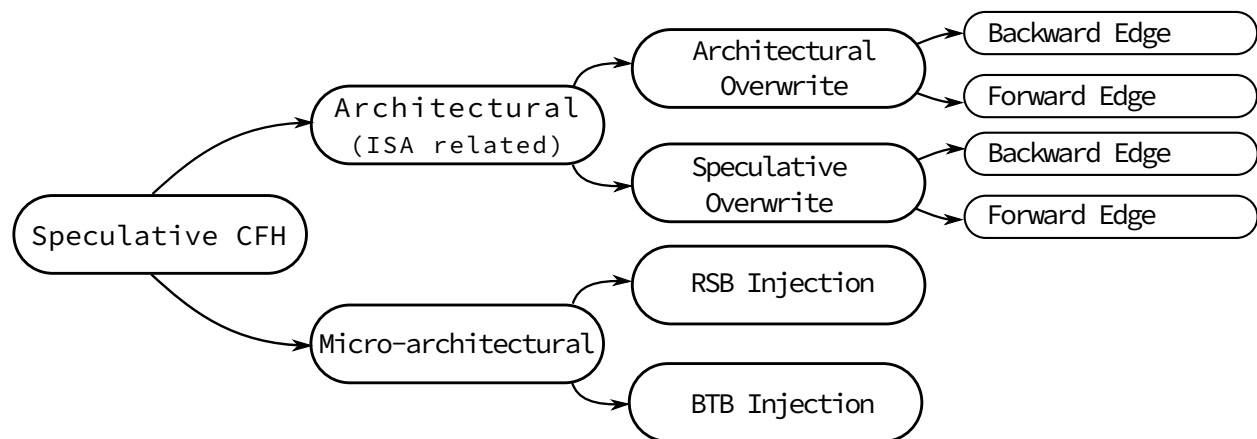


Figure 5.2: Overview of various Speculative control flow hijacking attacks.

Figure 5.2 shows a breakdown of the various instances in the SPEAR attack class in the context of different variants of speculative control flow hijacks. Classic speculative control flow hijacking attacks can be performed through microarchitectural components such as the Branch Target Buffer (BTB) and Return Stack Buffer (RSB) [10, 31, 32]. At the same time, the speculative control flow can also be influenced by instruction sequences that only affect architectural components, such as registers or memory: I refer to these as SPEAR attacks. For instance, executing the *call %rbx* x86 instruction speculatively when the value of *%rbx* is available to the execution unit will result in speculative execution continuing at the address in the *%rbx* register. Therefore, if the *%rbx* register can be controlled by the attacker, a speculative control flow hijack can occur. This control by the attacker can either be architectural or speculative, as I will see next.

Similarly, a *push %rbx; ret* instruction sequence with the register value available would also simply continue execution at the provided address, with no need to predict where speculative execution continues via the RSB. Hence, SPEAR-vulnerable code patterns can concern both forward edges (*jmp* and *call*) and backward edges (*ret*).


```

1 ;Copy of RET Value
2     mov rax,[rsp]
3     mov QWORD[stored_ret], rax
4
5 ;Architectural Overwrite
6 ; (Attacker Controlled)
7     mov rax, QWORD[target]
8     mov [rsp], rax
9
10 ;Evict RET Value Copy
11     clflush [stored_ret]
12     lfence
13
14 ;Backward Edge Integrity Check
15 ; (Speculation Trigger)
16     mov rax, [rsp]
17     cmp rax, QWORD[stored_ret]
18     jne my_exit
19
20 ;Backward Edge Hijack
21     ret

```

Listing 3: Architectural backward edge overwrite.

The SPEAR classification offers us a convenient way to reason on attacks triggered by control flow data overwrite. SPEAR covers all attack scenarios studied in this dissertation, namely, speculative bypass of memory safety mechanisms; in addition, it covers other known attacks, such as the speculative overwrite of a backward edge [35], and the speculative bypass of manually-inserted array bounds checks [81].

Architectural overwrite

The case where an attacker controls the control-flow-influencing register architecturally, i.e., via the Instruction Set Architecture (ISA), is closely related to traditional memory corruption attacks. These attacks can nowadays be mitigated by mechanisms such as stack smashing protection (SSP) and, in general, CFI implementations that check the validity of control flow metadata before control flow is transferred, thus detecting and preventing outcomes induced by attacker-controlled overwrites. SPEAR architectural overwrite attacks focus on the opportunity that the attacker has to speculatively bypass the checks introduced by these mitigations.

I provide in Listing 3 the snippet of code that illustrate the backward edge case for architectural overwrites. A similar snippet for the forward edge case is reported in Listing 4. The structure of both cases is similar: the original value of the edge (line 2) is preserved in a safe location, after which, I assume that the architectural overwrite is performed (line 5) with an attacker-controlled value (e.g., through a buffer overflow). Afterwards, the program executes an integrity check on the forward or backward edge (line 12) before performing the control flow transfer (e.g., SSP or CFI check). To increase the success rate of the attack, I try to maximize the speculation window caused by the integrity check, for instance by evicting its reference value – in the snippet, this

```

1  mov rax, [orig_target]
2  mov QWORD[stored_target], rax ;Copy of Target Value
3
4  mov rax, QWORD[hijacked_target]
5  mov QWORD[target], rax        ;Architectural Overwrite
6                                ;(Attacker Controlled)
7
8  clflush [stored_target]       ;Evict Target Value Copy
9  lfence
10
11 mov rax, QWORD[target]
12 cmp rax, QWORD[stored_target] ;Forward Edge Integrity Check
13                                ;(Speculation Trigger)
14 jne my_exit
15
16 call QWORD[target]            ;Forward Edge Hijack

```

Listing 4: Architectural forward edge overwrite.

```

1  ;Speculative execution trigger
2  ...
3
4  ;Speculative Overwrite
5  ; (Attacker Controlled)
6      mov rax, QWORD[hijacked_target]
7      mov QWORD[target], rax
8
9  ;Forward Edge Hijack
10     call QWORD[target]

```

Listing 5: Speculative forward edge overwrite

step is captured by a *clflush* instruction (line 8). If the CPU mispredicts the outcome of the check, it might execute either a *ret* (backward edge) or a *call* (forward edge) which transfers the control towards the attacker-controlled value used in the architectural overwrite (line 16).

Speculative overwrite

Alternatively, the attacker may control the control-flow-influencing register speculatively. This means that in a first phase, speculative execution is triggered (for example by a conditional branch). In a second phase, the attacker speculatively influences the control flow edge, thus hijacking speculative control flow. The control-flow-influencing value may be the result of a load from an address that is generated during the speculative execution phase, or it may be loaded from a location that is speculatively overwritten by a preceding store operation, resulting in speculative store-to-load forwarding.

I provide in Listing 6 the snippet of code that illustrate the backward edge case for speculative overwrites. A similar snippet for the forward edge case is reported in Listing 4. Both cases share the same structure. First, speculative execution is triggered by a condition (line 1). Then, the speculative overwrite is performed through some instruction within the speculated part of the code. Here, the value used for the overwrite is under attacker control (line 4). Finally, the overwritten value is used for control flow transfer allowing the attacker to hijack the speculative control flow (line 7).

Family	Architectural		Speculative	
	<i>Fwd</i>	<i>Bwd</i>	<i>Fwd</i>	<i>Bwd</i>
Intel Broadwell	99.5	94.9	99.5	98.7
Intel Skylake	97.6	98.3	98.2	92.1
Intel Coffee Lake	99.8	98.1	99.7	99.4
Intel Kabylake	99.5	95.9	100	99.5
AMD Ryzen	100	100	100	100

Table 5.1: Success rate (in percentage) computed over 1000 iterations for architectural or speculative overwrites of backward and forward edges performed on various architectures families.

SPEAR experimental results

I follow the methodology of Mambretti et al. [82] and test all four snippets using the Speculator tool [83], which aids the detection of speculative control flow transfers by using performance monitor counters (PMC) and *speculation markers*.

The SPEAR experimental results are shown in Table 5.1. Each success rate is computed on 1000 iterations. In the *architectural overwrites* case, speculative control flow hijacks are observed at least 95% of the time for Listing 3 and 97% of the time for Listing 4 on all tested architectures. The results prove that control flow is indeed speculatively transferred to the overwritten location, thereby bypassing the checks during speculative execution. Therefore, I conclude that SPEAR attacks with architectural overwrites can result in speculative control flow hijacks. In the *speculative overwrites*

```

1      ...                ;Speculative execution trigger
2
3      mov rax, QWORD[target]
4      mov [rsp], rax      ;Speculative Overwrite
5                          ;(Attacker Controlled)
6
7      ret                ;Backward Edge Hijack

```

Listing 6: Speculative backward edge overwrite.

case, for the backward edge case the success rate is at least 92% while for the forward edge case it is at least 98% . The experiment results demonstrate that speculative overwrites are feasible and lead to speculative control flow hijacks provided a sufficiently large speculation window exists to facilitate the edge overwrite followed by the dereference.

5.1.2 Speculation window and eviction

SPEAR attacks require the existence of a speculation window to permit the execution of the control flow transfer and the side channel send operation, a common precondition for all speculative execution attacks. This requires a speculative execution trigger, i.e., an instruction that causes a wide-enough window of dependent instructions that are executed but not retired. This is usually achieved when the process accesses uncached data: the speculation window then corresponds to the time for the access to main memory to complete. In Listing 4 for example, this is achieved with the *clflush* instruction. To show the necessity of a wide speculation window, I re-run the snippet without *clflush* in the Speculator tool and verify that indeed the control flow hijack only takes place in about one run out of 1000. When it does, the window is only a couple of instructions wide. I therefore conclude that without eviction, or other similar approaches to lengthen the speculation window, SPEAR attacks are unlikely to be practical.

In all snippets referenced by this section, the speculation window is artificially lengthened by flushing one of the memory operands of the compare instruction. This may not be realistic, as it imposes a strong requirement on the victim code to include a flush (or comparable) instruction. Instead, because the last level cache (LLC) is shared and often inclusive, the same effect can be accomplished more realistically by an external attacker thread computing an *eviction set* and performing a small number of accesses to addresses in this set.

An LLC eviction set competes for the same LLC slice and cache set as the target address to be evicted. Existing techniques for performing such attacks typically assume knowledge of the targeted physical address, as the LLC is physically indexed. As a consequence of rowhammer attacks, this is no longer realistic, as most OSes have removed access to physical mappings for unprivileged users. In Linux, privileged-only */proc/PID/pagemap* access [84] was introduced in release 4.0.

I demonstrate here that such eviction attacks can still be performed without knowledge of the physical address. To this end, I perform the eviction in two steps. The first step consists of the identification of an eviction set for a cache line in a page under the attacker’s control, by following the approach of Maurice et al. [85]. The second step consists in releasing this page to

the OS, and executing the victim process such that it reuses the previously-created page. This permits the reuse of the eviction set constructed and verified to be working in the first step. To increase the victim data eviction success rate, I follow the eviction set loading method proposed by Liu et al. [86]. I show details of such a practical attack in Section 5.2.1 for SSP.

5.1.3 Speculative ROP

To perform a complete speculative execution attack, the speculative control flow hijack must be followed by a *side channel send* gadget with a secret input. Unfortunately, Spectre v1-type Flush+Reload side channel send gadgets are known to be difficult to find [10, 4]. As in classical control flow hijacks [87], however, a speculative code reuse attack can be performed by chaining the speculative execution of gadgets to construct a Flush+Reload side channel send sequence.

To chain the gadget sequences, I proceed in a similar way to traditional ROP attacks, with sequences ending in *ret* instructions, yet with two additional requirements. These requirements for performing speculative code reuse are the following: *i)* execution of all instructions in the gadgets must fit into the speculation window; *ii)* all code pages in which the gadgets reside must be present and mapped in the victim process.

The first requirement is a consequence of the behavior of speculative execution. In particular, all return values used to chain gadgets need either to be in store buffers or in cache. Indeed, whether the return addresses are speculatively or architecturally written to the stack, execution of return instructions will make use of these addresses if they are available, with the CPU preferring those values for steering front-end fetches over values provided by the RSB. If the return address is not in cache (or in store buffers), loading the return address from memory will exceed the speculation window in practice and only RSB-based branch prediction will be in use, which will result in failure of the attack. A similar approach and analogy exists with forward edges for code reuse. Using the Speculator tool, I obtain experimentally that the maximum number of empty gadgets that fit in the largest speculation window is 20 on my Kaby Lake i7-8550 test platform.

The second requirement is needed to avoid page misses during gadgets execution. In the event of a page miss, speculative execution might halt or nested speculation might be triggered. Despite of the two strict requirements, I show in Section 5.2.1 that speculative ROP can be achieved for a practical use case.

5.2 Case studies

In this section, I analyze different case studies where memory safety mechanisms can be bypassed with SPEAR attacks. In particular, in Section 5.2.1 I use a practical attack that speculatively bypasses SSP leveraging architectural overwrites of backward edges. Section 5.2.2 analyzes architectural overwrites of forward edges, targeting two prominent CFI frameworks, GCC VTV and LLVM CFI. In the GCC VTV case, I show how the integrity check of the forward edge can be used to perform a speculative control flow hijack. For LLVM CFI, I conclude that the constraints of its implementation does not allow SPEAR attacks to be mounted in practice, demonstrating the importance of careful feasibility analysis. Finally, in Section 5.2.3, I demonstrate two types of speculative bounds check bypasses in the Go language using speculative overwrites of a forward edge. I show how the attacker may influence the control flow target through both a load whose

address value is attacker controlled and a load of a value that was speculatively overwritten by the attacker. I demonstrate practical implementation of speculative ROP and LLC eviction techniques as part of the end-to-end practical attack on SSP, i.e., I implement all the stages in Figure 5.1. I do not further re-implement them in the case of CFI and Go, where they would equally apply and where I focus instead on the central part of the attack as a proof of their feasibility, i.e., I implement only Step 5 in Figure 5.1. Therefore, the success rates reported below refer either to all the stages together for the SSP case (7.19%) or just the hijack stage for the Go (above 80%) and the GCC VTV (85%) use cases, hence the large difference.

5.2.1 Attacking stack canaries

Stack canaries are one of the earliest mitigations against buffer overflows [75], and are widely used to this day. Among the most broadly adopted implementations are LLVM’s and GCC’s Stack Smashing Protection (SSP) and Microsoft’s /GS. At a high level, stack canaries work by inserting a value (the *canary*) between stack buffers and control-flow influencing data on the stack, in particular the saved return value. The integrity of the canary is then checked prior to using the saved return value. Local stack variables are reordered such that buffers, likely to be overflowed, reside adjacent to the canary while code pointers remain further away. This way, contiguous overflows of local stack buffers can be detected by the integrity check. The chosen canary value is randomly generated once during process execution start, and stored in a safe location.

Each compiler performs the instrumentation differently but in essence the mechanics are identical with respect to SPEAR attacks; I therefore focus on the example of LLVM on Linux *x86_64*. Implementations consist of two distinct *instrumentation atoms*. The instrumentation atoms on my target system are shown in Listing 7. The first, the prologue SSP atom, is placed after the function prologue and local variable allocation, and is responsible for storing the canary value on the current stack frame. The second, the epilogue SSP atom, is placed before local variable deallocation and the function epilogue. It compares for equality between the global and local canary values; if the values differ, the `__stack_chk_fail` function is called, terminating the program. If the local canary value was not modified during function execution, the function returns normally. I show next that this particular comparison can be the target of a SPEAR attack.

SPEAR attack on LLVM-SSP

The pattern of the SSP instrumentation closely resembles that of Listing 3. Under my threat model, an attacker with a buffer overflow against a function protected by SSP can perform a SPEAR architectural overwrite attack of the return value of that function. I describe a practical attack targeting a version of *libpng* with a reported buffer overflow (CVE-2004-0597): the bug is not exploitable in the traditional way owing to the fact that the function is compiled with SSP. I show how a speculative adversary can exploit the SPEAR architectural overwrite to leak arbitrary secrets from the victim.

The attack proceeds as follows: in the first step, the attacker overwrites the saved return address of the victim function. In the second step, the attacker leverages a misprediction in the conditional jump of the canary integrity check, thus transiently executing a return to the previously overwritten return address. This PHT-based misprediction is achieved by the attacker in a way similar to Spectre v1, by executing the canary integrity check with an intact local canary

```

1 func:
2     prologue
3
4     ; Store canary on the stack
5     mov rbx, QWORD[fs:0x28]
6     mov QWORD[stack_canary], rbx
7     ...
8     body
9     ...
10    ; Check for corrupted canary, if yes fail
11    mov rbx, QWORD[stack_canary]
12    xor rbx, QWORD[fs:0x28]
13    je exit
14    call __stack_chk_fail
15 exit:
16    epilogue
17    ret

```

Listing 7: Stack canary check instrumentation example.

sufficiently many times. As discussed in Section 5.1.2, another requirement is that a sufficiently long speculation window exist. I achieve this by evicting the global canary from the LLC, as I show in Section 5.2.1. The attacker is then able to perform a side-channel send operation by constructing a speculative ROP chain to access a secret as I show in Section 5.2.1.

LLC eviction of the global canary

I apply the two-step method described in Section 5.1.2 for the eviction of the global canary from LLC, and thus from all cache levels by the property of inclusiveness of caches on the target platform. The global canary value is always stored at a fixed offset in a page: I use this property to find eviction sets for this particular offset by using the undocumented Intel LLC slice function reverse engineered by Maurice et al. [85].

The attacker process first identifies a page with a known eviction set and then unmaps it to be reused by the victim to store its canary. This is achieved with two processes under attacker control, as follows. At first, one of them maps a hugepage and enters a loop in which it brings an eviction set into cache and waits for feedback from the second attacker process. The latter in turn probes its own stack canary and reports back a success as soon as the canary is no longer cached.

Once the eviction set is identified, the attacker releases the page, which is now ready to be reused by the victim process to store its canary. The page release is done via *madvise* which instructs the system about the process memory behavior, in this case indicating that a certain memory range will not be accessed soon (*MADV_DONTNEED*). I manually craft the memory area released by the attacker in order to shift the target page frame in the right position in the kernel buddy freelist. I empirically verify that the reuse of a page frame for the victim canary page occurs with 100% success rate when attacker controls victim startup. When the attacker does not control victim startup, the success rate drops (but remains above 50%), because synchronization is more difficult and all processes in the system consume resources from the buddy freelist. Factors that

```
1 mov rax, secret
2 shl rax, 8
3 add rax, shared_array
4 mov rax, [rax]
```

Listing 8: Example of Flush+Reload gadget.

influence the success rate include the order of the page in the freelist, the “distance” between the release operation by the attacker and the request operation by the victim process. I note that I do not use any artificial synchronization mechanism between the victim and attacker, which makes this attack widely applicable.

While data eviction is a common part for speculative execution attacks, I adapt an LLC eviction technique only used previously in the context of side channels. Existing techniques for Spectre attacks evict large quantities of data from the caches, lowering success rate. For the SSP attack, this technique ensures the ROP gadgets executed speculatively remain in cache. Their eviction would result in the attack failing, because the RSB would be used to predict return location.

Speculative ROP

I now focus on building and using a speculative ROP chain that accesses a secret and leaks it through a side channel. I use the Flush+Reload cache side channel initially used by Kocher et al. [10], although other side channels can be used similarly [34, 36, 33].

In Section 5.1.3, I have identified two major constraints on the attack: *i*) a limited number of instructions can fit into the speculation window; and, *ii*) all code pages in which the gadgets reside must be present and mapped with corresponding TLB entries. In addition to these requirements, I note that gadget code, as well as any data accessed by gadgets, must be available in cache during speculative execution. Typically, this is not an issue in speculative execution attacks because the attacker can run several attack iterations as warm up phase to bring the required data in the cache, whereas this attack is *single shot*: the process terminates after each attempt and this is an additional requirement.

Concerning the first requirement, the Flush+Reload side-channel send gadget only requires a few instructions: there are sufficiently short gadgets available, and length is therefore not an issue in practice. For the second requirement instead, I create a tool to search for gadgets in code that was recently accessed by the victim program, for which pages are present and mapped in the victim process. The tool traces the victim process and collects all executed shared (library) code pages, which are then fed into an existing ROP gadget search tool, ROPgadget [88]. I run the tool on the victim program and find 26 mapped code pages within the 4 different modules used by the victim: *libc*, *libpng*, *libz* and *ld*. In total, the tool discovered 2096 gadgets, out of which 406 are candidates for building the side-channel send gadget. Per-gadget occurrences are shown in Table 5.2. Finally, to ensure that all gadget sequences are in cache, a hyperthread-located attacker performs a ROP chain warm up phase by executing the chain in close temporal proximity with the SPEAR attack.

I build a 5-gadget ROP chain using the ROP gadgets found by my gadget search tool. The chain is functionally equivalent to the Flush+Reload gadget shown in Listing 8. The chain accesses a

Gadget type	Occurrence
pop reg ; ret	262
mov reg1, [reg0] ; ret	69
shl reg, 8 ; ret	4
add reg1, reg0 ; ret	71

Table 5.2: ROP gadgets used for building Spectre v1 chain with their corresponding occurrences. The search space is a subset of *libc*, *libpng*, *libz* and *ld* executable pages, obtained by filtering out pages unmapped in the victim’s address space and pages without a valid TLB mapping.

target address computed using a secret byte value, as in the initial Spectre attacks [10]. Because Flush+Reload requires shared memory, I choose the target address to reside in such a shared memory area between attacker and victim, the first 16 readable and executable pages of the *libpthread* library. To leak one byte I use an array size of 256. To avoid prefetching effects during side-channel receive, I choose the element size to be 256, i.e., four cache lines. The total array size equals 256×256 bytes, 16 pages.

By splitting the Flush+Reload gadget in small sequences of instructions as shown in Listing 8, I easily find the required gadgets within the constraints of the attack. The ROP chain that I find and use in the attack is shown in Listing 9. This chain pops the addresses (controlled by the attacker) of the start of the 16 pages and of the targeted secret from the stack. Next, the secret value is loaded at line 8. The next speculative gadgets multiply the secret value by 256 and compute the target address. The last speculative gadget dereferences the target address, resulting in a load being issued during speculative execution. This eventually brings the value into the cache to be observed by the attacker. The whole chain therefore allows the attacker to implement a universal read primitive over the victim process speculatively, using a Flush+Reload attack and the attacker’s control over the stack.

```

1  libpng.so.3.1.2.5 : 0x7960
2      pop rdx
3      ret
4  libpng.so.3.1.2.5 : 0x7f0a
5      pop rsi
6      ret
7  libpng.so.3.1.2.5 : 0x128ec
8      mov eax, dword ptr [rsi] ; load of the secret
9      mov byte ptr [rdi + 6a], al
10     ret
11 libpng.so.3.1.2.5 : 0x9f4b
12     shl rax, 8
13     add rax, rdx
14     ret
15 libpng.so.3.1.2.5 : 0x9fde
16     add eax, dword ptr [rax]
17     add byte ptr [rdi], cl
18     xchg eax, ebp
19     ret

```

Listing 9: Flush+Reload gadget ROP chain.

Attack evaluation and results

The attacker targets the *libpng* version 1.2.5 which is vulnerable to *CVE-2004-0597* [89].

CVE-2004-0597 is a stack buffer overflow which allows the attacker to read *length* bytes in *readbuf*. Due to improper sanitization of *length*, a read larger than *PNG_MAX_PALETTE_LENGTH* is allowed in a stack buffer. The target victim is a program that receives a *.png* file and parses the file using the unpatched *libpng-1.2.5*. When building the victim target with stack canaries enabled, the compiler will instrument *png_handle_tRNS* with the corresponding prologue and epilogue SSP atoms. As expected, SSP protects *png_handle_tRNS* from exploitation by stopping execution before the function returns. However, using a SPEAR architectural overwrite attack, I can perform a speculative control flow hijack. During the SPEAR attack, the attacker feeds *.png* files of the legitimate length to train the pattern history table to bypass the stack canary check. Then, the attacker provides a *length* larger than *PNG_MAX_PALETTE_LENGTH* that overwrites the value of the return address to trigger the speculative ROP attack.

I confirm the attack works and leaks bytes at arbitrary, attacker-chosen addresses from the victim memory, on Intel Skylake and Coffee Lake with latest microcode updates, and on Ubuntu 16.04 and 18.04 (both with kernel version 4.15.0) with all default Spectre mitigations enabled. Namely, both setups include `__user` pointer sanitization and `usercopy/swapgs` barriers mitigations against *Spectre v1*. Moreover, default mitigations against *Spectre v2* are present (*retpoline*, *IBPB*, *IBRS_FW*, and *RSB filling*), excepting *STIBP* which is disabled on Ubuntu 16.04. I report the attack evaluation results on Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz (Skylake) running Ubuntu 16.04.6 with kernel version 4.15.0. As described in Section 5.1, the attack has an initialization phase where eviction sets are identified, memory used by the side channel is flushed and the ROP sequence is primed. Concurrently with the submission of the malicious

```

1 void /* PRIVATE */
2 png_handle_tRNS(png_structp png_ptr,
3                 png_infop info_ptr,
4                 png_uint_32 length)
5 {
6     ...
7     png_byte readbuf[PNG_MAX_PALETTE_LENGTH];
8     ...
9     if (png_ptr->color_type == PNG_COLOR_TYPE_PALETTE) {
10         if (!(png_ptr->mode & PNG_HAVE_PLTE))
11             {
12                 /* Should be an error, but I can cope with it */
13                 png_warning(png_ptr, "Missing PLTE before tRNS");
14             }
15         ...
16         png_crc_read(png_ptr, readbuf, (png_size_t)length);
17         png_ptr->num_trans = (png_uint_16)length;
18     }
19     ...
20 }

```

Listing 10: *libpng* vulnerable snippet related to *CVE-2004-0597*

payload, the attacker also runs the eviction loop to lengthen the speculation window by causing the eviction of the stack canary in the victim.

I measure the attack success rate as the number of times the attacker is able to correctly guess a secret byte from the victim memory space, per total number of runs. I report means over 100 runs with 95% confidence level. The end-to-end attack success rate is $7.19\% \pm 0.62$, for a single run. In practice the attacker does, as in most other Spectre attacks, re-run the attack as many times as necessary to improve its guesses and reach close to 100% success rate. Therefore, I compute the leakage rate based on the attack time, which is measured as the full duration of repeating the attack 100 times against the re-startable victim. The duration includes the restart time of the victim and the attacker execution time. The end-to-end leakage rate of victim bytes is 0.3 bytes per second (with all correct guesses), which I deem sufficiently high for practical use. Due to different *binutils* versions in the two distribution versions, I observe a slight leakage rate drop in the Ubuntu 18.04 environment.

For improving the success rate, and therefore improving the leakage rate of the end-to-end SSP attack, one needs to improve the success of each individual stage of the attack showed in Figure 5.1. In addition, the attacker may run too fast or too slow with respect to the victim (the attacker simply attempts to synchronize with busy loops), which can also lead to failure of the attack. I have verified such synchronization is successful in my PoC in 78% of cases. I already report in Section 5.2.1 results for the LLC eviction stage: 100%. Because changes to the victim can affect the success rate, measuring the success of other individual steps within the end-to-end PoC is very difficult. However, based on these numbers and experiments outside the PoC, I infer that the greatest area for improvement in the leakage rate should come from improving the ROP gadget phase (e.g., limiting cases where the gadget code is not in cache) and side channel receive/send

(e.g., limiting cache noise from eviction activity and other sources, or using another side channel).

5.2.2 Attacking CFI

Control Flow Integrity (CFI) of forward edges aims to protect the integrity of code pointers used in indirect calls and jumps. CFI implementations contain two main parts: instrumenting all indirect control transfers to check their validity at runtime, and classifying valid control flow transfers (typically using static analysis at build time). I analyze here two prominent cases: the GCC Virtual Table Verification (VTV) [90] mechanism to prevent *c++* virtual table corruption, as well as LLVM-CFI [91], a publicly available, low overhead, forward-edge CFI implementation. In the GCC VTV, I prove that a SPEAR attack is possible, while in the LLVM-CFI case I conclude that eviction-related considerations result in the speculation window being too short for practical exploitation. In particular, this case study demonstrates that I cannot conclude that SPEAR attacks apply equally to all implementations of memory safety-related defenses, and case-by-case analysis is necessary.

GCC VTV

In the GCC VTV implementation, for every call to a virtual function in the program, the compiler inserts a check to make sure that the pointer used for the indirect call belongs to the virtual table of the object. Such check is represented by a call to the function `__VLTVerifyVtablePointer` implemented in *libvtable.so* library. Within this function, the pointer is looked up from the table; if found, the function simply returns to the program which will perform the call, otherwise, it gracefully fails. If an attacker can successfully evict the cache line related to the variable the pointer is tested against, speculative execution is triggered during the evaluation of the check. In that case, the indirect call to the virtual function is speculatively executed and the code at the corrupted pointer is executed. At this point, the attacker has performed speculative control flow hijack and can mount a data exfiltration attack as described in Section 5.2.1.

In my proof-of-concept implementation of this attack, I artificially evict from all cache levels the variable related to the vtable of the object within the *libvtable.so* code. Then, I create a *c++* program that defines two different classes each containing one virtual method. The first class is my target for the forward edge overwrite. To verify whether speculative control flow hijack takes place, I instrument the program to read performance monitor counters and set the speculative control flow hijack target to contain a *speculation marker*. I use the second class to instantiate the object that is later corrupted.

After object initialization, I perform a vtable pointer overwrite in my victim object making it point to the vtable of the first class. Finally, I perform the virtual call for the control flow transfer which is instrumented by GCC VTV with a call to the integrity check inside the *libvtable.so* library. During normal execution, this overwrite is detected by the library which reports the corruption and prevents the control flow transfer by terminating the application. With a SPEAR attack as described here, I verify that control flow hijacking occurs in 85% ($n=1000$), demonstrating that a SPEAR architectural forward-edge attack is viable against GCC VTV. I note also that the redirection is performed to a vtable of a completely unrelated class, a case which should be prevented by VTV. A real-world attack would additionally require evicting the compare variable, for example

```

1 type slice struct {
2     array unsafe.Pointer
3     len    int
4     cap    int
5 }

```

Listing 11: Arrays in Go.

by using the same method as in Section 5.2.1, as well as a way of achieving a side-channel send for the attacker, as in Section 5.2.1.

LLVM CFI

The CFI solution implemented in LLVM uses function types as *equivalence classes*: an indirect call to a function of a different type than the one specified by the programmer is forbidden by the CFI instrumentation. This is achieved by placing functions of an equivalence class in a jump table, thereby having as many jump tables (whose addresses are carefully chosen) as equivalence classes. The instrumentation for indirect calls then consist in simply checking that the address of the target fall within the range of the jump table, and at the right alignment.

This range check can be seen as a check against a compile-provided constant value, using the address of the provided target. Both of these components are by design available and cached while performing this check: evicting the code that contains the range check would result in speculative execution stopping, and evicting the address of the target would result in the iBTB being used for speculative execution. In either case, a SPEAR attack would fail. The attack may be triggered without any attempt to artificially extend the speculation window, but, as demonstrated experimentally in Section 5.1.2, the resulting speculation window is rare and short, making such attacks unlikely to be practical. I conclude that LLVM CFI is in practice not vulnerable to SPEAR attacks.

5.2.3 Attacking memory safe languages

Most modern languages are designed to ensure memory safety. Instrumental to achieving this property are bounds checks for load and store operations into arrays. In this section, I show how bounds checks may be speculatively bypassed, allowing the transient execution of out-of-bounds load and store operations. I show under which conditions this leads to a SPEAR attack.

I focus in this case study on the popular Go programming language, runtime and compiler. I present two variants, one where data that influences a forward control flow edge is architecturally overwritten and one where a forward edge is speculatively overwritten. In either case, the attacker is able to achieve a speculative control flow hijack. I prototype both variants and show the conditions under which the attack succeeds at a rate exceeding 80%.

Before detailing the two attacks, I give a brief introduction to the way the Go compiler manages arrays and bounds checks. Arrays in Go are represented in memory as the *struct* shown in Listing 11. The address of the contiguous chunk of virtual memory backing the array is stored in *array*. The number of elements that *array* can hold (and implicitly the size of the memory

```

1  mov rcx, [array]
2  cmp [array+0x8], rax
3  jbe runtime.panicindex
4  mov rax, [rcx+rax*8]

```

Listing 12: Bounds check in Go.

```

1  array[index].function()

```

Listing 13: Load-based speculative control flow hijack code pattern.

chunk since Go is statically typed and the size of the elements is always known) is stored in *cap*. The current number of elements that have been stored in the array is stored in *len*.

Whenever an array access is performed in Go, the compiler will add appropriate bounds checks. This is achieved in the course of the compiler pass to translate the abstract syntax tree (AST) into the static single assignment (SSA) intermediate representation by adding an *IsInBounds* meta-operation before every array load or store. *IsInBounds* takes two arguments, the index of the current access and the length of the array, and drives a conditional jump either to the basic block that performs the array access if the index is between zero and length minus one, or a jump to a function that raises a *panic* otherwise.

IsInBounds is translated by later passes into a sequence of instructions similar to the one shown in Listing 12. The snippet shows a load from an array of integers: at first *rcx* is loaded with the address of the memory array, a compare instruction is issued between the index of the array access in *rax* and the array length at *array+0x8*. If the index is negative or not strictly less than the length, the code jumps to a call to the *runtime.panicindex* function. Otherwise the array access is performed.

The conditional jump generated by the *IsInBounds* meta-operation may speculatively execute the wrong jump target and perform a transient load or store operation out of bounds. I show two distinct code patterns, one leveraging a load and one a store, that may lead to speculative control flow hijack.

Load-based SPEAR speculative attack

The first pattern is shown in Listing 13. It represents an instance of a SPEAR-speculative attack and consists of an interface function call, where the interface is stored into an array of interfaces *array*, dereferenced at position *index*. Note that the array must be an array of interfaces so that calling the function is achieved by an indirect call. For the attack to be successful, I need *index* to be attacker-controlled and the attacker must be able to store the value of two pointers in the memory space of the target process at a known location.

The first condition is met whenever a process accesses an array using an index that is received as an external input. The second condition is very commonly met since programs store user-provided input for processing. Knowledge of the location of the stored pointers depends on the

```

1 type iface struct {
2     tab *itab
3     data unsafe.Pointer
4 }
5
6 type itab struct {
7     inter *interfacetype
8     _type *_type
9     hash  uint32
10    _      [4]byte
11    fun    [1]uintptr
12 }

```

Listing 14: Structs used by interface calls.

```

1 fake iface:
2 0x0000: <fake itab>
3 0x0008: 0x0000000000000000
4 ...
5 fake itab:
6 0x1000: 0x0000000000000000
7 0x1008: 0x0000000000000000
8 0x1010: 0x0000000000000000
9 0x1018: <CFH target>
10 ...
11 CFH target:
12 0x2000: <attacker code>

```

Listing 15: Memory layout in preparation for the exploitation of load-based speculative control flow hijack. The attacker fake *iface* starts at offset 0x0. The fake *itab* prepared by the attacker starts at offset 0x1000. The control flow hijack target is located at offset 0x2000.

memory area being used, and is aided by the deterministic nature of the Go allocator.

Without loss of generality, I describe the case where *function* is the first function defined by the interface. Exploitation proceeds as follows: first, the attacker prepares the memory structures that are used when an interface call is performed. The structures are shown in Listing 14, and are used by dereferencing the *tab* pointer from the *iface* struct and then calling into the *fun* array.

In preparation for exploitation, the attacker ensures that the memory layout of the target program contains a pattern similar to that shown in Listing 15. Assuming that the attacker wants to speculatively redirect the control flow to address 0x2000, the attacker creates a fake *itab* structure (in the example at 0x1000) such that the first entry in the *fun* pointer array points to the desired target. Then the attacker creates a fake *iface* structure (in the example at offset 0x0) such that the *tab* pointer points to the aforementioned *itab* structure. With the memory thus prepared, the attacker supplies the index into the array such that the resulting address (the base address plus index multiplied by the size of an *iface* structure) equals the fake *iface* structure (0x0 in my

```

1 array[index] = value
2 ...
3 interface.function()

```

Listing 16: Store-based speculative control flow hijack code pattern.

example). With the index thus set the program will call the *runtime.panicindex* function; however if the conditional jump of the bounds check is mispredicted, the dereference and subsequent indirect call will take place transiently. Note that, contrary to the case studies in Section 5.2.1 and Section 5.2.2, the attack is not necessarily “single shot”: if the program calls *recover*, the attacker might be able to execute the vulnerable sequence multiple times.

I prototype the attack to evaluate its effectiveness in a proof of concept. The proof of concept only aims to establish the feasibility of the attack: in particular I do not integrate into an end-to-end attack and refer to Section 5.2.1 for cache eviction and speculative ROP. The PoC contains the pattern of Listing 13 called in a loop to train the pattern history table and ensure that the bounds check conditional jump as strongly non-taken. The index used to access the array in the loop is in bounds during the training phase and is then set to the target index computed as described above in the last iteration.

To verify whether speculative control flow hijack takes place, I instrument the program to read PMCs during the execution of the loop, and set the speculative control flow hijack target to contain a speculation marker. The *runtime.panicindex* function is modified to read and persist PMC values for each execution.

This instrumentation permits us to verify that speculative control flow hijack indeed takes place. The success rate is influenced by several factors that I review here. The most relevant factor is the size of the speculation window, which is influenced by how quickly the correct jump target is determined. The speculation window is maximized if the variables used in the compare instruction that drives the jump – especially the array length – are not present in any of the levels of the cache. In order to get empirical evidence of this fact, I instrument the program with a *clflush* instruction right before the array dereference to ensure that the array length is not cached. In practice, an attacker may achieve the same result by performing cache eviction code sequences. However flushing the cache alone does not ensure a high success rate: this is because the array length is stored right after the base address of the array, whose address is loaded into memory as the first instruction of the dereference sequence. I verify that if the two memory locations belong to different cache lines, the speculation window is maximized. Another factor that influences the success rate is whether the target of the speculative control flow hijack is already in the instruction cache. I make sure that this be the case by insert a call to the marker function in the warm up phase before the loop. I report success rates exceeding 80% ($n=1000$) when the array length is flushed and is in a separate cache line as the base address on multiple platforms (Xeon CPU E5-2640, Core i7-8650U, Core i7-6700K) and different versions of the Go runtime (1.13.4, 1.12, 1.10.4).

Store-based SPEAR speculative attack

The second pattern is shown in Listing 16.

The pattern consists of a store operation of an attacker-controlled value at an attacker-controlled location into an array. The elements stored in the array must permit storage of a pointer. Smaller sizes would permit partial control over the speculative control flow hijack target. The pattern requires that the array store be followed by an interface call. The interface call does not need to be related to the array. It only needs to be in close proximity of the store operation so that it may still be speculatively executed. This pattern does not require any ability to perform preparatory store operations in the memory space of the target program. The pattern makes use of store-to-load forwarding, since the store in the array is used to (speculatively) overwrite a function pointer which is later (speculatively) loaded and called. This corresponds to the “speculative overwrite of forward edge” variant of a SPEAR attack.

The store part of the pattern consists of a speculative version of a “write-what-where” condition. It may be exploited in several ways to hijack the interface call: the most basic one would be to overwrite the *tab* pointer in the *iface* struct (see Listing 14). However this would either require the attacker to perform a set of preparatory stores identical to those discussed in Section 5.2.3, or it would restrict the freedom of the attacker to choose a target out of the existing interface pointers. Another strategy would be for the attacker to overwrite the *fun* pointer in the *itab* structure directly. These structures are stored in a non-writable virtual memory region. However, given that the store takes place speculatively, the attacker is able to bypass the write restrictions and overwrite the pointer. Therefore, I choose to prototype this simpler and more effective variant.

Exploitation proceeds as follows: at first the attacker speculatively overwrites the *fun* pointer in the *itab* of the interface that is later dereferenced. This is achieved, as the attacker controls value and index. The former is set to the address of the desired speculative control flow hijack target; the latter is set such that base array and index multiplied by the size of the array elements add up to the address of the *fun* pointer to be overwritten. As in the previous section, with the index thus set the program will panic; however if the bounds check is mispredicted, the store-to-load forwarding and subsequent indirect call will take place, achieving speculative control flow hijack.

I prototype the attack to evaluate its effectiveness employing a similar instrumentation as the previous section, with PMCs and speculation markers employed to identify successful runs, and a loop to set the predictor state. The success rate is similarly influenced by ensuring that the variables driving the conditional branch are not cached, and that the speculative control flow hijack target is in cache. Under these conditions, I report success rates exceeding 80% ($n=1000$) on the same platforms listed in Section 5.2.3.

5.2.4 SPEAR attack against Rust bounds checking

The implementation of Rust panicking mechanism is abundant of SPEAR speculative control flow hijacking patterns similar to those discussed in the Go case study (Section 5.2.3). Here, I examine the safety features employed by Rust for index expressions and demonstrate a proof of concept SPEAR attack against out of range access hardening.

In Rust, memory safety for index expressions is established during Mid-level Intermediate Representation (MIR) building, with static and dynamic arrays, slices and strings being subject to sanitization. At compiler level, index expressions are instrumented with bounds checks which prevent out of range access. However, similarly to the case of Go, CPU misprediction of bounds check outcome leads to speculative out of bounds access.

The attack targets the array index access followed by an indirect call in Listing 17 at line 9. To

```

1  const PADDING_SIZE: usize = 7;
2  pub type Fptr = fn(u64) -> u64;
3
4  pub struct Data {
5      _padding: [u64; PADDING_SIZE], //
6      buf: Box<[Fptr]>,
7  }
8  let data: Data = Box::new(Data { ... }); //
9  data.buf[index](); //

```

Listing 17: SPEAR speculative control flow hijacking target in Rust. The *index* value is attacker controlled. I assume that the attacker writes the CFH target in memory prior to the attack.

```

1  mov rsi, [index]
2  mov rax, [buf len]
3  cmp rsi, rax ;
4  jle ok ;
5  call <core::panicking::panic_bounds_check>
6  ok:
7  mov rcx, [buf]
8  mov rdx, [index] ;
9  ; Calls function pointer when index is in bounds
10 call QWORD[rcx+rdx*8] ;

```

Listing 18: Disassembly of rust index expression bounds check instrumentation.

trigger the panicking system, the array is accessed with an attacker controlled index which is out of bounds. Rust MIR instruments the array index access with a bounds check. I analyze the index expression bounds check instrumentation at Assembly level in Listing 18. The instrumentation starts with array length loading and comparison against the attacker provided index, at line 3. Depending on the comparison outcome, the execution proceeds with accessing the array element requested or aborting in case of in-bounds requirement violation.

When the comparison between index and length is slow (due to uncached operands), the CPU may mispredict the result and continue execution speculatively, on the wrong path.

In the PoC, the victim data structure is chosen such that the array length can be evicted prior to the attack. The array length is stored together with the array data pointer in *buf*. At line 8 the *Data* object is initialized using *Box*, therefore the object is placed on heap. This avoids Rust default stack allocation which lowers the array length eviction success. Furthermore, the eviction may affect attack critical data, like the *buf* data pointer. In the PoC, *Data* uses a large enough padding so that the array length and the data pointer land on different cache lines.

The *buf* length eviction triggers mispeculation of the jump direction taken (Listing 18, line 4). Inside the speculation window, an out of bounds array access with the attacker controlled index leads to reading a function pointer from an attacker-owned memory area. Subsequently, the attacker controlled function pointer is the *call* instruction destination (line 10), therefore facilitating speculative execution of attacker chosen code (in this case, a speculation marker). Despite of the CPU rolling back the speculative execution effects on registers and memory, I use Intel Performance Monitoring Counters for counting speculation marker hits. I carry out the experiments on an Intel Skylake machine running Ubuntu 18.04. I measure an overall success rate of 90% ($n=1000$) for the SPEAR attack against Rust bounds checking mechanism. As for Go and GCC VTV, this success rate refers to the hijack phase only.

5.3 Mitigations against SPEAR

In this section, I implement and analyze serializing-based (*lfence*) and masking-based mitigations for SPEAR-architectural attacks (SSP) in Section 5.3.1 and SPEAR-speculative ones (Go) in Section 5.3.2. I show that in both cases the masking-based solution results in a low overhead. Finally, I discuss possible mitigations for GCC VTV case in Section 5.3.3.

5.3.1 Mitigations for SSP

I investigate two possible mitigations for the SPEAR-architectural attack against SSP. A serializing instruction such as *lfence* can be inserted after loading the canary in the epilogue instrumentation, thereby ensuring that the comparison can only lead to a short enough speculation window. Alternatively, the return value can be masked architecturally with a generated value that is set to 0 when the check fails (the canary is corrupted), and all ones when it passes, as shown in Listing 19.

I implement both mitigations as compiler passes in *clang+llvm*. The masking-based mitigation implementation is an extension of Speculative Load Hardening [38]. SSP is architecture specific, therefore my solution is built for *x86_64* Linux systems. I run the SSP mitigations benchmarking on Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz. I measure the normalized runtime of both *return address masking* and *lfence* on SPECint CPU 2006. The normalized runtime is computed as runtime

```

1  mov rax, QWORD[fs:0x28]
2  mov rcx, QWORD[stack_canary]
3  xor rdx, rdx
4  cmp rax, rcx
5  setne dl
6  add rdx, 0xffffffffffffffff
7  and QWORD[rsp + 8], rdx

```

Listing 19: Masking mitigation sequence; *rax* contains global canary value and *rcx* contains the stack canary; *rsp+8* points to the return address.

over the baseline runtime constituted by execution with SSP Disabled. For reference, I additionally plot the normalized runtime for all existing SSP implementations, SSP Loose (*-fstack-protector* flag), SSP Strong (*-fstack-protector-strong* flag), and SSP All (*-fstack-protector-all* flag).

The results are shown in Figure 5.3a. The *lfence* mitigation shows a high overhead in 9 out of 12 benchmarks, the highest being 100%, in the SSP All case with *xalancbm*k. Return address masking incurs a significantly lower, albeit still not negligible performance penalty, reaching a maximum of 13% for the same benchmark.

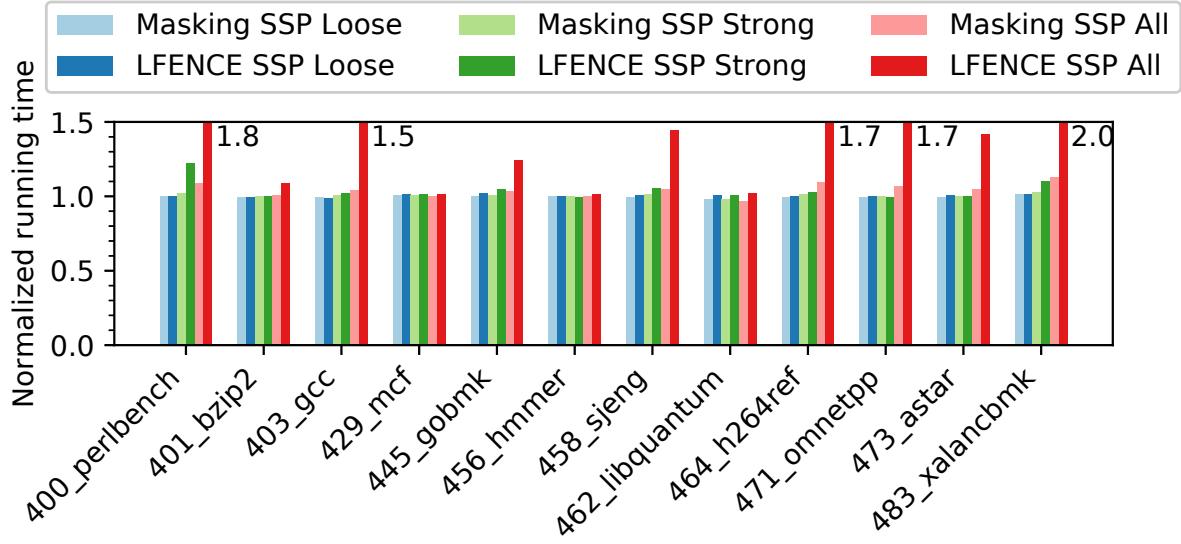
Based on this evaluation, I find the return address masking mitigation to be viable and superior to the *lfence* mitigation: the overhead of vanilla SSP (shown in Figure 5.3b on SPECint CPU 2006 is at most 9%, in the case of SSP All on *xalancbm*k). In addition, I note that most Linux distributions either use the SSP Loose or SSP Strong options, both of which incur a low overhead on all SSP benchmarks: I record a maximum of 2.1% overhead over the SSP Disabled baseline. With return address masking, the maximum overhead becomes 2.7% over the SSP Disabled baseline. I conclude that return address masking does not impose a significant overhead with the most commonly used SSP compiler options.

5.3.2 Mitigations for the Go compiler

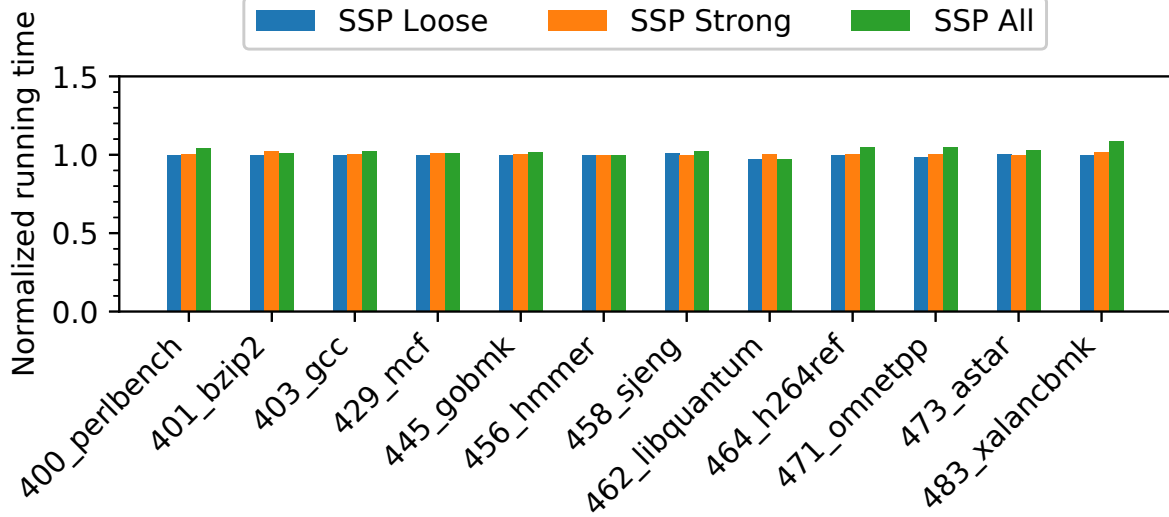
I investigate possible mitigations for the SPEAR-speculative attack on Go. The mitigations consist of two different compiler passes that ensure that the vulnerability is no longer exploitable. The first is based on *lfence*, whereas the second is based on branchless index masking sequences. As part of responsible disclosure I have notified the Go team, who have implemented 2 families of compiler-based mitigations for Spectre, namely, index masking (through the *-spectre=index* compiler switch) and retpoline (through the *-spectre=ret* compiler switch).

The feature was released as part of Go 1.15 [92].

The first mitigation consists of adding an *lfence* instruction after the *cmp* instruction in the sequence that implements the *IsInBounds* meta-operation. With reference to Listing 12, the *lfence* instruction is inserted after the *cmp* on line 2. The insertion ensures that all prior instructions have completed, which means that there will be no misprediction of the branch target and any out-of-bound access will result in a panic with no transient execution. The instruction is added explicitly in the pass that translates the AST into SSA form by defining a new *Lfence* meta-operation and adding it after each *IsInBounds* operation. I ensure that the operation is neither reordered nor eliminated.



(a) SSP with speculative bypass mitigations.



(b) Vanilla SSP.

Figure 5.3: Overhead computed as normalized runtime over SSP Disabled baseline.

```

1  cmp    rcx, rdx
2  jae    <raise-panic>
3  mov    rbx, rdx
4  sub    rdx, rcx
5  sbb    rcx, rcx
6  and    rcx, rbx
7  shl    rcx, 0x4
8  mov    rax, [rax+rcx*1]

```

Listing 20: Masking mitigation sequence; *rdx* contains the index and *rcx* contains the length of the array and *rax* contains the base address of the array.

The second mitigation I investigate entails the addition of an appropriate masking sequence that ensures that the index is set to a “safe” value in case of out-of-bounds accesses. The masking sequence amounts to a no-op in case the access is in bounds by performing an *and* operation on the index with a sign extended -1 mask. If the access is not in bounds, in my implementation, the masking operation forces an access of the element at index 0 in the array by performing an *and* operation on the index with a 0 mask. I can see the masking sequence in Listing 20: after the usual *cmp* and *jmp* instructions, length and index are subtracted in order to set the carry flag. Then, the *sbb* instruction is used to set a register to -1 in case of an in-bounds access or 0 otherwise. The array is subsequently accessed after performing an *and* operation on the index with the mask thus obtained. The pattern might be further optimized by using the *cmp* instruction of the bounds check to set the carry flag. This, however, is not always possible since the compiler will use a compare instruction with an immediate whenever possible. The immediate can only be the second source operand, forcing the direction of the comparison instruction. For the sake of simplicity I therefore rely on an extra subtraction operation. The masking instruction sequence is added by defining three new meta-operations – *OpMaskStep1*, *OpMaskStep2* and *OpMaskStep3* – which are later lowered into a *sub*, *sbb* and *and* instruction, respectively.

I measure the overhead of both mitigations by building the Go runtime version 1.12.0 and running the full benchmark suite. I run the experiments on a 40-core Xeon E5-2640 machine with 64 GiB of RAM. Figure 5.4 displays the empirical cumulative distribution function of the overhead of each of the two mitigation strategies. I can see how the *lfence*-based approach incurs a high overhead (143% mean and 84% median) due to the fact that *lfence* will terminate any speculative execution and thus severely curtail the instruction throughput. On the other hand, the masking approach shows a much lighter overhead (12% mean and 6% median) since the instructions involved are simple and do not cause any memory-related operation.

5.3.3 Mitigations for GCC VTV

The same mitigations considered in Section 5.3.1 and Section 5.3.2 work in the GCC VTV use case. Serializing mechanisms (e.g., *lfence*) are a viable solution, albeit likely with high overhead. A branchless masking solution or retpoline could also be used in this context for what I expect to have better performance, however I did not implement these.

I believe a better approach, from a performance point of view, for GCC VTV would be a

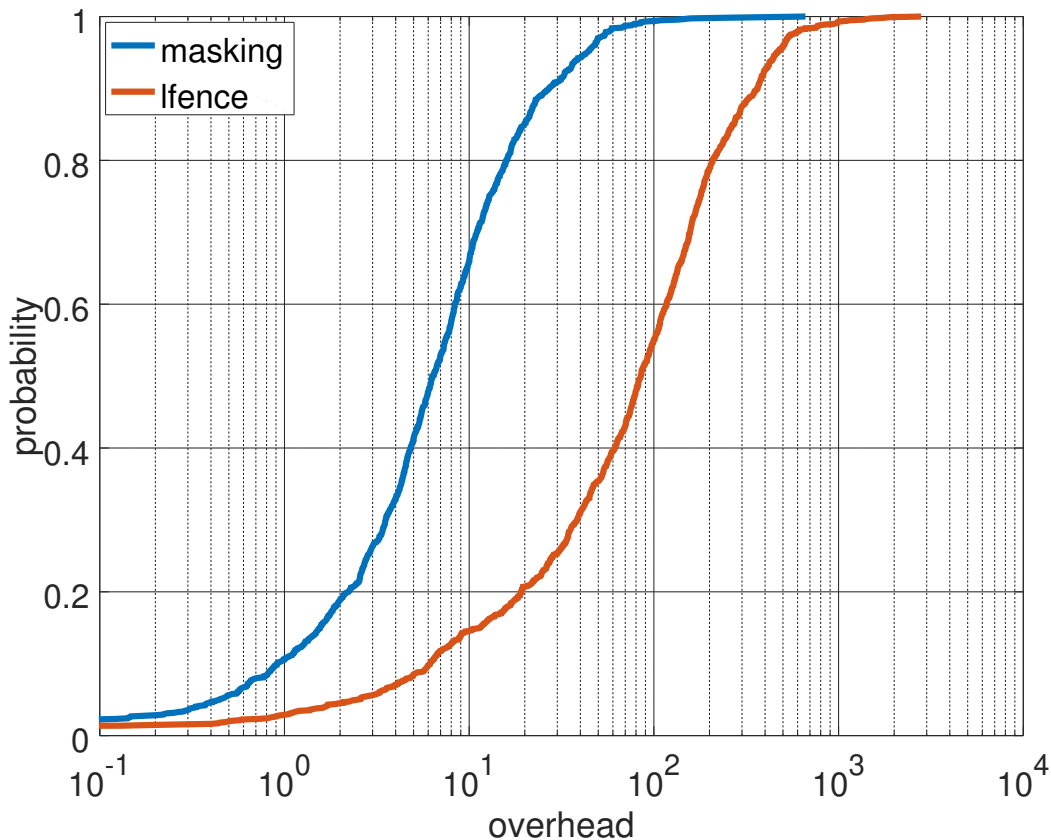


Figure 5.4: Empirical CDF of the logarithm of the overhead percentage for the considered mitigations. Overhead data is gathered by running the full set of benchmarks of the Go runtime version 1.12.0.

re-design with the principles observed for LLVM CFI described in Section 5.2.2 where the metadata and the pointer that have to be verified co-exist within the same cache line. This condition prevents the attacker to achieve the correct data eviction and, consequently, the speculation window to perform the attack is too small.

5.4 Discussion on SPEAR

Applicability to other use cases.

Beyond the highlighted use-cases, SPEAR attacks may be employed against other targets. For example, other memory-safe languages may be targeted with SPEAR attacks to speculatively bypass bounds checks as I show for the Go programming language. Preliminary investigation suggests that this is likely to be possible, since instruction sequences for bounds checks similar to those detailed in Section 5.2.3 are also present in *Rust* and *Java* (for JITted blocks). I analyze in more detail the Rust use case and report my findings in Appendix 5.2.4.

Theoretically, any security check that directly or indirectly gates a control flow transfer may be turned into a SPEAR attack. For instance, all the heap hardening mechanisms that verify the integrity of the heap metadata and pointers within *libc* can potentially lead to one of the SPEAR variant through the speculative use of a corrupted data to decide the application control flow. However, as demonstrated in the LLVM CFI case, a case-by-case analysis is necessary to establish whether SPEAR attacks are applicable.

Data leaked in SPEAR-architectural attacks.

SPEAR attacks allow an adversary to leak sensitive information from the victim address space. In the case of SSP, I demonstrate that arbitrary memory can be leaked, one byte per iteration. While I can target any memory location, I cannot target data that is not deterministic across runs. In particular, I cannot target to leak the stack canary, given that its value is re-randomized at every program start. I note that SPEAR-speculative attacks do not have this constraint, given that they do not require a program restart.

General applicability of speculative ROP.

The speculative ROP and LLC eviction techniques are demonstrated as part of the SSP, SPEAR-architectural overwrite of a backward edge, use case. Nevertheless these techniques are generally applicable for the exploitation of other SPEAR use cases, with exploitability always depending on the scenario at hand. For the general forward edge cases, I note that this requires, as in classical ROP attacks, a technique known as a stack pivot, which consists in the attacker setting up a fake return stack somewhere under its control in memory, and having the first control flow hijack point to an instruction setting the stack pointer to that address (for instance, the *push rax; pop rsp; ret* stack pivot gadget). Using the Speculator tool, I verify that such stack pivots do work for SPEAR-architectural as well as SPEAR-speculative attacks.

General applicability of LLC eviction.

In my end-to-end attack over SSP, I employ a new more precise LLC eviction technique which is described in details in 5.2.1. The necessity for developing my own, more precise, LLC eviction technique stems from the fact that my attack poses two additional requirements. The first is the fact that I require the eviction process to be very selective, since I cannot allow elements such as the addresses injected on the stack or the gadgets code to be evicted because that will stall speculative execution and prevent the completion of the attack. The second is that the eviction process needs to complete within a short amount of time to avoid the scenario where the line containing the canary is first evicted and then re-cached by the natural execution of the victim while the eviction process completes. With my technique, I can keep the number of possible cache-sets as small as possible and therefore minimize the length of the eviction process. I explore an existing LLC flush method discovered by Oren et al. [93] which could potentially fit the second requirement. However, I conclude that this method is too intrusive in a setting where the attacker relies on cached data and code (victim secret, ROP gadgets) available in the speculation window.

Disclosure

I submitted the PoC exploits and my findings to the Go security team on November 22nd, 2019. As a result of my notification, the Go security team has deployed hardening measures (index masking and retpoline) which were released in Go 1.15.

5.5 Testing Tools

In this last part of my dissertation, I analyze and use the 4 state-of-the-art tools to discover if a system is vulnerable to transient execution attacks. I divide the tools into two main categories: *information gathering* and *empirical*.

5.5.1 Information gathering tools

These tools use the kernel, the *cpuid* instruction and the microcode version as sources to collect the necessary information. The kernel provides information about Spectre and Meltdown vulnerabilities through the *sysfs* virtual file system interface. They use the *cpuid* instruction to determine if the CPU supports mitigations such as IBRS. Due to the differences among the Linux distributions and the way the necessary pieces of information are presented by the several Linux kernel versions, these tools must constantly be adapted to parse information correctly on all the different distributions.

The *spectre-meltdown-checker* [94] script was released soon after the disclosure of Spectre and Meltdown. This script inspects the local system for information related to transient execution patches. The *spectre-meltdown-checker* script even understands if the kernel space is hardened against transient execution attacks by also disassembling the kernel image and counting the number of *lfence* instructions. The tool is constantly updated by the community with tests for all the newly disclosed attacks. It is considered the state-of-the-art tool for verifying the system status.

The *mdstool-cli* [95] tool was published together with the disclosure of the RIDL vulnerability to the public. It aims to detect if the system is vulnerable to such attacks and previously discovered ones. As for SPECTRE-MELTDOWN-CHECKER, MDSTOOL-CLI inspects locations inside the system such as *sysfs* and the results of the *cpuid* instruction. While more recent, MDSTOOL-CLI is less exhaustive compared to SPECTRE-MELTDOWN-CHECKER and only checks whether the CPU self-reports itself as vulnerable to a specific class of attacks. Mitigations status and availability information are also gathered but they are not factor in the final report.

5.5.2 Empirical tools

As the name suggests, these tools use an active approach to assess if the machine is vulnerable or not to transient execution attacks. The tools under this category run several tests to verify the presence of the attack vectors of each of the known transient execution attacks. Each test emulates a specific attack and determines whether the transient execution attack is feasible. Two methods can be employed to make the final determination: either cache side channels or Performance Monitoring Counters (PMCs).

The approach with PMCs is less noisy than the cache-based ones and provides a more accurate determination about the presence or absence of the attack vector. The main drawback of using PMCs is their limited availability on virtual machines: they are rarely virtualized.

Transient Fail was released by Canella et al. [9]. It includes a series of empirical tests/PoCs that cover all the known transient execution vulnerabilities. This tool attempts to trigger the vulnerabilities locally and determines whether the attempt is successful by observing micro architectural side effects on the cache. Based on those, it is possible to infer whether a specific attack vector is present in the tested system.

Speculator [82] leverages CPU performance counters to evaluate speculative execution. I extend SPECULATOR with tests for each known transient execution attack since they are not available in the original version of the tool. I use *markers* presented along with SPECULATOR [82] as signals to verify whether the attack was successful. This mode of operation differs from the one used in TRANSIENTFAIL that relies instead on known cache side-channels (e.g., *Prime+Probe*, *Flush+Reload*). While Das et al. [96] suggest that PMCs should not be used in security applications to detect attacks, Mambretti et al. [82] prove instead that they can be reliably used to monitor tests results in the context of transient execution attacks. Moreover, SPECULATOR implementation carefully follows Das et al. [96] guideline to eliminate common mistakes in PMC usage.

```

1 CVE-2017-5715 aka 'Spectre Variant 2, branch target injection'
2 * Mitigated according to the /sys interface: YES
3   (Enhanced IBRS, IBPB: conditional, RSB filling)
4 * Mitigation 1
5   * Kernel is compiled with IBRS support: YES
6     * IBRS enabled and active: YES
7   * Kernel is compiled with IBPB support: YES
8     * IBPB enabled and active: YES
9 * Mitigation 2
10  * Kernel has branch predictor hardening (arm): NO
11  * Kernel compiled with retpoline option: YES
12  * Kernel supports RSB filling: UNKNOWN
13    (kernel image missing)
14 > Status: NOT VULNERABLE (IBRS + IBPB are mitigating the vulnerability)

```

Listing 5.1: Sample output from a commonly used tool to check vulnerability of a system against transient execution attacks

At the time of my analysis, SPECTRE-MELTDOWN-CHECKER and MDSTOOL-CLI were last updated at the end of May 2019, while TRANSIENTFAIL on August 2019. Another tool, called SafeSide [97], is under development with the same goal and design of TRANSIENTFAIL. However, due to the early stages of the tool when my experiments were run and its great similarity with TRANSIENTFAIL, I decided not to include it in my comparison.

5.6 Methodology

In this section, I describe the methodology I follow to evaluate the tools. At first I describe a few meaningful contexts in which the tools may be run: I focus on 6 practical use cases in which a

CPU Family	Kernel
Intel Ivy Bridge	4.15.0
Intel Haswell	4.15.0
Intel Broadwell	5.0.0
Intel Skylake	4.15.0
Intel Kaby Lake	4.15.0
Intel Kaby Lake R	4.15.0
Intel Cascade Lake	4.15.0
AMD Ryzen	4.15.0
AMD Ryzen 2	4.15.0

Table 5.3: CPU families the tools have been tested with the corresponding kernel version

system administrator may want to execute one or more of the tools to measure the degree of security of their system against a speculative adversary, keeping the use case of that system into account. I also describe the systems in which I tested the various tools.

5.6.1 Use Cases

This section details the use cases I considered to evaluate the impact of transient execution attacks. I categorize use cases based on the security domain at which attacker and victim operate. The considered cases are the following.

- *Sandbox to process (S-P)*: I consider an attacker running sandboxed code, such as Javascript, eBPF, or NaCL [98], aiming to leak data from the process that runs the sandbox.
- *User to user (U-U)*: I consider an attacker controlling an unprivileged process, targeting a privileged process such as a process running as *root* on the same machine.
- *User to kernel (U-K)*: I consider a local attacker targeting an OS kernel.
- *VM Guest to Guest (G-G)*: I consider an attacker that controls a VM guest OS, and targets other VM guests running on the same host.
- *VM Guest to Host (G-H)*: I consider an attacker that controls a VM guest OS, and targets the hypervisor (also known as VM Monitor).
- *Host to SGX (H-SGX)*: I consider an attacker in control of the system (i.e., able to load its own kernel) targeting an SGX enclave. SgxPectre [99] presents a series of attacks in this setting, exploiting different variants of Spectre v2.

5.6.2 Systems and Platforms

This section describes the choice of platform, architecture and Linux kernel version that are used to evaluate the tools. I test each tool on several different CPU families and kernel versions: Table 5.3 shows the CPU families and the corresponding kernel version where the 4 tools have been tested. I run the *information gathering* tools on each system and collect their final report. The approach of

this set of tools consists of simply parsing information gathered in the system, thus the execution time of both MDSTOOL-CLI and SPECTRE-MELTDOWN-CHECKER is roughly constant. Concerning *empirical* tools, I execute each test available in SPECULATOR and TRANSIENTFAIL on the machines listed in Table 5.3. I set a 20 seconds timeout for all TRANSIENTFAIL empirical test runs with the exception of the Spectre-BTB one where the threshold was set to 250 seconds because its runs are significantly slower. Most of the tests are meant to run indefinitely while others until a secret is revealed. The timeout is necessary to avoid infinite iterations. Spectre-BTB is a rather noisy attack and the test is run in batches with an incremental number of iterations (e.g., 100k, 200k, 300k etc.) until either the attack is successful or the time expired. I adjusted its timeout to allow this incremental search of the right number of iterations. For SPECULATOR instead, I execute all tests with 10k runs per instance. While in the case of TRANSIENTFAIL tests can be run as a standard user, SPECULATOR requires root access on each machine to access the PMC interface.

I run all tools on systems hosted by 17 different infrastructure-as-a-service (IaaS) cloud providers: I collect samples for all three main IaaS offerings, namely:

- *multi-tenant* solutions: each tenant gets access to a virtual machine deployed on a set of physical resources shared with other customers;
- *single-tenant* solutions: the tenant gets access to a machine with a hypervisor whose physical resources are not shared with other customers;
- *bare metal* solutions: the machine is delivered without any virtualization solution and is fully devoted to a single customer.

Testing on real cloud systems as opposed to self hosted and managed hardware provides useful additional insight on the use of these tools in real-world scenarios: the choice of virtualization platform and its effects on the availability of the PMC infrastructure, the impact of workload from other tenants on the cache side channels, and default configurations for the systems are just a few. More will be discussed in Section 5.7.1.

Overall I perform tests on 28 different machines as many cloud providers support more than one of the above mentioned solutions. To keep my analysis as general as possible, I collect the results from clouds of different sizes and market share: according to Gartner [100], the clouds I test, cover together more than the 75% of the IaaS market share in 2018.

5.7 Testing Tools Analysis

In this section, I report results related to the comparison between the methodologies underpinning the tools, underlining pros and cons of each. I also present the results of my analysis on the 17 tested providers. Finally, I report, based on the use cases described in Section 5.6.1, which methodology might be better suited to determine the security of a system with respect to transient execution attacks.

	Parsing Error	Use Case Imprecision	Attack Vector only	Cache Noise	PMC req.	Root req.
MDSTOOL-CLI	✓	✓				
SPECTRE-MELTDOWN-CHECKER		✓				✓
TRANSIENTFAIL			✓	✓		
SPECULATOR			✓		✓	✓

Table 5.4: Major pitfalls and limitations observed for each tool. I indicate with ✓ that the pitfall is present, whereas I leave blank otherwise.

Tools	Use Case	Spectre					Meltdown				
		PHT	BTB	RSB	STL	US	RW	P	PK	BR	GP
SPECTRE-MELTDOWN-CHECKER	S-P	✗	✗	✗	⇒		✗		✗	✗	
	U-U	✗	✗	✗							
	U-K	✓	✓	⇒	✓	✓	✗	✓		✗	⇒
	G-G	✗	✗	✗				✓			⇒
	G-H	✗	✗	✗				✗			
	H-SGX	✗	✗	✗				✓			
MDSTOOL-CLI	S-P	✗	✗	✗	✗		✗		✗	✗	
	U-U	✗	✗	✗							
	U-K	⇒	⇒	✗	⇒	⇒	✗	⇒		✗	✗
	G-G	✗	✗	✗				✗			✗
	G-H	✗	✗	✗				✗			
	H-SGX	✗	✗	✗				✗			
TRANSIENTFAIL	synthetic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPECULATOR	synthetic	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗

Table 5.5: Classification of the result types for each of the attacks with respect to the use cases described in Section 5.6.1. Empirical tools do not focus on specific use cases but rather on the existence of the attack vector. The table reflects this by referring the use case as synthetic. Results are reported as: ✓ if the tool reports information about a certain attack within the use case considered; ⇒ if the information can be inferred but it is not directly reported; ✗ where either no information can be inferred from the tool; finally the cell is left blank where the attack cannot be performed or it is not feasible under the specific use case.

```

1 CVE-2017-5753 aka 'Spectre Variant 1, bounds check bypass'
2 * Mitigated according to the /sys interface:
3   YES (Mitigation: usercopy/swapgs barriers and __user pointer sanitization)
4 * Kernel has array_index_mask_nospec:
5   YES (1 occurrence(s) found of x86 64 bits array_index_mask_nospec())
6 * Kernel has the Red Hat/Ubuntu patch:
7   NO
8 * Kernel has mask_nospec64 (arm64):
9   NO
10 > STATUS : NOT VULNERABLE (Mitigation: usercopy/swapgs barriers and __user pointer
    sanitization)

```

Listing 5.2: SPECTRE-MELTDOWN-CHECKER sample output for Spectre PHT

5.7.1 Tools comparison

Table 5.4 presents a comparison between the tools I employed in my study. I enumerate here advantages and disadvantages of each, providing examples from my analysis as support.

Pitfalls

Parsing errors. Information gathering tools may wrongly parse system information, thereby providing wrong data to the user. For example, the MDSTOOL-CLI tool uses `sysfs` file system provided by the Linux kernel to detect which mitigations are used against Spectre-PHT. The content of this file has changed over different kernel versions, and MDSTOOL-CLI only recognizes output with an outdated format, which leads it to conclude that a system with mitigations for Spectre-PHT is vulnerable, in contradiction with the kernel-provided output. In contrast, SPECTRE-MELTDOWN-CHECKER parses the output correctly for all tested kernel versions.

A similar parsing problem arises on older kernels when the `/sys` interface is not present. While SPECTRE-MELTDOWN-CHECKER reports that the interface is missing, MDSTOOL-CLI does not and reports that CPUs are safe against Spectre-PHT, Spectre-BTB, Meltdown-US and Meltdown-P. This issue may mislead the tool user into considering a system safe. I discovered the flawed MDSTOOL-CLI reports while running my experiments on one of the cloud providers listed in Table 5.7.

Inaccuracies from implicit assumptions. A second, very common issue for information gathering tools is related to the assumptions that are implicitly made over the considered use case: the tools report results in very generic terms, whereas in reality they only analyze a specific use case. For example, both SPECTRE-MELTDOWN-CHECKER and MDSTOOL-CLI report results about Spectre-BTB, when in reality they only consider the effects of the Spectre-BTB attack vector on a user-to-kernel use case. Similar arguments can be made for most transient execution attacks, wherein the same attack vector applies to multiple use cases: the tacit assumptions and lack of precision can be misleading. A related and more subtle issue concerns the confusion between the attack vector and mitigations against specific attacks leveraging it. As an example, the Linux kernel includes software mitigations to prevent a subset of Spectre-PHT attacks; SPECTRE-MELTDOWN-CHECKER checks whether they are enabled and – based on that – reports whether the system is vulnerable to Spectre-PHT. This report, shown in Listing 5.2, is however misleading because the mitigations

it evaluates do not necessarily protect from all Spectre-PHT attacks. First, Spectre-PHT attacks are not limited to attacks targeting the kernel: any user-space program could be the target of such an attack, as long as the target program contains a vulnerable code pattern and has not been compiled with mitigations such as SLH [16]. Second, Spectre-PHT attacks against the Linux kernel may still be feasible in isolated cases, as the mitigations are based on false-negative prone static analysis and manual code analysis.

Not considering same-address space training. I have identified that Spectre-BTB attacks are implicitly assumed by empirical tools to be in the cross-address space (cAS or cHT) variant. This is problematic and may lead to a false sense of security: Spectre-BTB attacks may also be performed in the same address-space (sAS) setting. Mitigations such as Intel’s IBRS, STIBP, and IBPB only apply to the cross address space setting.

Attack vector only. Empirical tools draw conclusions based on synthetic attack scenarios. Thus, they are only able to report results on the presence or absence of the specific transient attack vector on which an end-to-end attack may be built. However, the presence of the attack vector does not necessarily mean that an attack could be mounted in a particular use case, i.e., that the system is vulnerable. For example, in the context of Spectre-BTB in the cHT setting, an empirical tool can verify with a synthetic attack if the branch predictor is shared or not between the logic cores of a system. Although the attack is synthetic, a negative result means that the attack vector is not present, and that no attack of this kind can be performed. However, a positive result does not mean a system is vulnerable: it only means that further requirements need to be fulfilled for a valid end-to-end attack.

Cache noise. TRANSIENTFAIL faces problems when the system under test has competing cache activity. This can give the user wrong or inaccurate results. During my cloud analysis, I experienced such a problem on one provider where none of the TRANSIENTFAIL proof-of-concept attacks ran correctly.

PMC & root requirement. A problem specific to SPECULATOR is the availability of PMCs. In particular, during my analysis of cloud providers, in all but one cloud provider PMCs were not available, due to the lack of the performance counter interface in the virtualized environment. Also, SPECULATOR requires root privileges to use the PMCs infrastructure, which limits the use of the tool to the administrators of the system.

Tool	Commit hash
SPECTRE-MELTDOWN-CHECKER	91d0699
MDSTOOL-CLI	11b3240
TRANSIENTFAIL	7b0c9b2
SPECULATOR	4973a19

Table 5.6: Tools version used in the experiments

Tools vs use cases

Table 5.5 describes the effectiveness of each tool in evaluating transient execution attacks in each of the applicable use cases.

SPECTRE-MELTDOWN-CHECKER reports information for the *U-K* use case for almost all attack families. It is also the only tool the user can run to determine if Meltdown-P is patched in three

out of the six use cases, and the only tool that checks the CPU microcode to determine whether the machine is patched against Meltdown-GP. MDSTOOL-CLI limits its output to the information exposed by the kernel through `sysfs`; this mainly includes information about the state of the mitigations, with few details about the use cases under analysis. Nevertheless, a user with good knowledge of the mitigations listed in the final report of this tool can infer if the machine is vulnerable in the *U-K* use case.

Empirical tools verify instead the existence of the attack vectors at the base of Spectre and Meltdown variants. Both TRANSIENTFAIL and SPECULATOR execute test applications in user space in order to verify the feasibility of an attack. However, the attacks are usually run in the same address space of the victim, or from an attacker to a victim process not hardened against the attack. Therefore, the empirical tools reports are not linked with the use cases considered for the other tools; results are thus labeled as *synthetic* in Table 5.5.

5.7.2 Analysis

In this section, I present results collected by running all tools on systems hosted by 17 of the most prominent cloud providers

Cloud	Multi-tenant	Single-tenant	B. Metal
AWS	✓	✓	✓
Alibaba	✓		
Azure	✓	✓	
IBM Cloud	✓	✓	✓
GCP	✓	✓	
Digital Ocean	✓		
OVH			✓
Hetzner	✓		✓
Oracle	✓	✓	✓
Packet			✓
Scaleway	✓		✓
Vultr	✓	✓	✓
Bigstep			✓
Cloudsigma	✓		
Tencent	✓		
RamNode	✓		
Zenlayer			✓

Table 5.7: List of the 17 cloud provider tested and their available configurations

Information gathering tools

Results from SPECTRE-MELTDOWN-CHECKER and MDSTOOL-CLI show that 16 out of the 17 tested cloud providers make use of the proper mitigations to harden the kernel against transient execution attacks: only one provider uses a kernel version (4.4.0) that does not include mitigations against Spectre and Meltdown attacks.

Regarding Spectre, 16 cloud providers run a kernel properly recompiled with *lfence* instructions and retpoline. Additionally, all enable mitigations such as IBPB, STIBP and SSBD. The default setup of these mitigations is *conditionally enabled* which means that the mitigations are not enforced to user space applications unless specifically requested, however the patched kernel in these settings uses them when needed. Hence, the kernel cannot be attacked in any of the depicted scenarios (sAS, cAS and cHT).

Regarding Meltdown, all the tested Intel machines, excepting those of one provider, use KPTI and PTE inversion to block respectively Meltdown-US and Meltdown-P. However, the Meltdown-US test from the TRANSIENTFAIL suite showed that KPTI does not stop Meltdown-US over pages mapped as kernel-pages at runtime. Moreover, SPECTRE-MELTDOWN-CHECKER reports that the CPU microcode of such machines is patched to stop Meltdown-GP. As for AMD-based machines, Meltdown attacks do not affect them.

Empirical tools

Here, I report the results obtained by running the *empirical* tools on the considered cloud providers. Tests for Spectre-PHT and Spectre-BTB succeed in the cAS case on all machines. Not all cloud providers are affected in the cHT case since on virtual machines, hyper-threading is not always available, making this scenario unfeasible. Specifically, I find that 4 clouds disable hyper-threading in their multi-tenant solutions. Results for Spectre-RSB are negative, indicating that this attack vector is prevented owing to the fact that RSB filling is enabled: at every context switch, possibly poisonous RSB entries are flushed. The cHT scenario is unfeasible because the RSB is only shared between processes running interleaved on the same logical core. Spectre-STL succeeds on all machines since SSBD is only conditionally enabled, being inactive in practice: SSBD must be fully enabled to prevent the attack in both user-space and kernel-space. Tests for Meltdown-RW succeed on all Intel-based nodes except for those based on the new Cascade Lake microarchitecture. For what concerns Meltdown-BR, the tests are successful on Intel CPUs supporting the *mpx* instructions, while they report negative results on Ivy Bridge and Broadwell families. None of the Meltdown variants show positive results on AMD.

I conclude that the *empirical* tools generally report most known attack vectors as present and exploitable. Instead, the *information gathering* tools report the security stance of the system in the U-K use case, which generally results in the system being classified as not vulnerable due to the widely enabled kernel mitigations. The results discrepancy is justified by the following:

- the default configuration of all tested systems only conditionally enable user-space (U-U) mitigations;
- the code implementing the synthetic attack scenarios in SPECULATOR and TRANSIENTFAIL restricts generalization to other target applications on the system. By design, these tools are unaware of the mitigations enabled outside of the synthetic environment.

5.8 Recommendations

During my analysis, I find that all available tools have shortcomings about their ability to determine whether a system is vulnerable. Based on the experience gained during my work, I propose 5 main

recommendations directed towards systems administrators using such tools, as well as developers of these tools.

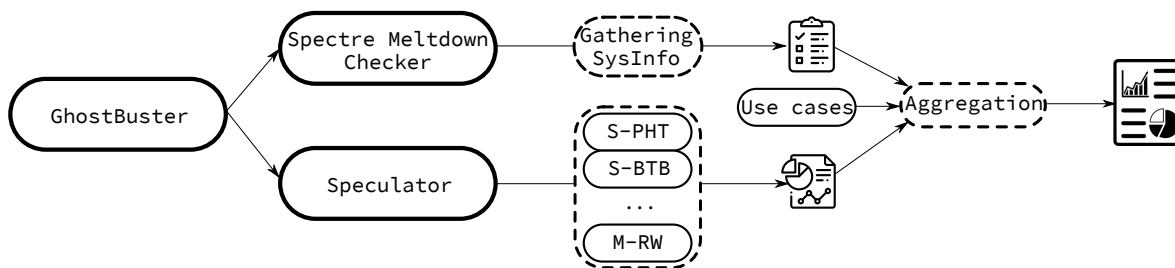


Figure 5.5: GHOSTBUSTER's overview. GHOSTBUSTER leverages SPECTRE-MELTDOWN-CHECKER and my modified version of SPECULATOR to assess the system security using both known methodologies, *gathering* and *empirical*. Then, it aggregates the results in a final report factoring in also the various use cases I identified to give a more accurate picture to the user. With solid circles I describe the major components of GHOSTBUSTER while with dotted circles I highlight operations performed.

5.8.1 Limit cache noise

When working with empirical tools such as TRANSIENTFAIL, it is very important to pay particular attention to workloads running on the same physical machine. Heavy cache activity, for instance from the last-level-cache (LLC) that is often shared across cores might cause the test to report that an attack is not feasible while actually the failure is caused by temporary cache activity. If the user/administrator has the ability to control the load on the system, I recommend to pause any workload during the test's execution. Additionally, I recommend to run the tests several times with enough time between each run. Finally, when possible, I recommend using PMC-based approaches whenever available, as they are less prone to noise.

```

1  ===== SPECTRE STL =====
2  * Attack success rate (synthetic test): 93.0%
3  * Attack vector: Present
4  * Difficulty: High - No practical attack demonstrated
5
6  ----- USE CASES -----
7  * S-P: SSB is not fully disabled. Check that the victim program is compiled with seccomp()/prctl().
8  * U-U: SSB is not fully disabled. Check that the victim program is compiled with seccomp()/prctl().
9  * U-K: SSB is not fully disabled. Your kernel is vulnerable.
10 * G-G: Check if SSB is fully disabled on the host machine.
11 * G-H: Check if the kernel on the host machine supports disabling SSB.
12 * H-SGX: SSB is not fully disabled.

```

Listing 5.3: GHOSTBUSTER sample output for Spectre STL

5.8.2 Define the right use case and understand your threat model

Most of the mitigations currently available (described in Section 2.4) incur a medium to high overhead. In fact, it is very common for these mitigations to be disabled by default. Enabling one or more of them when not necessary is generally not indicated. Therefore, it is important for a system administrator or user to understand which of the attacks flagged by one of the tools for a specific system falls under the use case(s) the administrator/user cares about. An example of such a case would be enabling a system-wide mitigation like STIBP when the only use case considered important is the User to Kernel (U-K) one. This is because, by default, STIBP already prevents attacks such as Spectre-BTB under the U-K scenario and it would be a performance waste to enforce it system-wide.

```
1 CVE-2018-3639 aka 'Variant 4, speculative store bypass'
2 * Mitigated according to the /sys interface: YES
3   (Speculative Store Bypass disabled via prctl and seccomp)
4 * Kernel supports disabling speculative store bypass (SSB):
5   YES (found in /proc/self/status)
6 * SSB mitigation is enabled and active: YES
7   (per-thread through prctl)
8 > STATUS: NOT VULNERABLE (Mitigation: Speculative Store Bypass disabled via prctl and
   seccomp)
```

Listing 5.4: SPECTRE-MELTDOWN-CHECKER sample output for Spectre STL

5.8.3 Understanding information gathering tools results

Tools should report as clearly as possible the assumptions underlying the reported results and the considered use cases. In the current state of these tools, I recommend to verify either through the tool documentation (if any) or the source code of the tool (if available) which use cases is under analysis. Our analysis suggests it is possible that the assumptions made by existing tools do not match those of the user, leading to a false sense of security or a lack of action.

```
1 ===== SPECTRE BTB same address-space =====
2 * Attack success rate (synthetic test): 95.00%
3 * Attack vector: Present
4 * Difficulty: High - No practical attacks demonstrated.
5 ----- USE CASES -----
6 * S-P: Check that the target process is compiled with lfence or retpoline.
7 * U-U: Check that the target process is compiled with lfence or retpoline.
8 * U-K: This kernel is not vulnerable: Full retpoline + IBPB are mitigating the vulnerability.
9 * G-G: Check that the target process is compiled with lfence or retpoline.
10 * G-H: Check that host kernel is compiled with retpoline and supports RSB filling.
11 * H-SGX: Check that the target process is compiled with lfence or retpoline.
```

Listing 5.5: GHOSTBUSTER sample output for Spectre BTB same address-space

5.8.4 Use a mixed approach

Based on my analysis, I make the observation that the two types of tools, information gathering and empirical, report different types of information. While using only one or the other gives a very limited snapshot of the system security, their combination allows to gather more robust information about the system in general but also allows for a greater ability to infer information and inform the user. For instance, if I consider the Spectre-BTB in the cross address-space case and I successfully verify the presence of such attack vector using either SPECULATOR or TRANSIENTFAIL. Now, I can combine this result with the output on SPECTRE-MELTDOWN-CHECKER and connect which use cases this attack vector might affect. For example, assuming that SPECTRE-MELTDOWN-CHECKER tells us that STIBP is at default settings (conditionally enabled), I can infer that the attack vector I verified with the empirical tools affects the U-U and S-P use cases but it does not affect U-K.

5.8.5 Static analysis

A clear limitation of information gathering tools considered in this work is that, for attacks such as Spectre-PHT, their verification is at best a coarse approximation. This is because none of the tools inspects the target application at code level to verify if proper mitigations are inserted (e.g., lfencing sensible branches or branchless masking). While no comprehensive static analysis tool is available, known techniques such as lfence counting (which is implemented in SPECTRE-MELTDOWN-CHECKER only for the current kernel image) can help determine whether such protections are in place. I recommend future work to integrate this type of static analysis to be able to inform users better, in particular with respect to U-U use cases targeting important user space programs and libraries, such as OpenSSH or OpenSSL.

```
1  ===== SPECTRE BTB cross address-space =====
2  * Attack success rate (synthetic test): 75.00% (spatial colocation), 56.00% (temporal colocation)
3  * Attack vector: Present
4  * Difficulty: Low - Practical attacks demonstrated in user-to-kernel and user-to-user use cases.
5  ----- USE CASES -----
6  * S-P: This use case does not apply for this attack.
7  * U-U: Check that the target process is compiled with lfence or retpoline.
8      IBPB is conditionally enabled: check that the target process invokes it with prctl/seccomp.
9      STIBP is conditionally enabled: check that the target process invokes it with prctl/seccomp.
10 * U-K: This kernel is not vulnerable: Full retpoline + IBPB are mitigating the vulnerability.
11 * G-G: Enable STIBP and IBRS system-wide on the guest machine. If STIBP and IBPB are conditionally enabled
      , check that the target process invokes them with prctl/seccomp or is compiled with retpoline.
12 * G-H: Check that IBRS is enabled on the host, or that it is compiled with retpoline and uses IBPB.
13 * H-SGX: Not vulnerable: IBRS is enabled.
```

Listing 5.6: GHOSTBUSTER sample output for Spectre BTB cross address-space

5.9 GhostBuster

Given that none of the four available tools provide a complete and consumable answer as to whether a system is vulnerable to the various classes of transient execution attacks, I prototype a new tool, GHOSTBUSTER, shown in Figure 5.5, that takes into account the recommendations presented in the previous section and provides a system administrator with accurate information to decide whether their system is vulnerable.

The tool is built on the foundation of *empirical* and *information gathering* methodologies combined, as a result of the insights collected during my analysis. GHOSTBUSTER is a meta-tool combining a modified version of SPECULATOR and SPECTRE-MELTDOWN-CHECKER, which allows us to use the best of each available approach. Another key difference from existing tools is that GHOSTBUSTER provides information explicitly based on use cases presented in Section 5.6.1. The use case information is integrated with SPECULATOR and SPECTRE-MELTDOWN-CHECKER outputs during the *Aggregation* phase as depicted in Figure 5.5.

In GHOSTBUSTER, the first tool I leverage is an enhanced version of SPECULATOR. For GHOSTBUSTER, I include two sets of empirical tests, the PMC based I used during my analysis in Section 5.5.2 and a second set, a cache based series of tests similar to the ones used in TRANSIENTFAIL. The two set of tests for empirical verification are necessary to make sure I can have a fallback mechanism when one of the two is not available, once again making the best out of available approaches to avoid the pitfalls I identified. As mentioned in Section 5.7.1, PMC-based tests cannot be used in a virtualized environment because such interface is not exported to the guest. Similarly, there are cases in which the system has too much LLC activity, making the tests using the cache unreliable and therefore requiring the PMC-based tool. When the system setup allows so, GHOSTBUSTER runs both empirical test sets to have confirmation of the results and detects any mismatch. If GHOSTBUSTER detects a huge amount of LLC activity, it prompt a warning to the user to signal that possible problems can arise while running TRANSIENTFAIL.

GHOSTBUSTER uses the SPECTRE-MELTDOWN-CHECKER output in the analysis phase, for its comprehensive report on supported mitigations on the target system, together with their activation status. This enables GHOSTBUSTER to connect the synthetic results provided by the tests with the actual use cases. Subsequent to information gathering and analysis, GHOSTBUSTER presents the results based on each use case. It presents information about the difficulty level D of the attack, which I provide based on whether real-world attacks using that attack vector exist and how easy it is for the attacker to meet the requirements for the attack. Formally, I compute it as follows:

$$D = 100 - \left[40 \text{ } rw + \left(60 - \sum_{j=0}^N W_j R_j \right) \right] \quad (5.1)$$

$$= \begin{cases} [0, 40) & \text{Low} \\ [40, 70) & \text{Medium} \\ [70, 100] & \text{High} \end{cases} \quad (5.2)$$

where $rw \in \{0, 1\}$ indicates whether the considered attack has a real world instance, $R_j \in \{0, 1\}$ indicates if the identified attack requirement j is present for the attack, N is the total number of possible requirements found for a certain type of attack, and W_j represents the difficulty weight for each of the requirements, that for the sake of simplicity I set $W_j = \frac{60}{N}$ for each requirement j .

When possible, GHOSTBUSTER reports the system status, vulnerable or not vulnerable. In cases when such a conclusion cannot be drawn, as may be the case with Spectre-PHT or Spectre-BTB where the attack and mitigations are program dependent, GHOSTBUSTER provides a checklist that the user can follow to verify the security of such application. I prefer to provide a checklist for some of the cases instead of trying an automatic approach because there are no error-free methods to detect, for instance, if an application is instrumented with SLH against Spectre-PHT. False positive results might induce a false sense of security which is against the principles GHOSTBUSTER is designed with, so I suggest the user what exactly requires manual verification instead.

Listing 5.1 shows the output of SPECTRE-MELTDOWN-CHECKER and Listing 5.5 and Listing 5.6 show the output of GHOSTBUSTER in relation to the Spectre-BTB attack. These outputs are taken from the same machine in the same settings. SPECTRE-MELTDOWN-CHECKER provides raw information about the mitigations status (e.g., present/not present). For instance, it confirms the availability of mitigations such as IBRS and IBPB, and marks them as active. Also, it shows that the kernel is compiled with retpoline. Finally it informs the user that the system is *not vulnerable* because both mitigations IBRS and IBPB are present on the machine. I deem this output to be misleading because the same machine tested under SPECULATOR is reported vulnerable to the Spectre-BTB attack vector despite the picture depicted by SPECTRE-MELTDOWN-CHECKER output. In practice, this means existing known attacks such as SMoTherSpectre [34] leaking bytes from OpenSSL are feasible on this machine.

In contrast, GHOSTBUSTER provides more precise information. First, it provides the attack success rate that empirical tests have obtained and it confirms the presence of the attack vector. This information is retrieved thanks to my enhanced version of SPECULATOR fork.

Second, based on the output of SPECTRE-MELTDOWN-CHECKER I am able to provide a more detailed view regarding each one of the use cases. It is possible to notice that GHOSTBUSTER considers the system protected against Spectre-BTB under the U-K use case. This confirms the output of SPECTRE-MELTDOWN-CHECKER from Listing 5.1 that strictly focuses on the kernel protection from transient execution attacks. Instead, for cases such as U-U, GHOSTBUSTER recognizes that there are settings (e.g., same address space) in which no information regarding mitigations is available and therefore no final decision can be taken based on available information. In such cases GHOSTBUSTER provides suggestions such as to verify that the target application is compiled with lfence/retpoline, thereby not misleading the user into a false sense of security. Third, GHOSTBUSTER considers, when necessary, the various attack settings (cAS/cHT/sAS). For Spectre-BTB, GHOSTBUSTER provides different outputs for same and cross address space and adjusts the recommendation accordingly to the scenario. For instance, for the U-U use case in the cross address space setting, it suggests to the user to verify if the target application requests to enable STIBP and IBPB to the kernel through either *seccomp* or *prctl* interface. In fact, it even distinguishes between the need for STIBP (mitigating temporal colocation, cHT) and for IBPB (mitigation spatial colocation, cAS) depending on whether none, either, or both of the two empirical tests are successful. In the output shown here, both tests pass and both attack vectors are present, therefore the recommendation is to enable both STIBP and IBPB.

Another example for comparison between GHOSTBUSTER and SPECTRE-MELTDOWN-CHECKER is provided by Listing 5.4 and Listing 5.3 for the Spectre-STL attack. Here, SPECTRE-MELTDOWN-CHECKER detects that the system supports Speculative Store Bypass (SSB) and simply declares the system *not vulnerable* based on this information. In reality, the system should be considered vulnerable for most of the use cases because the SSB mitigation is enabled only conditionally,

which is reflected in the GHOSTBUSTER output. Therefore, in use cases such as U-U and S-P, the target application must be checked and forced to use either *seccomp* or *prctl* which would enforce the mitigation for the current process. Nevertheless, GHOSTBUSTER also makes sure to let the administrator know that this is a minor threat, given that no known attacks exist to this date.

This type of output comparison is valid for all the supported attacks, which I do not report for sake of space. As shown, GHOSTBUSTER enhances current tools output to include use cases and more targeted information, thus standing out as a more accurate and usable tool. Although my current GHOSTBUSTER implementation is meant for x86/x86_64 Linux machines, the principles incorporated and its design remain valid for other architectures (e.g ARM) and other Operating Systems (e.g., Windows).

To conclude, GHOSTBUSTER provides a more detailed and accurate view of a system's vulnerability to transient execution attacks by simply combining the best features of existing approaches and presenting them in an understandable way. While for certain use cases the assessment is binary, vulnerable or not vulnerable, for others, GHOSTBUSTER guides the user on testing whether the target application meets the correct safety requirements against an attack under the considered use case.

Chapter 6

Future Work

Speculative execution attacks are a very new class of vulnerabilities and most of their limits and capabilities remain to be discovered. Most of the work so far has been done on the x86 architecture. However, speculative execution attacks affect any architecture that supports speculative execution. Extending the focus to less mainstream architecture could potentially benefit the understanding of the effects of speculative execution attacks. I propose to port SPECULATOR to other architectures such as ARM and Power. In this way, I can verify if similar performance counters that can serve as speculative execution markers exists and can be used similarly as I did under x86. Also, comparing how speculative execution is implemented in various architectures could underline the most successful design decisions that can serve as base to create future designs.

A direction for future research is the automation of some of the several stages speculative execution attacks comprehend. These attacks are highly dependent from many factors to be successful (e.g., code alignment, correct cache priming). Manually checking for those elements is a very tedious and error prone task. In general, having automated analysis that is able to keep those parameters into account would greatly benefit the ability of exploiting these vulnerabilities in real world settings.

Cache side channels have been extensively studied in the past and a lot of literature is available in this area. However, speculative execution attacks use of cache side channels differs from previous known attacks. In a normal cache side channel attack, there are generally less requirements during the cache priming phase. This allows for a broader and less precise cache eviction that allow the attacker to obtain high quality signal. This is generally not possible in speculative execution attacks because there are stricter requirements on what should be cached and what should be evicted. While in my work we made progress in achieving a more precise eviction, further improvements in developing even finer grain techniques are required to augment the exploitability of these vulnerabilities.

Chapter 7

Papers

7.1 Related Publications

Parts of this thesis have been published in peer-reviewed conferences. Hereafter the list:

- Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. “Two methods for exploiting speculative control flow hijacks”. In *Proceedings of the 13th USENIX Workshop on Offensive Technologies (WOOT)*, Santa Clara, California, USA, 2019. [33]
- Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. “Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations”. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, San Juan, Puerto Rico, 2019. [82]
- Andrea Mambretti, Pasquale Convertini, Alessandro Sorniotti, Alexandra Sandulescu, Engin Kirda, and Anil Kurmus. “GhostBuster: Understanding and Overcoming the Pitfalls of Transient Execution Vulnerability Checkers”. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Honolulu, Hawaii, USA, 2021. [101]
- Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. “Bypassing Memory Safety Mechanism through Speculative Control Flow Hijacks”. In *Proceedings of the 6th IEEE European Symposium on Security and Privacy (IEEE Euro S&P)*, Vienna, Austria, 2021. [102]

7.2 Other work

Besides the work that is part of my dissertation, I worked and contributed on several other projects that I briefly introduce in this section.

7.2.1 Lava: Large-scale automated vulnerability addition [1]

Automated bug finding has been an extensively studied area in the last decade. A large amount of tools have been developed, often employing very different techniques from one another (e.g., fuzzing, symbolic execution, concolic execution etc.). However, comparing these techniques remain a complicated task and it is hard to understand which and where one technique perform better than another. First and foremost, the evaluation of each one of this work focuses on solely discovering zero-day vulnerabilities—i.e., vulnerabilities that were not previously known. This approach does not offer to compare the technique presented with others because the applications used are often different. Moreover, it does not give information about the tool precision since the total number of zero-day vulnerabilities is not known.

Towards solving this problem, I present Lava which is a tool aimed to create ground truth corpora of buggy programs that can be used as benchmark to compare bug finding techniques. With Lava, it is possible to analyze an application execution under a specific input and automatically identifying parts of the input that are not involved in control flow decisions. Doing so guarantees that the execution path from the program entry point to the injected vulnerability will not change if those input bytes are modified. Lava adds a bug trigger based on these bytes for each of the injected vulnerabilities. The vulnerabilities inserted by Lava can be of different types of memory corruption and resides on the execution path based on the initial input. Lava reports all the information related to each one of the injected vulnerability so that can be used to verify the output of the bug finding tools.

7.2.2 HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing [2]

Most of contemporary fuzz testing techniques focus on memory corruption vulnerabilities. Meanwhile, algorithmic complexity (AC) vulnerabilities, which are a common attack vector for denial-of-service attacks, remain an understudied threat. AC vulnerabilities encompass all those vulnerabilities that, given a legitimate input (i.e., an input that does not exceed the bandwidth capability of the application), are able to maximize the resource usage either in space or in time such to cause denial-of-service.

In this work, I present HotFuzz a tool that apply micro-fuzzing, a novel technique that is method-based, to discover AC vulnerabilities in Java applications. HotFuzz targets Java due to its wide adoption in the wild for all sorts of applications. Through a modified version of the JVM, called EyeVM, HotFuzz is able to monitor the resource usage in time or space of a single method execution and is able to record outliers. HotFuzz marks as suspicious all those inputs that are causing dramatic changes compared to the baseline. As baseline, HotFuzz uses the recorded behavior of the test-suite each method within each library comes with.

7.2.3 Trellis: Privilege separation for multi-user applications made easy [3]

Across the security field, access control of resources is one of the most studied areas. Several methodologies and schemes to perform access control exist in each domain where security is key (e.g., physical building access, network access, software resources access etc.). In the software realm, most of the access control in a shared environment is done at the operating system level.

For instance, under Linux, the filesystem access is regulated by three type of permission—i.e., user, group, and other— and three type of accesses—i.e., read, write, and execute. Without the right permissions, a certain user, in a shared system, cannot read, write, or execute a file. This allow to protect resources from unauthorized accesses whenever needed. This type of access control mechanism is very common and generally well tested from possible logic vulnerabilities that would otherwise allow to bypass it.

However, the definition of users and/or access controls is not always defined at the operating system level. That is generally the case for multi-user applications that are found in enterprise environment (e.g., SAP CRM). In these applications, the burden for defining such access control is left to the developers which often might lack of the security know-how. A mistake in the design or in the implementation of such mechanism can jeopardize the whole system security. One example is presented by Mulliner et al. [103] that discovered that some of these applications use the Graphic User Interface (GUI) as medium to enforce access control. This GUI enforcement is inadequate and can be easily bypass through the use of GUI inspectors.

In this work, I present Trellis that is a framework for expressing hierarchical access control policies in applications and enforcing these policies during execution. Through partial annotation, developers can define within the code base the privilege levels required by the application. Trellis forwards this annotation at compile time and creates a privilege separated binary. At runtime, these policies are extracted by the binary and enforced by the operating system kernel.

7.2.4 HONEYBUG: hypervisor-based approach for gathering attacker insights

The modern cyber-attack landscape offers a large spectrum of possible attackers. This ranges from beginners like “script kiddies”, to more powerful attackers such as criminal organization, and state-sponsored attackers. Each attacker has different goals that motivate her activities. Often times, it is impossible to attribute who is behind a certain attack and from which category such attacker belongs to. Towards solving this problem a large number of work is done in terms of honeypot systems that are spread within the network perimeter of a network. The goal of such honeypot is to study the movement of the attacker within a fake system deceiving the attacker from the real target and allowing time to react to the attack. However, such honeypot tend to be quite finger-printable due to their confined environment and are only able to observe the attacker only when it is already inside the system. Very little work has been done to observe the attacker during the exploitation phase which is where most of the information about the attacker capabilities can be gathered. Most of the related work incur in high overhead that is one of them main fingerprints the attacker is able to detect.

In this work, I present Honeybug, a novel lightweight honeypot design with main focus to observe exploitation of vulnerabilities while they are happening. To do so, I make us of Lava [1] to build applications with several known bugs. I modified Lava such that each bug location is instrumented during compilation with a nop-equivalent sequence of bytes that will be overwritten at runtime to link the bug to a Honeybug custom hypervisor. The hypervisor is in charge of monitoring the bug at runtime. When the application is loaded, the memory page where the bug resides is cloned and one of the copies is overwritten with a hook to the hypervisor at the location of the nop-equivalent sequence. I use the split-EPT [104] method to hide my instrumentation to the

attacker. Basically, upon reads on the location where there is the instrumentation, the hypervisor shows the nop-equivalent sequence of instructions. While upon execution, it executes the sequence of instructions that call into the hypervisor notifying that the bug is under exploitation. The use of an hypervisor instead of cache split techniques or emulators is that the hypervisor is called only upon specific circumstances leaving a very tiny footprint. A mechanism such Honeybug allows for several possibilities when the exploitation detection is detected, for instance, the application can be restart, or the bug can be shown but not really executed and so on. In this project, I aim to just collect attacker interaction to be able to determine the attackers ability and possibly being able to place each attacker to the right category based on the skills shown.

While the bulk of the prototype has been done, this project is missing the final evaluation before being submitted for publication.

Chapter 8

Conclusions

In this dissertation, I concentrated on the subclass of transient execution attacks induced by speculative execution, such as Spectre-PHT and Spectre-BTB. At the time of the writing, this area of research is still relatively new, and new attack variants and new defenses are proposed on a monthly basis.

First, I proposed a novel technique based on performance counters to deterministically be able to reverse engineer the behavior of the micro-architecture. I integrated the use of this technique in a tool, SPECULATOR, that is tailored to study small code snippets without introducing measurement noise.

This technique and SPECULATOR were crucial for all the results gathered in this dissertation. With them, I was able to gather insights on the micro-architecture (e.g., behavior during nested speculation, or the effects of the *clflush* instruction in the speculation window etc.), and investigate new side-channels gadgets beyond the known data cache (e.g., instruction cache, and branch target buffer). Furthermore, using SPECULATOR I tested the resilience of modern memory corruption vulnerabilities such as CFI and SSP in the context of transient execution attacks and demonstrated that they can be leveraged to carry information leakage attacks.

Finally, I inspected the state of the art tools to discover if a system is vulnerable to transient execution attacks. My analysis showed that each of the available tools has several pitfalls that might mislead the user and give a false sense of security. As a result, I designed and proposed an hybrid tool that I called GHOSTBUSTER that take advantage of the best features of each of the existing tools and provide a more comprehensive view on the security stance of the system. GHOSTBUSTER is based on SPECULATOR and SPECTRE-MELTDOWN-CHECKER.

Bibliography

- [1] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “LAVA: Large-scale Automated Vulnerability Addition,” in *IEEE Symposium on Security and Privacy (Oakland)*, May 2016.
- [2] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, “Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing,” in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.
- [3] A. Mambretti, K. Onarlioglu, C. Mulliner, W. Robertson, E. Kirda, F. Maggi, and S. Zanero, “Trellis: Privilege Separation for Multi-User Applications Made Easy,” in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Sep. 2016.
- [4] G. P. Zero, “Reading privileged memory with a side-channel,” <https://googleprojectzero.blogspot.ch/2018/01/reading-privileged-memory-with-side.html>, 2018.
- [5] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *IEEE Symposium on Security and Privacy*, 2019.
- [6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *USENIX Security Symposium*, 2018.
- [7] J. Corbet, “Kaiser: hiding the kernel from user space,” <https://lwn.net/Articles/738975/>, 2017.
- [8] A. Sez nec and P. Michaud, “A case for (partially) tagged geometric history length branch prediction,” *J. Instr. Level Parallelism*, vol. 8, 2006.
- [9] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtvushkin, and D. Gruss, “A Systematic Evaluation of Transient Execution Attacks and Defenses,” in *USENIX Security Symposium*, 2019, extended classification tree at <https://transient.fail/>.
- [10] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *IEEE Symposium on Security and Privacy*, 2018.
- [11] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks on aes, and countermeasures,” *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, Jan 2010. [Online]. Available: <https://doi.org/10.1007/s00145-009-9049-y>
- [12] B. W. Lampson, “Lazy and speculative execution in computer systems,” in *ACM SIGPLAN Conference on Functional Programming*, 2008.
- [13] P. Kocher, <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>, 2018.
- [14] P. Turner, “Retpoline: a software construct for preventing branch-target-injection,” <https://support.google.com/faqs/answer/7625886>, 2018.
- [15] J. Corbet, “Meltdown/spectre mitigation for 4.15 and beyond,” <https://lwn.net/Articles/744287/>, 2018.
- [16] C. Carruth, “Speculative Load Hardening,” <https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html>, 2018.

- [17] P. Turner, “Retpoline: a software construct for preventing branch-target-injection,” <https://support.google.com/faqs/answer/7625886>, 2018.
- [18] Intel, “Deep dive: Indirect branch restricted speculation,” <https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation>, 2018.
- [19] —, “Deep dive: Indirect branch predictor barrier,” <https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier>, 2018.
- [20] —, “Deep dive: Single thread indirect branch predictors,” <https://software.intel.com/security-software-guidance/insights/deep-dive-single-thread-indirect-branch-predictors>, 2018.
- [21] D. W. Wall, “Limits of instruction-level parallelism,” in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 176–188. [Online]. Available: <http://doi.acm.org/10.1145/106972.106991>
- [22] K. B. Theobald, G. R. Gao, and L. J. Hendren, “Speculative execution and branch prediction on parallel machines,” in *International Conference on Supercomputing*, 1993.
- [23] “Intel Software Developer Manual,” <https://software.intel.com/en-us/articles/intel-sdm>, 2018.
- [24] “Intel Architectures Optimization Reference Manual,” <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>, 2018.
- [25] “Preliminary Processor Programming Reference (PPR) for AMD Family 17h Models 00h-0Fh Processors,” http://support.amd.com/TechDocs/54945_PPR_Family_17h_Models_00h-0Fh.pdf, 2017.
- [26] A. Fog, “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers,” <https://www.agner.org/optimize/microarchitecture.pdf>, 2018.
- [27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Computer Architecture News*, vol. 39, no. 2, Aug. 2011.
- [28] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2671225.2671271>
- [29] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+flush: A fast and stealthy cache attack,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 279–299.
- [30] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+abort: A timer-free high-precision l3 cache attack using intel TSX,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 51–67. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen>
- [31] G. Maisuradze and C. Rossow, “Ret2spec: Speculative execution using return stack buffers,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: ACM, 2018, pp. 2109–2122. [Online]. Available: <http://doi.acm.org/10.1145/3243734.3243761>
- [32] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *USENIX Workshop On Offensive Technologies*, 2018.
- [33] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, “Two methods for exploiting speculative control flow hijacks,” in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: <https://www.usenix.org/conference/woot19/presentation/mambretti>

- [34] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019.*, 2019, pp. 785–800. [Online]. Available: <https://doi.org/10.1145/3319535.3363194>
- [35] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," <https://people.csail.mit.edu/vlk/spectre11.pdf>, 2018.
- [36] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Cham: Springer International Publishing, 2019, pp. 279–299.
- [37] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 897–912.
- [38] C. Carruth, "Speculative load hardening," <https://llvm.org/docs/SpeculativeLoadHardening.html>, 2018.
- [39] D. Williams, "Sanitize speculative array de-references," <https://lore.kernel.org/patchwork/patch/874621/>, 2018.
- [40] <https://repo.or.cz/w/smash.git>, 2018.
- [41] "The Linux Kernel user's and administrator's guide," <https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html>, 2019.
- [42] M. Bynens, "V8 Untrusted code mitigations," <https://github.com/v8/v8/wiki/Untrusted-code-mitigations>, 2018.
- [43] L. Wagner, "Mitigations landing for new class of timing attack," <https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/>, 2018.
- [44] "JIT mitigations for Spectre," <https://github.com/Microsoft/ChakraCore/commit/08b82b8d33e9b36c0d6628b856f280234c87ba13>, 2018.
- [45] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *Commun. ACM*, vol. 63, no. 7, p. 93–101, Jun. 2020. [Online]. Available: <https://doi.org/10.1145/3399742>
- [46] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 954–968. [Online]. Available: <https://doi.org/10.1145/3352460.3358274>
- [47] "DOLMA: Securing speculation with the principle of transient non-observability," in *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin>
- [48] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy," in *MICRO 2021 - 54th Annual IEEE/ACM International Symposium on Microarchitecture, Proceedings*, ser. Proceedings of the Annual International Symposium on Microarchitecture, MICRO. IEEE Computer Society, Oct. 2021, pp. 607–622, publisher Copyright: © 2021 Association for Computing Machinery.; null ; Conference date: 18-10-2021 Through 22-10-2021.
- [49] S. Ainsworth, *GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation*. New York, NY, USA: Association for Computing Machinery, 2021, p. 592–606. [Online]. Available: <https://doi.org/10.1145/3466752.3480074>
- [50] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 428–441. [Online]. Available: <https://doi.org/10.1109/MICRO.2018.00042>

- [51] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–86. [Online]. Available: <https://doi.org/10.1145/3352460.3358314>
- [52] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 264–276.
- [53] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Sjölander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 723–735. [Online]. Available: <https://doi.org/10.1145/3307650.3322216>
- [54] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating Conflict-Based cache attacks with a practical Fully-Associative design," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1379–1396. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar>
- [55] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.
- [56] A. C. de Melo, "The New Linux 'perf' tools," <http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf>, 2010.
- [57] T. Röhl, J. Eitzinger, G. Hager, and G. Wellein, "LIKWID Monitoring Stack: A Flexible Framework Enabling Job Specific Performance monitoring for the masses," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.
- [58] J. Levon, "Oprofile," <http://oprofile.sourceforge.net>, 2002.
- [59] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Proc. of the 2006 Ottawa Linux Symposium*, 2006, pp. 269–288.
- [60] M. Pettersson, "Perfctr," <http://user.it.uu.se/~mikpe/linux/perfctr/>, 2006.
- [61] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.
- [62] D. Zapanu, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 23–32.
- [63] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013.
- [64] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," *CoRR*, 2018. [Online]. Available: <http://arxiv.org/abs/1807.07940>
- [65] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2109–2122. [Online]. Available: <http://doi.acm.org/10.1145/3243734.3243761>
- [66] AMD, "Software optimization guide for amd family 17h processors," https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf, 2017.
- [67] A. Fog, "Test results for amd ryzen," <https://www.agner.org/optimize/blog/read.php?i=838&v=t>, 2017.
- [68] "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon Processors," https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.
- [69] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," <https://people.csail.mit.edu/vlk/spectre11.pdf>, 2018.

- [70] S. Ramakesavan and J. Rodriguez, “Intel Memory Protection Extensions Enabling Guide,” <https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide>, 2016.
- [71] O. Aci mez, “Yet another microarchitectural attack: exploiting i-cache,” in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007.
- [72] O. Aci mez, B. B. Brumley, and P. Grabher, “New results on instruction cache attacks,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2010.
- [73] D. Evtvushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over aslr: Attacking branch predictors to bypass aslr,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 40.
- [74] A. Sotirov, “Bypassing memory protections: The future of exploitation,” in *USENIX Security*, 2009.
- [75] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *IEEE Symposium on Security and Privacy*, 2013.
- [76] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, “Memory errors: The past, the present, and the future,” in *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, ser. RAID’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 86–106. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33338-5_5
- [77] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtvushkin, and D. Gruss, “A systematic evaluation of transient execution attacks and defenses,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/canella>
- [78] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, “Spectre is here to stay: An analysis of side-channels and speculative execution,” 2019.
- [79] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 298–307. [Online]. Available: <https://doi.org/10.1145/1030083.1030124>
- [80] R. Hund, C. Willems, and T. Holz, “Practical timing side channel attacks against kernel space aslr,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 191–205.
- [81] C. Robertson and et al., “C++ developer guidance for speculative execution side channels,” <https://docs.microsoft.com/en-us/cpp/security/developer-guidance-speculative-execution>, 2018.
- [82] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, “Speculator: A tool to analyze speculative execution attacks and mitigations,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 747–761. [Online]. Available: <https://doi.org/10.1145/3359789.3359837>
- [83] M. et al., “Speculator,” <https://github.com/ibm-research/speculator/wiki>, 2019.
- [84] “pagemap: do not leak physical addresses to non-privileged userspace,” <https://lwn.net/Articles/642074/>.
- [85] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, “Reverse engineering intel last-level cache complex addressing using performance counters,” in *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, 2015, pp. 48–65. [Online]. Available: https://doi.org/10.1007/978-3-319-26362-5_3
- [86] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: <https://doi.org/10.1109/SP.2015.43>
- [87] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2133375.2133377>

- [88] “ROPgadget,” <http://shell-storm.org/project/ROPgadget>.
- [89] “CVE-2004-0597,” <https://nvd.nist.gov/vuln/detail/CVE-2004-0597>, 2004.
- [90] C. Tice, G. Inc, T. Roeder, G. Inc, P. Collingbourne, G. Inc, S. Checkoway, Úlfar Erlingsson, G. Inc, L. Lozano, G. Inc, and G. Pike, “Enforcing forward-edge control-flow integrity,” in *in GCC & LLVM. In 23rd USENIX Security Symposium (USENIX Security 14) (Aug. 2014)*, USENIX Association, 2014, pp. 941–955.
- [91] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov>
- [92] R. Cox, “Spectre,” 2020. [Online]. Available: <https://github.com/golang/go/wiki/Spectre>
- [93] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, “The spy in the sandbox: Practical cache attacks in javascript and their implications,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1406–1418. [Online]. Available: <https://doi.org/10.1145/2810103.2813708>
- [94] S. Lesimple, “spectre-meltdown-checker script,” 2018. [Online]. Available: <https://github.com/speed47/spectre-meltdown-checker>
- [95] VUsec, “mdstool-cli tool,” 2019. [Online]. Available: <https://github.com/vusec/ridl>
- [96] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, “Sok: The challenges, pitfalls, and perils of using hardware performance counters for security,” 09 2018.
- [97] Google, “Google safeside project,” 2019. [Online]. Available: <https://github.com/google/safeside>
- [98] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” IEEE, pp. 79–93, 2009.
- [99] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution,” in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 142–157.
- [100] Gartner, “Gartner says worldwide iaas public cloud services market grew 31.3in 2018,” 2018. [Online]. Available: <https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-says-worldwide-iaas-public-cloud-services-market-grew-31point3-percent-in-2018>
- [101] A. Mambretti, P. Convertini, A. Sorniotti, A. Sandulescu, E. Kirda, and A. Kurmus, “Ghostbuster: understanding and overcoming the pitfalls of transient execution vulnerability checkers,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 307–317.
- [102] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, “Bypassing memory safety mechanisms through speculative control flow hijacks,” in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 633–649.
- [103] C. Mulliner, W. Robertson, and E. Kirda, “Hidden gems: Automated discovery of access control vulnerabilities in graphical user interfaces,” in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 149–162.
- [104] J. Torrey, “More shadow walker: Tlb-splitting on modern x86,” *Blackhat USA*, 2014.