UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Memory Safety for Today’s Languages and Architectures

Permalink
https://escholarship.org/uc/item/50m2k5wj

Author
Disselkoen, Craig

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Memory Safety for Today's Languages and Architectures

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Craig Disselkoen

Committee in charge:

Professor Deian Stefan, Co-Chair
Professor Dean Tullsen, Co-Chair
Professor Farinaz Koushanfar
Professor Sorin Lerner
Professor Stefan Savage

2022

Copyright
Craig Disselkoen, 2022
All rights reserved.

The dissertation of Craig Disselkoen is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

EPIGRAPH

Whatever you do, work heartily, as for the Lord and not for men

—Colossians 3:23, ESV

TABLE OF CONTENTS

Dissertation Approval Page . . . iii

Epigraph . . . iv

Table of Contents . . . v

List of Figures . . . ix

List of Tables . . . x

Acknowledgements . . . xi

Vita . . . xiv

Abstract of the Dissertation . . . xvi

Introduction . . . 1
- 1 Memory safety . . . 1
- 2 Side-channel attacks and constant-time programming . . . 2
- 3 Spectre attacks . . . 3
- 4 ARM MTE . . . 4
- 5 WebAssembly (Wasm) . . . 5
- 6 Overview . . . 5

Chapter 1 PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX . . . 9
- 1.1 Background and related work . . . 11
  - 1.1.1 Cache attacks . . . 11
  - 1.1.2 Relevant microarchitecture . . . 15
  - 1.1.3 Transactional memory and TSX . . . 19
- 1.2 Potential TSX-based attacks . . . 22
  - 1.2.1 Naïve TSX-based attack . . . 23
  - 1.2.2 PRIME+ABORT–L1 . . . 24
  - 1.2.3 PRIME+ABORT–L3 . . . 25
  - 1.2.4 Finding eviction sets . . . 26
- 1.3 Results . . . 29
  - 1.3.1 Characteristics of the Intel Skylake architecture . . . 29
  - 1.3.2 Dynamically generating eviction sets . . . 30
  - 1.3.3 Detecting memory accesses . . . 32
  - 1.3.4 Attacks on AES . . . 36
- 1.4 Potential countermeasures . . . 37
- 1.5 Disclosure . . . 41

1.6 Conclusion 42

Chapter 2 Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork 43
2.1 Motivation 44
2.1.1 Constant-time programming 44
2.1.2 Constant-time verification 46
2.2 Constant-time verification with Pitchfork 47
2.2.1 Taint propagation 47
2.2.2 Analyzing protocol-level code 50
2.3 Implementation of Pitchfork 51
2.4 Evaluation 54
2.5 Future and related work 58
2.6 Conclusion 59

Chapter 3 Finding Spectre Vulnerabilities in Crypto Code with Pitchfork 61
3.1 Motivating example 63
3.2 Speculative constant-time 64
3.3 Detecting violations 65
3.3.1 Schedule generation 66
3.3.2 Implementation and evaluation 67

Chapter 4 SoK: Practical Foundations for Software Spectre Defenses 69
4.1 Preliminaries 72
4.1.1 Spectre vulnerabilities 73
4.1.2 Breaking cryptography with Spectre 73
4.1.3 Breaking software isolation with Spectre 75
4.1.4 Security properties and execution semantics 76
4.2 Choices in semantics 78
4.2.1 Leakage models 79
4.2.2 Non-interference and policies 85
4.2.3 Execution models 89
4.2.4 Nondeterminism 92
4.2.5 Higher-level abstractions 95
4.2.6 Expressivity and microarchitectural features 98
4.3 Related work 101
4.3.1 Systematization of Spectre attacks and defenses 101
4.3.2 Hardware-based Spectre defenses 102
4.3.3 Software-hardware co-design 103
4.3.4 Other transient execution attacks 103
4.4 Conclusion 104

Chapter 5 Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade 106
5.1 Overview 109
5.1.1 Two kinds of speculative leaks 109
5.1.2 Eliminating speculative leaks 111
5.1.3 Automatically and efficiently repairing speculative leaks . . 113
5.1.4 Attacker model 116
5.2 Implementation 116
5.3 Evaluation 118
5.4 Related work 124
5.5 Limitations and future work 125
5.6 Conclusion 127

Chapter 6 Progressive Memory Safety for WebAssembly 128
6.1 Motivation 131
6.1.1 Memory safety 131
6.1.2 WebAssembly (Wasm) 132
6.1.3 Hardware support for memory safety 134
6.2 Design goals 135
6.3 Design 137
6.3.1 Handles and segments 137
6.3.2 Slicing handles 139
6.3.3 Segment allocation and deallocation 139
6.3.4 Handle integrity 140
6.4 Implementation strategies 142
6.4.1 Spatial safety 142
6.4.2 Handle integrity 144
6.4.3 Temporal safety 145
6.5 Compiling to MSWasm 147
6.6 Discussion 149
6.6.1 Compiling MSWasm 149
6.6.2 Beyond heap memory safety 150
6.6.3 Alternative paths to memory safety 151
6.6.4 Future directions for hardware 152

Chapter 7 DMS: Deterministic Spatial Memory Safety with ARM MTE 154
7.1 Background 155
7.1.1 Memory safety 156
7.1.2 Probabilistic enforcement with ARM MTE 156
7.2 Deterministic spatial safety with DMS 158
7.2.1 Pointer classifications 159
7.2.2 Safety checks for dirty pointers 162
7.3 Conclusion 164

Conclusion 165

Bibliography 166

LIST OF FIGURES

Figure 1.1: Comparison of the operation of various cache attacks 12
Figure 1.2: "Double coverage" of prototype groups generated by Algorithm 1 31
Figure 1.3: Access detection rates for PRIME+ABORT 33
Figure 1.4: Access detection rates for unmodified PRIME+PROBE 33
Figure 1.5: Access detection rates for our modified implementation of PRIME+PROBE 34
Figure 1.6: PRIME+ABORT and PRIME+PROBE attacks against AES 38

Figure 2.1: Excerpt from the function mbedtls_internal_aes_decrypt() 48
Figure 2.2: Excerpt from the function NSC_DecryptFinal() 50
Figure 2.3: Excerpt from the function NSC_Decrypt() 54
Figure 2.4: Excerpt from the function NSC_EncryptUpdate() 57

Figure 4.1: Code snippet which an attacker can exploit using Spectre 74

Figure 5.1: Code fragment adapted from the HACL* SHA2 implementation 110
Figure 5.2: Running example, with two different possible patches 113
Figure 5.3: Subset of the def-use graph of the example program 114
Figure 5.4: Runtime of SHA256 (CT-Wasm) as the workload size varies 123
Figure 5.5: Runtime of SHA256 (CT-Wasm) as the workload size varies, presented on a per-byte basis 123

LIST OF TABLES

Table 1.1: Relevant cache parameters in the Intel Skylake architecture 16
Table 1.2: Availability of Intel TSX in recent Intel CPUs 19
Table 1.3: Causes of transactional aborts in Intel TSX 20
Table 1.4: Runtimes of PRIME+ABORT- and PRIME+PROBE-based versions of Algorithm 1 30

Table 2.1: Functions verified by Pitchfork, up to its assumptions 55

Table 4.1: Comparison of various semantics and tools 80
Table 4.1: Comparison of various semantics and tools, continued 81
Table 4.2: Speculative security properties and their equivalent non-interference statements 88

Table 5.1: Performance results for BLADE 119

ACKNOWLEDGEMENTS

First of all, none of this would have been possible without my amazing coauthors, who contributed to discussions, code, paper text, and just generally providing encouragement and keeping things on track. Specifically (in alphabetical order) thanks to Gilles Barthe, Jay Bosamiya, Fraser Brown, Sunjay Cauligi, Aidan Denlinger, Nathan Froyd, Zhao Gang, Tal Garfinkel, Anitha Gollamudi, Klaus v. Gleissenthall, Radha Jagadeesan, Alan Jeffrey, Ranjit Jhala, Evan Johnson, Rami Kıcı, David Kohlbrenner, Michael LeMay, Sorin Lerner, Amit Levy, Alexandra Michael, Daniel Moghimi, Shravan Narayan, Bryan Parno, Marco Patrignani, Leo Porter, Eric Rahm, John Renner, Tamara Rezk, James Riely, Ravi Sahita, Hovav Shacham, Michael Smith, Deian Stefan, Dean Tullsen, Anjo Vahldiek-Oberwagner, Marco Vassena, and Conrad Watt.

But from this list (and beyond), a few names definitely stand out. Thanks to David Kohlbrenner, who took a new and very green PhD student under his wing for my first year or two. Thanks for showing me the ropes and making me excited about security. Thanks to Rob McGuinness, who despite never being my co-author, somehow still felt like one. Thanks for pushing me to play games and have fun and see friends once in a while. And thanks in particular to Sunjay Cauligi, Shravan Narayan, and John Renner, who helped keep me (mostly) sane during the long marathon that is a PhD. You gave me coding help during the frequent times I found myself stuck; you were always free for a tangential discussion, coffee break, or a walk across campus; and you were there for me when I needed you.

A very especial thanks to my advisors, Dean Tullsen and Deian Stefan. To Dean for taking a chance on a less-than-typical PhD application, and supporting my gradual transition into language-based security even though his main experience and expertise were elsewhere. And to Deian for believing in me, putting up with my faults, and teaching me so much about writing, talking, and research. This PhD wouldn't have happened without either of you.

Thanks to my parents Brent and Beth Disselkoen for being awesome and supportive and always good sources of advice. (And for not complaining when we stubbornly continue to live far away from Iowa.)

And finally, thank you to my wife Monica, who has truly been by my side through all of this. Thanks for moving across the country with me, for being the breadwinner of the family, and for believing in me every step of the way.

The Introduction, in part, uses material from all works listed below.

Chapter 1, in part, is a reprint of the material as it appears in the USENIX Security Symposium 2017. Disselkoen, Craig; Kohlbrenner, David; Porter, Leo; Tullsen, Dean. USENIX, 2017. The dissertation author was the primary investigator and author of this material.

Chapter 2, in part, is a reprint of the material as it was submitted to TECHCON 2020. Disselkoen, Craig; Cauligi, Sunjay; Tullsen, Dean; Stefan, Deian. SRC, 2020. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, contains material reprinted from the Proceedings of the ACM Conference on Programming Language Design and Implementation. Cauligi, Sunjay; Disselkoen, Craig; v. Gleissenthall, Klaus; Tullsen, Dean; Stefan, Deian; Rezk, Tamara; Barthe, Gilles. ACM, 2020. The dissertation author was the primary investigator and author of the reprinted material.

Chapter 4, in part, is a reprint of the material as it appears in the 43rd IEEE Symposium on Security and Privacy (S&P '22). Cauligi, Sunjay; Disselkoen, Craig; Moghimi, Daniel; Barthe, Gilles; Stefan, Deian. IEEE, 2022. The dissertation author was a primary investigator and author of this material.

Chapter 5, in part, contains material reprinted from the Proceedings of the ACM on Programming Languages (Issue POPL), 2021. Vassena, Marco; Disselkoen, Craig; v. Gleissenthall, Klaus; Cauligi, Sunjay; Kıcı, Rami; Jhala, Ranjit; Tullsen, Dean; Stefan, Deian. ACM, 2021. The dissertation author was the primary investigator and author of the reprinted material.

Chapter 6, in part, is a reprint of the material as it appears in the Workshop on Hardware and Architectural Support for Security and Privacy (HASP), 2019. Disselkoen, Craig; Renner, John; Watt, Conrad; Garfinkel, Tal; Levy, Amit; Stefan, Deian. ACM, 2019. The dissertation author was the primary investigator and author of this material.

Chapter 6 also contains material currently under submission for publication. Michael, Alexandra; Gollamudi, Anitha; Disselkoen, Craig; Denlinger, Aidan; Bosamiya, Jay; Watt, Conrad; Parno, Bryan; Patrignani, Marco; Vassena, Marco; Stefan, Deian. The dissertation author was the primary investigator and author of the reprinted material.

Chapter 7, in part, is currently being prepared for submission for publication of the material. Disselkoen, Craig; Tullsen, Dean; Stefan, Deian. The dissertation author was the primary investigator and author of this material.

VITA

2015	Bachelor of Science in Engineering, Computer Emphasis Dordt University
2015	Bachelor of Arts, Mathematics Dordt University
2015-2016	Research and Teaching Assistant Dordt University
2016-2022	Graduate Research Assistant, Computer Science University of California San Diego
2017	Engineering Intern (PhD) Qualcomm
2018	Research Intern (PhD) Mozilla
2019	Master of Science, Computer Science University of California San Diego
2021	Technical Intern (PhD) Correct Computation
2022	Doctor of Philosophy, Computer Science University of California San Diego

PUBLICATIONS

C. Disselkoen, D. Kohlbrenner, L. Porter, D. Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX." USENIX Security Symposium, August 2017.

M. Smith, C. Disselkoen, S. Narayan, F. Brown, D. Stefan. "Browser history re:visited." USENIX Workshop on Offensive Technologies (WOOT), August 2018.

C. Disselkoen, R. Jagadeesan, A. Jeffrey, J. Riely. "Code That Never Ran: Modeling Attacks on Speculative Evaluation." IEEE Symposium on Security and Privacy (S&P), May 2019.

C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, D. Stefan. "Position Paper: Progressive Memory Safety for WebAssembly." Workshop on Hardware and Architectural Support for Security and Privacy (HASP), June 2019.

S. Cauligi, C. Disselkoen, K. v Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, G. Barthe. "Constant-Time Foundations for the New Spectre Era." 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2020.

S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, D. Stefan. "Retrofitting Fine Grain Isolation in the Firefox Renderer." USENIX Security Symposium, August 2020.

C. Disselkoen, S. Cauligi, D. Tullsen, D. Stefan. "Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork." TECHCON, September 2020.

M. Vassena, C. Disselkoen, K. v Gleissenthall, S. Cauligi, R. Kıcı, R. Jhala, D. Tullsen, D. Stefan. "Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade." 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), January 2021.

S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A.Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, D. Stefan. "Swivel: Hardening WebAssembly against Spectre." USENIX Security Symposium, August 2021.

S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, D. Stefan. "SoK: Practical Foundations for Spectre Defenses." IEEE Symposium on Security and Privacy (S&P), May 2022.

A. Michael, A. Gollamudi, C. Disselkoen, A. Denlinger, J. Bosamiya, C. Watt, B. Parno, M. Patrignani, M. Vassena, D. Stefan. "MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code." In submission.

C. Disselkoen, D. Tullsen, M. LeMay, D. Stefan. "DMS: Deterministic Spatial Memory Safety with ARM MTE." Unpublished.

ABSTRACT OF THE DISSERTATION

Memory Safety for Today's Languages and Architectures

by

Craig Disselkoen

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Deian Stefan, Co-Chair
Professor Dean Tullsen, Co-Chair

Memory safety vulnerabilities remain one of the most critical sources of exploitable security problems in today's software. Despite the growing popularity of modern, memory-safe languages, much of today's software remains written in C and C++, which are prone to these vulnerabilities; and rewriting all of this C and C++ code would be prohibitively expensive and time-consuming. At the same time, microarchitectural side-channel attacks threaten to violate memory safety in increasingly complex ways. But, new languages such as WebAssembly (Wasm), and new hardware features such as ARM MTE, give programmers new tools in the fight against

memory safety vulnerabilities — and with clever use of these tools, we can obtain strong security guarantees for today's software.

In this dissertation, we present a variety of tools for improving memory safety for today's C and C++ codebases, on today's side-channel-prone microarchitectures. In the domain of *finding* memory-safety vulnerabilities, we first demonstrate how new microarchitectural features sometimes introduce new side-channel attacks (Chapter 1); then, we present program analysis tools which help keep programs secure from that class of side-channel attacks (Chapter 2) and from a newer and particularly relevant class of side-channel attacks, Spectre attacks (Chapter 3). In the remainder of the dissertation we focus on automatically *preventing* memory-safety vulnerabilities. We systematically compare and critique proposed software-based defenses against Spectre (Chapter 4); then we present one such defense, a tool which automatically and efficiently secures cryptographic programs against Spectre (Chapter 5). Starting with Chapter 6 we return to non-side-channel memory safety vulnerabilities, proposing an extension to Wasm which provides memory safety even *inside* its software sandbox; and finally, in Chapter 7 we present a compiler-based defense which works in conjunction with ARM MTE to automatically secure C and C++ programs from spatial memory safety vulnerabilities.

Introduction

Every day, attackers use exploits to hijack devices, leak confidential information, and spread malware. Even widely-used and thoroughly-tested software is routinely found to contain exploitable vulnerabilities: In 2021 alone, for instance, the CVE database recorded 308 vulnerabilities in Google Chrome, 485 in Windows 10, 312 in macOS, 574 in Android, and 360 in iOS [45]. In one extreme but noteworthy example of the damage that can be caused by these kind of software vulnerabilities, in 2018, according to Citizen Lab and UN reports [10], the Saudi Arabian government used exploits developed by the NSO Group to spy on journalists and dissidents, leading to the murder of Saudi Arabian journalist Jamal Khashoggi. Understandably, one major focus of the security community is defending against these exploits, and the software vulnerabilities they rely on.

1 Memory safety

In a large majority of today's exploits, attackers are leveraging *memory safety* vulnerabilities in existing code. Memory safety vulnerabilities include, e.g., *out-of-bounds* violations, where an attacker can access memory outside of the intended array or object; and *use-after-free* errors (UAFs), where an attacker can continue to access the memory assigned to an array or object after that memory has been freed and reused for a different purpose. Memory safety problems are widely exploited in real systems: They represent around 70% of all vulnerabilities in Microsoft

products, according to a 2019 Microsoft report [151]; and they have also caused around 70% of high- or critical-severity security bugs in Chromium since 2015 [41].

One obvious solution to memory-safety vulnerabilities is to rewrite all of our security-critical software in memory-safe programming languages, such as Rust or Go. However, today's software systems are enormous, and rewriting them in a new language is an onerous proposition. For instance, Chromium alone contains over 14 million lines of C and C++ [201]; rewriting it would be prohibitively expensive and time-consuming.

In this dissertation, we propose ways to automatically improve memory safety for the huge body of existing code already written in C and C++. If we can enforce memory safety automatically and invisibly to the programmer, we reap the security benefits without incurring the tremendous cost of rewriting in a safe language.

2 Side-channel attacks and constant-time programming

Today's systems have even more kinds of memory safety problems to worry about than those of yesteryear. Not only do systems still have an abundance of "traditional" memory safety vulnerabilities, such as out-of-bounds violations and UAFs, but in recent years researchers have repeatedly demonstrated how attackers can use *side-channel attacks* to violate memory safety in more insidious ways. In particular, using a side-channel attack, an attacker may be able to learn values stored in out-of-bounds memory, without directly reading the values: The attacker can simply watch the changes made to microarchitectural state by the target program. In this dissertation, the side channel we focus on is the CPU cache: how secret information may influence the CPU cache state, and how an attacker can exfiltrate the secret information from the CPU cache state even without direct read access to the targeted memory.

The de-facto approach to defending critical code — in particular cryptographic code — from cache side-channel attacks is *constant-time programming*. Constant-time programming can be distilled down to three rules:

1. Secret values must not influence the program's control flow;
2. Secret values must not influence the addresses of memory accesses;
3. Secret values must not influence the inputs to any variable-time machine operation (such as integer division on many processors).

Unfortunately, it is notoriously hard to write constant-time code: Not only do experts fail to adequately write truly constant-time code [20, 26, 152], but even the process of fixing these mistakes can lead to further vulnerabilities [4, 199]. In this dissertation, we not only describe novel ways for attackers to exploit cache side channels, but we also propose a method (and tool) for developers to ensure their code correctly follows constant-time programming.

3 Spectre attacks

Even if the target program is written extremely carefully — using bounds checks and constant-time programming techniques — attackers can often still leak the program's secrets anyway by turning to *transient execution attacks* such as Spectre. Spectre attacks exploit particular microarchitectural features, such as branch predictors, which are widely adopted by today's processors. With these features, the processor may ignore important safety checks when executing speculatively, relying on its ability to roll back execution if its prediction was incorrect. Unfortunately, today's processors do not roll back the state of many microarchitectural structures — in particular, the CPU cache state. Thus, an attacker can learn secret data by observing changes to the cache state made by transiently executed code. Software defenses for Spectre need to go beyond the safety checks and constant-time programming principles which were sufficient for

other kinds of cache side-channel attacks. In this dissertation, we examine already-proposed software Spectre defenses and their theoretical underpinnings; and even further, we propose novel software defenses of our own.

4 ARM MTE

While microarchitectural features can be the cause of high-profile vulnerabilities (e.g., cache side-channel attacks and Spectre), they can also be part of the broader solution for all types of memory safety vulnerabilities. For one example relevant to this dissertation, ARM recently introduced its Memory Tagging Extension (MTE) [12]. MTE adds a 4-bit hardware tag to each 16-byte (aligned) *granule* of memory, and also repurposes 4 otherwise-unused bits of virtual address as a tag value for each pointer. On every pointer dereference (load or store), the hardware compares the pointer's tag value to the tag value of the memory being accessed, and if they do not match, a fault is generated.

MTE hardware naturally lends itself to providing low-overhead memory-safety enforcement, as described in ARM's own whitepaper [12]. When an MTE-aware memory allocator chooses a tag value (perhaps randomly) for each new memory allocation, the hardware provides probabilistic spatial and temporal safety. For instance, suppose a pointer with tag value `T` is incremented to point out-of-bounds of its intended memory allocation. With high probability, the out-of-bounds memory will have a tag value other than `T`, so when the pointer is dereferenced, the tags will mismatch, causing a hardware fault. Or suppose a memory region with tag value `T` is freed and reallocated. If an old (dangling) pointer with tag value `T` attempts to access the now-reallocated memory, with high probability the memory now has a tag value other than `T`, so the tags will mismatch, causing a hardware fault.

In this dissertation, we suggest novel ways to use MTE (e.g., to improve the performance of MSWasm in Chapter 6), and we also propose a system which builds on top of MTE's probabilistic

protection in order to provide fully deterministic spatial memory safety through cooperating hardware and software checks.

5 WebAssembly (Wasm)

One of today's tools for fighting memory safety vulnerabilities is WebAssembly (Wasm). Wasm is a platform-independent bytecode designed to run C/C++ and similar languages at near native speed in the browser. Wasm is designed to allow browsers to run code in a sandbox, isolating the impact of memory safety vulnerabilities in Wasm code from the rest of the browser. Unfortunately, keeping the browser safe from Wasm code is not the same as keeping Wasm code safe from itself — isolation doesn't prevent attackers from exploiting memory-safety bugs to compromise the Wasm code and any data it handles. In this dissertation, we propose defense mechanisms for memory safety vulnerabilities *inside* the Wasm sandbox; and we also use Wasm as a solid starting point for defenses against more sophisticated attacks, namely Spectre.

6 Overview

In this dissertation we address the problem of memory safety vulnerabilities from several different angles, with a particular emphasis on how these vulnerabilities can be detected and prevented in today's languages and on today's architectures.

In Chapter 1, we demonstrate how a then-novel Intel architectural feature called TSX (Transactional Synchronization Extensions) can actually open programs up to a new kind of memory safety vulnerability. Specifically, we show how attackers can use TSX to leak the contents of memory they shouldn't have access to. We present an attack PRIME+ABORT which is reminiscent of the well-known cache side-channel attack PRIME+PROBE, but outperforms

PRIME+PROBE in both accuracy and efficiency — and does so without relying on timers, allowing it to bypass many common side-channel defense mechanisms.

In Chapter 2, we present a novel tool called Pitchfork that helps programmers keep their programs secure from side-channel attacks — including the PRIME+ABORT attack described in Chapter 1 — by automatically finding violations of the constant-time programming paradigm. Pitchfork uses symbolic execution to find code locations where constant-time programming is violated and secret data may be leaked. Critically, Pitchfork is designed to analyze not only cryptographic primitives, but also protocol-level cryptographic code — flaws in which have been responsible for high-profile vulnerabilities such as Lucky 13 [4]. We describe how we used Pitchfork to verify protocol-level code in both `libsignal` [198] and Mozilla's NSS cryptographic library [157]. Our verification effort, however, also revealed several constant-time vulnerabilities in NSS, including a critical memory-safety vulnerability which was assigned CVE-2019-11745.

In Chapter 3, we extend Pitchfork's analysis to speculative execution, showing how a variation of Pitchfork can be used to find Spectre vulnerabilities in cryptographic code. This version of Pitchfork found Spectre vulnerabilities in protocol-level code in the cryptographic libraries libsodium and OpenSSL.

Starting with Chapter 4, we turn our attention from *finding* memory-safety vulnerabilities in existing code, to automatically *preventing* them in today's languages and on today's architectures. In Chapter 4, we present a novel systematization of the wide variety of software-based Spectre defenses created or proposed from 2018 (when Spectre became public) until the end of 2021, comparing these defenses in terms of the strength of their theoretical underpinnings and the resulting security guarantees. Based on our analysis, we present opinionated recommendations about the best directions for this research area moving forward. Specifically, we encourage developers to use leakage models derived from constant-time programming, rather than weaker models which only consider leaks via the data cache; we emphasize the necessity of considering multiple Spectre variants rather than just Spectre-PHT (even if the program analysis tool itself only directly

reasons about Spectre-PHT); and we recommend *against* modelling microarchitectural details such as intricate cache structures or port contention, which introduce unnecessary complexity and sacrifice portability.

In Chapter 5, we present BLADE, a tool which automatically secures cryptographic code against Spectre attacks. BLADE is built on the insight that to stop leaks via speculative execution, it suffices to *cut* the dataflow from expressions that speculatively introduce secrets (*sources*) to those that leak them through the cache (*sinks*), rather than prohibit speculation altogether. We formalize this insight in a static type system that types each expression as either *transient*, i.e., possibly containing speculative secrets, or as being *stable*; BLADE prevents speculative leaks by requiring that all *sink* expressions are stable. Finally, we implement BLADE in the Cranelift WebAssembly compiler, where it ensures that attackers cannot break out of the Wasm sandbox even using (particular variants of) Spectre attacks. We evaluate our approach by repairing several verified, yet vulnerable WebAssembly implementations of cryptographic primitives, and find that BLADE can fix existing programs that leak via speculation *automatically*, without user intervention, and *efficiently*, imposing less than 20% performance overhead.

In Chapter 6, we return to traditional (non-side-channel) memory safety vulnerabilities, such as out-of-bounds and use-after-free. We show how, despite Wasm's strong sandboxing, these vulnerabilities remain a concern even inside the Wasm sandbox. Then we propose MSWasm, which automatically prevents these vulnerabilities in Wasm code. MSWasm captures important low-level C/C++ memory semantics such as pointers and memory allocation in Wasm, at compile time, allowing MSWasm implementations to leverage this information at deployment time to enforce memory safety. We discuss how MSWasm can be efficiently implemented on today's and tomorrow's architectures, using features such as ARM MTE. And we present a full-scale compiler based on CHERI LLVM which compiles C/C++ to MSWasm, showing how MSWasm can enforce memory safety for arbitrary C/C++ code.

Finally, in Chapter 7, we present DMS, a compiler-based defense which automatically secures arbitrary C/C++ programs against memory safety vulnerabilities, without requiring them to be compiled through Wasm. DMS supplements ARM MTE with a layer of software checks in order to achieve fully deterministic spatial memory safety enforcement, with much better performance than would be possible in software alone. DMS identifies which memory accesses are provably (deterministically) safe, and which accesses may "slip through the cracks" of MTE's probabilistic enforcement. Then, DMS inserts software checks at each insecure memory access. Together, MTE's hardware enforcement and DMS's software checks provide fully deterministic spatial safety for legacy applications. With DMS, we can secure today's programs from memory safety vulnerabilities on tomorrow's architectures.

Chapter 1

PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX

State-of-the-art cache attacks [75, 82, 104, 114, 136, 169, 170, 174, 233] leverage differences in memory access times between levels of the cache and memory hierarchy to gain insight into the activities of a victim process. These attacks require the attacker to frequently perform a series of timed memory operations (or cache management operations [75]) to learn if a victim process has accessed a critical address (e.g., a statement in an encryption library).

These attacks are highly dependent on precise and accurate timing, and defenses can exploit this dependence. In fact, a variety of defenses have been proposed which undermine these timing-based attacks by restricting access to highly precise timers [92, 124, 144, 212].

In this chapter, we introduce an alternate mechanism for performing cache attacks, which does not leverage timing differences (timing side channels) or require timed operations of any type. Instead, it exploits Intel's implementation of Hardware Transactional Memory, which is

called TSX [103]. We demonstrate a novel cache attack based on this mechanism, which we will call PRIME+ABORT.

The intent of Transactional Memory (and TSX) is to both provide a simplified interface for synchronization and to enable optimistic concurrency: processes abort only when a conflict exists, rather than when a potential conflict may occur, as with traditional locks [86, 89]. Transactional memory operations require transactional data to be buffered, in this case in the cache which has limited space. Thus, the outcome of a transaction depends on the state of the cache, potentially revealing information to the thread that initiates the transaction. By exploiting TSX, an attacker can monitor the cache behavior of another process and receive an abort (call-back) if the victim process accesses a critical address. This chapter demonstrates how TSX can be used to trivially detect writes to a shared block in memory; to detect reads and writes by a process co-scheduled on the same core; and, most critically, to detect reads and writes by a process executing anywhere on the same processor. This latter attack works across cores, does not assume that the victim uses or even knows about TSX, and does not require any form of shared memory.

The advantages of this mechanism over conventional cache attacks are twofold. The first is that PRIME+ABORT does not leverage any kind of timer; as mentioned, several major classes of countermeasures against cache attacks revolve around either restricting access or adding noise to timers. PRIME+ABORT effectively bypasses these countermeasures.

The second advantage is in the efficiency of the attack. The TSX hardware allows for a victim's action to directly trigger the attacking process to take action. This means the TSX attack can bypass the detection phase required in conventional attacks. Direct coupling from event to handler allows PRIME+ABORT to provide over $3\times$ the throughput of comparable state-of-the-art attacks.

The rest of this chapter is organized as follows. Section 1.1 presents background and related work; Section 1.2 introduces our novel attack, PRIME+ABORT; Section 1.3 describes

experimental results, making comparisons with existing methods; in Section 1.4, we discuss potential countermeasures to our attack; and Section 1.6 concludes the chapter.

1.1 Background and related work

1.1.1 Cache attacks

Cache attacks [75, 82, 104, 114, 136, 169, 170, 174, 233] are a well-known class of side-channel attacks which seek to gain information about which memory locations are accessed by some victim program, and at what times. In an excellent survey, Ge et al. [67] group such attacks into three broad categories: PRIME+PROBE, FLUSH+RELOAD, and EVICT+TIME. Since EVICT+TIME is only capable of monitoring memory accesses at the program granularity (whether a given memory location was accessed during execution or not), in this paper we focus on PRIME+PROBE and FLUSH+RELOAD, which are much higher resolution and have received more attention in the literature. Cache attacks have been shown to be effective for successfully recovering AES [114], ElGamal [136], and RSA [233] keys, performing keylogging [76], and spying on messages encrypted with TLS [106].

Figure 1.1 outlines all of the attacks which we will consider. At a high level, each attack consists of a pre-attack portion, in which important architecture- or runtime-specific information is gathered; and then an active portion which uses that information to monitor memory accesses of a victim process. The active portion of existing state-of-the-art attacks itself consists of three phases: an *initialization* phase, a *waiting* phase, and a *measurement* phase. The initialization phase prepares the cache in some way; the waiting phase gives the victim process an opportunity to access the target address; and then the measurement phase performs a timed operation to determine whether the cache state has changed in a way that implies an access to the target address has taken place.

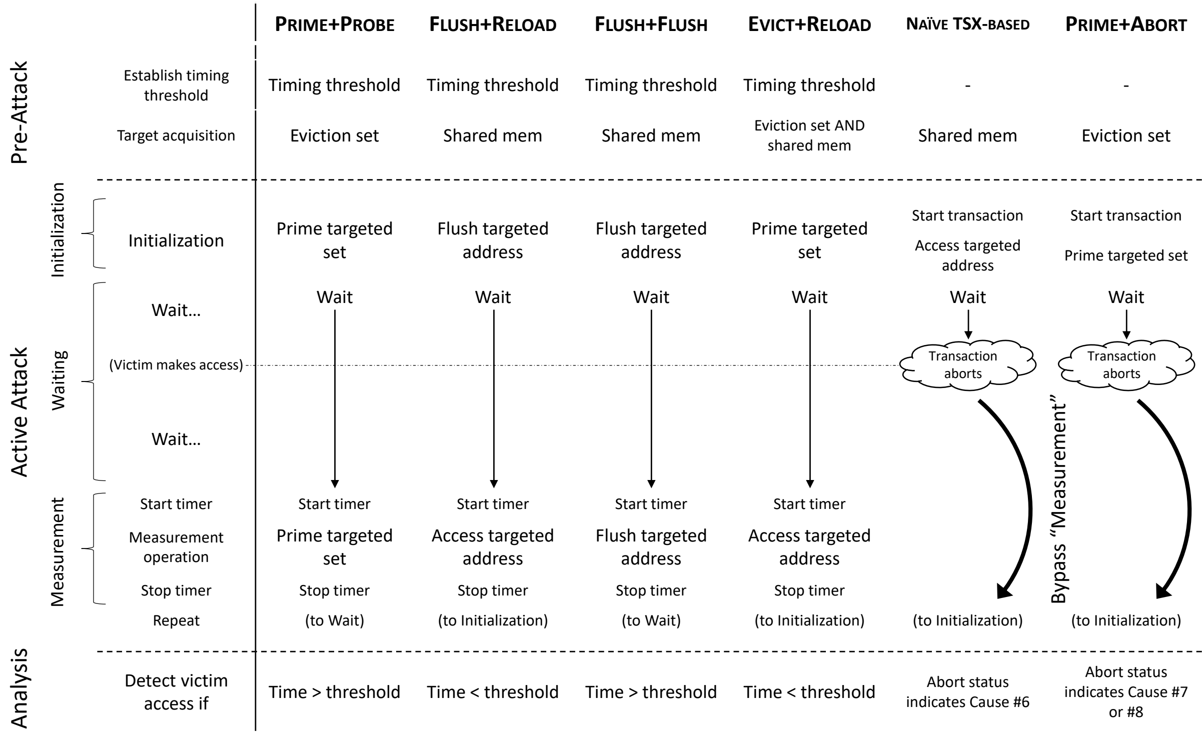


Figure 1.1: Comparison of the operation of various cache attacks, including our novel attacks.

Specifics of the initialization and measurement phases vary by cache attack (discussed below). Some cache attack implementations make a tradeoff in the length of the waiting phase between accuracy and resolution—shorter waiting phases give more precise information about the timing of victim memory accesses, but may increase the relative overhead of the initialization and measurement phases, which may make it more likely that a victim access could be *missed* by occurring outside of one of the measured intervals. In our testing, not all cache attack implementations and targets exhibited obvious experimental tradeoffs for the waiting phase duration. Nonetheless, fundamentally, all of these existing attacks can only gain temporal information at the waiting-interval granularity.

1.1.1.1 PRIME+PROBE

PRIME+PROBE [104, 114, 136, 170, 174] is the oldest and largest family of cache attacks, and also the most general. PRIME+PROBE does not rely on shared memory, unlike most other

cache attacks (including FLUSH+RELOAD and its variants, described below). The original form of PRIME+PROBE [170, 174] targets the L1 cache, but recent work [104, 114, 136] extends it to target the L3 cache in Intel processors, enabling PRIME+PROBE to work across cores and without relying on hyperthreading (Simultaneous Multithreading [205]). Like all L3 cache attacks, L3 PRIME+PROBE can detect accesses to either instructions or data; in addition, L3 PRIME+PROBE trivially works across VMs.

PRIME+PROBE targets a single cache set, detecting accesses by any other program (or the operating system) to any address in that cache set. In its active portion's initialization phase (called "prime"), the attacker accesses enough cache lines from the cache set so as to completely fill the cache set with its own data. Later, in the measurement phase (called "probe"), the attacker reloads the same data it accessed previously, this time carefully observing how much time this operation took. If the victim did not access data in the targeted cache set, this operation will proceed quickly, finding its data in the cache. However, if the victim accessed data in the targeted cache set, the access will evict a portion of the attacker's primed data, causing the reload to be slower due to additional cache misses. Thus, a slow measurement phase implies the victim accessed data in the targeted cache set during the waiting phase. Note that this "probe" phase can also serve as the "prime" phase for the next repetition, if the monitoring is to continue.

Two different kinds of initial one-time setup are required for the pre-attack portion of this attack. The first is to establish a timing threshold above which the measurement phase is considered "slow" (i.e. likely suffering from extra cache misses). The second is to determine a set of addresses, called an *eviction set*, which all map to the same (targeted) cache set (and which reside in distinct cache lines). Finding an eviction set is much easier for an attack targeting the L1 cache than for an attack targeting the L3 cache, due to the interaction between cache addressing and the virtual memory system, and also due to the "slicing" in Intel L3 caches (discussed further in Sections 1.1.2.1 and 1.1.2.2).

1.1.1.2 FLUSH+RELOAD

The other major class of cache attacks is FLUSH+RELOAD [82, 233]. FLUSH+RELOAD targets a specific address, detecting an access by any other program (or the operating system) to that exact address (or another address in the same cache line). This makes FLUSH+RELOAD a much more precise attack than PRIME+PROBE, which targets an entire cache set and is thus more prone to noise and false positives. FLUSH+RELOAD also naturally works across cores because of shared, inclusive, L3 caches (as explained in Section 1.1.2.1). Again, like all L3 cache attacks, FLUSH+RELOAD can detect accesses to either instructions or data. Additionally, FLUSH+RELOAD can work across VMs via the page deduplication exploit [233].

The pre-attack of FLUSH+RELOAD, like that of PRIME+PROBE, involves determining a timing threshold, but is limited to a single line instead of an entire "prime" phase. However, FLUSH+RELOAD does not require determining an eviction set. Instead, it requires the attacker to identify an exact target address; namely, an address in the attacker's virtual address space which maps to the physical address the attacker wants to monitor. Yarom and Falkner [233] present two ways to do this, both of which necessarily involve shared memory; one exploits shared libraries, and the other exploits page deduplication, which is how FLUSH+RELOAD can work across VMs. Nonetheless, this step's reliance on shared memory is a critical weakness in FLUSH+RELOAD, limiting it to only be able to monitor targets in shared memory.

In FLUSH+RELOAD's initialization phase, the attacker "flushes" the target address out of the cache using Intel's `CLFLUSH` instruction. Later, in the measurement phase, the attacker "reloads" the target address (by accessing it), carefully observing the time for the access. If the access was "fast", the attacker may conclude that another program accessed the address, causing it to be reloaded into the cache.

An improved variant of FLUSH+RELOAD, FLUSH+FLUSH [75], exploits timing variation in the `CLFLUSH` instruction itself; this enables the attack to combine its measurement and initialization phases, much like PRIME+PROBE. A different variant, EVICT+RELOAD [76], performs

the initialization phase by evicting the cacheline with PRIME+PROBE's "prime" phase, allowing the attack to work without the `CLFLUSH` instruction at all—e.g., when the instruction has been disabled, as in Google Chrome's NaCl [71].

1.1.1.3 Timer-free cache attacks

All of the attacks so far discussed—PRIME+PROBE, FLUSH+RELOAD, and variants—are still fundamentally timing attacks, exploiting timing differences as their underlying attack vector. One recent work which, like this work, proposes a cache attack without reference to timers is that of Guanciale et al. [79]. Instead of timing side channels, Guanciale et al. rely on the undocumented hardware behavior resulting from disobeying ISA programming guidelines, specifically with regards to virtual address aliasing and self-modifying code. However, they demonstrate their attacks only on the ARM architecture, and they themselves suggest that recent Intel x86-64 processors contain mechanisms that would render their attacks ineffective. In contrast, our attack exploits weaknesses specifically in recent Intel x86-64 processors, so in that respect our attack can be seen as complementary to Guanciale et al.'s work. We believe that our work, in addition to utilizing a novel attack vector (Intel's hardware transactional memory support), is the first timer-free cache attack to be demonstrated on commodity Intel processors.

1.1.2 Relevant microarchitecture

1.1.2.1 Caches

Basic background

Caches in modern processors store data that is frequently or recently used, in order to reduce access time for that data on subsequent references. Data is stored in units of *cache lines* (a fixed architecture-dependent number of bytes). Caches are often organized hierarchically, with a small but fast "L1" cache, a medium-sized "L2" cache, and a large but comparatively slower "L3"

Table 1.1: Relevant cache parameters in the Intel Skylake architecture

	L1-Data	L1-Inst	L2	L3
Size	32 KB	32 KB	256 KB	2-8 MB[1]
Assoc	8-way	8-way	4-way	16-way
Sharing	Per-core	Per-core	Per-core	Shared
Line size	64 B	64 B	64 B	64 B

[1] depending on model. This range covers all Skylake processors (server, desktop, mobile, embedded) currently available as of January 2017 [96].

cache. At each level of the hierarchy, there may either be a dedicated cache for each processor core, or a single cache shared by all processor cores.

Commonly, caches are *set-associative*, which allows any given cacheline to reside in only one of N locations in the cache, where N is the *associativity* of the cache. This group of N locations is called a *cache set*. Each cacheline is assigned to a unique cache set by means of its *set index*, typically a subset of its address bits. Once a set is full (the common case), any access to a cacheline with the given set index (but not currently in the cache) will cause one of the existing N cachelines with the same set index to be removed, or *evicted*, from the cache.

Intel cache organization

Recent Intel processors contain per-core L1 instruction and data caches, per-core unified L2 caches, and a large L3 cache which is shared across cores. In this paper we focus on the Skylake architecture which was introduced in late 2015; important Skylake cache parameters are provided in Table 1.1.

Inclusive caches

Critical to all cross-core cache attacks, the L3 cache is inclusive, meaning that everything in all the per-core caches must also be held in the L3. This has two important consequences which are key to enabling both L3-targeting PRIME+PROBE and FLUSH+RELOAD to work across cores. First, any data accessed by any core must be brought into not only the core's private L1 cache, but also the L3. If an attacker has "primed" a cache set in the L3, this access to a

different address by another core necessarily evicts one of the attacker's cachelines, allowing PRIME+PROBE to detect the access. Second, any cacheline evicted from the L3 (e.g., in a "flush" step) must also be invalidated from all cores' private L1 and L2 caches. Any subsequent access to the cacheline by any core must fetch the data from main memory and bring it to the L3, causing FLUSH+RELOAD's subsequent "reload" phase to register a cache hit.

Set index bits

The total number of cache sets in each cache can be calculated as (total number of cache lines) / (associativity), where the total number of cache lines is (cache size) / (line size). Thus, the Skylake L1 caches have 64 sets each, the L2 caches have 1024 sets each, and the shared L3 has from 2K to 8K sets, depending on the processor model.

In a typical cache, the lowest bits of the address (called the *line offset*) determine the position within the cache line; the next-lowest bits of the address (called the *set index*) determine in which cache set the line belongs, and the remaining higher bits make up the *tag*. In our setting, the line offset is always 6 bits, while the set index will vary from 6 bits (L1) to 13 bits (L3) depending on the number of cache sets in the cache.

Cache slices and selection hash functions

However, in recent Intel architectures (including Skylake), the situation is more complicated than this for the L3. Specifically, the L3 cache is split into several *slices* which can be accessed concurrently; the slices are connected on a ring bus such that each slice has a different latency depending on the core. In order to balance the load on these slices, Intel uses a proprietary and undocumented hash function, which operates on a physical address (except the line offset) to select which slice the address "belongs" to. The output of this hash effectively serves as the top N bits of the set index, where 2^N is the number of slices in the system. Therefore, in the case of an 8 MB L3 cache with 8 slices, the set index consists of 10 bits from the physical address and 3 bits calculated using the hash function. For more details, see [114], [145], [234], [93], or [105].

This hash function has been reverse-engineered for many different processors in Intel's Sandy Bridge [114, 145, 234], Ivy Bridge [93, 105, 145], and Haswell [105, 145] architectures, but to our knowledge has not been reverse-engineered for Skylake yet. Not knowing the precise hash function adds additional difficulty to determining eviction sets for PRIME+PROBE—that is, finding sets of addresses which all map to the same L3 cache set. However, our attack (following the approach of Liu et al. [136]) does not require knowledge of the specific hash function, making it more general and more broadly applicable.

1.1.2.2 Virtual memory

In a modern virtual memory system, each process has a set of *virtual addresses* which are mapped by the operating system and hardware to *physical addresses* at the granularity of pages [53]. The lowest bits of an address (referred to as the page offset) remain constant during address translation. Pages are typically 4 KB in size, but recently larger pages, for instance of size 2 MB, have become widely available for use at the option of the program [114, 136]. Crucially, an attacker may choose to use large pages regardless of whether the victim does or not [136].

Skylake caches are physically-indexed, meaning that the physical address of a cache line (and not its virtual address) determines the cache set which the line is mapped into. Like the slicing of the L3 cache, physical indexing adds additional difficulty to the problem of determining eviction sets for PRIME+PROBE, as it is not immediately clear which virtual addresses may have the same set index bits in their corresponding physical addresses. Pages make this problem more manageable, as the bottom 12 bits (for standard 4 KB pages) of the address remain constant during translation. For the L1 caches, these 12 bits contain the entire set index (6 bits of line offset + 6 bits of set index), so it is easy to choose addresses with the same set index. This makes the problem of determining eviction sets trivial for L1 attacks. However, L3 attacks must deal with both physical indexing and cache slicing when determining eviction sets. Using large pages helps, as the 21-bit large-page offset completely includes the set index bits (meaning they

Table 1.2: Availability of Intel TSX in recent Intel CPUs, based on data drawn from Intel ARK [96] in January 2017. Since Broadwell, all server CPUs and a majority of i7/i5 CPUs support TSX.

Series (Release[1])	Server[2]	i7/i5	i3/m/etc[3]
Kaby Lake (Jan 2017)	3/3 (100%)	23/32 (72%)	12/24 (50%)
Skylake (Aug 2015)	23/23 (100%)	27/42 (64%)	4/34 (12%)
Broadwell (Sep 2014)	77/77 (100%)	11/22 (50%)	2/18 (11%)
Haswell (Jun 2013)	37/85 (44%)	2/87 (2%)	0/82 (0%)

[1] for the earliest available processors in the series
[2] Xeon and Pentium-D
[3] (i3/m/Pentium/Celeron)

remain constant during translation), leaving only the problem of the hash function. However, the hash function is not only an unknown function itself, but it also incorporates bits from the entire physical address, including bits that are still translated even when using large pages.

1.1.3 Transactional memory and TSX

Transactional Memory (TM) has received significant attention from the computer architecture and systems community over the past two decades [87, 89, 195, 235]. First proposed by Herlihy and Moss in 1993 as a hardware alternative to locks [89], TM is noteworthy for its simplification of synchronization primitives and for its ability to provide optimistic concurrency.

Unlike traditional locks which require threads to wait if a conflict is possible, TM allows multiple threads to proceed in parallel and only abort in the event of a conflict [181]. To detect a conflict, TM tracks each thread's read and write sets and signals an abort when a conflict is found. This tracking can be performed either by special hardware [87, 89, 235] or software [195].

Intel's TSX instruction set extension for x86 [86, 103] provides an implementation of hardware TM and is widely available in recent Intel CPUs (see Table 1.2).

Table 1.3: Causes of transactional aborts in Intel TSX

1. Executing certain instructions, such as CPUID or the explicit XABORT instruction
2. Executing system calls
3. OS interrupts[1]
4. Nesting transactions too deeply
5. Access violations and page faults
6. Read-Write or Write-Write memory conflicts with other threads or processes (including other cores) at the cacheline granularity—whether those other processes are using TSX or not
7. A cacheline which has been written during the transaction (i.e., a cacheline in the transaction's *write set*) is evicted from the L1 cache
8. A cacheline which has been read during the transaction (i.e., a cacheline in the transaction's *read set*) is evicted from the L3 cache

[1] This means that any transaction may abort, despite the absence of memory conflicts, through no fault of the programmer. The periodic nature of certain interrupts also sets an effective maximum time limit on any transaction, which has been measured at about 4 ms [219].

TSX allows any program to identify an arbitrary section of its code as a *transaction* using explicit `XBEGIN` and `XEND` instructions. Any transaction is guaranteed to either: (1) **complete**, in which case all memory changes which happened during the transaction are made visible *atomically* to other processes and cores, or (2) **abort**, in which case all memory changes which happened during the transaction, as well as all other changes (e.g. to registers), are discarded. In the event of an abort, control is transferred to a fallback routine specified by the user, and a status code provides the fallback routine with some information about the cause of the abort.

From a security perspective, the intended uses of hardware transactional memory (easier synchronization or optimistic concurrency) are unimportant, so we will merely note that we can place arbitrary code inside both the transaction and the fallback routine, and whenever the transaction aborts, our fallback routine will immediately be given a callback with a status code. There are many reasons a TSX transaction may abort; important causes are listed in Table 1.3. Most of these are drawn from the Intel Software Developer's Manual [103], but the specifics of Causes #7 and #8—in particular the asymmetric behavior of TSX with respect to read sets and write sets—were suggested by Dice et al. [55]. Our experimental results corroborate their suggestions about these undocumented implementation details.

While a transaction is in process, an arbitrary amount of data must be buffered (hidden from the memory system) or tracked until the transaction completes or aborts. In TSX, this is done in the caches—transactionally written lines are buffered in the L1 data cache, and transactionally read lines marked in the L1–L3 caches. This has the important ramification that the cache size and associativity impose a limit on how much data can be buffered or tracked. In particular, if cache lines being buffered or tracked by TSX must be evicted from the cache, this necessarily causes a transactional abort. In this way, details about cache activity may be exposed through the use of transactions.

TSX has been addressed only rarely in a security context; to the best of our knowledge, there are only two works on the application of TSX to security to date [77, 110]. Guan et al. use TSX as part of a defense against memory disclosure attacks [77]. In their system, operations involving the plaintext of sensitive data necessarily occur inside TSX transactions. This structurally ensures that this plaintext will never be accessed by other processes or written back to main memory (in either case, a transactional abort will roll back the architectural state and invalidate the plaintext data).

Jang et al. exploit a timing side channel in TSX itself in order to break kernel address space layout randomization (KASLR) [110]. Specifically, they focus on Abort Cause #5, access violations and page faults. They note that such events inside a transaction trigger an abort but not their normal respective handlers; this means the operating system or kernel are *not* notified, so the attack is free to trigger as many access violations and page faults as it wants without raising suspicions. They then exploit this property and the aforementioned timing side channel to determine which kernel pages are mapped and unmapped (and also which are executable).

Neither of these works enable new attacks on memory accesses, nor do they eliminate the need for timers in attacks.

1.2 Potential TSX-based attacks

We present three potential attacks, all of which share their main goal with cache attacks—to monitor which cachelines are accessed by other processes and when. The three attacks we will present leverage Abort Causes #6, 7, and 8 respectively. Figure 1.1 outlines all three of the attacks we will present, as the PRIME+ABORT entry in the figure applies to both PRIME+ABORT–L1 and PRIME+ABORT–L3.

All of the TSX-based attacks which we will propose have the same critical structural benefit in common. This benefit, illustrated in Figure 1.1, is that these attacks have no need for a "measurement" phase. Rather than having to conduct some (timed) operation to determine whether the cache state has been modified by the victim, they simply receive a hardware callback through TSX immediately when a victim access takes place. In addition to the reduced overhead this represents for the attack procedure, this also means the attacker can be actively waiting almost indefinitely until the moment a victim access occurs—the attacker does not need to break the attack into predefined intervals. This results in a higher resolution attack, because instead of only coarse-grained knowledge of when a victim access occurred (i.e. which predefined interval), the attacker gains precise estimates of the relative timing of victim accesses.

All of our proposed TSX-based attacks also share a structural weakness when compared to PRIME+PROBE and FLUSH+RELOAD. Namely, they are unable to monitor multiple targets (cache sets in the case of PRIME+PROBE, addresses in the case of FLUSH+RELOAD) simultaneously while retaining the ability to distinguish accesses to one target from accesses to another. PRIME+PROBE and FLUSH+RELOAD are able to do this at the cost of increased overhead; effectively, a process can monitor multiple targets concurrently by performing multiple initialization stages, having a common waiting stage, and then performing multiple measurement stages, with each measurement stage revealing the activity for the corresponding target. In contrast, although our TSX-based attacks could monitor multiple targets at once, they would be unable to

distinguish events for one target from events for another without additional outside information. Some applications of PRIME+PROBE and FLUSH+RELOAD rely on this ability (e.g. [169]), and adapting them to rely on PRIME+ABORT instead would not be trivial. However, others, including the attack presented in Section 1.3.4, can be straightforwardly adapted to utilize PRIME+ABORT as a drop-in replacement for PRIME+PROBE or FLUSH+RELOAD.

We begin by discussing the simplest, but also least generalizable, of our TSX-based attacks, ultimately building to our proposed primary attack, PRIME+ABORT–L3.

1.2.1 Naïve TSX-based attack

Abort Cause #6 enables a potentially powerful, but limited attack.

From Cause #6, we can get a transaction abort (which for our purposes is an immediate, fast hardware callback) whenever there is a read-write or write-write conflict between our transaction and another process. This leads to a natural and simple attack implementation, where we simply open a transaction, access our target address, and wait for an abort (with the proper abort status code); on abort, we know the address was accessed by another process.

The style of this attack is reminiscent of FLUSH+RELOAD [233] in several ways. It targets a single, precise cacheline, rather than an entire cache set as in PRIME+PROBE and its variants. It does not require a (comparatively slow) "prime eviction set" step, providing fast and low-overhead monitoring of the target cacheline. Also like FLUSH+RELOAD, it requires the attacker to acquire a specific address to target, for instance exploiting shared libraries or page deduplication.

Like the other attacks using TSX, it benefits in performance by not needing the "measurement" phase to detect a victim access. In addition to the performance benefit, this attack would also be harder to detect and defend against. It would execute without any kind of timer, mitigating several important classes of defenses (see Section 1.4). It would also be resistant to most types of cache-based defenses; in fact, this attack has so little to do with the cache at all that it could

hardly be called a cache attack, except that it happens to expose the same information as standard cache attacks such as FLUSH+RELOAD or PRIME+PROBE do.

However, in addition to only being able to monitor target addresses in shared memory (the key weakness shared by all variants of FLUSH+RELOAD), this attack has another critical shortcoming. Namely, it can only detect read-write or write-write conflicts, not read-read conflicts. This means that one or the other of the processes—either the attacker or the victim—must be issuing a write command in order for the access to be detected, i.e. cause a transactional abort. Therefore, the address being monitored must not be in read-only memory. Combining this with the earlier restriction, we find that this attack, although powerful, can only monitor addresses in writable shared memory. We find this dependence to render it impractical for most real applications, and for the rest of the paper we focus on the other two attacks we will present.

1.2.2 PRIME+ABORT–L1

The second attack we will present, called PRIME+ABORT–L1, is based on Abort Cause #7. Abort Cause #7 provides us with a way to monitor evictions from the L1 cache in a way that is precise and presents us with, effectively, an immediate hardware callback in the form of a transactional abort. This allows us to build an attack in the PRIME+PROBE family, as the key component of PRIME+PROBE involves detecting cacheline evictions. This attack, like all attacks in the PRIME+PROBE family, does not depend in any way on shared memory; but unlike other attacks, it will also not depend on timers.

Like other PRIME+PROBE variants, our attack requires a one-time setup phase where we determine an eviction set for the cache set we wish to target; but like early PRIME+PROBE attacks [170, 174], we find this task trivial because the entire L1 cache index lies within the page offset (as explained earlier). Unlike other PRIME+PROBE variants, for PRIME+ABORT this is the sole component of the setup phase; we do not need to find a timing threshold, as we do not rely on timing.

The main part of PRIME+ABORT–L1 involves the same "prime" phase as a typical PRIME+PROBE attack, except that it opens a TSX transaction first. Once the "prime" phase is completed, the attack simply waits for an abort (with the proper abort status code). Upon receiving an abort, the attacker can conclude that some other program has accessed an address in the target cache set. This is similar to the information gleaned by ordinary PRIME+PROBE.

The reason this works is that, since we will hold an entire cache set in the write set of our transaction, any access to a different cache line in that set by another process will necessarily evict one of our cachelines and cause our transaction to abort due to Cause #7. This gives us an immediate hardware callback, obviating the need for any "measurement" step as in traditional cache attacks. This is why we call our method PRIME+ABORT—the abort replaces the "probe" step of traditional PRIME+PROBE.

1.2.3 PRIME+ABORT–L3

PRIME+ABORT–L1 is fast and powerful, but because it targets the (core-private) L1 cache, it can only spy on threads which share its core; and since it must execute simultaneously with its victim, this means it and its victim must be in separate hyperthreads on the same core. In this section we present PRIME+ABORT–L3, an attack which overcomes these restrictions by targeting the L3 cache. The development of PRIME+ABORT–L3 from PRIME+ABORT–L1 mirrors the development of L3-targeting PRIME+PROBE [104, 114, 136] from L1-targeting PRIME+PROBE [170, 174], except that we use TSX. PRIME+ABORT–L3 retains all of the TSX-provided advantages of PRIME+ABORT–L1, while also (like L3 PRIME+PROBE) working across cores, easily detecting accesses to either instructions or data, and even working across virtual machines.

PRIME+ABORT–L3 uses Abort Cause #8 to monitor evictions from the L3 cache. The only meaningful change this entails to the active portion of the attack is performing reads rather than writes during the "prime" phase, in order to hold the primed cachelines in the read set of the

transaction rather than the write set. For the pre-attack portion, PRIME+ABORT–L3, like other L3 PRIME+PROBE attacks, requires a much more sophisticated setup phase in which it determines eviction sets for the L3 cache. This is described in detail in the next section.

1.2.4 Finding eviction sets

The goal of the pre-attack phase for PRIME+ABORT is to determine an eviction set for a specified target address. For PRIME+ABORT–L1, this is straightforward, as described in Section 1.1.2.2. However, for PRIME+ABORT–L3, we must deal with both physical indexing and cache slicing in order to find L3 eviction sets. Like [136] and [104], we use large (2 MB) pages in this process as a convenience. With large pages, it becomes trivial to choose virtual addresses that have the same physical set index (i.e. agree in bits 6 to N, for some processor-dependent N, perhaps 15), again as explained in Section 1.1.2.2. We will refer to addresses which agree in physical set index (and in line offset, i.e. bits 0 to 5) as *set-aligned* addresses.

We generate eviction sets dynamically using the algorithm from Mastik [232] (inspired by that in [136]), which is shown as Algorithm 1. However, for the subroutine where Mastik uses timing methods to evaluate potential eviction sets (Algorithm 2), we use TSX methods instead (Algorithm 3).

Algorithm 3, a subroutine of Algorithm 1, demonstrates how Intel TSX is used to determine whether a candidate eviction set can be expected to consistently evict a given target cacheline. If "priming" the eviction set (accessing all its lines) inside a transaction followed by accessing the target cacheline consistently results in an immediate abort, we can conclude that a transaction cannot hold both the eviction set and the target cacheline in its read set at once, which means that together they contain at least ($associativity + 1$, or 17 in our case) lines which map to the same cache slice and cache set.

Conceptually, the algorithm for dynamically generating an eviction set for any given address has two phases: first, creating a "prototype group", and second, specializing it to form an

Algorithm 1: Dynamically generating a prototype eviction set for each cache slice, as implemented in [232]

Input: a set of potentially conflicting cachelines *lines*, all set-aligned
Output: a set of prototype eviction sets, one eviction set for each cache slice; that is, a "prototype group"

```
group ← {};
workingSet ← {};
while lines is not empty do
    repeat forever :
        line ← random member of lines;
        remove line from lines;
        if workingSet evicts line then              // Algorithm 2 or 3
            c ← line;
            break;
        end
        add line to workingSet;
    end
    foreach member in workingSet do
        remove member from workingSet;
        if workingSet evicts c then                 // Algorithm 2 or 3
            add member back to lines;
        else
            add member back to workingSet;
        end
    end
    foreach line in lines do
        if workingSet evicts line then              // Algorithm 2 or 3
            remove line from lines;
        end
    end
    add workingSet to group;
    workingSet ← {};
end
return group;
```

Algorithm 2: PRIME+PROBE (timing-based) method for determining whether an eviction set evicts a given cacheline, as implemented in [232]

```
Input: a candidate eviction set es and a cacheline line
Output: true if es can be expected to consistently evict line
times ← {};
repeat 16 times :
    access line;
    repeat 20 times :
        foreach member in es do
            access member;
        end
    end
    timed access to line;
    times ← times + {elapsed time};
end
if median of times > predetermined threshold then return true;
else return false;
```

eviction set for the desired target address. The algorithms shown (Algorithms 1, 2, and 3) together constitute the first phase of this larger algorithm. In this first phase, we use only set-aligned addresses, noting that all such addresses, after being mapped to an L3 cache slice, necessarily map to the same cache set inside that slice. This phase creates one eviction set for each cache slice, targeting the cache set inside that slice with the given set index. We call these "prototype" eviction sets, and we call the resulting group of one "prototype" eviction set per cache slice a "prototype group".

Once we have a prototype group generated by Algorithm 1, we can obtain an eviction set for any cache set in any cache slice by simply adjusting the set index of each address in one of the prototype eviction sets. Not knowing the specific cache-slice-selection hash function, it will be necessary to iterate over all prototype eviction sets (one per slice) in order to find the one which collides with the target on the same cache slice. If we do not know the (physical) set index of our target, we can also iterate through all possible set indices (with each prototype eviction set) to find the appropriate eviction set, again following the procedure from Liu et al. [136].

Algorithm 3: PRIME+ABORT (TSX-based) method for determining whether an eviction set evicts a given cacheline

```
Input: a candidate eviction set es and a cacheline line
Output: true if es can be expected to consistently evict line

aborts ← 0;
commits ← 0;

while aborts < 16 and commits < 16 do
    begin transaction;
    foreach member in es do
        access member;
    end
    access line;
    end transaction;
    if transaction committed then increment commits;
    else if transaction aborted with appropriate status code then increment aborts;
end
if aborts >= 16 then return true;
else return false;
```

1.3 Results

1.3.1 Characteristics of the Intel Skylake architecture

Our test machine has an Intel Skylake i7-6600U processor, which has two physical cores and four virtual cores. It is widely reported (e.g., in all of [93, 105, 114, 136, 145, 234]) that Intel processors have one cache slice per physical core, based on experiments conducted on Sandy Bridge, Ivy Bridge, and Haswell processors. However, our testing on the Skylake dual-core i7-6600U leads us to believe that it has four cache slices, contrary to previous trends which would predict it has only two. We validate this claim by using Algorithm 1 to produce four distinct eviction sets for large-page-aligned addresses. Then we test our four distinct eviction sets on many additional large-page-aligned addresses not used in Algorithm 1. We find that each large-page-aligned address conflicts with exactly one of the four eviction sets (by Algorithm 3), and further, that the conflicts are spread relatively evenly over the four sets. This convinces us that each of our four eviction sets represents set index 0 on a different cache slice, and thus that there are indeed four cache slices in the i7-6600U.

Table 1.4: Runtimes of PRIME+ABORT- and PRIME+PROBE-based versions of Algorithm 1 to generate a "prototype group" of eviction sets (data based on 1000 runs of each version of Algorithm 1)

	PRIME+ABORT	PRIME+PROBE
Min	4.5 ms	68.3 ms
1Q	10.1 ms	76.6 ms
Median	15.0 ms	79.3 ms
3Q	21.3 ms	82.0 ms
Max	64.7 ms	91.0 ms

Having determined the number of cache slices, we can now calculate the number of low-order bits in an address that must be fixed to create groups of set-aligned addresses. For our i7-6600U, this is 16. Henceforth we can use set-aligned addresses instead of large-page-aligned addresses, which is an efficiency gain.

1.3.2 Dynamically generating eviction sets

In the remainder of the Results section, we compare PRIME+ABORT–L3 to the L3 version of PRIME+PROBE as implemented in [232]. We begin by comparing the PRIME+ABORT and PRIME+PROBE versions of Algorithm 1 for dynamically generating prototype eviction sets.

Table 1.4 compares the runtimes of the PRIME+ABORT and PRIME+PROBE versions of Algorithm 1. The PRIME+ABORT-based method is over 5$\times$ faster than the PRIME+PROBE-based method in the median case, over 15$\times$ faster in the best case, and over 40% faster in the worst case.

Next, we compare the "coverage" of prototype groups (sets of four prototype eviction sets) derived and tested with the two methods. We derive 10 prototype groups with each version of Algorithm 1; then, for each prototype group, we use either timing-based or TSX-based methods to test 1000 additional set-aligned addresses not used for Algorithm 1 (a total of $10,000$ additional set-aligned addresses for PRIME+ABORT and $10,000$ for PRIME+PROBE). The testing

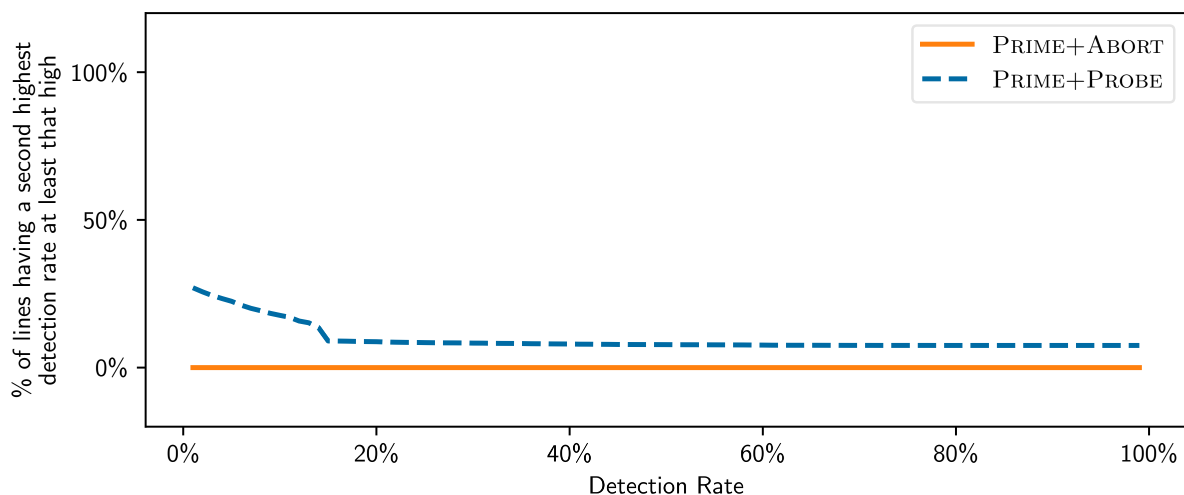


Figure 1.2: "Double coverage" of prototype groups generated by PRIME+ABORT- and PRIME+PROBE-based versions of Algorithm 1. With PRIME+PROBE, some tested cachelines are reliably detected by more than one prototype eviction set. In contrast, with PRIME+ABORT each tested cacheline is reliably detected by only one prototype eviction set.

procedure is akin to a single iteration of the outer loop in Algorithm 2 or 3 respectively. Using this procedure, each of the $10,000$ set-aligned addresses is tested $10,000$ times against each of the four prototype eviction sets in the prototype group. This produces four "detection rates" for each set-aligned address (one per prototype eviction set). We assume that the highest of these four detection rates corresponds to the prototype eviction set from the same cache slice as the tested address, and we call this detection rate the "max detection rate" for the set-aligned address. Both PRIME+ABORT and PRIME+PROBE methods result in "max detection rates" which are consistently indistinguishable from 100%. However, we note that out of the 100 million trials in total, 13 times we observed the PRIME+PROBE-based method fail to detect the access (resulting in a "max detection rate" of 99.99% in 13 cases), whereas with the PRIME+ABORT-based method, all 100 million trials were detected, for perfect max detection rates of 100.0%. This result is due to the structural nature of transactional conflicts—it is impossible for a transaction with a read set of size $(1 + associativity)$ to ever successfully commit; it must always abort.

Since each address maps to exactly one cache slice, and ideally each eviction set contains lines from only one cache slice, we expect that any given set-aligned address conflicts with only

one out of the four prototype eviction sets in a prototype group. That is, we expect that out of the four detection rates computed for each line (one per prototype eviction set), one will be very high (the "max detection rate"), and the other three will be very low. Figure 1.2 shows the "second-highest detection rate" for each line—that is, the maximum of the remaining three detection rates for that line, which is a measure of false positives. For any given detection rate on the x-axis, the figure shows what percentage of the $10,000$ set-aligned addresses had a false-positive detection rate at or above that level. Whenever the "second-highest detection rate" is greater than zero, it indicates that the line appeared to be detected by a prototype eviction set meant for an entirely different cache slice (i.e. a false positive detection). In Figure 1.2, we see that with the PRIME+PROBE-based method, around 22% of lines have "second-highest detection rates" over 5%, around 18% of lines have "second-highest detection rates" over 10%, and around 7.5% of lines even have "second-highest detection rates" of 100%, meaning that more than one of the "prototype eviction sets" each detected that line in 100% of the $10,000$ trials. In contrast, with the PRIME+ABORT-based method, none of the $10,000$ lines tested had "second-highest detection rates" over 1%. PRIME+ABORT produces very few false positives and cleanly monitors exactly one cache set in exactly one cache slice.

1.3.3 Detecting memory accesses

Figures 1.3, 1.4, and 1.5 show the success of PRIME+ABORT and two variants of PRIME+PROBE in detecting the memory accesses of an artificial victim thread running on a different physical core from the attacker. The victim thread repeatedly accesses a single memory location for the duration of the experiment—in the "treatment" condition, it accesses the target (monitored) location, whereas in the "control" condition, it accesses an unrelated location. We introduce delays (via busy-wait) of varying lengths into the victim's code in order to vary the frequency at which it accesses the target location (or unrelated location for control). Figures 1.3, 1.4, and 1.5 plot the number of events observed by the respective attackers, vs. the actual number

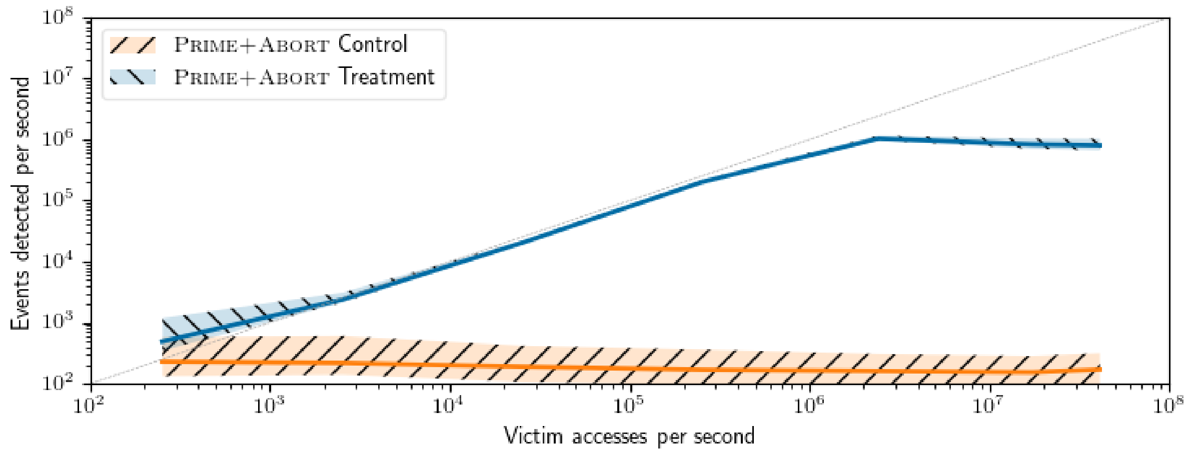


Figure 1.3: Access detection rates for PRIME+ABORT in the "control" and "treatment" conditions. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

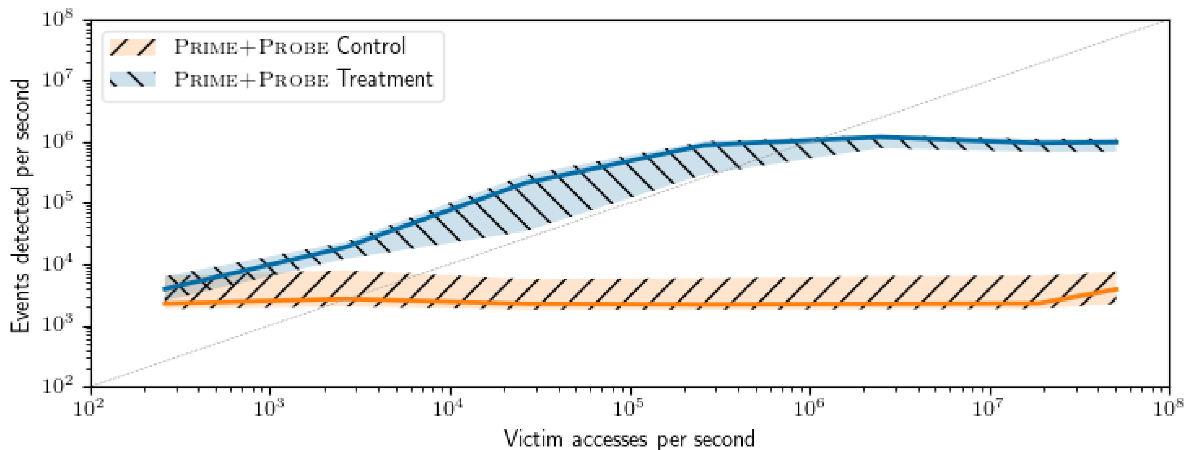


Figure 1.4: Access detection rates for unmodified PRIME+PROBE in the "control" and "treatment" conditions. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

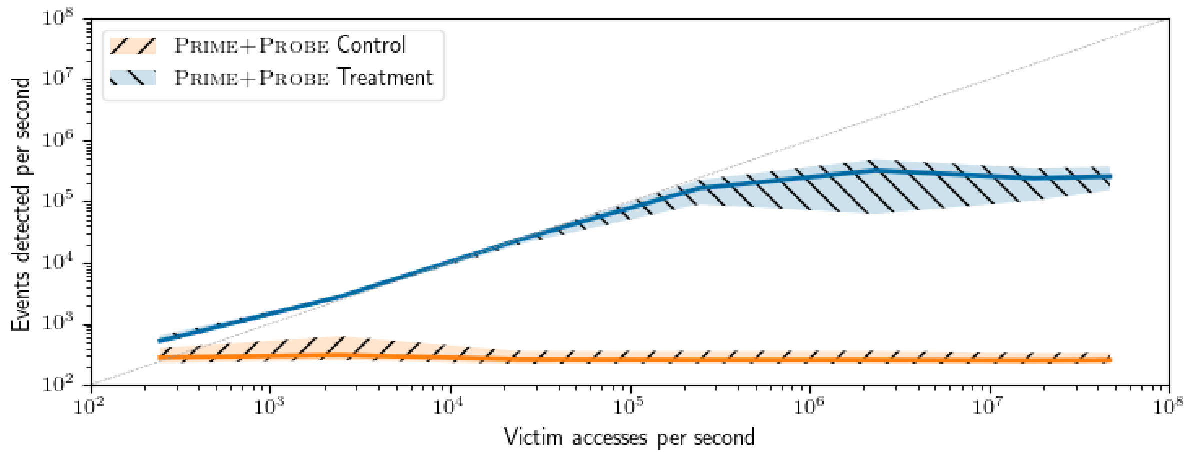


Figure 1.5: Access detection rates for our modified implementation of PRIME+PROBE which "collapses" streaks. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

of accesses by the victim, in "control" and "treatment" scenarios. Data were collected from 100 trials per attacker, each entailing separate runs of Algorithm 1 and new targets. The $y = x$ line is shown for reference in all figures; it indicates perfect performance for the "treatment" condition, with all events detected but no false positives. Perfect performance in the "control" condition, naturally, is values as low as possible in all cases.

We see in Figure 1.3 that PRIME+ABORT detects a large fraction of the victim's accesses at frequencies up to several hundred thousand accesses per second, scaling up smoothly and topping out at a maximum detection speed (on our test machine) of around one million events per second. PRIME+ABORT exhibits this performance while also displaying relatively low false positive rates of around 200 events per second, or one false positive every 5000 µs. The close correlation between number of detected events and number of victim accesses indicates PRIME+ABORT's low overheads—in fact, we measured its transactional abort handler as executing in 20-40 ns—which allow it to be essentially "always listening" for victim accesses. Also, it demonstrates PRIME+ABORT's ability to accurately count the number of victim accesses, despite

only producing a binary output (access or no access) in each transaction. Its high speed and low overheads allow it to catch each victim access in a separate transaction.

Figure 1.4 shows the performance of unmodified PRIME+PROBE as implemented in Mastik [232][1]. We see false positive rates which are significantly higher than those observed for PRIME+ABORT—over 2000 events per second, or one every 500 µs. Like PRIME+ABORT, this implementation of PRIME+PROBE appears to have a top speed around one million accesses detected per second under our test conditions. But most interestingly, we observe significant "oversampling" at low frequencies—PRIME+PROBE reports many more events than actually occurred. For instance, when the victim thread performs 2600 accesses per second, we expect to observe 2600 events per second, plus around 2000 false positives per second as before. However, we actually observe over $18,000$ events per second in the median case. Likewise, when the victim thread provides $26,000$ accesses per second, we observe over $200,000$ events per second in the median case. Analysis shows that for this implementation of PRIME+PROBE on our hardware, single accesses can cause long streaks of consecutive observed events, sometimes as long as hundreds of observed events. We believe this to be due to the interaction between this PRIME+PROBE implementation and our hardware's L3 cache replacement policy. One plausible explanation for why PRIME+ABORT is not similarly afflicted, is that the replacement policy may prioritize keeping lines that are part of active transactions, evicting everything else first. This would be a sensible policy for Intel to implement, as it would minimize the number of unwanted/unnecessary aborts. In our setting, it benefits PRIME+ABORT by ensuring that a "prime" step inside a transaction cleanly evicts all other lines.

To combat the oversampling behavior observed in PRIME+PROBE, we investigate a modified implementation of PRIME+PROBE which "collapses" streaks of observed events, meaning that a streak of any length is simply counted as a single observed event. Results with this modified

[1]We make one slight modification suggested by the maintainer of Mastik: every probe step, we actually perform multiple probes, "counting" only the first one. In our case we perform five probes at a time, still alternating between forwards and backwards probes. All of the results which we present for the "unmodified" implementation include this slight modification.

implementation are shown in Figure 1.5. We see that this strategy is effective in combating oversampling, and also reduces the number of false positives to around 250 per second or one every 4000 μs. However, this implementation of PRIME+PROBE performs more poorly at high frequencies, having a top speed around 300,000 events per second compared to the one million per second of the other two attacks. This effect can be explained by the fact that as the victim access frequency increases, streaks of observed events become more and more likely to "hide" real events (multiple real events occur in the same streak)—in the limit, we expect to observe an event during every probe, but this approach will observe only a single streak and indicate a single event occurred.

Observing the two competing implementations of PRIME+PROBE on our hardware reveals an interesting tradeoff. The original implementation has good high frequency performance, but suffers from both oversampling and a high number of false positives. In contrast, the modified implementation has poor high frequency performance, but does not suffer from oversampling and exhibits fewer false positives. For the remainder of this paper we consider the modified implementation of PRIME+PROBE only, as we expect that its improved accuracy and fewer false positives will make it more desirable for most applications. Finally, we note that PRIME+ABORT combines the desirable characteristics of both PRIME+PROBE implementations, as it exhibits the fewest false positives, does not suffer from oversampling, and has good high frequency performance, with a top speed around one million events per second.

1.3.4 Attacks on AES

In this section we evaluate the performance of PRIME+ABORT in an actual attack by replicating the attack on OpenSSL's T-table implementation of AES, as conducted by Gruss et al. [75]. As those authors acknowledge, this implementation is no longer enabled by default due to its susceptibility to these kinds of attacks. However, as with their work, we use it for the purpose of comparing the speed and accuracy of competing attacks. Gruss et al. compared PRIME+PROBE,

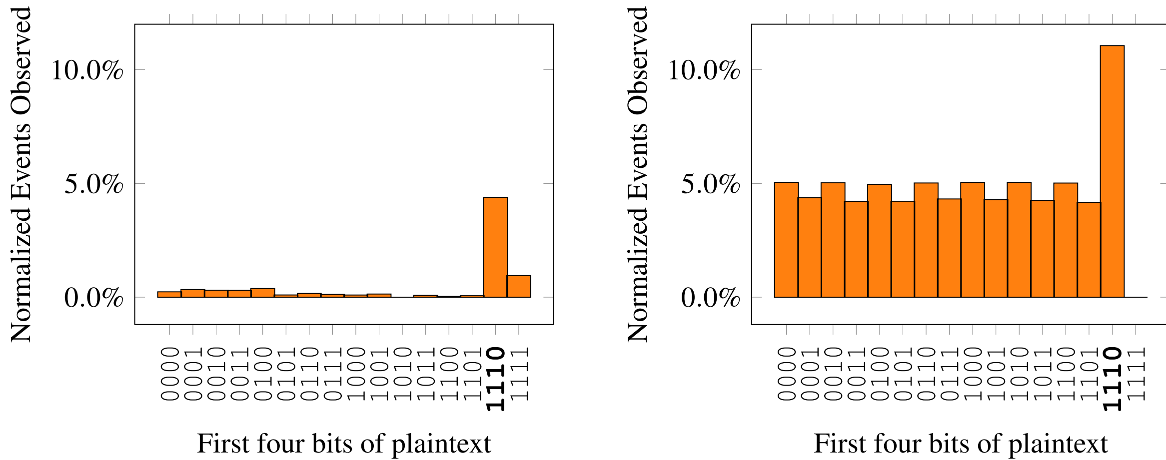
FLUSH+RELOAD, and FLUSH+FLUSH [75]; we have chosen to compare PRIME+PROBE and PRIME+ABORT, as these attacks do not rely on shared memory. Following their methods, rather than using previously published results directly, we rerun previous attacks alongside ours to ensure fairness, including the same hardware setup.

Figures 1.6a and 1.6b provide the results of this experiment. In this chosen-plaintext attack, we listen for accesses to the first cacheline of the first T-Table (`Te0`) while running encryptions. We expect that when the first four bits of our plaintext match the first four bits of the key, the algorithm will access this cacheline one extra time over the course of each encryption compared to when the bits do not match. This will manifest as causing more events to be detected by PRIME+ABORT or PRIME+PROBE respectively, allowing the attacker to predict the four key bits. The attack can then be continued for each byte of plaintext (monitoring a different cacheline of `Te0` in each case) to reveal the top four bits of each key byte.

In our experiments, we used a key whose first four bits were arbitrarily chosen to be `1110`, and for each method we performed one million encryptions with each possible 4-bit plaintext prefix (a total of sixteen million encryptions for PRIME+ABORT and sixteen million for PRIME+PROBE). As shown in Figures 1.6a and 1.6b, both methods correctly predict the first four key bits to be `1110`, although the signal is arguably cleaner and stronger when using PRIME+ABORT.

1.4 Potential countermeasures

Many countermeasures against side-channel attacks have already been proposed; Ge et al. [67] again provide an excellent survey. Examining various proposed defenses in the context of PRIME+ABORT reveals that some are effective against a wide variety of attacks including PRIME+ABORT, whereas others are impractical or ineffective against PRIME+ABORT. This leads us to advocate for the prioritization and further development of certain approaches over others.



(a) PRIME+ABORT attack against AES. Shown is, for each condition, the percentage of additional events that were observed compared to the condition yielding the fewest events.

(b) PRIME+PROBE attack against AES. Shown is, for each condition, the percentage of additional events that were observed compared to the condition yielding the fewest events.

Figure 1.6: PRIME+ABORT and PRIME+PROBE attacks against AES

We first examine classes of side-channel countermeasures that are impractical or ineffective against PRIME+ABORT and then move toward countermeasures which are more effective and practical.

Timer-based countermeasures

A broad class of countermeasures ineffective against PRIME+ABORT are approaches that seek to limit the availability of precise timers, either by injecting noise into timers to make them less precise, or by restricting access to timers in general. There are a wide variety of proposals in this vein, including [92], [124], [144], [212], and various approaches which Ge et al. classify as "Virtual Time" or "Black-Box Mitigation". PRIME+ABORT should be completely immune to all timing-related countermeasures.

Partitioning time

Another class of countermeasures that seems impractical against PRIME+ABORT is the class Ge et al. refer to as Partitioning Time. These countermeasures propose some form of "time-sliced exclusive access" to shared hardware resources. This would technically be effective

against PRIME+ABORT, because the attack is entirely dependent on running simultaneously with its victim process; any context switch causes a transactional abort, so the PRIME+ABORT process must be active in order to glean any information. However, since PRIME+ABORT targets the LLC and can monitor across cores, implementing this countermeasure against PRIME+ABORT would require providing each user process time-sliced exclusive access to the LLC. This would mean that processes from different users could never run simultaneously, even on different cores, which seems impractical.

Disabling TSX

A countermeasure which would ostensibly target PRIME+ABORT's workings in particular would be to disable TSX entirely, similarly to how hyperthreading has been disabled entirely in cloud environments such as Microsoft Azure [143]. While this is technically feasible—in fact, due to a hardware bug, Intel already disabled TSX in many Haswell CPUs through a microcode update [94]—TSX's growing prevalence (Table 1.2), as well as its adoption by applications such as `glibc` (pthreads) and the JVM [110], indicates its importance and usefulness to the community. System administrators are probably unlikely to take such a drastic step.

Auditing

More practical but still not ideal is the class of countermeasures Ge et al. refer to as Auditing, which is based on behavioral analysis of running processes. Hardware performance counters in the target systems can be used to monitor LLC cache misses or miss rates, and thus detect when a PRIME+PROBE- or FLUSH+RELOAD-style attack is being conducted [40, 75, 241] (as any attack from those families will introduce a large number of cache misses—at least in the victim process). As a PRIME+PROBE-style attack, PRIME+ABORT would be just as vulnerable to these countermeasures as other cache attacks are. However, any behavioral auditing scheme is necessarily imperfect and subject to misclassification errors in both directions. Furthermore, any auditing proposal targeting PRIME+ABORT which specifically monitors TSX-related events, such as transactions opened or transactions aborted, seems less likely to be effective, as many

benign programs which utilize TSX generate a large number of both transactions and aborts, just as PRIME+ABORT does. This makes it difficult to distinguish PRIME+ABORT from benign TSX programs based on these statistics.

Constant-time programming

The class of countermeasures referred to as *constant-time programming* includes a variety of approaches, some of which are likely to be effective against PRIME+ABORT. These countermeasures are generally software techniques to ensure important invariants are preserved in program execution regardless of (secret) input data, with the aim of mitigating side channels of various types. For our purposes, it is not enough to merely ensure that critical functions in a program always execute in constant time regardless of secret data. This is insufficient to defend against PRIME+ABORT, as PRIME+ABORT can track cache accesses without relying on any kind of timing side-channel. Instead, following the description of constant-time programming in the Introduction (Section 2), we must ensure that no data access or control-flow decision made by the program ever depends on any secret data. This approach is effective against PRIME+ABORT, as monitoring cache accesses (either for instructions or data) would not reveal anything about the secret data being processed by the program.

Randomizing hardware operations

Another interesting class of defenses proposes to insert noise into hardware operations so that side-channel measurements are more difficult. Although PRIME+ABORT is immune to such efforts related to timers, other proposals aim to inject noise into other side-channel vectors, such as cache accesses. For instance, RPcache [220] proposes to randomize the mapping between memory address and cache set, which would render PRIME+ABORT and other cache attacks much more difficult. Other proposals aim to, for instance, randomize the cache replacement policy. Important limitations of this kind of noise injection (noted by Ge et al.) include that it generally can only make side-channel attacks more difficult or less efficient (not completely impossible), and that higher levels of mitigation generally come with higher performance costs. However,

these kinds of schemes seem to be promising, providing relatively lightweight countermeasures against a quite general class of side-channel attacks.

Cache set partitioning

Finally, a very promising class of countermeasures proposes to partition cache sets between processes, or disallow a single process to use all of the ways in any given LLC cache set. This would be a powerful defense against PRIME+ABORT or any other PRIME+PROBE variant. Some progress has been made towards implementing these defenses, such as CATalyst [135], which utilizes Intel's "Cache Allocation Technology" [95]; or "cache coloring" schemes such as STEALTHMEM [117] or that proposed by [69]. One undesirable side effect of this approach is that it would reduce the maximum size of TSX transactions, hindering legitimate users of the hardware transactional memory functionality. However, the technique is still promising as an effective defense against a wide variety of cache attacks. For more examples and details of this and other classes of side-channel countermeasures, we again refer the reader to Ge et al. [67].

Our work with PRIME+ABORT leads us to recommend the further pursuit of those classes of countermeasures which are effective against all kinds of cache attacks including PRIME+ABORT, specifically constant-time programming, randomizing cache operations, or providing mechanisms for partitioning cache sets between processes.

1.5 Disclosure

We disclosed this vulnerability to Intel on January 30, 2017, explaining the basic substance of the vulnerability and offering more details. We also indicated our intent to submit our research on the vulnerability to USENIX Security 2017 in order to ensure Intel was alerted before it became public. We did not receive a response.

1.6 Conclusion

PRIME+ABORT leverages Intel TSX primitives to yield a high-precision, cross-core cache attack which does not rely on timers, negating several important classes of defenses. We have shown that leveraging TSX improves the efficiency of algorithms for dynamically generating eviction sets; that PRIME+ABORT has higher accuracy and speed on Intel's Skylake architecture than previous L3 PRIME+PROBE attacks while producing fewer false positives; and that PRIME+ABORT can be successfully employed to recover secret keys from a T-table implementation of AES. Additionally, we presented new evidence useful for all cache attacks regarding Intel's Skylake architecture: that it may differ from previous architectures in number of cache slices, and that it may use different cache replacement policies for lines involved in TSX transactions.

Acknowledgments

We thank our anonymous reviewers for their helpful advice and comments. We also especially thank Yuval Yarom for his assistance in improving the quality of this work.

This material is based in part upon work supported by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1, in part, is a reprint of the material as it appears in the USENIX Security Symposium 2017. Disselkoen, Craig; Kohlbrenner, David; Porter, Leo; Tullsen, Dean. USENIX, 2017. The dissertation author was the primary investigator and author of this material.

Chapter 2

Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork

Constant-time programming is the de-facto approach to writing critical code—in particular cryptographic code—that is robust against timing side-channel attacks, including attacks such as PRIME+ABORT (Chapter 1). Unfortunately, it is notoriously hard to write constant-time code: Not only do experts fail to adequately write truly constant-time code [20, 26, 152], but even the process of fixing these mistakes can lead to further vulnerabilities [4, 199]. Almost yearly, attackers break cryptosystems using these attacks. For example, recently, timing side-channels were found in several libraries implementing elliptic curve cryptography, as well as in code running on smart cards and TPM chips [109, 155].

The only way we can hope to eliminate these kinds of vulnerabilities in software is with rigorous verification. Formal verification allows us to mathematically prove that code is constant-time (and give a counterexample when it's not). To this end, we present a verification

tool, Pitchfork, which ensures functions are constant-time with respect to secret values such as encryption keys or message plaintexts.

We designed Pitchfork to easily analyze both cryptographic primitives and protocol-level cryptographic code. Protocol-level code has been left underanalyzed by much of the previous work on constant-time verification, which has instead focused on cryptographic primitives. However, flaws in the implementation of protocol-level code are responsible for high-profile vulnerabilities such as Lucky 13 [4]. Indeed, as we describe later, Pitchfork has found several constant-time violations in protocol-level code from real-world codebases.

Pitchfork uses underconstrained symbolic execution augmented with dynamic taint tracking to verify that code is constant-time. In particular, it uses a *shadow memory* to track secrets even as they are stored to and loaded from memory. Pitchfork also allows the user to specify *function hooks* which are executed in lieu of a call to a particular function; this allows Pitchfork to focus on protocol-level code while ignoring implementation details of cryptographic primitives. With these techniques, Pitchfork was able to verify protocol-level code in both `libsignal` [198] and Mozilla's NSS cryptographic library [157]. Our verification effort, however, also revealed several constant-time vulnerabilities in NSS, including a critical vulnerability which was assigned CVE-2019-11745.

2.1 Motivation

In this section, we give a brief introduction to constant-time programming and show how Pitchfork verifies constant-time properties.

2.1.1 Constant-time programming

Timing side-channel attacks have repeatedly been used to leak sensitive data, particularly in cryptographic code [4, 121, 199]. Timing vulnerabilities in cryptographic code arise when

secret data influences either control flow or the addresses of memory accesses. For example, the function `check_password` below is vulnerable to timing attacks, as secret data—namely, the correct password—influences a branch condition.

```
int check_password(char* password, char* guess) {
    for (int i = 0; i < 32; i++) {
        if (password[i] != guess[i]) {
            return -1;  // fail
        }
    }
    return 0;  // success
}
```

Specifically, this function's execution time depends on how many of the initial characters of the `guess` were correct. If an attacker is allowed to make repeated guesses, they can efficiently discover the secret password by successively brute-forcing each character, moving to the next character when the function's execution time increases.

Writing code that is free from timing vulnerabilities is difficult. In practice, experts use a collection of constant-time recipes, which can be distilled down to three rules:

1. Secret values must not influence the program's control flow—otherwise, an attacker could infer secret values from the program's execution time.[1]

2. Secret values must not influence the addresses of memory accesses—otherwise, an attacker could infer secret values using cache side-channel attacks.

3. Secret values must not influence the inputs to any variable-time machine operation (such as integer division on many processors).

[1]In addition, an attacker may be able to infer secret values from a cache side-channel attack, if the control flow leads to different memory access patterns, even for public data.

To fix the above function, we need to remove the control flow dependency on the secret password data:

```
int check_password(char* password, char* guess) {
    int rv = 0;
    for (int i = 0; i < 32; i++) {
        rv |= (password[i] ^ guess[i]);
    }
    return rv;
}
```

Here, we keep track of the final return value `rv`, updating it using bitwise operations so that its final value is nonzero if any character mismatched. Crucially, all 32 characters are compared regardless of whether the initial characters were guessed correctly.

Timing vulnerabilities in cryptographic code can be very difficult to find, often going unnoticed for years even in major cryptographic libraries. Furthermore, attempts to fix these vulnerabilities are often incomplete, or even introduce new timing vulnerabilities in the process. As an example, the recent fix for a timing side-channel vulnerability in Mozilla's NSS cryptographic library [157] failed to fully eliminate all dependency of the control flow on secret data. Using Pitchfork, we analyzed the "fixed" code and found the remaining vulnerability. That vulnerability was fixed in a subsequent patch, and the resulting code was verified with Pitchfork before it was committed. This example demonstrates the need for a tool which can rigorously verify cryptographic code to be constant-time.

2.1.2 Constant-time verification

Unfortunately, most safety-critical constant-time code has not been formally verified to be constant-time. Instead, it simply undergoes manual review—a process which has repeatedly missed timing vulnerabilities in the past [4, 199]. Some work has been done on programming

languages with formally verifiable constant-time properties [37, 223, 243], but these tools cannot easily be applied to existing C/C++ code. Furthermore, previous constant-time verification efforts focus on verifying cryptographic primitives [6, 49]; however, modern cryptographic primitives such as Salsa20, Poly1305, and Ed25519 are designed to be constant-time by construction. In contrast, flaws in the implementation of protocol-level code have resulted in high-profile vulnerabilities such as Lucky 13 [4]. Finally, previous tools for constant-time verification [6] don't provide meaningful feedback to aid the development process; they simply report whether a given program is constant-time or not.

Our experience using these tools identified the need for a tool which can verify not only cryptographic primitives, but more complex cryptographic protocol code. We also need this tool to provide detailed feedback to the developer about vulnerabilities, such as the vulnerability's location in the source code and the conditions under which it occurs.

2.2 Constant-time verification with Pitchfork

We designed Pitchfork to fulfill these needs: to verify both cryptographic primitives and protocol-level code, and to provide detailed feedback to developers about vulnerabilities. Pitchfork uses underconstrained symbolic execution augmented with dynamic taint tracking to directly verify the three simple properties given in Section 2.1.1. In this section, we describe how Pitchfork analyzes cryptographic code to verify these properties.

2.2.1 Taint propagation

Consider the `mbed-crypto` [13] AES decryption function in Figure 2.1. Pitchfork is designed to find code locations where *secrets* could leak via timing channels. Hence, to start, the developer must indicate an initial set of variables, function arguments, or struct fields which contain secret data. Here, we want to ensure that the AES keys are not leaked. Accordingly, we

```
int mbedtls_internal_aes_decrypt(
      mbedtls_aes_context *ctx,
      const unsigned char input[16],
      unsigned char output[16]
) {
    // ...
    RK = ctx->rk;  // secret AES round keys

    X0 = ( (uint32_t) input[0]       )    \
       | ( (uint32_t) input[1] <<  8 )    \
       | ( (uint32_t) input[2] << 16 )    \
       | ( (uint32_t) input[3] << 24 );   \
    X0 ^= *RK++;
    // {repeat to define X1, X2, X3,
    //    using input[4] through input[15]}

    for ( i = ( ctx->nr >> 1 ) - 1; i > 0; i-- ) {
        Y0 = *RK++ ^ RT0[ ( X0       ) & 0xFF ] ^ \
                     RT1[ ( X3 >>  8 ) & 0xFF ] ^ \
                     RT2[ ( X2 >> 16 ) & 0xFF ] ^ \
                     RT3[ ( X1 >> 24 ) & 0xFF ];
        // loop body continues ...
    }

    // function continues ...
}
```

Figure 2.1: Excerpt from the function `mbedtls_internal_aes_decrypt()`, with macros expanded and inlined, and some formatting adjusted to fit the page.

need to mark the AES keys secret, and other inputs (including the ciphertext) as public. To do this, we note that these keys are stored in the `rk` field of the struct `mbedtls_aes_context`:

```
typedef struct mbedtls_aes_context {
    int nr;         /* The number of rounds. */
    uint32_t *rk;   /* AES round keys. */
    // {more fields ...}
} mbedtls_aes_context;
```

In this struct, we mark the `nr` field as public, and the `rk` field—which holds the AES keys—as a (public) pointer to an array of secret data. We do this by creating a Rust object describing the struct:

```
_struct("mbedtls_aes_context", vec![
    default(),  // nr
    pub_pointer_to(array_of(sec_i32(), 64)),  // rk
    // ... more fields ...
])
```

and then passing this object to Pitchfork using its API.

Pitchfork then *propagates* the tainted-secret values as it symbolically executes the function, marking the result of each operation secret if any of its inputs were secret. Furthermore, to track secret values even as they are stored to and loaded from memory, Pitchfork uses a *shadow memory*: While the primary memory stores symbolic values (as in standard symbolic execution), the shadow memory stores for each address a flag indicating whether that address's current contents are tainted or not. Thus on line 13 when the pointer `*RK` is dereferenced, Pitchfork looks up the corresponding address in the shadow memory and finds that the value currently stored there is secret—it stems from the `rk` struct field which was previously marked secret. Therefore, Pitchfork also marks the resulting value `X0` as secret.

```
1  if (context->doPad) {
2    if (context->padDataLength != 0) {
3      rv = (*context->update)(context->cipherInfo, pLastPart, &outlen, maxout,
4                              context->padBuf, context->blockSize);
5      if (rv != SECSuccess) {
6        // ...
7      } else {
8        unsigned int padSize = (unsigned int) pLastPart[context->blockSize - 1];
9        if ((padSize > context->blockSize) || (padSize == 0)) {
10         crv = CKR_ENCRYPTED_DATA_INVALID;
11       } else {
12         // ...
13       }
14     }
15   }
16 }
```

Figure 2.2: Excerpt from the function `NSC_DecryptFinal()` in the NSS cryptographic library, version 3.46. Some comments not relevant to the current discussion have been omitted, and some formatting has been adjusted to fit the page.

Finally, Pitchfork reports a constant-time violation when secret-tainted data is used in a branch condition or memory address. In this example, Pitchfork reports a violation on line 18, as the array index (the expression `X0 & 0xFF`) uses `X0`, which has been marked secret.

2.2.2 Analyzing protocol-level code

In this section, we show how Pitchfork finds a real constant-time violation in the protocol-level function `NSC_DecryptFinal` (Figure 2.2) from Mozilla's Network Security Services (NSS) cryptographic library [157]. NSS is used by many applications, most notably Firefox.

When analyzing protocol-level code, we don't want to get bogged down analyzing complicated but uninteresting functions. Thus, Pitchfork allows the user to specify *function hooks* which are executed in lieu of a call to a particular function. For example, on line 3, the call to the function pointer `context->update` invokes a block cipher decryption primitive. Since we wish to analyze the protocol and not the primitives, we allow the user to choose to supply a function hook which is executed instead of the decryption primitive itself. The function hook writes data marked

secret into the output buffer `pLastPart`, and writes an appropriately constrained length value to the output-length parameter `&outlen`. Then, Pitchfork can analyze the critical protocol-level code abstract from the implementation details of the block cipher primitive, which can be verified separately.

After `context->update` writes (secret) decrypted data into the buffer `pLastPart`, the following code strips padding from the block cipher output. On line 8, the code determines the size of this padding by reading the last byte of the decrypted data. Since the buffer contents were marked secret, Pitchfork also marks the resulting value `padSize` as secret. Then, on line 9, the code uses `padSize` as part of a branch condition. This is dangerous, as the conditional branch may leak information about the padding through a timing side-channel: Specifically, it could lead to a *padding-oracle attack*, allowing an attacker to recover the plaintext. Pitchfork identifies this leak and correctly reports a constant-time violation.

We use function hooks for several purposes. In the example above, we used a function hook to ignore the implementation details of a block cipher primitive, which could be analyzed separately. Similarly, function hooks allow Pitchfork to avoid analyzing the implementation of random number generation functions, concurrency primitives, or logging frameworks, instead treating these functions as black boxes. This allows Pitchfork's analysis to focus on the code which is most likely to contain vulnerabilities.

2.3 Implementation of Pitchfork

Haybale

At the core of Pitchfork is Haybale, a new symbolic execution engine which we implemented for this work. Haybale implements *underconstrained symbolic execution*: it can analyze single functions rather than entire programs, as in UC-KLEE [182] or Sys [25]. For instance, Haybale can analyze each function in a library individually, using symbolic reasoning to consider

all possible values of the function arguments, rather than having to analyze an entire executable which contains calls to the library. Haybale, like other existing symbolic execution engines, explores all *paths* through a function (sequences of control flow decisions). Moreover, Haybale allows the user to perform various analyses on these paths—e.g., find values of the function inputs which exercise that path, ask whether the path could result in a particular return value, or determine whether a particular pointer value encountered along the path could ever be `NULL`.

Incremental solving

Haybale leverages the *incremental solving* mode of modern SMT solvers to perform efficient backtracking while exploring related paths. With incremental solving, SMT solvers can partially revert their state, removing recent constraints but retaining important solving work already completed. When Haybale completes its analysis of a path, it uses incremental solving to revert to the program state where the completed path diverges from the next path to analyze. This allows the SMT solver to reuse its analysis of the entire common prefix of the two paths.

Symbolic memory

The symbolic representation of memory contents is considered a crucial design consideration in symbolic execution engines [25, 29, 38, 60]. For Haybale, we chose a simple flat memory model inspired by Sys [25], in which the memory is represented by a single symbolic array which maps from indices to 8-bit values. Our experience, surprisingly, showed us that this simple model outperformed a more complex model based on 64-bit memory "cells", which was designed to optimize for the common case of 64-bit or 32-bit memory operations. We hypothesize that modern SMT solvers are optimized to handle array operations well—Haybale uses Boolector [164], an SMT solver specializing in bitvector theory which has done well at recent SMT competitions—and a flat memory model is the simplest way to express Haybale's intent to the SMT solver.

Pitchfork

Pitchfork extends Haybale with dynamic taint tracking in order to determine which values in registers and/or memory are influenced by secrets and which are not. Pitchfork expects the user to annotate some function arguments or struct fields as secret—e.g., secret keys or message plaintext—and then it propagates these annotations through the program as it executes, marking the result of an operation secret if any of its inputs were secret. Whenever it encounters conditional branches or memory accesses, Pitchfork reports a constant-time violation if the branch condition or memory address is marked secret.

LLVM

Pitchfork and Haybale operate at the level of LLVM IR. This allows them to analyze code written in C/C++, Rust, Swift, Go, or any other language which can compile to LLVM IR. Compared to some other symbolic execution tools which analyze executable binaries [38, 197], operating on LLVM IR allows Pitchfork to remain closer to the source level, allowing it to, e.g., report the source line number and filename for any constant-time violations it finds. It also allows Pitchfork to leverage the type-level information in the LLVM bitcode; Pitchfork uses this to greatly accelerate and simplify the process of providing public/secret annotations for function arguments and even complicated data structures.

Rust

Both Pitchfork and Haybale are implemented in the Rust language. This is a departure from previous notable symbolic execution engines, which have been implemented in C++ [29], Python [197], or Haskell [25]. Compared to Python, using Rust avoids the performance overheads of garbage collection or a global interpreter. It also provides strong static typing, which avoids the problem of having a long-running analysis halted by a simple type error in results-reporting code. In contrast, compared with C++, we found Rust's strong memory safety guarantees helped us avoid many tricky bugs such as segmentation faults or use-after-frees, although we did still experience a few of these bugs at the interface between Haybale and Boolector [164], which is written in C. (Our experience in this respect aligns with that reported by the Sys developers in

```
1  if (context->doPad && context->multi) {
2    // ...
3    crv = NSC_DecryptUpdate(hSession, pEncryptedData, ulEncryptedDataLen, pData,
     ↪  &updateLen);
4    if (crv == CKR_OK) {
5      maxoutlen -= updateLen;
6      pData += updateLen;
7    }
8    finalLen = maxoutlen;
9    crv2 = NSC_DecryptFinal(hSession, pData, &finalLen);
10   if (crv == CKR_OK && crv2 == CKR_OK) {
11     *pulDataLen = updateLen + finalLen;
12   }
13   return crv == CKR_OK ? crv2 : crv;
14 }
```

Figure 2.3: Excerpt from the function `NSC_Decrypt()` in the NSS cryptographic library, version 3.46.

[25].) Although Rust demands some extra work to satisfy its strict rules (e.g., its borrow checker), this paid off when dealing with tricky memory safety bugs, and gave us confidence to write and refactor our code freely.

2.4 Evaluation

We used Pitchfork to verify several functions from Mozilla's NSS cryptographic library [157]—widely used by many applications including Firefox—and from the `libsignal` cryptographic library [198]. Table 2.1 shows the results of Pitchfork's analysis on `libsignal` commit `71954c5` and on NSS commit `ee786a6d6`; most of these functions were verified by Pitchfork to be constant-time, up to its assumptions.[2] For some functions, Pitchfork's analysis timed out; in these cases, more aggressive assumptions—e.g., more constraints on input variables, or more function hooks to assume correct the implementations of more helper functions—may allow Pitchfork to verify the functions more quickly.

[2]Like UC-KLEE [182], Pitchfork makes a number of assumptions which simplify the solver's burden: e.g., it bounds the number of iterations allowed for loops, bounds the sizes of some operations like `memcpy`, and chooses an arbitrary layout of objects in memory rather than considering all possibilities.

Table 2.1: Functions verified by Pitchfork, up to its assumptions. NSS functions are taken from NSS commit `ee786a6d6`; a † indicates that Pitchfork found a constant-time violation in that function in NSS version 3.46, which was fixed before commit `ee786a6d6`. `libsignal` functions are taken from commit `71954c5`. Verified* means that Pitchfork verified the function, but due to its assumptions and/or constraints on the function inputs, its analysis did not reach 100% code coverage. Timed out means that Pitchfork's analysis did not complete due to timing out (or reaching other resource limits), but Pitchfork did not find any vulnerabilities in the allotted time.

Library	Function	Result
NSS	`sftk_SSLMACVerify`	Verified †
	`stfk_SSLMACSign`	Verified
	`NSC_EncryptUpdate`	Timed out †
	`NSC_EncryptFinal`	Timed out
	`NSC_Encrypt`	Timed out
	`NSC_DecryptUpdate`	Verified*
	`NSC_DecryptFinal`	Verified †
	`NSC_Decrypt`	Timed out †
	`ssl3_AESGCM`	Verified
	`ssl3_ChaCha20Poly1305`	Verified
	`ssl3_SignHashesWithPrivKey`	Timed out
	`ssl3_MACEncryptRecord`	Timed out
	`ssl3_ConsumeHandshake`	Timed out
	`ssl_ConstructServerHello`	Timed out
`libsignal`	`sender_chain_key_create`	Verified*
	`sender_chain_key_get_iteration`	Verified
	`sender_chain_key_create_message_key`	Verified*
	`sender_chain_key_create_next`	Verified*
	`sender_chain_key_get_seed`	Verified
	`ratchet_chain_key_create`	Verified*
	`ratchet_chain_key_get_key`	Verified
	`ratchet_chain_key_get_index`	Verified
	`ratchet_chain_key_get_message_keys`	Verified*
	`ratchet_chain_key_create_next`	Verified*
	`ratchet_root_key_create`	Verified*
	`ratchet_root_key_create_chain`	Verified*
	`ratchet_root_key_get_key`	Verified
	`ratchet_root_key_compare`	Verified
	`group_cipher_encrypt`	Verified*
	`group_cipher_decrypt`	Verified*
	`session_cipher_create`	Verified

Pitchfork found several constant-time violations in NSS version 3.46, in the functions marked with † in Table 2.1. (As shown in Table 2.1, Pitchfork's analysis of NSS commit `ee786a6d6` confirms that these violations have been fixed.) We discussed one of those vulnerabilities in Section 2.2.2. In the remainder of this section, we discuss two other vulnerabilities in NSS 3.46 which were found by Pitchfork, including CVE-2019-11745.

NSC_Decrypt

Figure 2.3 shows a constant-time violation in NSS version 3.46, which was found by Pitchfork. In this code, `NSC_Decrypt()` first calls `NSC_DecryptUpdate()` and then `NSC_DecryptFinal()` before returning. We can see the efforts taken by the programmers to have this protocol-level code remain constant-time; for instance, even if the call to `NSC_DecryptUpdate()` returns an error value, `NSC_Decrypt()` still calls `NSC_DecryptFinal()`, discarding the result, so as not to expose the return value of `NSC_DecryptUpdate` via a timing channel. However, Pitchfork's analysis of the called function `NSC_DecryptFinal()` shows that its return value, stored in `crv2` on line 9, could be influenced by secrets—in particular, whether the padding was valid in the decrypted plaintext. This dependence of the return value on the padding validity persists even after the vulnerability in `NSC_DecryptFinal()` shown in Figure 2.2 (Section 2.2.2) is fixed: although the fixed `NSC_DecryptFinal()` no longer *leaks* that value through the timing channel, the return value is still dependent on the padding validity, due to the program logic. Unfortunately, the code shown here immediately leaks the `NSC_DecryptFinal()` return value through a timing channel again, as `crv2` is used to influence a branch condition on line 10. Pitchfork correctly flags this as a violation; the correct fix is to use constant-time techniques to update `*pulDataLen` without a timing dependency on `crv2`.

Unlike in the previous examples, here we see how Pitchfork found a violation through its ability to *enter* a call rather than use a function hook to ignore the implementation. It was only because of Pitchfork's analysis of the called function `NSC_DecryptFinal()` that Pitchfork found the value `crv2` to be tainted, thus discovering the constant-time violation. Both function

```
1  if (context->doPad) {
2    /* deal with previous buffered data */
3    if (context->padDataLength != 0) {
4      /* fill in the padded to a full block size */
5      for (i = context->padDataLength; (ulPartLen != 0) && i < context->blockSize; i++)
       ↪  {
6          context->padBuf[i] = *pPart++;
7          ulPartLen--;
8          context->padDataLength++;
9      }
10     /* not enough data to encrypt yet? then return */
11     if (context->padDataLength != context->blockSize) {
12       // ...
13     }
14     /* encrypt the current padded data */
15     rv = (*context->update)(context->cipherInfo, pEncryptedPart, &padoutlen,
16                             context->blockSize, context->padBuf, context->blockSize);
17     if (rv != SECSuccess) {
18       // ...
19     }
20     pEncryptedPart += padoutlen;
21     maxout -= padoutlen;
22   }
23   // ...
24 }
25 rv = (*context->update)(context->cipherInfo, pEncryptedPart, &outlen, maxout,
26                         pPart, ulPartLen);
27 // ...
```

Figure 2.4: Excerpt from the function `NSC_EncryptUpdate()` in the NSS cryptographic library, version 3.46. Some formatting has been adjusted to fit the page.

hooks and full interprocedural analysis are important tools; Pitchfork supports both, giving the user maximum freedom to make the tradeoffs appropriate for each analysis.

CVE-2019-11745

Figure 2.4 shows an excerpt from the function `NSC_EncryptUpdate()` in NSS 3.46, containing the critical-severity bug CVE-2019-11745 which was found by Pitchfork.

Specifically, in the call to `context->update` on line 15, the fourth parameter indicates the maximum length which can be written to the output buffer `pEncryptedPart`. The code sets this parameter to `context->blockSize`, even though only `maxout` bytes remain in the output buffer, and nowhere does `NSC_EncryptUpdate()` confirm that `maxout >= context->blockSize`. This

may result in a small out-of-bounds write; the call to `context->update` on line 15 can write up to about `context->blockSize` bytes off of the end of the output buffer.

However, that's not the most serious consequence of the error here, nor was it the problem discovered by Pitchfork. After writing to the output buffer, `context->update` writes to the variable `padoutlen` indicating the number of bytes written. Supposing `context->update` does write the allowed maximum of `context->blockSize` bytes to the buffer, we will have `padoutlen == context->blockSize`. Then, on line 21, the code updates `maxout`, which formerly contained the true number of bytes remaining in the output buffer, by subtracting `padoutlen`, which is greater than `maxout` in this case. Unfortunately, since `maxout` is declared with an `unsigned` type, the subtraction will wrap around and give a very large positive value for `maxout`. Subsequently, on line 25, the next call to `context->update` may write up to `maxout` many bytes—i.e., effectively arbitrarily many bytes—to the output buffer, far out of bounds of the buffer's allocation.

This arbitrarily large out-of-bounds write easily results in a constant-time violation for Pitchfork to detect. In particular, Pitchfork reported the possibility for the out-of-bounds write to overwrite a later pointer with a secret value, so that dereferencing that pointer is a constant-time violation.

2.5 Future and related work

There has been a lot of work related to constant-time programming, including both constant-time verification and constant-time language design.

Constant-time verification

Both `ct-verif` [6] and Binsec/Rel [49] verify constant-time properties for existing code. `ct-verif`, like Pitchfork, focuses on existing code in languages such as C/C++; while Bin-

sec/Rel verifies compiled binaries instead. Both of these efforts focus on verifying cryptographic primitives, whereas Pitchfork is designed to verify protocol-level code.

Constant-time language design

Other work on constant-time verification proposes rewriting existing cryptographic code in new languages designed for formal verification. For example, FaCT [37] is a C-like language with formal constant-time guarantees; Jasmin [5] more closely resembles an assembly language; and HACL* [243] provides a formally-verified cryptographic library written in the F* language. Similarly, CT-Wasm [223] extends the WebAssembly portable bytecode language with support for annotating secret data and providing constant-time semantics. In contrast, Raccoon [183] transforms existing LLVM IR to make it constant-time.

Spectre

Even if we eliminate all side-channels due to constant-time violations, we still have to worry about Spectre attacks [123] and their variants (e.g., [91, 119, 125]). Spectre attacks have received significant attention recently due to the extensive powers they provide to attackers. However, Spectre vulnerabilities are notoriously difficult to exploit. Indeed, for this reason, traditional constant-time violations are *even more dangerous* than Spectre vulnerabilities, as they are far easier for attackers to exploit. In this chapter, following most recent work on constant-time programming (e.g., [6, 37, 223]), we consider only sequential execution and leave speculative execution for future work. In the next chapter, we will extend Pitchfork to defend against Spectre vulnerabilities as well — building a program analysis tool which captures both traditional timing side-channel vulnerabilities and Spectre vulnerabilities.

2.6 Conclusion

Pitchfork uses underconstrained symbolic execution to identify violations of the constant-time programming principles in real cryptographic libraries, including a critical vulnerability

in NSS which was assigned CVE-2019-11745. Pitchfork can analyze both cryptographic primitives and protocol-level code, making it a valuable tool for defending today's cryptographic implementations from side-channel vulnerabilities.

Availability

Both Pitchfork (our constant-time verifier) and Haybale (the general symbolic execution engine on which Pitchfork is built) are available open-source. Pitchfork can be found on GitHub at https://github.com/PLSysSec/haybale-pitchfork, or on Rust's package manager at https://crates.io/crates/haybale-pitchfork. Likewise, Haybale can be found on GitHub at https://github.com/PLSysSec/haybale or on Rust's package manager at https://crates.io/crates/haybale.

Acknowledgments

We would like to thank the NSS developers and Patrick Liu for their help and useful discussions. This work was supported in part by a gift from Cisco, the NSF under Grant Number CCF-1918573, the Global Research Outreach program of Samsung Research, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapter 2, in part, is a reprint of the material as it was submitted to TECHCON 2020. Disselkoen, Craig; Cauligi, Sunjay; Tullsen, Dean; Stefan, Deian. SRC, 2020. The dissertation author was the primary investigator and author of this material.

Chapter 3

Finding Spectre Vulnerabilities in Crypto Code with Pitchfork

Protecting secrets in software is hard. Security and cryptography engineers must write programs that protect secrets, both at the source level and when they execute on real hardware. Unfortunately, as we have seen already in the previous chapters, hardware too easily divulges information about a program's execution via *cache side-channels* — e.g., an attacker can learn secrets by simply observing (via timing or methods such as PRIME+ABORT) the effects of a program on the CPU cache [67].

The most robust way to deal with cache side-channels is via *constant-time programming* — the paradigm used to implement almost all modern cryptography [14, 43, 52, 176, 177]. Constant-time programs can neither branch on secrets nor access memory based on secret data.[1] These restrictions ensure that programs do not leak secrets via timing side-channels — at least, given some simplifying assumptions about the microarchitectural features of the hardware.

Unfortunately, these guarantees are moot for most modern hardware: Spectre [123], Meltdown [133], ZombieLoad [189], RIDL [209], and Fallout [30] are all dramatic examples of attacks

[1] More generally, constant-time programs cannot use secret data as input to any variable-time operation — e.g., floating-point multiplication.

that exploit microarchitectural features. These attacks reveal that code that is deemed constant-time in the usual sense may, in fact, leak information on processors with microarchitectural features. The decade-old constant-time recipes are no longer enough.[2]

In this chapter, we explore constant-time in the presence of microarchitectural features that have been exploited in recent attacks: out-of-order and speculative execution. We focus on constant-time for two key reasons. First, *impact*: constant-time programming is largely used in real-world crypto libraries and high-assurance code, where developers already go to great lengths to eliminate leaks via side-channels. Second, *foundations:* constant-time programming is already rooted in foundations, with well-defined semantics [16, 37]. These semantics consider very powerful attackers — e.g., attackers in [16] have control over the cache and the scheduler. An advantage of considering powerful attackers is that the semantics can overlook many hardware details — e.g., since the cache is adversarially controlled, there is no point in modeling it precisely — making constant-time amenable to automated verification and enforcement.

Contributions

In this chapter, we first define *speculative constant-time*, an extension of constant-time for machines with out-of-order and speculative execution. This definition allows us to discover microarchitectural side channels in a principled way — all four classes of Spectre attacks as classified by Canella et al. [32], for example, manifest as violations of speculative constant-time.

Then, we present an extension of Pitchfork which detects violations of our speculative constant-time property. Although in Chapter 2 Pitchfork was built on Haybale, this speculative extension of Pitchfork is built on top of the `angr` symbolic execution engine [197]. Like other symbolic analysis tools, Pitchfork suffers from path explosion, which limits the depth of speculation we can analyze. Nevertheless, we are able to use Pitchfork to detect leaks in the well-known Kocher test cases [122] for Spectre v1, as well as our more extensive test suite which includes Spectre v1.1 variants. Furthermore, subsequent work in [36] (but not part of this dissertation) uses

[2] OpenSSL found this situation so hopeless that they recently updated their security model to explicitly exclude "physical system side channels" [168].

Pitchfork to analyze, and find leaks in, real cryptographic code from the libsodium, OpenSSL, and curve25519-donna libraries.

Open source

Pitchfork and our test suites are open source and available at `https://pitchfork.programming.systems`.

3.1 Motivating example

In this section, we show why classical constant-time programming is insufficient when attackers can exploit microarchitectural features. The example below is a variant of the well-known Spectre v1 attack [123].

```
1 int A[4]; // non-secret data
2 int B[4]; // non-secret data
3 int Key[8]; // secret key
4 void vulnerable(int idx) {
5   if (idx < 4) {
6     int y = A[idx];
7     int z = B[y];
8   }
9 }
```

If the bounds check on line 5 passes, the program uses `idx` to index into a public array `A`, saves the value into `y`, and uses `y` to index into another public array `B`. On the other hand, if the bounds check fails, the program skips the array operations. In a sequential execution, this program neither loads nor branches on secret values; it thus trivially satisfies the constant-time discipline.

However, modern processors do not execute sequentially. Instead, they continue fetching instructions before prior instructions are complete. In particular, a processor may continue

fetching instructions beyond a conditional branch, before evaluating the branch condition. In that case, the processor *guesses* which branch will be taken — for example, the processor may erroneously guess that the branch condition in line 5 evaluates to `true`, even though `idx` is `9`. It will therefore continue down the "true" branch speculatively. In hardware, such guesses are made by a branch predictor, which may have been mistrained by an adversary.

In this chapter, we conservatively assume that the adversary has total control of the branch predictor, and thus can cause the processor to mispredict any branch the adversary wishes. If the attacker causes the processor to mispredict the bounds check on line 5 — for instance, to predict `true` when `idx` is `9` — then the processor will speculatively execute the array operations anyway. At line 6, the processor will read `A[9]`, an out-of-bounds value which in this case may be equivalent to `Key[0]` — part of the secret key. Then, at line 7, the processor uses this transient secret value to index into `B`. Accessing this address affects the CPU cache state, so an attacker armed with a cache side-channel attack could subsequently determine which address in `B` was accessed, and thus recover the secret value. Though this secret leakage cannot happen under sequential execution (the code obeys constant-time programming), this example highlights how constant-time programming is insufficient when attackers can exploit microarchitectural features.

3.2 Speculative constant-time

Clearly, we need a new notion of what it means for a program to be secure from side-channel attacks. We propose an extension of constant-time security which we call *speculative constant-time*.

Like classical constant-time, our notion of speculative constant-time does not directly model caches, nor any of the microarchitectural predictors which could be exploited by an attacker. Rather, our notion of speculative constant-time is based on externally visible effects, i.e., memory accesses and control flow. We can thus reason about *any* possible cache implementation — instead

of trying to define exactly which cache accesses may leak to the attacker under a specific cache eviction policy, we conservatively assume that the attacker has perfect knowledge of the memory addresses of *all* cache accesses. Likewise, following classical constant-time, we prohibit control flow from depending on secret data; this makes it unnecessary for us to track various other side channels. Indeed, while channels such as port contention or register renaming produce distinct measurable effects [123], they only serve to leak the path taken through the code — and thus modeling these observations separately would be redundant. Most importantly, our speculative constant-time diverges from classical constant-time by allowing the attacker total control over branch prediction. The attacker can supply arbitrary predictions for conditional branches, indirect branches, return addresses, and other microarchitectural predictors.

This approach has two important consequences. First, it allows our semantics to remain *tractable* and *amenable to verification*. For instance, we do not need to model the behavior of the cache or any branch predictor. Second, our notion of speculative constant-time is *robust*, i.e., it holds for all possible branch predictors and replacement policies — assuming that they do not leak secrets directly, a condition that is achieved by all practical hardware implementations.

3.3 Detecting violations

We develop a tool Pitchfork which detects violations of this speculative constant-time property (SCT). Attackers in this framework have broad powers over many different microarchitectural predictors and even out-of-order scheduling; to make the analysis more tractable, Pitchfork does not detect SCT violations based on alias prediction, indirect jumps, or return stack buffers. Nevertheless, Pitchfork still exposes attacks based on Spectre variants 1, 1.1, and 4.

Conceptually, Pitchfork first generates a set of *schedules* representing various *worst-case* attackers. Each schedule represents one combination of predictions and scheduling decisions which is legal for the attacker to make for the program. Pitchfork's set of worst-case schedules

is far smaller than the set of all possible schedules for the program, but is nonetheless sound: if there is an SCT violation in any possible schedule, then there will be an SCT violation in one of the worst-case schedules. Pitchfork then checks for secret leakage by symbolically executing the program under each schedule.

3.3.1 Schedule generation

Given a program, Pitchfork generates a set of schedules representing various worst-case attackers. Pitchfork's schedule generation is parametrized by a *speculation bound*, which limits the size of the reorder buffer, and thus the depth of speculation.

In general, Pitchfork constructs worst-case schedules to maximize speculation. These schedules eagerly fetch instructions until the reorder buffer is full, i.e., the size of the reorder buffer equals the speculation bound. Once the reorder buffer is full, the schedules only retire instructions as necessary to fetch new ones.

When conditional branches are to be fetched, Pitchfork constructs schedules containing both possible outcomes: one where the branch is guessed true and one where the branch is guessed false. For the mispredicted outcome, Pitchfork's schedules execute the branch as late as possible (i.e., it is the oldest instruction in the reorder buffer and the reorder buffer is full), which delays the rollback of mispredicted paths.

To account for load-store forwarding hazards and potential mispredictions, Pitchfork constructs schedules containing all possible forwarding outcomes. For every load instruction l in the program, Pitchfork finds all prior stores s_i within the speculation bound that would resolve to the same address. Then, for each such store, Pitchfork constructs a schedule that would cause that store to forward its data to l. Additionally, Pitchfork constructs a schedule where none of the prior stores s_i have resolved addresses, forcing the load instruction to read from memory.

For all instructions other than conditional branches and memory operations, Pitchfork only constructs schedules where these instructions are executed eagerly and in order. Reordering of

these instructions is uninteresting: either the instructions naturally commute, or data dependencies prevent the reordering (i.e., the reordered schedule is invalid for the program). This intuition matches with the property that any out-of-order execution of a given program has the same final result regardless of its schedule.

3.3.2 Implementation and evaluation

We implement Pitchfork on top of the `angr` binary-analysis tool [197]. Pitchfork uses `angr` to symbolically execute a given program according to each of its worst-case schedules, flagging any resulting secret leakage.

To sanity check Pitchfork, we create and analyze a set of Spectre v1 and v1.1 test cases, and ensure we flag their SCT violations. Our test cases are based off the well-known Kocher Spectre v1 examples [122]. Since many of the Kocher examples exhibit violations even during sequential execution, we create a new set of Spectre v1 test cases which only exhibit violations when executed speculatively. We also develop a similar set of test cases for Spectre v1.1 data attacks.

Pitchfork necessarily inherits the limitations of `angr`'s symbolic execution. For instance, `angr` concretizes addresses for memory operations instead of keeping them symbolic. Furthermore, exploring every speculative branch and potential store-forward within a given speculation bound leads to an explosion in state space. In our tests, we were able to support speculation bounds of up to 20 instructions. We were able to increase this bound to 250 instructions when we disabled checking for store-forwarding hazards. Though these bounds do not capture the speculation depth of some modern processors, Pitchfork still correctly finds SCT violations in all our test cases. We consider the design and implementation of a more scalable tool future work.

Author's note

Although not part of this dissertation, the published version of this chapter ([36]) contains much additional material on the theoretical foundations for SCT. It also describes how Pitchfork was used to find SCT violations in real cryptographic libraries.

Acknowledgments

Chapter 3, in part, contains material reprinted from the Proceedings of the ACM Conference on Programming Language Design and Implementation. Cauligi, Sunjay; Disselkoen, Craig; v. Gleissenthall, Klaus; Tullsen, Dean; Stefan, Deian; Rezk, Tamara; Barthe, Gilles. ACM, 2020. The dissertation author was the primary investigator and author of the reprinted material.

Chapter 4

SoK: Practical Foundations for Software Spectre Defenses

Spectre attacks have upended the foundations of computer security [123]. With Spectre, attackers can steal secrets across security boundaries — both hardware boundaries provided by the process abstraction [229], and software boundaries provided by memory safe languages and software-based fault isolation (SFI) techniques [216]. In response, the security community has been working on program analysis tools to both find Spectre vulnerabilities and to guide mitigations (e.g., compiler passes) that can be used to secure programs in the presence of this class of attacks. But Spectre attacks — and speculative execution in general — violate our typical assumptions and abstractions and have proven particularly challenging to reason about and defend against.

Many existing defense mechanisms against Spectre are either incomplete (and thus miss possible attacks) or overly conservative (and thus slow). For example, the MSVC compiler's `/Qspectre` pass — one of the first compiler-based defenses against Spectre [171] — inserts mitigations by finding Spectre gadgets (or patterns). Since these patterns are not based on any rigorous analysis, the compiler misses similarly vulnerable code patterns [167]. As another

example, Google Chrome adopted process isolation as its core defense mechanism against Spectre attacks [184]. This is also unsound: Canella et al. [32], for example, show that Spectre attacks can be performed across the process boundary. On the other side of the spectrum, inserting fences at every load or control flow point is sound but prohibitively slow [162].

Language-based security can help us achieve — or at least understand the trade-offs of giving up on — *performance* and *provable security guarantees*. Historically, the security community has turned to language-based security to solidify intricate defense techniques — from SFI enforcement on x86 [156], to information flow control enforcement [186], to eliminating side-channel attacks with constant-time programming [16]. At the core of language-based security are *program semantics* — rigorous models of program behavior which serve as the basis for *formal security policies or foundations*. These policies help us carefully and explicitly spell out our assumptions about the attacker's strength and ensure that our tools are sound with respect to this class of attackers — e.g., that Spectre vulnerability-detection or -mitigation tools find and mitigate the vulnerabilities they claim to find and mitigate.

Formal foundations are key to performance too. Without formalizations, Spectre defenses are usually either overly conservative (which leads to unnecessary and slow mitigations) or crude (and thus vulnerable). For example, speculative load hardening [34] is *safe* — it safely eliminates Spectre-PHT attacks — but is overly conservative and slow: It assumes that *all* array indexing operations must be hardened. In practice, this is not the case [101, 211]. Crude techniques like oo7 [218] are both inefficient *and* unsafe — they impose unnecessary restrictions yet also miss vulnerable code patterns. Foundations allow us to craft defenses that are minimal (e.g., they target the precise array indexes that need hardening [81, 211]) and provably secure.

Alas, not all foundations are equally practical. Since speculative execution breaks common assumptions about program semantics — the cornerstone of language-based methods — existing Spectre foundations explore different design choices, many of which have important ramifications on defense tools and the software produced or analyzed by these tools (Table 4.1). For instance,

one key choice is the *leakage model* of the semantics, which determines what the attacker is allowed to observe. Another choice is the *execution model*, which simultaneously captures the attacker's strength and which Spectre variants the resulting analysis (or mitigation) tool can reason about. These choices in turn determine which *security policies* can be verified or enforced by these tools.

While formal design decisions fundamentally impact the soundness and precision of Spectre analysis and mitigation tools, they have not been systematically explored by the security community. For example, while there are many choices for a leakage model, the constant-time [16] and sandbox isolation [81] models are the most pragmatic; leakage models that only consider the data cache trade off security for no clear benefits (e.g., scalability or precision). As another example, the most practical execution models borrow (again) from work on constant-time: They are detailed enough to capture practical attacks, but abstract across different hardware — and are thus useful for both software-based verification and mitigation techniques. Models which capture microarchitectural details like cache structures make the analysis unnecessarily complicated: They do not fundamentally capture additional attacks and give up on portability.

Contributions

In this chapter, we systematize the community's knowledge on Spectre foundations and identify the different design choices made by existing work and their tradeoffs. This complements existing, excellent surveys [31, 32, 230] on the low-level details of Spectre attacks and defenses which do not consider foundations or, for example, high-level security policies. Throughout, we discuss the limitations of existing formal frameworks, the defense tools built on top of these foundations, and future directions for research. In summary, we make the following contributions:

- Study existing foundations for Spectre analysis in the form of semantics, discuss the different design choices which can be made in a semantics, and describe the tradeoffs of each choice.

- Compare many proposed Spectre defenses — both with and without formal foundations — using a unifying framework, which allows us to understand differences in the security guarantees they offer.
- Identify open research problems, both for foundations and for Spectre software defenses in general.
- Provide recommendations both for developers and for the research community that could result in tools with stronger security guarantees.

Scope

In this systematization, we focus on software-only defenses against Spectre attacks. We focus on *Spectre* because most other transient attacks (e.g., Meltdown [133], LVI [207], MDS [100], or Foreshadow [206]) can efficiently be addressed in the hardware, through microcode updates or new hardware designs. (This is also the reason existing software-based tools against transient execution attacks focus solely on Spectre, as we discuss in Section 4.3.4.) We focus on *defenses* because prior work, notably Canella et al. [32], already give an excellent overview of the types of Spectre vulnerabilities and the powerful capabilities they give attackers. And we focus on *software-only* defenses — although proposals for hardware defenses are extremely valuable, hardware design cycles (and hardware upgrade cycles) are very long. Moreover, software foundations are useful for understanding hardware and hardware-software co-designs (e.g., they directly affect execution and leakage models). Having secure software foundations allows us to defend against today's attacks on today's hardware, and tomorrow's as well.

4.1 Preliminaries

In this section, we first discuss Spectre attacks and how they violate security in two particular application domains: high-assurance cryptography and isolation of untrusted code.

Then, we provide an introduction to formal semantics for security and its relevance to secure speculation in these application domains.

4.1.1 Spectre vulnerabilities

Spectre [8, 15, 91, 119, 123, 125, 140, 240] is a recently discovered family of vulnerabilities stemming from *speculative execution* on modern processors. Spectre allows attackers to learn sensitive information by causing the processor to mispredict the targets of control flow (e.g., conditional jumps or indirect calls) or data flow (e.g., aliasing or value forwarding). When the processor realizes it has mispredicted, it *rolls back* execution, erasing the programmer-visible effects of the speculation. However, *microarchitectural* state — such as the state of the data cache — is still modified during speculative execution; these changes can be leaked during speculation and can persist even after rollback. As a result, the attacker can recover sensitive information from the microarchitectural state, even if the sensitive information was only speculatively accessed.

Figure 4.1 gives an example of a vulnerable function: An attacker can exploit branch misprediction to leak arbitrary memory via the data cache. The attacker first primes the branch to predict that the condition `i < arrALen` is true by causing the code to repeatedly run with appropriate (small) values of `i`. Then, the attacker provides an out-of-bounds value for `i`. The processor (mis)predicts that the condition is still true and *speculatively* loads out-of-bounds (potentially secret) data into `x`; subsequently, it uses the value `x` as part of the address of a memory read operation. This encodes the value of `x` into the data cache state — depending on the value of `x`, different cache lines will be accessed and cached. Once the processor resolves the misprediction, it rolls back execution, but the data cache state persists. The attacker can later interpret the data cache state in order to infer the value of `x`.

4.1.2 Breaking cryptography with Spectre

```
if (i < arrALen) {  // mispredicted
  int x = arrA[i];  // x is oob value
  int y = arrB[x];  // leaked via address!
  // ...
```

Figure 4.1: Code snippet which an attacker can exploit using Spectre. If an attacker can control `i` and cause the processor to transiently enter the branch, the attacker can load an arbitrary value from memory into `x`, which is then leaked via the following memory access.

High-assurance cryptography has long relied on *constant-time programming* [16] in order to create software which is secure from timing side-channel attacks. Constant-time programming ensures that program execution does not depend on secrets. It does this via three rules of thumb [16, 19]: control flow (e.g., conditional branches) should not depend on secrets, memory access patterns (e.g., offsets into arrays) should not be influenced by secrets, and secrets should not be used as operands to variable-latency instructions (e.g., floating-point instructions or integer division on many processors). These rules ensure that secrets remain safe from an attacker powerful enough to perform cache attacks, exfiltrate data via branch predictor state, or snoop data via port contention [22].

In the face of Spectre, constant-time programming is not sufficient. The snippet in Figure 4.1 is indeed constant-time if `arrA` contains only public data (and `i` and `arrALen` are also public). Yet, a Spectre attack can still abuse this code to leak secrets from anywhere in memory.

Cache-based leaks are not the only way for an attacker to learn cryptographic secrets: In the following example, an attacker can again (speculatively) leak out-of-bounds data, but this time the leak is via control flow.

```
if (i < arrALen) {
  int x = arrA[i];
  switch(x) {  // leak via branching!
    case 'A': /* ... */
```

```
case 'B': /* ... */
// ...
```

This code uses `x` as part of a branch condition (in a `switch` statement). Just as before, the attacker can speculatively read arbitrary memory into `x`. They can then leak the value of `x` in several ways, including: (1) Based on the different execution times of the various cases; (2) through the data cache, based on differing (benign) memory accesses performed in the various cases; (3) through the instruction cache or micro-op cache [185], based on which instructions were (speculatively) accessed; or (4) through port contention [22], branch predictor state [107], or other microarchitectural resources that differ among the branches.

4.1.3 Breaking software isolation with Spectre

Spectre attacks also break important guarantees in the domain of *software isolation*. In this domain, a host application executes untrusted code and wants to ensure that the untrusted code cannot access any of the host's data. Common examples of software isolation include JavaScript or WebAssembly runtimes, or even the Linux kernel, through eBPF [63]. Spectre attacks can break the memory safety and isolation mechanisms commonly used in these settings [111, 141, 162, 196].

We demonstrate with a small example:

```
int guest_func() {
  get_host_val(1);
  get_host_val(1);
  // ... repeat ...
  char c = get_host_val(99999);
  // ... leak c
}
```

```
char get_host_val(int idx) {
  if (idx < 100) { // check if within bounds
    return host_arr[idx];
  } else {
    return 0;
} }
```

Here, an attacker-supplied guest function `guest_func` calls the host function `get_host_val` to get values from an array. Although `get_host_val()` implements a bounds check, the attacker can still speculatively access out-of-bounds data by mistraining the branch predictor — breaking any isolation guarantees. Once the attacker (speculatively) obtains an out-of-bounds value of their choosing, they can leak the value (e.g., via data cache, etc.) and recover it after the speculative rollback. In this setting, we need to ensure that, *even speculatively*, untrusted code cannot break isolation.

4.1.4 Security properties and execution semantics

Formally, we will define safety from Spectre attacks as a security property of a *formal (operational) semantics*. The semantics abstractly captures how a processor executes a program as a series of state transitions. The states, which we will write as σ, include any information the developer will need to track for their analysis, such as the current instruction or command and the contents of memory and registers. The developer then defines an *execution model* — a set of transition rules that specify how state changes during execution. For example, in a semantics for a low-level assembly, a rule for a `store` instruction will update the resulting state's memory with a new value.

The rules in the execution model determine how and when speculative effects happen. For example, in a sequential semantics, a conditional branch will evaluate its condition then step to the appropriate branch. A semantics that models branch prediction will instead *predict* the

condition result and step to the predicted branch. We adapt notation from Guarnieri et al. [81], writing $[\![\cdot]\!]^{seq}$ to represent the execution model for standard sequential execution. We notate other execution models similarly; for example, $[\![\cdot]\!]^{pht}$ models prediction for Spectre-PHT attacks — i.e., conditional branch prediction. Other execution models are listed in Table 4.2.

Next, to precisely specify the attacker model, the developer must define which *leakage observations* — information produced during an execution step — are visible to an attacker. For example, they may decide that rules with memory accesses leak the addresses being accessed. The set of leakage observations in a semantics' rules is its *leakage model*. We again borrow notation from Guarnieri et al. [81], which defines the leakage models $[\![\cdot]\!]_{ct}$ and $[\![\cdot]\!]_{arch}$. The $[\![\cdot]\!]_{ct}$ model exposes leakage observations relevant to constant-time security: The sequence of control flow (the *execution trace*) and the sequence of addresses accessed in memory (the *memory trace*).[1] The $[\![\cdot]\!]_{arch}$ model, on the other hand, exposes all values loaded from memory in addition to the addresses themselves (or equivalently, it exposes the trace of register values) [81]. Under this model, an attacker is allowed to observe all architectural computation; for a value to remain unobserved, it cannot be accessed at all over the course of execution, adversarial or otherwise. Since the leakage observations in $[\![\cdot]\!]_{arch}$ are a strict superset of those in $[\![\cdot]\!]_{ct}$, we say that $[\![\cdot]\!]_{arch}$ is *stronger* than $[\![\cdot]\!]_{ct}$ (i.e., it models a more powerful attacker). These properties make $[\![\cdot]\!]_{arch}$ most useful for software isolation, as any out-of-bounds accesses will immediately show up in an $[\![\cdot]\!]_{arch}$ leakage trace.

Surprisingly, the $[\![\cdot]\!]_{ct}$ and $[\![\cdot]\!]_{arch}$ leakage models generalize well to speculative execution — for example, if we want to construct a semantics for Spectre-PHT attacks, we need only modify a sequential constant-time semantics to account for branch misprediction. Indeed, the execution model and leakage model of a semantics are orthogonal; we call the combination of the two the *contract* provided by the semantics — a sequential constant-time semantics has the contract $[\![\cdot]\!]_{ct}^{seq}$, while our hypothetical Spectre-PHT semantics would provide the contract $[\![\cdot]\!]_{ct}^{pht}$.

[1] Like Guarnieri et al. [81], we omit variable-latency instructions from our formal model for simplicity.

Formally, the contract governs the attacker-visible information produced when executing a program: Given a program p, a semantics with contract $[\![\cdot]\!]_\ell^\alpha$, and an initial state σ, we write $[\![p]\!]_\ell^\alpha(\sigma)$ for the sequence (or *trace*) of leakage observations the semantics produces when executing p.

After determining a proper contract, the developer must finally define the *policy* that their security property enforces: Precisely which data can and cannot be leaked to the attacker. Formally, a policy π is defined in terms of an equivalence relation $\simeq_\pi$ over states, where $\sigma_1 \simeq_\pi \sigma_2$ iff σ_1 and σ_2 agree on all values that are public (but may differ on sensitive values).

Armed with these definitions, we can state security as a *non-interference property*: A program satisfies *non-interference* if, for any two π-equivalent initial states for a program p, an attacker cannot distinguish the two resulting leakage traces when executing p. A developer has several choices when crafting a suitable semantics and security policy; these choices greatly influence how easy or difficult it is to detect or mitigate Spectre vulnerabilities. We cover these choices in detail in Section 4.2: Sections 4.2.1 and 4.2.2 discuss choices in leakage models $[\![\cdot]\!]_\ell$ and security policies π. Sections 4.2.3 and 4.2.4 discuss tradeoffs for different execution models $[\![\cdot]\!]^\alpha$ and the transition rules in a semantics. In Section 4.2.5, we discuss how the input language of the semantics affects analysis; and finally, in Section 4.2.6, we discuss which microarchitectural features to include in formal models.

4.2 Choices in semantics

The foundation of a well-designed Spectre analysis tool is a carefully constructed formal semantics. Developers face a wide variety of choices when designing their semantics — choices which heavily depend on the attacker model (and thus the intended application area) as well as specifics about the tool they want to develop. Cryptographic code requires different security properties, and therefore different semantics and tools, than in-process isolation. Many of these choices also look different for *detection* tools, focused only on finding Spectre vulnerabilities,

vs. *mitigation* tools, which transform programs to be secure. In this section, we describe the important choices about semantics that developers face, and explain those choices' consequences for Spectre analysis tools and for their associated security guarantees. We also point out a number of open problems to guide future work in this area.

What makes a practical semantics?

A practical semantics should make an appropriate tradeoff between *detail* and *abstraction*: It should be detailed enough to capture the microarchitectural behaviors which we're interested in, but it should also be abstract enough that it applies to all (reasonable) hardware. For example, we do not want the security of our code to be dependent on a specific cache replacement policy or branch predictor implementation.

In the non-speculative world, formalisms for constant-time have been successful: The principles of constant-time programming (no secrets for branches, no secrets for addresses) create secure code without introducing processor-specific abstractions. Speculative semantics should follow this trend, producing portable tools which can defend against powerful attackers on today's (and tomorrow's) microarchitectures.

4.2.1 Leakage models

Any semantics intended to model side-channel attacks needs to precisely define its attacker model. An important part of the attacker model for a semantics is the *leakage model* — that is, what information does the attacker get to observe? Leakage models intended to support sound mitigation schemes should be *strong* — modeling a powerful attacker — and *hardware-agnostic*, so that security guarantees are portable. That said, the best choice for a leakage model depends in large part on the intended application domain.

Leakage models for cryptography

As we saw in Section 4.1.2, high-assurance cryptography implementations have long relied on the constant-time programming model; thus, semantics intended for cryptographic

Table 4.1: Comparison of various semantics and tools (on following page; legend appears here). Semantics are sorted by *Level*, then alphabetically; works without semantics are ordered last. [1]Extension to other variants is discussed, but not performed. [2]Semantics includes indirect jumps and rules to update the indirect branch predictor state, but cannot mispredict indirect jump targets. [3]"Weak" variants of semantics leak loaded values during non-speculative execution. [4]Detects only "speculative type confusion vulnerabilities", a specific subset of Spectre-PHT. [5]Mitigates Spectre-PHT without inserting fences. [6]Defends by effectively preventing speculation, so leakage model is irrelevant. [7]Effectively $[\![\cdot]\!]_{mem}$ for loads, but detects any speculative store to an attacker-controlled address, which is more similar to $[\![\cdot]\!]_{arch}$ for stores. [8]Swivel operates on WebAssembly, which does not have fences. However, Swivel can insert fences in its assembly backend.

Level – How abstract is the semantics? (Section 4.2.5)		Leakage – What can the attacker observe? (Section 4.2.1)		Variants (Section 4.2.3)
Low	Assembly-style, with branch instructions	P – Path / instructions executed	L – Values loaded from memory	P – Spectre-PHT
Medium	Structured control flow such as if-then-else	B – Speculation rollbacks	R – Values in registers	B – Spectre-BTB
High	In the style of weak memory models	M – Addresses of memory operations	S – Branch predictor state	R – Spectre-RSB
—	The work has no associated formal semantics	C – Cache lines / cache state	T – Step counter / timer	S – Spectre-STL

Fence – Does it reason about speculation fences?		Hijack – Can it model or mitigate speculative hijack?	
✓	Fully reasons about fences in the target/input code	✓	Models/mitigates speculative hijack attacks
∽	The mitigation tool inserts fences, but the analysis does not reason about fences in the target/input code (and thus cannot verify the mitigated code as secure)	→	Models/mitigates forward-edge (ijmp) hijack only
×	Does not reason about, or insert, fences	∽	Models/mitigates hijack only via speculative stores
		×	Does not model/mitigate speculative hijack attacks

Nondet. – How is nondeterminism handled? (Section 4.2.4)
OOO – Models out-of-order execution? (Section 4.2.6)
Win. – Can reason about speculation windows? (Section 4.2.3)

Tool – Does the paper include a tool?		Implementation – How does the tool detect or mitigate vulnerabilities? (Section 4.2.4)			
Det	Tool detects insecure programs or verifies secure programs	Taint	Taint tracking (abstract execution)	Manual	Manual effort
Mit	Tool modifies programs to ensure they are secure	Safety	Memory safety (abstract execution)	Fuzz	Fuzzing
Val	Tool is only used to validate the semantics, does not automatically perform any security analysis	SelfC	Self composition (abstract execution)	Flow	Data flow analysis
—	Does not include a tool	Cache	Cache must-hit analysis (abstract execution)	Struct	Structured compilation
*	Tool's connection to the semantics is incomplete or unclear (e.g., tool does not implement the full semantics)	Model	Model checking over the whole program		
		+	Includes additional work or constraints to remove sequential trace (Section 4.2.2)		

Table 4.1: Comparison of various semantics and tools, continued

Semantics or tool name	Level	Leakage		Variants	Nondet.	Fence	OOO	Win.	Hij.	Tool	Impl.
Cauligi et al. [36] (Pitchfork)	Low	$[\![\cdot]\!]_{ct}$	P,B,M	P,B,R,S	Directives	✓	✓	✓	✓	Det*	Taint
Cheang et al. [39]	Low	$[\![\cdot]\!]_{arch}$	P,M,S,R	P	Oracle	✓	×	✓	×	Det/Mit	SelfC+
Daniel et al. [48] (Binsec/Haunted)	Low	$[\![\cdot]\!]_{ct}$	P,M	P,S	Mispredict	×	×	✓	×	Det	SelfC
Guanciale et al. [78] (InSpectre)	Low	$[\![\cdot]\!]_{ct}$	P,M	P,B,R,S	—	✓	✓	×	✓	—	—
Guarnieri et al. [80] (Spectector)	Low	$[\![\cdot]\!]_{ct}$	P,B,M	P	Oracle	✓	×	✓	→	Det	SelfC+
Guarnieri et al. [81]	Low	(parametrized)		P[1]	Oracle	✓	✓	✓	×	Det	SelfC+
Mcilroy et al. [147]	Low	$[\![\cdot]\!]_{cache}$	T	P[2]	Oracle	∽	×	✓	→	Mit*	Manual
Barthe et al. [18] (Jasmin)	Medium	$[\![\cdot]\!]_{ct}$	P,B,M	P,S	Directives	✓	×	×	×	Det	Safety
Patrignani and Guarnieri [173]	Medium	$[\![\cdot]\!]_{ct}$	P,B,M,L[3]	P[1]	Mispredict	✓	×	✓	×	—	—
Vassena et al. [211] (Blade)	Medium	$[\![\cdot]\!]_{ct}$	B,M	P	Directives	✓	✓	×	×	Mit	Flow
Colvin and Winter [42]	High	$[\![\cdot]\!]_{mem}$	M	P	Weak-mem	✓	✓	×	×	Val	Model
Disselkoen et al. [56]	High	$[\![\cdot]\!]_{mem}$	M	P	Weak-mem	✓	✓	×	×	—	—
P. de León and Kinder [175] (Kaibyo)	High	$[\![\cdot]\!]_{mem}$	M	P,S	Weak-mem	✓	✓	✓	×	Det	Model
AISE [228]	—	$[\![\cdot]\!]_{cache}$	C	P	Mispredict	×	×	✓	×	Det	Cache+
ASTCVW [120]	—	$[\![\cdot]\!]_{arch}$	L	P[4]	—	×	×	×	×	Det	Taint
ELFbac [111]	—	$[\![\cdot]\!]_{arch}$	L	P	—	×[5]	×	×	✓	Mit	Struct
KLEESpectre [217] (w/ cache)	—	$[\![\cdot]\!]_{cache}$	C	P	Mispredict	✓	×	✓	×	Det	Cache
KLEESpectre [217] (w/o cache)	—	$[\![\cdot]\!]_{mem}$	M	P	Mispredict	✓	×	✓	×	Det	Taint
oo7 [218] (v1 pattern)	—	$[\![\cdot]\!]_{mem}$	M	P	—	∽	×	✓	×	Det/Mit	Flow
oo7 [218] (“weak” and v1.1 patterns)	—	$[\![\cdot]\!]_{arch}$	L	P	—	∽	×	✓	∽	Det/Mit	Flow
Specfuscator [191]	—	—[6]	—	P,B,R	—	×[5]	×	×	✓	Mit	Struct
SpecFuzz [167]	—	$[\![\cdot]\!]_{arch}$	L	P	Mispredict	—	—	—	✓	Det	Fuzz
SpecTaint [180]	—	$[\![\cdot]\!]_{mem}$[7]	M	P	Mispredict	✓	×	✓	∽	Det	Taint
SpecuSym [83]	—	$[\![\cdot]\!]_{cache}$	C	P	Mispredict	×	×	✓	×	Det	SelfC+
Swivel [162] (poisoning protection)	—	$[\![\cdot]\!]_{mem}$	M	P,B,R	—	∽[8]	×	×	✓	Mit	Struct
Swivel [162] (breakout protection)	—	$[\![\cdot]\!]_{arch}$	L	P,B,R	—	∽[8]	×	×	✓	Mit	Struct
Venkman [196]	—	$[\![\cdot]\!]_{arch}$	L	P,B,R	—	∽	×	×	✓	Mit	Struct

programs naturally choose the $[\![\cdot]\!]_{\text{ct}}$ leakage model. Like the constant-time programming model in the non-speculative world, the $[\![\cdot]\!]_{\text{ct}}$ leakage model is strong and hardware-agnostic, making it a solid foundation for security guarantees. The $[\![\cdot]\!]_{\text{ct}}$ leakage model is a popular choice among existing formalizations: As we highlight in Table 4.1, over half of the formal semantics for Spectre use the $[\![\cdot]\!]_{\text{ct}}$ leakage model (or an equivalent) [18, 36, 48, 78, 80, 173, 211]. Guarnieri et al. [81] leave the leakage model abstract, allowing the semantics to be used with several different leakage models, including $[\![\cdot]\!]_{\text{ct}}$.

Leakage models for isolation

Sections 4.1.3 and 4.1.4 describe the $[\![\cdot]\!]_{\text{arch}}$ leakage model, which is a better fit for modeling speculative isolation, e.g., for a WebAssembly runtime executing untrusted code [162] or a kernel defending against memory region probing [70]. Under $[\![\cdot]\!]_{\text{arch}}$, *all* values in the program are observable — this is what lets it easily model properties for software isolation: If we define a policy π where all values and memory regions outside the isolation boundary are secret, then software isolation security (or speculative memory safety) is simply non-interference with respect to $[\![\cdot]\!]_{\text{arch}}$ (and this π).

The $[\![\cdot]\!]_{\text{arch}}$ leakage model appears less frequently than $[\![\cdot]\!]_{\text{ct}}$ in formal models: Only two of the semantics in Table 4.1 ([39, 81]) use the $[\![\cdot]\!]_{\text{arch}}$ leakage model. On the other hand, Spectre sandbox isolation frameworks such as Swivel [162], Venkman [196], and ELFbac [111] implicitly use the $[\![\cdot]\!]_{\text{arch}}$ model, as do the detection tools SpecFuzz [167], ASTCVW [120], SpecTaint [180], and the "weak" and "v1.1" modes of oo7 [218]. The three isolation frameworks all explicitly prevent memory reads or writes to any locations outside of the isolation boundary — i.e., enforcing non-interference under $[\![\cdot]\!]_{\text{arch}}$. The detection tools, meanwhile, look for gadgets that can speculatively access *arbitrary* (or attacker-controlled) memory locations — i.e., breaking speculative memory safety. Unfortunately, these tools are not formalized, so their leakage models are not made explicit (nor clear).

Weaker leakage models

The remaining semantics and tools in Table 4.1 consider only the memory trace of a program, but not its execution trace. The $[\![\cdot]\!]_{\text{mem}}$ leakage model, like $[\![\cdot]\!]_{\text{ct}}$, allows an attacker to observe the sequence of memory accesses during the execution of the program; the $[\![\cdot]\!]_{\text{cache}}$ leakage model instead only tracks (an abstraction of) cache state. The attacker in this model can only observe cached addresses at the granularity of cache lines. A few tools have even weaker leakage models — for instance, oo7 only emits leakages that can be influenced by malicious input (see Section 4.2.3) and KLEESpectre (with cache modeling enabled) only allows the attacker to observe the final state of the cache upon termination.

All of these models, including $[\![\cdot]\!]_{\text{mem}}$ and $[\![\cdot]\!]_{\text{cache}}$, are weaker than $[\![\cdot]\!]_{\text{ct}}$ — they model less powerful attackers who cannot observe control flow. As a result, they miss attacks which leak via the instruction cache or which otherwise exploit timing differences in the execution of the program. They even miss some attacks that exploit the data cache: If a sensitive value influences a branch, an attacker could infer the sensitive value through the data cache based on differing (benign) memory access patterns on the two sides of the branch, even if no sensitive value directly influences a memory address. For instance, in the following code, even though `cond` is not used to calculate the memory address, an attacker can infer the value of `cond` based on whether `arr[a]` gets cached or not:

```
if (cond)
  b = arr[a];
else
  b = 0;
```

Because the $[\![\cdot]\!]_{\text{mem}}$ and $[\![\cdot]\!]_{\text{cache}}$ leakage models miss these attacks, they cannot provide the strong guarantees necessary for secure cryptography or software isolation. Tools which want to provide sound verification or mitigation should instead choose a strong leakage model appropriate for their application domain, such as $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$.

That said, weaker leakage models are still useful in certain settings: Tools which are interested in only certain vulnerability classes can use these weaker models to reduce the number of false positives in their analysis or reduce the complexity of their mitigation. Even though these models may miss some Spectre attacks, detection tools can still use the $[\![\cdot]\!]_{\text{cache}}$ or $[\![\cdot]\!]_{\text{mem}}$ models to find Spectre vulnerabilities in real codebases. Using a leakage model which ignores control flow leakage may help the detection tool scale to larger codebases.

Some tools [83, 217] also provide the ability to reason about what attacks are possible with particular cache configurations — e.g., with a particular associativity, cache size, or line size. This is a valuable capability for a detection tool: It helps an attacker zero in on vulnerabilities which are more easily exploitable on a particular target machine. However, security guarantees based on this kind of analysis are not portable, as executing a program on a different machine with a different cache model invalidates the security analysis. Tools that instead want to make guarantees for all possible architectures, such as verifiers or compilers, will need more conservative leakage models — models that assume the entire memory trace (and execution trace) is always leaked.

Open problems: Leakage models for weak-memory-style semantics

We have described leakage models only in terms of observations of execution traces; this is a natural way to define leakage for *operational semantics*, where execution is modeled simply as a set of program traces. However, the weak-memory-style speculative semantics proposed by Colvin and Winter [42], Disselkoen et al. [56], and Ponce de León and Kinder [175] have a more structured view of program execution (for instance, using dependency analysis or pomsets [68]). These semantics define leakage equivalent to the $[\![\cdot]\!]_{\text{mem}}$ leakage model; it remains an open problem to explore how to define $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$ leakage in this more structured execution model — in particular, what it means for such a semantics to allow an attacker to observe control-flow.

Open problems: Leakage models for language-based isolation

As with most work on Spectre foundations, we focus on cryptography and software-based isolation. Spectre, though, can be used to break most other software abstractions as well — from module systems [84] and object capabilities [139] to language-based isolation techniques like information flow control [186]. How do we adopt these abstractions in the presence of speculative execution? What formal security property should we prove? And what leakage model should be used?

4.2.2 Non-interference and policies

After the leakage model, we must determine what *secrecy policy* we consider for our attacker model — i.e., which values can and cannot be leaked. Domains such as cryptography and isolation already have defined policies for sequential security properties: For cryptography, memory that contains secret data (e.g., encryption keys) is considered sensitive; isolation simply declares that all memory outside the program's assigned sandbox region should not be leaked.

The straightforward extension of sequential non-interference to speculative execution is to enforce the same leakage model (e.g., $[\![\cdot]\!]_{\text{ct}}$) with the same security policy — no secrets should be leaked whether in normal or speculative execution. We refer to this simple extension as a *direct* non-interference property, or *direct NI*.

Definition 1 (Direct non-interference)**.** Program p satisfies *direct non-interference* with respect to a given contract $[\![\cdot]\!]$ and policy π if, for all pairs of π-equivalent initial states σ and σ', executing p with each initial state produces the same trace. That is, $p \vdash NI(\pi, [\![\cdot]\!])$ is defined as

$$\forall \sigma, \sigma' : \sigma \simeq_\pi \sigma' \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma').$$

We elide writing π for brevity — e.g., $NI([\![\cdot]\!]_{\text{ct}}^{\text{pht}})$ expresses constant-time security under Spectre-PHT semantics.

Alternatively, we may instead want to assert that the speculative trace of a program has no *new* sensitive leaks as compared to its sequential trace. This is a useful property for compilers and mitigation tools that may not know the secrecy policy of an input program, but want to ensure the resulting program does not leak any additional information. We term this a *relative* non-interference property, or *relative NI*; a program that satisfies relative NI is no less secure than its sequential execution.

Definition 2 (Relative non-interference)**.** Program p satisfies *relative non-interference* from contract $[\![\cdot]\!]_{\mathrm{a}}^{\mathrm{seq}}$ to $[\![\cdot]\!]_{\mathrm{b}}^{\beta}$ and with policy π if: For all pairs of π-equivalent initial states σ and σ', if executing p under $[\![\cdot]\!]_{\mathrm{a}}^{\mathrm{seq}}$ produces equal traces, then executing p under $[\![\cdot]\!]_{\mathrm{b}}^{\beta}$ produces equal traces. That is, $p \vdash NI(\pi, [\![\cdot]\!]_{\mathrm{a}}^{\mathrm{seq}} \Rightarrow [\![\cdot]\!]_{\mathrm{b}}^{\beta})$ is defined as

$$\forall \sigma, \sigma' : \sigma \simeq_{\pi} \sigma' \wedge [\![p]\!]_{\mathrm{a}}^{\mathrm{seq}}(\sigma) = [\![p]\!]_{\mathrm{a}}^{\mathrm{seq}}(\sigma') \\ \implies [\![p]\!]_{\mathrm{b}}^{\beta}(\sigma) = [\![p]\!]_{\mathrm{b}}^{\beta}(\sigma').$$

For non-terminating programs, we can compare finite prefixes of $[\![p]\!]^{\beta}$ against their sequential projections to $[\![p]\!]^{\mathrm{seq}}$ — since speculative execution must preserve sequential semantics, there will always be a valid sequential projection. As before, we may elide π for brevity.

Interestingly, any relative non-interference property $NI(\pi, [\![\cdot]\!]_{\mathrm{a}}^{\mathrm{seq}} \Rightarrow [\![\cdot]\!]_{\mathrm{b}}^{\beta})$ for a program p can be expressed equivalently as a direct property $NI(\pi', [\![\cdot]\!]_{\mathrm{b}}^{\beta})$, where $\pi' = \pi \setminus canLeak(p, [\![\cdot]\!]_{\mathrm{a}}^{\mathrm{seq}})$. That is, we treat anything that could possibly leak under contract $[\![\cdot]\!]_{\mathrm{a}}^{\mathrm{seq}}$ as public. Relative NI is thus a (semantically) weaker property than direct NI, as it implicitly declassifies anything that might leak during sequential execution.

However, relative NI is still a stronger property than a conventional implication. For example, the property $NI([\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{seq}}) \Rightarrow NI([\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{pht}})$ makes no guarantees at all about a program that is not sequentially constant-time. Conversely, the relative NI property $NI([\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{seq}} \Rightarrow [\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{pht}})$ guarantees that even if a program is not sequentially constant-time, the sensitive information an

attacker can learn during the program's speculative execution is limited to what it already might leak sequentially.

In Table 4.2, we classify security properties of different works by which direct or relative NI properties they verify or enforce. We find that tools focused on verifying cryptography or memory isolation verify direct NI properties, whereas frameworks concerned with compilation or inserting Spectre mitigations for general programs tend towards relative NI.

Verifying programs

Direct NI unconditionally guarantees that sensitive data is not leaked, whether executing sequentially or speculatively. This makes it ideal for domains that already have clear policies about what data is sensitive, such as cryptography (e.g., secret keys) or software isolation (e.g., memory outside the sandbox). Indeed, tools that target cryptographic applications ([18, 36, 48, 211]) all verify that programs satisfy the direct *speculative constant-time* (SCT) property.

Additionally, we find that current tools that verify relative NI [39, 80] are indeed capable of verifying direct NI, but intentionally add constraints to their respective checkers to "remove" sequential leaks from their speculative traces. Although this is just as precise, it is an open problem whether tools can verify relative NI for programs without relying on a direct NI analysis.

Verifying compilers

On the other hand, compilers and mitigation tools are better suited to verify or enforce relative NI properties: The compiler guarantees that its output program contains no new leakages as compared to its input program. This way, developers can reason about their programs assuming a sequential model, and the compiler will mitigate any speculative effects. For instance, if a program p is already sequentially constant-time $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}})$, then a compiler that enforces $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$ will compile p to a program that is *speculatively* constant-time $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$. Similarly, if a program is properly sandboxed under sequential execution $NI(\llbracket \cdot \rrbracket_{\text{arch}}^{\text{seq}})$ and is compiled with a compiler that introduces no new *arch* leakage, the resulting program will remain sandboxed even under speculative execution [81].

Table 4.2: Speculative security properties in prior works and their equivalent non-interference statements. We write $\approx NI(\cdots)$ for unsound approximations of non-interference properties.

[1] [218] tracks taint of *attacker influence* rather than value sensitivity. [2] These works all derive their property from the definition given in [36] and share the same property name despite differences in execution mode. [3] The analysis tool of [36], Pitchfork, only verifies the weaker property $NI(\llbracket\cdot\rrbracket^{\text{pht-stl}}_{\text{ct}})$. [4] The definitions of SNI and wSNI are parameterized over the target leakage model. [5] The definition of wSNI in [81] does not require that the initial states be π-equivalent.

Property or tool name	Non-interference prop.	Precision
Mcilroy et al. [147]	$\approx NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper
oo7 [218] $\Phi_{spectre}$ $\Phi^{weak}_{spectre}$, $\Phi^{v1.1}_{spectre}$	$\approx NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{mem}})$ $\approx NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{arch}})$	taint[1]
Cache analysis [83, 228] [217]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{cache}})$	hyper taint
Weak memory modeling [42, 56] [175]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{mem}})$ $NI(\llbracket\cdot\rrbracket^{\text{pht-stl}}_{\text{mem}})$	hyper
Speculative constant-time (SCT)[2] [211]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	taint
Speculative constant-time (SCT)[2] [18, 48]	$NI(\llbracket\cdot\rrbracket^{\text{pht-stl}}_{\text{ct}})$	hyper
Speculative constant-time (SCT)[2] [36]	$NI(\llbracket\cdot\rrbracket^{\text{pbrs}}_{\text{ct}})$[3]	hyper, taint
Speculative non-interference (SNI) [80, 81]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\underline{\ }})$[4]	hyper
Robust speculative non-interference (RSNI) [173] Robust speculative safety (RSS) [173]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper taint
Conditional noninterference [78]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pbrs}}_{\text{ct}})$	hyper
Weak speculative non-interference (wSNI) [81]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\underline{\ }})$[4,5]	hyper
Weak robust speculative non-interference (RSNI^{-}) [173] Trace property-dependent observational determinism (TPOD) [39] Weak robust speculative safety (RSS^{-}) [173]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper hyper taint

Execution models (Section 4.2.3)	
$\llbracket\cdot\rrbracket^{\text{seq}}$	Sequential execution
$\llbracket\cdot\rrbracket^{\text{pht}}$	Captures Spectre-PHT
$\llbracket\cdot\rrbracket^{\text{pht-stl}}$	Captures Spectre-PHT/-STL
$\llbracket\cdot\rrbracket^{\text{pbrs}}$	Captures Spectre-PHT/-BTB/-RSB/-STL

Precision of the defined security property	
hyper	Non-interference hyperproperty, requires two π-equivalent executions
taint	Sound approximation using taint tracking, requires only one execution

Similarly, Patrignani and Guarnieri [173] explore whether compilers preserve *robust* non-interference properties. A security property is *robust* if a program remains secure even when linked against adversarial code (i.e., if the program is called with arbitrary or adversarial inputs). A compiler *preserves* a non-interference property if, after compilation from a source to a target language, the property still holds. In Patrignani and Guarnieri's framework, the source language describes sequential execution while the target language has speculative semantics, making their notion of compiler preservation very similar to enforcing relative NI.

4.2.3 Execution models

To reason about Spectre attacks, a semantics must be able to reason about the leakage of sensitive data in a speculative *execution model*. A speculative execution model is what differentiates a speculative semantics from standard sequential analysis, and determines what speculation the abstract processor can perform. For developers, choosing a proper execution model is a tradeoff: On the one hand, the choice of behaviors their model allows — i.e., which microarchitectural predictors they include — determines which Spectre variants their tools can capture. On the other hand, considering additional kinds of mispredictions inevitably makes their analysis more complex.

Spectre variants and predictors

Most semantics and tools in Table 4.1 only consider the conditional branch predictor, and thus only Spectre-PHT attacks. (Mis)predictions from the conditional branch predictor are constrained — there are only two possible choices for every decision — so the analysis remains fairly tractable. Jasmin [18], Binsec/Haunted [48], Pitchfork [36], and Kaibyo [175] all additionally model *store-to-load* (STL) predictions, where a processor forwards data to a memory load from a prior store to the same address. If there are multiple pending stores to that address, the processor may choose the wrong store to forward the data — this is the root of a Spectre-STL attack. STL predictions are less constrained than predictions from the conditional

branch predictor: In the absence of additional constraints, they allow for a load to draw data from any prior store to the same address.

Other control-flow mechanisms are significantly more complex: Return instructions and indirect jumps can be *speculatively hijacked* to send execution to arbitrary (attacker-controlled) points in the program.[2] An attacker can trivially hijack a victim program if they can control (mis)prediction of the RSB (for returns) [125, 140] or BTB (for indirect jumps) [123]. Even without this ability, an attacker can hijack control-flow if they speculatively overwrite the target address of a return or jump (e.g., by exploiting a prior PHT misprediction) [119, 142, 200]. Formally, these attacks still fit within our non-interference framework — if a program can be arbitrarily hijacked, then it will be unable to satisfy any non-interference property. However, to formally verify that this is the case, a semantics must model these behaviors.

Although capturing all speculative behaviors in a semantics is possible, the resulting analysis is neither practical nor useful; in practice, developers need to make tradeoffs. For example, the semantics proposed by Cauligi et al. [36] can simulate all of the aforementioned speculative attacks, but their analysis tool Pitchfork only detects PHT- and STL-based vulnerabilities. On the other hand, tools like oo7 (with the "v1.1" pattern) [218] and SpecTaint [180] conservatively assume that writes to transient addresses can overwrite *anything*, and thus immediately flag this behavior as vulnerable.

The InSpectre semantics [78] proceeds in the opposite direction — it allows the processor to predict arbitrary values, even the values of constants. InSpectre also allows more out-of-order behavior than most other semantics (see Section 4.2.6) — in particular, it allows the processor to commit writes to memory out-of-order. As a result, InSpectre is very expressive: It is capable of describing a wide variety of Spectre variants both known and unrealized. But, as a result, InSpectre cannot feasibly be used to verify programs; instead, the authors pose InSpectre as a framework for reasoning about and analyzing microarchitectural features themselves.

[2] Including, on x86-family processors, into the *middle* of an instruction [21].

Speculation windows

Several semantics and tools in Table 4.1 limit speculative execution by way of a *speculation window*. This models how hardware has finite resources for speculation, and can only speculate through a certain number of instructions or branches at a time.

Explicitly modeling a speculation window serves two purposes for detection tools. One, it reduces false positives: a mispredicted branch will not lead to a speculative leak thousands of instructions later. Two, it bounds the complexity of the semantics and thus the analysis. Since the abstract processor can only speculate up to a certain depth, an analysis tool need only consider the latest window of instructions under speculative execution. Some semantics refine this idea even further: Binsec/Haunted [48], for example, uses different speculation windows for load-store forwarding than it uses for branch speculation.

Speculation windows are also valuable for mitigation tools: although tools like Blade [211] and Jasmin [18] are able to prove security without reasoning about speculation windows, modeling a speculation window reduces the number of fences (or other mitigations) these tools need to insert, improving the performance of the compiled code.

Eliminating variants

Instead of modeling all speculative behaviors, compilers and mitigation tools can use clever techniques to sidestep particularly problematic Spectre variants. For example, even though Jasmin [18] does not model the RSB, Jasmin programs do not suffer from Spectre-RSB attacks: The Jasmin compiler inlines all functions, so there are no returns to mispredict. Mitigation tools can also disable certain classes of speculation with hardware flags [99]. After eliminating complex or otherwise troublesome speculative behavior, a tool need only consider those that remain.

Cross-address-space attacks

Previous systematizations of Spectre attacks [32] differentiate between *same-address-space* and *cross-address-space* attacks. Same-address-space attacks rely on repeatedly causing the victim code to execute in order to train a microarchitectural predictor. Cross-address-space

attacks are more powerful, as they allow an attacker to perform the training step on a branch within the attacker's own code.

Most of the semantics and tools in Table 4.1 make no distinction between same-address-space and cross-address-space attacks, as they ignore the mechanics of training and consider all predictions to be potentially malicious. A notable exception is oo7 [218], which explicitly tracks *attacker influence*. Specifically, oo7 only considers mispredictions for conditional branches which can be influenced by attacker input. Thus, oo7 effectively models only same-address-space attacks. Unfortunately, as a result, oo7 misses Spectre vulnerabilities in real code, as demonstrated by Wang et al. [217].

4.2.4 Nondeterminism

Speculative execution is inherently *nondeterministic*: Any given branch in a program may proceed either correctly or incorrectly, regardless of the actual condition value. More generally, speculative hijack attacks can send execution to entirely indeterminate locations. All of the semantics in Table 4.1 allow these nondeterministic choices to be actively adversarial — for instance, given by attacker-specified directives [36, 211] or by consulting an abstract oracle [39, 80, 81, 147]. These semantics all (conservatively) assume that the attacker has full control of microarchitectural prediction and scheduling; we explore the different techniques they use to verify or enforce security in the face of adversarial nondeterminism.

Exploring nondeterminism

Several Spectre analysis tools are built on some form of abstract execution: They simulate speculative execution of the program by tracking ranges or properties of different values. By checking these properties throughout the program, these tools determine if sensitive data can be leaked. Standard tools for (non-speculative) abstract execution are designed only to consider concrete execution paths; they must be adapted to handle the many possible nondeterministic execution paths from speculation. SpecuSym [83], KLEESpectre [217], and AISE [228] han-

dle this nondeterminism by following an *always-mispredict* strategy. When they encounter a conditional branch, they first explore the execution path which mispredicts this branch, up to a given speculation depth. Then, when they exhaust this path, they return to the correct branch. This technique, though, only handles the conditional branch predictor; i.e., Spectre-PHT attacks. Pitchfork [36] and Binsec/Haunted [48] adapt the *always-mispredict* strategy to account for out-of-order execution and Spectre-STL. Although it may not be immediately clear that *always-mispredict* strategies are sufficient to prove security — especially when the attacker can make any number of antagonistic choices — these strategies do indeed form a sound analysis [36, 48, 80].

Unfortunately, simulating execution only works for semantics where nondeterminism is relatively constrained: Conditional branches are a simple boolean choice, and store-to-load predictions are limited by the speculation window. If we pursue other Spectre variants, we will quickly become overwhelmed — again, an unconstrained hijack gadget can redirect control to almost anywhere in a program. The *always-mispredict* strategy here is nonsensical at best; abstract execution is thus necessarily limited in what it can soundly explore.

Abstracting out nondeterminism

Mitigation tools have more flexibility dealing with nondeterminism: Tools like Blade [211] and oo7 [218] apply dataflow analysis to determine which values may be leaked along *any* path, instead of reasoning about each path individually. Then, these tools insert speculation barriers to preemptively block potential leaks of sensitive data. This style of analysis comes at the cost of some precision: Blade, for example, conservatively treats *all* memory accesses as if they may speculatively load sensitive values, as its analysis cannot reason about the contents of memory. Similarly, oo7's "v1.1" pattern detection conservatively flags all (attacker-controlled) transient *stores*, as they may lead to speculative hijack. However, Blade and oo7 — and mitigation tools in general — can afford to be less precise than verification or detection tools; these, conversely, must maintain higher precision to avoid floods of false positives.

Restricting nondeterminism

Compilers such as Swivel [162], Venkman [196], and ELFbac [111] restructure programs entirely, imposing their own restricted set of speculative behavior at the software layer. ELFbac allocates sensitive data in separate memory regions and uses page permission bits to disallow untrusted code from accessing these regions — regardless of how a program may misspeculate, it will not be able to read (and thus cannot leak) sensitive data. Swivel and Venkman compile code into carefully aligned blocks so that control flow always land at the tops of protected code blocks, even speculatively; Swivel accomplishes this by clearing the BTB state after untrusted execution, while Venkman recompiles all programs on the system to mask addresses before jumping. Both systems also enforce speculative control-flow integrity (CFI) checks to prevent speculative hijacking, whether by relying on hardware features [103] or by implementing custom CFI checks with branchless assembly instructions. Developers that use these compilers can then reason about their programs much more simply, as the set of speculative behaviors is restricted enough to make the analysis tractable. Of the techniques discussed in this section, this line of work seems the most promising: It produces mitigation tools with strong security guarantees, without relying on an abundance of speculation barriers (as often results from dataflow analysis) or resorting to heavyweight simulation (e.g., symbolic execution).

Open problems: Rigorous performance comparison

To the best of our knowledge, no work has rigorously compared the performance of all the tools in Table 4.1. Perhaps the most complete comparison is by Daniel et al. [48], who compare the detection tools KLEESpectre, Pitchfork, and Binsec/Haunted in terms of the analysis time required to detect known violations in a few chosen targets. A general and objective performance comparison is difficult, if not impossible: The tools in Table 4.1 operate on different types of programs (general-purpose, cryptographic, sandboxing) and different languages (x86, LLVM, WebAssembly). They also provide different security guarantees, as we discuss above. An intermediate step towards an expanded performance comparison, which would be a valuable contribution on its own, would be to develop a larger corpus of known attacks on realistic (medium-

to-large-size) programs. This corpus would help evaluate both the security and performance of existing or newly-proposed tools.

4.2.5 Higher-level abstractions

Spectre attacks — and speculative execution — fundamentally break our intuitive assumptions about how programs should execute. Higher-level guarantees about programs no longer apply: Type systems or module systems are meaningless when even basic control flow can go awry. In order to rebuild higher-level security guarantees, we first need to repair our model of how programs execute, starting from low-level semantics. Once these foundations are firmly in place, only then can we rebuild higher-level abstractions.

Semantics for assembly or IRs

The majority of formal semantics in Table 4.1 operate on abstract assembly-like languages, with commands that map to simple architectural instructions. Semantics at this level implement control flow directly in terms of jumps to *program points* — usually indices into memory or an array of program instructions — and treat memory as largely unstructured. Since these low-level semantics closely correspond to the behavior of real hardware, they capture speculative behaviors in a straightforward manner, and provide a foundational model for higher-level reasoning. Similarly, many concrete analysis tools for constant-time or Spectre operate directly on binaries or compiler intermediate representations (IRs) [36, 48, 49, 80, 217]. These tools operate at this lowest level so that their analysis will be valid for the program unaltered — compiler optimizations for higher-level languages can end up transforming programs in insecure ways [19, 48, 49]. As a result however, these tools necessarily lose access to higher-level information such as control flow structure or how variables are mapped in memory.

Semantics for structured languages

The semantics proposed by Jasmin [18], Patrignani and Guarnieri [173], and Blade [211] build on top of these lower-level ideas to describe what we term "medium-level" languages —

those with structured control flow and memory, e.g., explicit loops and arrays. For these medium-level semantics, it is less straightforward to express speculative behavior: For instance, instead of modeling speculation directly, Vassena et al. [211] first translate programs in their source language to lower-level commands, then apply speculative execution at that lower level.

In exchange, the structure in a medium-level semantics lends itself well to program analysis. For example, Vassena et al. are able to use a simple type system to prove security properties about a program. Barthe et al. [18] also take advantage of structured semantics: They prove that if a sequentially constant-time program is *speculatively (memory) safe* — i.e., all memory operations are in-bounds array accesses — then the program is also speculatively constant-time. Since their source semantics only accesses memory through array operations, they can statically verify whether a program is speculatively safe — and thus speculatively secure. An interesting question for future work is whether their concept of speculative (memory) safety combines with other sequential security properties to give corresponding guarantees, such as for sandboxing, information flow, or rich type systems.

Weak-memory-style semantics

Weak-memory-style semantics present a fundamentally different approach, lifting the concept of speculative execution directly to a higher level. As these models are abstracted away from microarchitectural details, they are well-suited for analyzing Spectre variants in terms of data flow: Indeed, both Colvin and Winter [42] and Disselkoen et al. [56] treat Spectre-PHT as a constrained form of instruction reordering, while Ponce de León and Kinder [175] analyze dependency relations between instructions.

However, it remains challenging to translate a flexible semantics of this style into a concrete analysis tool: Of the three works discussed here, only Ponce de León and Kinder present a tool which can automatically perform a security analysis of a target program,[3] though even they admit that it is slower than comparable tools based on operational semantics. That said, this

[3]Colvin and Winter do present a tool, but it is only used to mechanically explore manually translated programs.

high-level approach to speculative semantics is certainly underexplored compared to the larger body of work on operational semantics, and is worthy of further investigation.

Compiler mitigations

With adequate foundations in place, one avenue to regaining higher-level abstractions is to modify compilers of higher-level languages to produce speculatively secure low-level programs. Many compilers already include options to conservatively insert speculation barriers or hardening into programs, which (when done properly) provides strong security guarantees. Although some such hardening passes have been verified [173], they are overly conservative and incur a significant performance cost. Other compiler mitigations been shown unsound [167] — or worse, even introduce new Spectre vulnerabilities [48] — further reinforcing that these techniques must be grounded in a formal semantics.

Open problems: Formalization of new compilation techniques

Swivel [162], Venkman [196], and ELFbac [111] show how the structure of code itself can provide security guarantees at a reduced performance cost. For instance, both Venkman and Swivel demonstrate that organizing instructions into *bundles* or *linear blocks* (respectively) can mitigate speculative hijacks, making these transient attacks tractable to analyze and prevent. However, none of these compiler-based approaches are yet grounded in a formal semantics. Formalizing these systems would increase our confidence in the strong guarantees they claim to provide.

Open problems: New languages

Another promising approach is to design new languages which are inherently safe from Spectre attacks. Prior work has produced secure languages like FaCT [37], which is (sequentially) constant-time by construction. An extension of FaCT, or a new language built on its ideas, could prevent Spectre attacks as well. Vassena et al. [211] have already taken a first step in this direction: They construct a simple *while*-language which is guaranteed safe from Spectre-PHT attacks when compiled with their fence insertion algorithm. It would be valuable to extend this further, both to

more realistic (higher-level) languages, and to more Spectre variants. The key question is whether dedicated language support can provide a path to secure code that outperforms the de-facto approach — that is, compiling standard C code and inserting Spectre mitigations.

4.2.6 Expressivity and microarchitectural features

One theme of this chapter is that a good (practical) semantics needs to have an appropriate amount of *expressivity*: On one hand, we want a semantics which is expressive — able to model a wide range of possible behaviors (e.g., Spectre variants). This allows us to model powerful attackers. On the other hand, a semantics which allows too many possible behaviors makes many analyses intractable. Indeed, a fundamental purpose of semantics is to provide a reasonable abstraction or simplification of hardware to ease analysis; a semantics which is too expressive simply punts this problem to the analysis writer. Thus, choosing how much expressivity to include in a semantics represents an interesting tradeoff.

By far the most important choice for the expressivity of a semantics is which misprediction behaviors to allow — i.e., which Spectre variants to reason about (discussed in Section 4.2.3). But beyond speculative execution itself, there are many other microarchitectural features which are relevant for a security analysis, and which have been — or could be — modeled in a speculative semantics. These features also affect the expressivity of the semantics, which means that choosing whether to include them results in similar tradeoffs.

Out-of-order execution

Many speculative semantics simulate a processor feature called *out-of-order execution*: They allow instructions to be executed in any order, as long as those instructions' dependencies (operands) are ready. Out-of-order execution is mostly orthogonal to speculative execution; in fact, out-of-order execution is not required to model Spectre-PHT, -BTB, or -RSB — speculative execution alone is sufficient. However, out-of-order execution is included in most modern

processors, and for that reason,[4] many speculative semantics also model it. Modeling out-of-order execution may provide an easier or more elegant way to express a variety of Spectre attacks, as opposed to modeling speculative execution alone. Furthermore, Disselkoen et al. [56] and Guanciale et al. [78] demonstrate how to abuse out-of-order execution to conduct (at least theoretical) novel side-channel attacks.[5]

Although modeling out-of-order execution might make a semantics simpler, the additional expressivity makes the resulting analysis more complex. Fully modeling out-of-order execution leads to an explosion in the number of possible executions of a program; naively incorporating out-of-order execution into a detection or mitigation tool results in an intractable analysis. Indeed, while Guarnieri et al. [81] and Colvin and Winter [42] present analysis tools based on their respective out-of-order semantics, they only analyze very simple Spectre gadgets and not code used in real programs. Instead, for analysis tools based on out-of-order semantics to scale to real programs, developers need to use lemmas to reduce the number of possibilities the analysis needs to consider. As one example, Pitchfork [36] operates on a set of "worst-case schedules" which represent a small subset of all possible out-of-order schedules — the developers formally show that this reduction does not affect the soundness of Pitchfork's analysis.

Caches and TLBs

Some speculative semantics and tools [83, 147, 217, 228] include abstract models of caches, tracking which addresses may be in the cache at a given time. One could imagine also including detailed models of TLBs. As discussed in Section 4.2.1, modeling caches or TLBs is probably not helpful, at least for mitigation or verification tools — not only does it make the semantics more complicated, but it potentially leads to non-portable guarantees. In particular, including a model of the cache usually leads to the $[\![\cdot]\!]_{\text{cache}}$ leakage model, rather than the $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$ leakage models which provide stronger defensive guarantees. Following in the tradition

[4]Or perhaps, because out-of-order execution is often discussed alongside (or even confused with) speculative execution.

[5]Disselkoen et al. [56] propose to abuse compile-time instruction reordering, which is different from microarchitectural out-of-order execution, but related.

of constant-time programming in the non-speculative world, it seems wiser for our analyses and mitigations to be based on microarchitecture-agnostic principles as much as possible, and not depend on details of the cache or TLB structure.

Other leakage channels

There are a variety of specific microarchitectural mechanisms which can result in leakages beyond the ones we directly focus on in this chapter. For instance, in the presence of multithreading, port contention in the processor's execution units can reveal sensitive information [22]; and many processor instructions, e.g., floating-point or SIMD instructions, can reveal information about their operands through timing side channels [9]. Most existing semantics do not model these specific effects. However, the commonly-used $[\![\cdot]\!]_{ct}$ and $[\![\cdot]\!]_{arch}$ leakage models are already strong enough to capture leakages from most of these sources: For instance, port contention can only reveal sensitive data if the sensitive data influenced which instructions are being executed — and the $[\![\cdot]\!]_{ct}$ leakage model already considers the sensitive data to be leaked once it influences control flow. For variable-time instructions, most definitions of $[\![\cdot]\!]_{ct}$ do not capture this leakage — but extending those definitions is straightforward [6]. In both of these examples, the $[\![\cdot]\!]_{arch}$ leakage model captures all leaks, as it (even more conservatively) already considers the sensitive data as leaked once it reaches a register — long before the data can influence control-flow or be used in an instruction. Although modeling any of these effects more precisely can increase the precision with which an analysis detects potential vulnerabilities, the tradeoff in analysis complexity is probably not worth it, and for mitigation and verification tools, the $[\![\cdot]\!]_{ct}$ and $[\![\cdot]\!]_{arch}$ leakage models provide stronger and more generalizable guarantees.

In a similar vein, most semantics and tools do not explicitly model parallelism or concurrency: They reason only about single-threaded programs and processors. Instead, they abstract away these details by giving attackers broad powers in their models — e.g., complete power over all microarchitectural predictions, and the capability to observe the full cache state after every execution step. The notable exceptions are the weak-memory-style semantics [42, 56, 175] —

multiple threads are an inherent feature for this style, making them a promising vehicle for further exploring the interaction between speculation and concurrency.

Open problems: Process isolation

In practice, a common response to Spectre attacks has been to move all secret data into a separate process — e.g., Chrome isolates different *sites* in separate processes [184]. This shifts the burden from application and runtime system engineers to OS engineers. Developing Spectre foundations to model the process abstraction will elucidate the security guarantees of such systems. This is especially useful, as the process boundary does not keep an attacker from performing out-of-place training of the conditional branch predictor, nor from leaking secrets via the cache state [32].

4.3 Related work

Both in industry and in academia, there has been a lot of interest in Spectre and other transient execution attacks. We discuss other systematization papers that address Spectre attacks and defenses, and we briefly survey related work which otherwise falls outside the scope of this chapter.

4.3.1 Systematization of Spectre attacks and defenses

Canella et al. [32] present a comprehensive systematization and analysis of Spectre and Meltdown attacks and defenses. They first classify transient execution attacks by whether they are a result of misprediction (Spectre) or an execution fault (Meltdown); and further classify the attacks by their root microarchitectural cause, yielding the nomenclature we use in this chapter (e.g., Spectre-PHT is named for the *Pattern History Table*). They then categorize previously known Spectre attacks, revealing several new variants and exploitation techniques. Canella et al. also propose a sequence of "phases" for a successful Spectre or Meltdown attack, and group

published defenses by the phase they target. A followup survey by Canella et al. [31] expands on the idea of attack phases, categorizing both hardware and software Spectre defenses according to which attack phase they prevent: Preparation, misspeculation, data access, data encoding, leakage, or decoding. Separately, Xiong et al. [230] also survey transient execution attacks, with a specific focus on the mechanics of exploits for these attacks. In contrast, our systematization focuses on the formal semantics behind Spectre analysis and mitigation tools rather than the specifics of attack variants or types of defenses.

4.3.2 Hardware-based Spectre defenses

In this chapter, we focus only on software-based techniques for existing hardware. The research community has also proposed several hardware-based Spectre defenses based on cache partitioning [118], cleaning up the cache state after misprediction [187], or making the cache invisible to speculation by incorporating some separate internal state [2, 116, 231]. Unfortunately, attackers can still use side channels other than the cache to exploit speculative execution [22, 190]. NDA [227], DOLMA [137], and Speculative Taint Tracking (STT) [237] block additional speculative covert channels by analyzing and classifying instructions that can leak information.

Fadiheh et al. [62] define a property for hardware execution that they term UPEC: A hardware that satisfies UPEC will not leak speculatively anything more than it would leak sequentially. In other words, UPEC is equivalent to the relative non-interference property $NI(\pi, [\![\cdot]\!]^{seq}_{arch} \Rightarrow [\![\cdot]\!]^{pht}_{arch})$.

The insights and recommendations from our work can guide future hardware mitigations; properties like $[\![\cdot]\!]_{ct}$ or $[\![\cdot]\!]_{arch}$ can serve as contracts of what software expects from hardware [81].

4.3.3 Software-hardware co-design

Although hardware-only approaches are promising for future designs, they require significant modifications and introduce non-negligible performance overhead for all workloads. Several works instead propose a software-hardware co-design approach. Taram et al. [202] propose context-sensitive fencing, making various speculative barriers available to software. Li et al. [131] propose memory instructions with a conditional speculation flag. Context [188] and SpectreGuard [65] allow software to mark secrets in memory. This information is propagated through the microarchitecture to block speculative access to the marked regions. SpecCFI [126] suggests a hardware extension similar to Intel CET [103] that provides target label instructions with speculative guarantees. Finally, several recent proposals allow partitioning branch predictors based on context provided by the software [214, 242]. As these approaches require both software and hardware changes, we should develop a formal semantics to apply them correctly.

4.3.4 Other transient execution attacks

We focus exclusively on Spectre, as other transient execution attacks are better addressed in hardware. For completeness, we briefly discuss these other attacks.

Meltdown variants

The Meltdown attack [133] bypasses implicit memory permission checks within the CPU during transient execution. Unlike Spectre, Meltdown does not rely on executing instructions in the victim domain, so it cannot be mitigated purely by changes to the victim's code. Foreshadow [206] and microarchitectural data sampling (MDS) [30, 100] demonstrate that transient faults and microcode assists can still leak data from other security domains, even on CPUs that are resistant to Meltdown. Researchers have extensively evaluated these Meltdown-style attacks leading to new vulnerabilities [153, 154, 189], but most recent Intel CPUs have hardware-level

mitigations for all these vulnerabilities in the form of microcode patches or proprietary hardware fixes [102].

Load value injection

Load value injection (LVI) [207] exploits the same root cause as Meltdown, Foreshadow, and MDS, but reverses these attacks: The attacker induces the transient fault into the victim domain instead of crafting arbitrary gadgets in their own code space. This inverse effect is subject to an exploitation technique similar to Spectre-BTB for transiently hijacking control flow. Although there are software-based mitigations proposed against LVI [101, 207], Intel only suggests applying them to legacy enclave software. Like Meltdown, LVI does not need software-based mitigation on recent Intel CPUs.

4.4 Conclusion

Spectre attacks break the abstractions afforded to us by conventional execution models, fundamentally changing how we must reason about security. We systematize the community's work towards rebuilding foundations for formal analysis atop the loose earth of speculative execution, evaluating current efforts in a shared formal framework and pointing out open areas for future work in this field.

We find that, as with previous work in the sequential domain, solid foundations for speculative analyses require proper choices for semantics and attacker models. Most importantly, developers must consider leakage models no weaker than $[\![\cdot]\!]_{\text{arch}}$ or $[\![\cdot]\!]_{\text{ct}}$. Weaker models—those that only capture leaks via memory or the data cache—lead to weaker security guarantees with no clear benefit. Next, though many frameworks focus on Spectre-PHT, sound tools must consider all Spectre variants. Although this increases the complexity of analysis, developers can combine analyses with structured compilation techniques to restrict or remove entire categories of Spectre attacks by construction. Finally, we recommend *against* modeling unnecessary

(micro)architectural details in favor of the simpler $[\![\cdot]\!]_{\text{arch}}$ and $[\![\cdot]\!]_{\text{ct}}$ models; details like cache structures or port contention introduce complexity and reduce portability.

When properly rooted in formal guarantees, software Spectre defenses provide a firm foundation on which to rebuild secure systems. We intend this systematization to serve as a reference and guide for those seeking to build or employ formal frameworks and to develop sound Spectre defenses with strong, precise security guarantees.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. We thank Matthew Kolosick for helping us understand some of the formal systems discussed and in organizing the paper. This work was supported in part by gifts from Intel and Google; by the NSF under Grant Numbers CNS-2120642, CCF-1918573 and CAREER CNS-2048262; by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and, by the Office of Naval Research (ONR) under project N00014-15-1-2750.

Chapter 4, in part, is a reprint of the material as it appears in the 43rd IEEE Symposium on Security and Privacy (S&P '22). Cauligi, Sunjay; Disselkoen, Craig; Moghimi, Daniel; Barthe, Gilles; Stefan, Deian. IEEE, 2022. The dissertation author was a primary investigator and author of this material.

Chapter 5

Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade

Implementing secure cryptographic algorithms is hard. The code must not only be functionally correct, memory safe, and efficient, it must also avoid divulging secrets indirectly through side channels like control-flow, memory-access patterns, or execution time. Consequently, much recent work focuses on how to ensure implementations do not leak secrets, e.g., via type systems [37, 223], verification [6, 179], and program transformations [17].

Unfortunately, these efforts can be foiled by speculative execution. Even if secrets are closely controlled via guards and access checks, the processor can simply ignore these checks when executing speculatively. This, in turn, can be exploited by an attacker to leak protected secrets.

In principle, memory fences block speculation, and hence, offer a way to recover the original security guarantees. In practice, however, fences pose a dilemma. Programmers can restore security by conservatively inserting fences after every load (e.g., using Microsoft's Visual Studio compiler pass [171]), but at huge performance costs. Alternatively, they can rely on heuristic approaches for inserting fences [218], but forgo guarantees about the absence of side-

channels. Since missing even one fence can allow an attacker to leak any secret from the address space, secure runtime systems — in particular, browsers like Chrome and Firefox — take yet another approach and isolate secrets from untrusted code in different processes to avoid the risk altogether [24, 159]. Unfortunately, the engineering effort of such a multi-process redesign is huge — e.g., Chrome's redesign took five years and roughly 450K lines of code changes [184].

In this chapter, we introduce BLADE, a new, fully automatic approach to provably and efficiently eliminate speculation-based leakage from constant-time cryptographic code. BLADE is based on the key insight that to prevent leaking data via speculative execution, it is not necessary to stop *all* speculation. Instead, it suffices to *cut* the data flow from expressions (sources) that could speculatively introduce secrets to those that leak them through the cache (sinks). We develop this insight into an automatic enforcement algorithm via four contributions.

***1. A new primitive* protect**

We propose an abstract primitive called **protect** that embodies several hardware mechanisms proposed in recent work [202, 237]. Crucially, and in contrast to a regular fence which stops *all* speculation, **protect** only stops speculation for a given *variable*. For example $x := \textbf{protect}(e)$ ensures that the value of e is assigned to x only *after* e has been assigned its *stable*, non-speculative value. Though we encourage hardware vendors to implement **protect** in future processors, for backwards compatibility, we implement and evaluate two versions of **protect** on existing hardware — one using fences, another using *speculative load hardening* (SLH) [35].

2. A type system for speculation

Our second contribution is an approach to conservatively approximating the dynamic semantics of speculation via a *static type sytem* that types each expression as either transient (**T**), i.e., expressions that *may* contain speculative secrets, or stable (**S**), i.e., those that cannot. Our system prohibits speculative leaks by requiring that all *sink* expressions that can influence intrinsic attacker visible behavior (e.g., cache addresses) are typed as stable. The type system does *not* rely on user-provided security annotations to identify sensitive sources and public sinks. Instead,

we *conservatively* reject programs that exhibit any flow of information from transient sources to stable sinks. This, in turn, allows us to automatically identify speculative vulnerabilities in existing cryptographic code (where secrets are not explicitly annotated).

3. Automatic protection

Existing programs that are free of **protect** statements are likely insecure under speculation (see Section 5.3 and [36]) and will be rejected by our type system. Thus, our third contribution is an algorithm that finds potential speculative leaks and automatically synthesizes a *minimal* number of **protect** statements to ensure that the program is speculatively constant-time (§ 5.1.3). To this end, we extend the type checker to construct a *def-use graph* that captures the data-flow between program expressions. The presence of a *path* from transient sources to stable sinks in the graph indicates a potential speculative leak in the program. To repair the program, we only need to identify a *cut-set*, a set of variables whose removal eliminates all the leaky paths in the graph. We show that inserting a **protect** statement for each variable in a cut-set suffices to yield a program that is well-typed, and hence, secure with respect to speculation. Finding such cuts is an instance of the classic Max-Flow/Min-Cut problem, so existing polynomial time algorithms let us efficiently synthesize **protect** statements that resolve the dilemma of enforcing security with *minimal* number of protections.

4.* BLADE *tool

Our final contribution is an automatic push-button tool, BLADE, which eliminates potential speculative leaks using this min-cut algorithm. BLADE extends the Cranelift compiler [27], which compiles WebAssembly (Wasm) to x86 machine code; thus, BLADE operates on programs expressed in Wasm. However, operating on Wasm is not fundamental to our approach — we believe that BLADE's techniques can be applied to other programming languages and bytecodes.

We evaluate BLADE by repairing verified yet vulnerable (to transient execution attacks) constant-time cryptographic primitives from Constant-time Wasm (CT-Wasm) [223] and HACL* [243] (§ 5.3). Compared to existing fully automatic speculative mitigation approaches (as

notably implemented in Clang), BLADE inserts an order of magnitude fewer protections (fences or SLH masks). BLADE's fence-based implementation imposes modest performance overheads: (geometric mean) 5.0% performance overhead on our benchmarks to defend from Spectre v1, or 15.3% overhead to also defend from Spectre v1.1. Both results are significant gains over current solutions. Our fence-free implementation, which automatically inserts SLH masks, is faster in the Spectre v1 case — geometric mean 1.7% overhead — but slower when including Spectre v1.1 protections, imposing 26.6% overhead.

5.1 Overview

This section gives a brief overview of the kinds of speculative leaks that BLADE identifies and how it repairs such programs by careful placement of **protect** statements. We then describe how BLADE (1) automatically repairs existing programs using our minimal **protect** inference algorithm and (2) proves that the repairs are correct using our transient-flow type system.

5.1.1 Two kinds of speculative leaks

Figure 5.1 shows a code fragment of the `SHA2_update_last` function, a core piece of the SHA2 cryptographic hash function implementation, adapted (to simplify exposition) from the HACL* library. This function takes as input a pointer `input_len`, validates the input (line 3), loads from memory the public length of the hash (line 4, ignore `protect` for now), calculates a target address `dst3` (line 6), and pads the buffer pointed to by `dst3` (line 8). Later, it uses `len` to determine the number of initialization rounds in the condition of the for-loop on line 10.

A. Leaking through a memory write

During normal, sequential execution this code is not a problem: the function validates the input to prevent classic buffer-overflow vulnerabilities. However, during speculation, an attacker can exploit this function to leak sensitive data. To do this, the attacker first has to *mistrain* the

```
1  void SHA2_update_last(int *input_len, ...)
2  {
3    if (! valid(input_len)) { return; }  // Input validation
4    int len = protect(*input_len);  // Can speculatively read secret data
5    ...
6    int *dst3 = len + base;  // Secret-tainted address
7    ...
8    *dst3 = pad;  // Secret-dependent memory access
9    ...
10   for ( i = 0; i < len + ...)  // Secret-dependent branch
11     dst2[i] = 0;
12   ...
13 }
```

Figure 5.1: Code fragment adapted from the HACL* SHA2 implementation, containing two potential speculative execution vulnerabilities: one through the data cache by writing memory at a secret-tainted address, and one through the instruction cache via a secret-tainted control-flow dependency. The patch computed by BLADE is the single **protect** inserted on line 4.

branch predictor to predict the next input to be valid. Since `input_len` is a function parameter, the attacker can do this by, e.g., calling the function repeatedly with legitimate addresses. After mistraining the branch predictor this way, the attacker manipulates `input_len` to point to an address containing secret data and calls the function again, this time with an invalid pointer. As a result of the mistraining, the branch predictor causes the processor to skip validation and load the secret into `len`, which in turn is used to calculate pointer `dst3`. The location pointed to by `dst3` is then written on line 8, leaking the secret data. Even though pointer `dst3` is invalid and the subsequent write will not be visible at the program level (the processor disregards it), the side-effects of the memory access persist in the cache and therefore become visible to the attacker. In particular, the attacker can extract the target address — and thereby the secret — using cache timing measurements [67].

B. Leaking through a control-flow dependency

The code in Figure 5.1 contains a second potential speculative vulnerability, which leaks secrets through a control-flow dependency instead of a memory access. To exploit this vulnerability, the attacker can similarly manipulate the pointer `input_len` to point to a secret

after mistraining the branch predictor to skip validation. But instead of leaking the secret directly through the data cache, the attacker can leak the value *indirectly* through a control-flow dependency: in this case, the secret determines how often the initialization loop (line 10) is executed during speculation. The attacker can then infer the value of the secret from a timing attack on the instruction cache or (much more easily) on iteration-dependent lines of the data cache.

5.1.2 Eliminating speculative leaks

Preventing the leak using memory fences

Since these leaks exploit the fact that input validation is speculatively skipped, we can prevent them by making sure that dangerous operations such as the write on line 8 or the loop condition check on line 10 are not executed until the input has been validated. Intel [97], AMD [7], and others [57, 171] recommend doing this by inserting a speculation barrier after critical validation check-points. This would place a *memory fence* after line 3, but anywhere between lines 3 and 8 would work. This fence would stop speculation over the fence: statements after the fence will not be executed until all statements up to the fence (including input validation) executed. While fences can prevent leaks, using fences as such is more restrictive than necessary — they stop speculative execution of all following instructions, not only of the instructions that leak — and thus incur a high performance cost [202, 203].

Preventing the leak efficiently

We do not need to stop all speculation to prevent leaks. Instead, we only need to ensure that potentially secret data, when read speculatively, cannot be leaked. To this end, we propose an alternative way to stop speculation from reaching the operations on line 8 and line 10, through a new primitive called **protect**. Rather than eliminate *all* speculation, **protect** only stops speculation along a *particular data-path*. We use **protect** to patch the program on line 4. Instead of assigning the value `len` directly from the result of the load, the memory load is guarded by a **protect**

statement. This ensures that the value assigned to `len` is always guaranteed to use the `input_len` pointer's final, nonspeculative value. This single **protect** statement on line 4 is sufficient to fix both of the speculative leaks described in Section 5.1.1 — it prevents any speculative, secret data from reaching lines 8 or 10 where it could be leaked to the attacker.

Implementation of protect

Our **protect** primitive provides an *abstract interface* for fine grained control of speculation. This allows us to eliminate speculation-based leaks precisely and only when needed. However, whether **protect** can eliminate leaks with tolerable runtime overhead depends on its concrete implementation. We consider and formalize two implementations: an ideal implementation and one we can implement on today's hardware.

To have fine grain control of speculation, **protect** must be implemented in *hardware* and exposed as part of the ISA. Though existing processors provide only coarse grained control over speculation through memory fence instructions, this might change in the future. For example, recently proposed microprocessor designs [202, 237] provide new hardware mechanisms to control speculation, in particular to restrict targeted types of speculation while allowing other speculation to proceed: this suggests that **protect** could be implemented efficiently in hardware in the future.

Even if future processors implement **protect**, we still need to address Spectre attacks on existing hardware. Hence, we formalize and implement **protect** in *software*, building on recent Spectre attack mitigations [188]. Specifically, we propose a self-contained approach inspired by Clang's Speculative Load Hardening (SLH) [35]. At a high level, Clang's SLH stalls speculative load instructions in a conditional block by inserting artificial data-dependencies between loaded addresses and the value of the condition. This ensures that the load is not executed before the branch condition is resolved. Unfortunately, this approach unnecessarily stalls *all* non-constant conditional load instructions, regardless of whether they can influence a subsequent load and thus can actually cause speculative data leaks. Furthermore, this approach is unsound — it can also

EXAMPLE	SUBOPTIMAL PATCH	OPTIMAL PATCH
$x := a[i_1]$	$x := \textbf{protect}(a[i_1])$	$x := a[i_1]$
$y := a[i_2]$	$y := \textbf{protect}(a[i_2])$	$y := a[i_2]$
$z := x + y$	$z := x + y$	$z := \textbf{protect}(x + y)$
$w := b[z]$	$w := b[z]$	$w := b[z]$

Figure 5.2: Running example. Program EXAMPLE is shown on the left, along with two different possible patches. The first patch is sub-optimal because it requires more **protect** statements than the optimal patch.

miss some speculative leaks, e.g., if a load instruction is not in the same conditional block that validates its address (like the code in Figure 5.1). In contrast to Clang, our approach applies SLH *selectively*, only to individual load instructions whose result flows to an instruction which might leak, and *precisely*, by using accurate bounds-check conditions to ensure that only data from valid addresses can be loaded.

5.1.3 Automatically and efficiently repairing speculative leaks

BLADE automatically finds potential speculative leaks and synthesizes a *minimal* number of **protect** statements to eliminate the leaks. We illustrate this process using the simple program EXAMPLE in Figure 5.2 as a running example. The program reads two values from an array ($x := a[i_1]$ and $y := a[i_2]$), adds them ($z := x + y$), and indexes another array with the result ($w := b[z]$). For simplicity, we omit bounds checks from the code shown, but assume that appropriate bounds checks are present at some point, perhaps before the code shown.

Like the SHA2 example from Figure 5.1, EXAMPLE contains a speculative execution vulnerability: the speculative array reads could bypass their bounds checks and so x and y can contain transient secrets (i.e., secrets introduced by misprediction). This secret data then flows to z, and finally leaks through the data cache when reading $b[z]$.

Def-use graph

To secure the program, we need to *cut the dataflow* between the array reads which could introduce *transient* secret values into the program, and the index in the array read where they

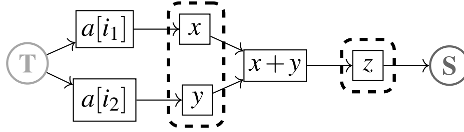


Figure 5.3: *Subset* of the def-use graph of EXAMPLE. The dashed lines identify two valid choices of cut-sets. The left cut requires removing two nodes and thus inserting two **protect** statements. The right cut shows a minimal solution, which only requires removing a single node.

are leaked through the cache. For this, we first build a *def-use graph* whose nodes and directed edges capture the data dependencies between the expressions and variables of a program. For example, consider (a subset of) the def-use graph of program EXAMPLE in Figure 5.3. In the graph, the edge $x \to x+y$ indicates that x is used to compute $x+y$. To track how transient values propagate in the def-use graph, we extend the graph with the special node T, which represents the *source* of *transient* values of the program. Since reading memory creates transient values, we connect the T node to all nodes containing expressions that explicitly read memory, e.g., $\mathrm{T} \to a[i_1]$. Following the data dependencies along the edges of the def-use graph, we can see that node T is transitively connected to node z, which indicates that z can contain transient data at run-time. To detect insecure uses of transient values, we then extend the graph with the special node **S**, which represents the *sink* of *stable* (i.e., non-transient) values of a program. Intuitively, this node draws all the values of a program that *must* be stable to avoid transient execution attacks. Therefore, we connect all expression used as array indices in the program to the **S** node, e.g., $z \to \mathbf{S}$. The fact that the graph in Figure 5.3 contains a *path* from T to **S** indicates that transient data flows through data dependencies into (what should be) a stable index expression and thus the program may be leaky.

Cutting the dataflow

In order to make the program safe, we need to *cut* the data-flow between T and **S** by introducing **protect** statements. This problem can be equivalently restated as follows: find a *cut-set*, i.e., a set of variables, such that removing the variables from the graph eliminates all paths from T to **S**. Each choice of cut-set defines a way to repair the program: simply add

a **protect** statement for each variable in the set. Figure 5.3 contains two choices of cut-sets, shown as dashed lines. The cut-set on the left requires two protect statements, for variables x and y respectively, corresponding to the suboptimal patch in Figure 5.2. The cut-set on the right is *minimal*, it requires only a single protect, for variable z, and corresponds to the optimal patch in Figure 5.2. Intuitively, minimal cut-sets result in patches that introduce as few **protect**s as needed and therefore allow more speculation. Luckily, the problem of finding a minimal cut-set is an instance of the classic Min-Cut/Max-Flow problem, which can be solved using efficient, polynomial-time algorithms [64]. For simplicity, BLADE adopts a uniform cost model and therefore synthesizes patches that contain a minimal *number* of **protect** statements, regardless of their position in the code and how many times they can be executed. Though our evaluation shows that even this simple model imposes modest overheads (§5.3), our implementation can be easily optimized by considering additional criteria when searching for a minimal cut set, with further performance gain likely. For example, we could assign weights proportional to execution frequency, or introduce penalties for placing **protect** inside loops.

(Lack of) secrecy annotations

Importantly, we do not rely on user annotations to identify secret sources and public sinks, but instead conservatively reject programs that exhibit any source-to-sink data flows. Intuitively, these security annotations would be unreliable anyway when programs are executed speculatively. In particular, public variables may still contain transient secrets and secret variables may still flow speculatively into public sinks. Hence, our def-use graph simply ignores security annotations. Notice that BLADE does *not* address the problem of enforcing a *sequential* constant-time discipline, which is the goal of existing type systems [223] and secure interfaces [243]. BLADE simply assumes that its input programs are already sequentially constant-time.

5.1.4 Attacker model

We delineate the extents of our security guarantees by discussing the attacker model considered in this work. We assume an attacker model where the attacker runs cryptographic code, written in Wasm, on a speculative out-of-order processor; the attacker can influence how programs are speculatively executed using the branch predictor[1] and choose the instruction execution order in the processor pipeline. The attacker can observe the effects of these actions on the cache, even if these effects are otherwise invisible at the ISA level. In particular, while programs run, the attacker can take precise timing measurements of the data- and instruction-cache with a cache-line granularity, and thus infer the value of secret data. These features allow the attacker to mount Spectre-PHT attacks [119, 123] and exfiltrate data through FLUSH+RELOAD [233] and PRIME+PROBE [204] cache side-channel attacks. We do not consider speculative attacks that rely on the Return Stack Buffer (e.g., Spectre-RSB [125, 140]), Branch Target Buffer (Spectre-BTB [123]), or Store-to-Load forwarding misprediction (Spectre-STL [91], recently reclassified as a Meltdown attack [154]). We similarly do not consider Meltdown attacks [133] or attacks that do not use the cache to exfiltrate data, e.g., port contention (SMoTherSpectre [22]).

5.2 Implementation

We implement BLADE as a compilation pass in the Cranelift [27] Wasm code-generator, which is used by the Lucet compiler and runtime [148]. BLADE first identifies all sources and sinks. Then, it finds the cut points using the Max-Flow/Min-Cut algorithm (§5.1.3), and either inserts fences at the cut points, or applies SLH to all of the loads which feed the cut point in the graph. This difference is why SLH sometimes requires code insertions in more locations.

[1] In particular, the attacker can make predictions based on control-flow history and memory-access patterns similar to real, adaptive predictors. Notice that these predictions *can* depend on secret information in general, but they are guaranteed to be secret-independent in the constant-time programs repaired by BLADE. Adversarial predictors that intentionally and deliberately leak secret data (e.g., reading secrets from memory) are out of scope.

Our SLH prototype implementation does not track the length of arrays, and instead uses a static constant for all array lengths when applying masking. Once compilers like Clang add support for conveying array length information to Wasm (e.g., via Wasm's custom section), our compilation pass would be able to take this information into account. This simplification in our experiments does not affect the sequence of instructions emitted for the SLH masks and thus BLADE's performance overhead is accurately measured.

Our Cranelift BLADE pass runs after the control-flow graph has been finalized and right before register allocation.[2] Placing BLADE before register allocation allows our implementation to remain oblivious of low-level details such as register pressure and stack spills and fills. Ignoring the memory operations incurred by spills and fills simplifies BLADE's analysis and reduces the required number of **protect** statements. This, importantly, does not compromise the security of its mitigations: In Cranelift, spills and fills are always to constant addresses which are inaccessible to ordinary Wasm loads and stores, even speculatively. (Cranelift uses guard pages — not conditional bounds checks — to ensure that Wasm memory accesses cannot access anything outside the linear memory, such as the stack used for spills and fills.) As a result, we can treat stack spill slots like registers. Indeed, since BLADE runs before register allocation, it already traces def-use chains across operations that will become spills and fills. Even if a particular spill-fill sequence would handle potentially sensitive transient data, BLADE would insert a **protect** between the original transient source and the final transient sink (and thus mitigate the attack).

Our implementation implements a single optimization: we do not mark constant-address loads as transient sources. We assume that the program contains no loads from out-of-bounds constant addresses, and therefore that loads from constant (Wasm linear memory) addresses can never speculatively produce invalid data. As we describe below, however, we omit this optimization when considering Spectre v1.1.

[2]More precisely: The Cranelift register allocation pass modifies the control-flow graph as an initial step; we insert our pass after this initial step but before register allocation proper.

At its core, our repair algorithm addresses Spectre v1 attacks based on PHT mispredictions. To also protect against Spectre variant 1.1 attacks, which exploit store forwarding in the presence of PHT mispredictions,[3] we perform two additional mitigations. First, we mark constant-address loads as transient sources (and thus omit the above optimization). Under Spectre v1.1, a load from a constant address may speculatively produce transient data, if a previous speculative store wrote transient data to that constant address — and, thus, BLADE must account for this. Second, our SLH implementation marks all stored *values* as sinks, essentially preventing any transient data from being stored to memory. This is necessary when considering Spectre v1.1 because otherwise, ensuring that a load is in-bounds using SLH is insufficient to guarantee that the produced data is not transient — again, a previous speculative store may have written transient data to that in-bounds address.

5.3 Evaluation

We evaluate BLADE by answering two questions: ***(Q1)*** How many **protect**s does BLADE insert when repairing existing programs? ***(Q2)*** What is the runtime performance overhead of eliminating speculative leaks with BLADE on existing hardware?

Benchmarks

We evaluate BLADE on existing cryptographic code taken from two sources. First, we consider two cryptographic primitives from CT-Wasm [223]:

- The Salsa20 stream cipher, with a workload of 64 bytes.
- The SHA-256 hash function, with workloads of 64 bytes (one block) or 8192 bytes (128 blocks).

Second, we consider automatically generated cryptographic primitives and protocols from the HACL* [243] library. We compile the automatically generated C code to Wasm using Clang's

[3] Spectre v1 and Spectre v1.1 attacks are both classified as Spectre-PHT attacks [32].

Table 5.1: **Ref**: Reference implementation with no Spectre mitigations; **Baseline-F**: Baseline mitigation inserting fences; **BLADE-F**: BLADE using fences as **protect**; **Baseline-S**: Baseline mitigation using SLH; **BLADE-S**: BLADE using SLH; **Overhead**: Runtime overhead compared to Ref; **Defs**: number of fences inserted (Baseline-F and BLADE-F), or number of loads protected with SLH (Baseline-S and BLADE-S)

		Without v1.1 protections			With v1.1 protections		
Benchmark	**Defense**	**Time**	**Overhead**	**Defs**	**Time**	**Overhead**	**Defs**
Salsa20 (CT-Wasm), 64 bytes	Ref	4.3 us	-	-	4.3 us	-	-
	Baseline-F	4.6 us	7.2%	3	8.6 us	101.7%	99
	BLADE-F	4.4 us	1.9%	0	4.3 us	1.7%	0
	Baseline-S	4.4 us	2.7%	3	5.3 us	24.3%	99
	BLADE-S	4.3 us	0.5%	0	5.4 us	26.4%	99
SHA-256 (CT-Wasm), 64 bytes	Ref	13.7 us	-	-	13.7 us	-	-
	Baseline-F	19.8 us	43.8%	23	20.3 us	48.0%	54
	BLADE-F	13.8 us	0.2%	0	14.5 us	5.4%	3
	Baseline-S	15.0 us	9.1%	23	15.1 us	10.0%	54
	BLADE-S	13.9 us	0.8%	0	15.2 us	10.9%	54
SHA-256 (CT-Wasm), 8192 bytes	Ref	114.6 us	-	-	114.6 us	-	-
	Baseline-F	516.6 us	350.6%	23	632.6 us	451.8%	54
	BLADE-F	113.7 us	-0.8%	0	193.3 us	68.6%	3
	Baseline-S	187.4 us	63.4%	23	208.0 us	81.5%	54
	BLADE-S	115.2 us	0.5%	0	216.5 us	88.9%	54
ChaCha20 (HACL*), 8192 bytes	Ref	43.7 us	-	-	43.7 us	-	-
	Baseline-F	85.2 us	94.8%	136	85.4 us	95.3%	142
	BLADE-F	44.4 us	1.5%	3	45.4 us	3.8%	7
	Baseline-S	52.8 us	20.8%	136	53.3 us	21.9%	142
	BLADE-S	43.6 us	-0.3%	3	53.8 us	22.9%	142
Poly1305 (HACL*), 1024 bytes	Ref	5.5 us	-	-	5.5 us	-	-
	Baseline-F	6.3 us	15.9%	133	6.4 us	17.2%	139
	BLADE-F	5.5 us	1.4%	3	5.6 us	2.2%	9
	Baseline-S	5.6 us	1.8%	133	5.7 us	4.4%	139
	BLADE-S	5.5 us	1.0%	3	5.6 us	2.5%	139
Poly1305 (HACL*), 8192 bytes	Ref	15.1 us	-	-	15.1 us	-	-
	Baseline-F	21.3 us	41.1%	133	21.4 us	41.2%	139
	BLADE-F	15.1 us	-0.0%	3	15.2 us	0.8%	9
	Baseline-S	16.2 us	7.2%	133	16.3 us	7.6%	139
	BLADE-S	15.2 us	0.7%	3	16.2 us	7.1%	139
ECDH Curve25519 (HACL*)	Ref	354.3 us	-	-	354.3 us	-	-
	Baseline-F	989.8 us	179.3%	1862	1006.4 us	184.0%	1887
	BLADE-F	479.9 us	35.4%	235	497.8 us	40.5%	256
	Baseline-S	507.0 us	43.1%	1862	520.4 us	46.9%	1887
	BLADE-S	386.1 us	9.0%	1419	516.8 us	45.9%	1887
Geometric means	Ref		-			-	
	Baseline-F		80.2%			104.8%	
	BLADE-F		5.0%			15.3%	
	Baseline-S		19.4%			25.8%	
	BLADE-S		1.7%			26.6%	

Wasm backend. (We do not use HACL*'s Wasm backend since it relies on a JavaScript embedding environment and is not well suited for Lucet.) Specifically, from HACL* we consider:

- The ChaCha20 stream cipher, with a workload of 8192 bytes.
- The Poly1305 message authentication code, with workloads of 1024 or 8192 bytes.
- ECDH key agreement using Curve25519.

We selected these primitives to cover different kinds of modern crypto workloads (including hash functions, MACs, encryption ciphers, and public key exchange algorithms). We omitted primitives that had inline assembly or SIMD since Lucet does not yet support either; we also omitted the AES from HACL* and TEA from CT-Wasm — modern processors implement AES in hardware (largely because efficient software implementations of AES are generally not constant-time [170]), while TEA is not used in practice. All the primitives we consider have been verified to be constant-time — free of cache and timing side-channels. However, the proofs assume a sequential execution model and do not account for speculative leaks as addressed in this work.

Experimental setup

We conduct our experiments on an Intel Xeon Platinum 8160 (Skylake) with 1TB of RAM. The machine runs Arch Linux with kernel `5.8.14`, and we use the Lucet runtime version `0.7.0-dev` (Cranelift version `0.62.0` with our modifications) compiled with `rustc` version `1.46.0`. We collect benchmarks using the Rust `criterion` crate version `0.3.3` [88] and report the point estimate for the mean runtime of each benchmark.

Reference and baseline comparisons

We compare BLADE to a reference (unsafe) implementation and a baseline (safe) implementation which simply **protect**s every Wasm memory load instruction. We consider two baseline variants: The baseline solution with Spectre v1.1 mitigation **protect**s every Wasm load instruction, while the baseline solution with only Spectre v1 mitigation **protect**s only Wasm load instructions with non-constant addresses. The latter is similar to Clang's Spectre mitigation pass, which applies SLH to each non-constant array read [35]. We evaluate both BLADE and the baseline

implementation with Spectre v1 protection and with both v1 and v1.1 protections combined. We consider both fence-based and SLH-based implementations of the **protect** primitive. In the rest of this section, we use Baseline-F and BLADE-F to refer to fence-based implementations of their respective mitigations and Baseline-S and BLADE-S to refer to the SLH-based implementations.

Results

Table 5.1 summarizes our results. With Spectre v1 protections, both BLADE-F and BLADE-S insert very few **protect**s and have negligible performance overhead on most of our benchmarks — the geometric mean overheads imposed by BLADE-F and BLADE-S are 5.0% and 1.7%, respectively. In contrast, the baseline passes insert between 3 and 1862 protections and incur significantly higher overheads than BLADE — the geometric mean overheads imposed by Baseline-F and Baseline-S are 80.2% and 19.4%, respectively.

With both v1 and v1.1 protections, BLADE-F inserts an order of magnitude fewer protections than Baseline-F, and has correspondingly low performance overhead — the geometric mean overhead of BLADE-F is 15.3%, whereas Baseline-F's is 104.8%. The geomean overhead of both BLADE-S and Baseline-S, on the other hand, is roughly 26%. Unlike BLADE-F, BLADE-S must mark all stored values as sinks in order to eliminate Spectre v1.1 attacks; for these benchmarks, this countermeasure requires BLADE-S to apply protections to every Wasm load, just like Baseline-S. Indeed, we see in the table that Baseline-S and BLADE-S make the exact same number of additions to the code.

We make two observations from our measurements. First, and somewhat surprisingly, BLADE does not insert any **protect**s for Spectre v1 on any of the CT-Wasm benchmarks. We attribute this to the style of code: the CT-Wasm primitives are hand-written and, moreover, statically allocate variables and arrays in the Wasm linear memory — which, in turn, results in many constant-address loads. This is unlike the HACL* primitives which are written in F*, compiled to C and then Wasm — and thus require between 3 and 235 **protect**s.

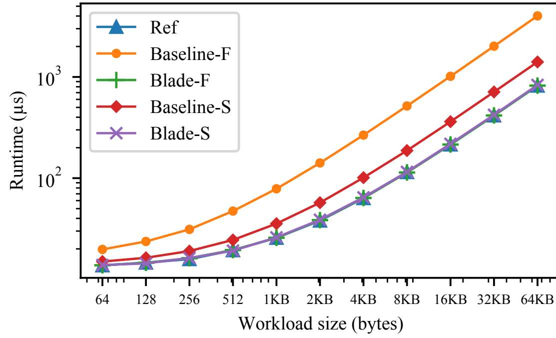
Second, we observe for the Spectre v1 version that SLH gives overall better performance than fences, as expected. This is true even in the case of Curve25519, where implementing **protect** using SLH (BLADE-S) results in a significant increase in the number of protections versus the fence-based implementation (BLADE-F). Even in this case, the more targeted restriction of speculation, and the less heavyweight impact on the pipeline, allows SLH to still prevail over the fewer fences. However, that advantage is lost when considering both v1 and v1.1 mitigation. In this case, the sharp increase in the number of **protect**s required for this solution end up making the fenced version more performant overall.

In reality, though, both versions are inadequate software emulations of what the **protect** primitive should be. Fences take a heavy toll on the pipeline and are far too restrictive of speculation, while SLH pays a heavy instruction overhead for each instance, and can only be applied directly to loads, not to arbitrary cut points. A hardware implementation of the **protect** primitive could combine the best of BLADE-F and BLADE-S: targeted restriction of speculation, minimal instruction overhead, and only as many defenses as BLADE-F, without the inflation in insertion count required by BLADE-S.

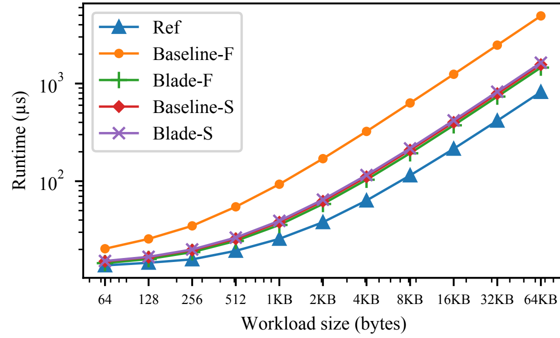
However, even without any hardware assistance, both versions of the BLADE tool provide significant performance gains over the current state of the art in mitigating Spectre v1, and over existing fence-based solutions when targeting v1 or both v1 and v1.1.

Additional analysis of performance overheads

Table 5.1 showed that the performance overhead for some benchmarks depends heavily on the workload size. We explore this relationship in more detail in Figure 5.4. Specifically, we see that for low workload sizes, the runtime is dominated by fixed costs for sandbox setup and teardown; the execution of the Wasm code contributes little to the performance. As the workload size increases, the overall performance overhead asymptotically approaches the overhead of the Wasm execution itself. This is shown even more clearly in Figure 5.5, where we see that the asymptotic overhead for SHA-256 with v1.1 protections is approximately 78% and 99%

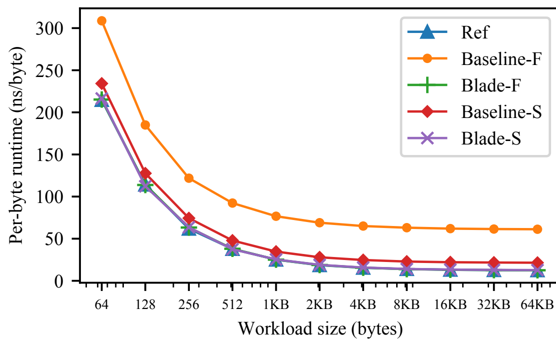


(a) Without v1.1 protections

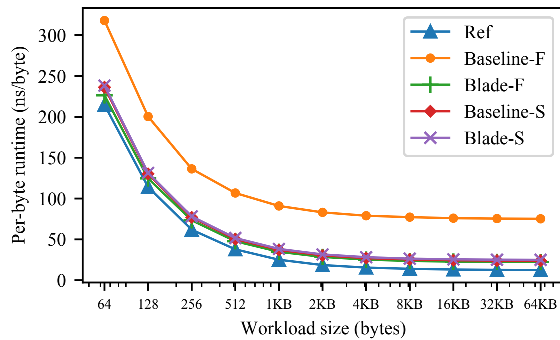


(b) With v1.1 protections

Figure 5.4: Runtime of SHA256 (CT-Wasm) as the workload size varies



(a) Without v1.1 protections



(b) With v1.1 protections

Figure 5.5: Runtime of SHA256 (CT-Wasm) as the workload size varies, presented on a per-byte basis

respectively for BLADE-F and BLADE-S, while the asymptotic overheads without v1.1 protections are unsurprisingly approximately zero for BLADE-F and BLADE-S, as they insert no defenses.

5.4 Related work

Detection and repair

Wu and Wang [228] detect cache side channels via abstract interpretation by augmenting the program control-flow to accommodate for speculation. SPECTECTOR [80] and PITCHFORK [36] use symbolic execution on x86 binaries to detect speculative vulnerabilities. Cheang et al. [39] and Bloem et al. [23] apply bounded model checking to detect potential speculative vulnerabilities respectively via 4-ways self-composition and taint-tracking. These efforts assume a *fixed* speculation bound, and they focus on vulnerability detection rather than proposing techniques to *repair* vulnerable programs. Furthermore, many of these works consider only *in-order* execution. In contrast, our type system enforces *speculative constant-time* when program instructions are executed *out-of-order* with *unbounded* speculation — and our tool BLADE automatically synthesizes repairs. Separately, OO7 [218] statically analyzes a binary from a set of untrusted input sources, detecting vulnerable patterns and inserting fences in turn. Our tool, BLADE, not only repairs vulnerable programs without user annotation, but ensures that program patches contain a minimum number of fences. Furthermore, BLADE formally guarantees that repaired programs are free from speculation-based attacks.

Concurrent to our work, Intel proposed a mitigation for a new class of LVI attacks [101, 207]. Like BLADE, they implement a compiler pass that analyzes the program to determine an optimal placement of fences to cut source-to-sink data flows. While we consider an abstract, ideal **protect** primitive, they focus on the optimal placement of fences in particular. This means that they optimize the fence placement by taking into account the coarse-grained effects of fences —

e.g., one fence providing a speculation barrier for multiple independent data-dependency chains.[4] This also means, however, their approach does not easily transfer to using SLH for cases where SLH would be faster.

Hardware-based mitigations

To eliminate speculative attacks, several secure hardware designs have been proposed. Taram et al. [202] propose context-sensitive fencing, a hardware-based mitigation that dynamically inserts fences in the instruction stream when dangerous conditions arise. INVISISPEC [231] features a special *speculative buffer* to prevent speculative loads from polluting the cache. STT [237] tracks speculative taints *dynamically* inside the processor micro-architecture and stalls instructions to prevent speculative leaks. Schwarz et al. [188] propose CONTEXT, a whole architecture change (applications, compilers, operating systems, and hardware) to eliminate *all* Spectre attacks. Though BLADE can benefit from a hardware implementation of **protect**, this work also shows that Spectre-PHT on existing hardware can be automatically eliminated in pure software with modest performance overheads.

5.5 Limitations and future work

BLADE only addresses Spectre-PHT attacks and does so at the Wasm layer. Extending BLADE to tackle other Spectre variants and the limitations of operating on Wasm is future work.

Other Spectre variants

The Spectre-BTB variant [123] mistrains the Branch Target Buffer (BTB), which is used to predict indirect jump targets, to hijack the (speculative) control-flow of the program. Although Wasm does not provide an unrestricted indirect jump instruction, the indirect function call instruction — which is used to call functions registered in a function table — can be abused

[4]Unlike our approach, their resulting optimization problem is NP-hard — and only sub-optimal solutions may be found through heuristics.

by an attacker. To address (in-process) Spectre-BTB, we could extend our type system to restrict the values used as indices into the function table to be typed as *stable*.

The other Spectre variant, Spectre-RSB [125, 140], abuses the return stack buffer. To mitigate these attacks, we could analyze Wasm code to identify potential RSB over/underflows and insert fences in response, or use mitigation strategies like RSB stuffing [98]. A more promising approach, however, is to use Intel's recent shadow stack, which ensures that returns cannot be speculatively hijacked [194].

Detecting Spectre gadgets at the binary level

BLADE operates on Wasm code — or more precisely, on the Cranelift compiler's IR — and can thus miss leaks inserted by the compiler passes that run after BLADE — namely, register allocation and instruction selection. Though these passes are unlikely to introduce such leaks, we leave the validation of the generated binary code to future work.

Spectre resistant compilation

An alternative to repairing existing programs is to ensure they are compiled securely from the start. Recent works have developed verified constant-time-preserving optimizing compilers for generating correct, efficient, and secure cryptographic code [5, 17]. Doing this for speculative constant-time, and understanding which optimizations break the SCT notion, is an interesting direction for future work.

Bounds information

BLADE-S relies on array bounds information to implement the speculative load hardening. For a memory-safe language, this information can be made available to BLADE when compiling to Wasm (e.g., as a custom section). When compiling languages like C, where array bounds information is not explicit, this is harder — and we would need to use program analysis to track array lengths statically [213]. Although such an analysis may be feasible for cryptographic code, it is likely to fall short for other application domains (e.g., due to dynamic memory allocation and pointer chasing). In these cases, we could track array lengths at runtime (e.g., by instrumenting

programs [161]) or, more simply, fall back to fences (especially since the overhead of tracking bounds information at runtime is typically high).

5.6 Conclusion

We presented BLADE, a fully automatic approach to provably and efficiently eliminate speculation-based leakage in unannotated cryptographic code. BLADE statically detects data flows from transient sources to stable sinks and synthesizes a minimal number of fence-based or SLH-based **protect** calls to eliminate potential leaks. Our evaluation shows that BLADE inserts an order of magnitude fewer protections than would be added by today's compilers, and that existing crypto primitives repaired with BLADE impose modest overheads when using both fences and SLH for **protect**.

Acknowledgments

We thank the reviewers and our shepherd Aseem Rastogi for their suggestions and insightful comments. Many thanks to Shravan Narayan, Ravi Sahita, and Anjo Vahldiek-Oberwagner for fruitful discussions. This work was supported in part by gifts from Fastly, Fujitsu, and Cisco; by the NSF under Grant Number CNS-1514435 and CCF-1918573; by ONR Grant N000141512750; by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity; and, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapter 5, in part, contains material reprinted from the Proceedings of the ACM on Programming Languages (Issue POPL), 2021. Vassena, Marco; Disselkoen, Craig; v. Gleissenthall, Klaus; Cauligi, Sunjay; Kıcı, Rami; Jhala, Ranjit; Tullsen, Dean; Stefan, Deian. ACM, 2021. The dissertation author was the primary investigator and author of the reprinted material.

Chapter 6

Progressive Memory Safety for WebAssembly

WebAssembly (Wasm) is a platform-independent bytecode designed to run C/C++ and similar languages at near native speed in the browser. Wasm's *linear memory model* — i.e., loads and stores to an untyped array of bytes, is the key feature that makes it possible for C/C++ compilers like Clang to easily and efficiently target Wasm. Unfortunately, this is also the reason memory safety vulnerabilities, like buffer overflows and use-after-frees (UAFs), remain a problem when C/C++ programs are compiled to Wasm [146, 151].

Wasm is designed to allow browsers to run code in a sandbox, isolating the impact of vulnerabilities in Wasm code from the rest of the browser.[1] But keeping the browser safe from Wasm code is not the same as keeping Wasm code safe from itself — isolation doesn't prevent attackers from exploiting memory-safety bugs to compromise the Wasm code and any data it handles.

This is worrisome. Wasm is supported by all major browser vendors, and already implemented in over 80% of all browsers on the web [54]. Wasm is also starting to find uses

[1] For example, Wasm is type-safe, separates code and data, and enforces coarse-grained control flow integrity (CFI) [226]. All these design choices simplify isolation.

beyond the browser — from server-side runtimes [165, 221], to IoT platforms [85], and edge computing [90, 210]. As Wasm proliferates, we risk creating yet another ecosystem where memory-safety vulnerabilities are rampant.

Unfortunately, we can't simply modify Wasm to enforce strong memory safety by default, like past bytecodes for high-level languages (e.g., the Java bytecode or .NET common interface language). Requiring strong memory safety would be an anathema to the simplicity and performance that have fueled Wasm's broad adoption. Instead, we argue for a progressive approach to strong memory safety that neither mandates high performance overheads that could hinder widespread adoption, nor gives up on the goal of memory safety in the name of performance.

This progressive approach is both necessary and timely. As hardware acceleration makes memory safety increasingly cheap, the boundary between safety and high performance will narrow. For example, Sparc's application data integrity (ADI) [193] and ARM's upcoming memory tagging extension (MTE) [72] can probabilistically detect and prevent many buffer overflow and UAF bugs at near zero overhead — orders of magnitude faster than what's possible without dedicated hardware [193]. Similarly, ARM's recent pointer authentication feature can efficiently mitigate pointer corruption [132]. Looking further out, it seems likely that ARM will adopt some version of CHERI [74, 222] to efficiently enforce full *spatial safety* and eliminate buffer overflow bugs altogether. Unfortunately, Wasm can't leverage these hardware features — too much high-level information is lost when compiling from C/C++ to Wasm's existing abstractions.

To bridge this gap, we propose Memory Safe WebAssembly (MSWasm), a backwards-compatible extension to Wasm that makes memory safety explicit at the language level. MSWasm extends Wasm's memory model with a new *segment memory* made up of *segments* — temporally safe extents of memory — and ensures that all accesses to the segment memory are via *handles*. Handles are strongly-typed first-class values that encapsulate bounds-checked, memory-safe pointers to the segment memory. With handles and segments, a C/C++ compiler can can encode all the semantics necessary to enforce *memory safety* [51] — in particular, *spatial safety*, *temporal*

safety, and *pointer integrity*. By allowing handles to be *sliced*, MSWasm even captures fine-grained *intra-object* safety, e.g., to prevent a buffer in one field of a `struct` from overflowing into the next.

These richer semantics provide MSWasm backends — compilers and JITs — with everything they need to use different hardware and software approaches to ensure safety. It's then up to the backend to determine what policy to enforce based on the available hardware and needs of the user.

Some users might value detecting critical memory safety bugs in production but are unwilling to tolerate much overhead for enforcement — an MSWasm backend for ARM could use memory tagging (when available) to achieve this efficiently. On the other hand, a developer deploying a critical service, such as an authentication server, might value security more than performance, and thus request that the MSWasm backend enforce the full set of MSWasm safety properties, regardless of hardware support. A third user, e.g., a game developer unconcerned with security, might simply want to eschew any overheads and get performance equivalent to what normal Wasm would offer.

Our hope is that by ensuring memory safety overheads never exceed what is acceptable to the user, compiling with the semantics necessary for full memory safety will become the default. This can help incentivize the development of new hardware features: with MSWasm, if a vendor develops a new feature and changes a single JIT (e.g., V8 in Chrome) they can almost immediately expose its value to billions of users. At the same time, as better hardware becomes widely available, MSWasm backends can seamlessly enable its use, providing a path to progressively better memory safety on the web, and other places where Wasm is used.

Organization

Next, we offer a brief overview of Wasm and its limited form of memory safety, then survey potential hardware features that could help (§ 6.1). We present MSWasm in § 6.3, and explore how it can improve memory safety for low-level languages like C/C++. In § 6.4 we discuss

different hardware and software mechanisms MSWasm backends could leverage to enforce safety. Finally we discuss extensions to MSWasm and alternative design choices in § 6.6.

6.1 Motivation

Our goal with MSWasm is to enhance Wasm for greater expressiveness, so that it can encode the semantics necessary to support different approaches to accelerating memory safety — more specifically, by adding a model of pointers and memory allocation so that this information isn't lost when lowering to Wasm.

To see why this is necessary, we will start by discussing the cause of memory safety vulnerabilities (§ 6.1.1); then sketch Wasm's basic structure, and why lowering to Wasm preserves these vulnerabilities (§ 6.1.2); and finally survey some of the current and future hardware support which MSWasm backends could use to prevent or mitigate memory-safety vulnerabilities (§ 6.1.3).

6.1.1 Memory safety

In loose terms, memory-safety bugs in C/C++ result from how compilers interpret undefined behavior. For example, one valid interpretation of writing beyond the end of an array in the C standard is to crash the program — an easily understandable semantic. However, array bounds checking can induce unwanted performance overheads, so compilers adopt the more dangerous interpretation: write to whatever other object happens to be in that memory location, and continue running.

This interpretation violates programmers' assumptions about separation between different data objects [51]; and when this interpretation meets malicious inputs, they become memory-safety vulnerabilities, as the programmer has inadvertently given a potentially malicious input control over unintended parts of program data and control flow.

To prevent these attacks, the compiler could take a more conservative interpretation, and halt on undefined behaviors that violate separation — i.e., enforce *memory safety*. In practice, this amounts to ensuring three properties:

- *spatial safety*, which prevents out-of-bounds reads and writes;
- *temporal safety*, which prevents exploitation of use-after-free;
- *pointer integrity*, which prevents pointers from being manufactured from non-pointer values (e.g., casting an integer to a pointer), and also makes it impossible to corrupt a pointer in memory to create a different valid pointer.

Together, these three properties ensure that every pointer dereference in a program returns data from the corresponding, valid object.

Enforcing these properties efficiently requires some amount of dynamic checking — such as tracking if a pointer's referent has been de-allocated to prevent use-after-free bugs. Often the overhead of implementing these checks in pure software is prohibitive; even optimized JIT-based approaches can incur over 2× performance slowdowns for enforcing full memory safety [163].

Thus, the status quo for C/C++ has been to rely on system-level mitigations such as ASLR and W⊕X rather than enforce memory safety outright. Fortunately, hardware vendors are increasingly adding features to bring down the overhead of memory safety.

6.1.2 WebAssembly (Wasm)

Wasm is a portable bytecode language, designed to be an efficient target for low-level languages [84]. On the Web, developers use Wasm to embed existing C/C++ libraries such as the libsodium crypto library, video decoding libraries, and game engines into webpages. But Wasm's reach extends far beyond the browser. Server-side, Node.js application developers, for example, use Wasm to safely embed fast native code alongside JavaScript. Even serverless

platforms (e.g., Fastly and Cloudflare) are making large bets on Wasm as the future of efficient, edge computing [90, 210].

Structurally, Wasm is a stack machine language that has well-typed stack (using simple primitive types: `i32`, `i64`, `f32`, and `f64`) and a linear model of memory, i.e., load and store to an "untyped array of bytes" [226] similar to native platforms. Consider, for example, a Wasm function that increments (by 3) a value in memory at a given address:

```
(func $add3 (param $addr i32)
  (i32.load (get_local $addr))
  (i32.add (i32.const 3))
  (i32.store (get_local $addr)))
```

This example shows how Wasm's values and stack operations are typed, but also how memory addresses simply have type `i32`; thus, loads and stores are free to arbitrarily read and write in Wasm's linear memory.

Because Wasm is designed to be an embedded in existing applications, Wasm code runs in an isolated sandbox. For example, even though Wasm code can access arbitrary indices in its own memory, there is no way for a Wasm instruction to access memory outside of its sandboxed area. The Wasm backend similarly protects return addresses with a separate stack and ensures that all indirect function calls go through well-typed entry points. Together, these protect Wasm code from stack-smashing attacks and make traditional return-oriented programming (ROP) attacks impractical.

Unfortunately, Wasm's safety is often misunderstood. For example, Wasm is sometimes called a "memory-safe language" [224]. This is not true: memory-safe languages provide *spatial safety*, *temporal safety*, and *pointer integrity* (§ 6.1.1); Wasm provides none of these.

Wasm, like native platforms, allows load and store instructions to operate on an untyped address space using arbitrary integer addresses. While attackers cannot carry out stack-smashing or ROP attacks, they can still exploit familiar memory-safety vulnerabilities in the Wasm linear

memory to read and write data, just as they have in C/C++ applications on native platforms for decades. These attacks neither enable nor require escaping the Wasm sandbox. Indeed, compromising the Wasm application itself is often enough — plenty of sensitive data (e.g., cryptographic keys in the case of libsodium) is located within the sandbox.

6.1.3 Hardware support for memory safety

Hardware tagged memory

Tagged memory associates additional metadata, a *tag*, with each region of memory. Research hardware-capability systems such as CHERI [222] and lowRISC [138] use tagged memory to ensure that capabilities cannot be forged or modified [113]. Other tagged memory systems such as Sparc ADI [193] associate, e.g., a 4-bit tag with each 64-byte aligned region of memory. In these systems, each pointer also contains a tag which is compared with the tag of the target memory on each load and store; if the tags don't match, the operation fails. The efficiency of this check makes memory tagging useful for enforcing a variety of protection policies [46]. For instance, memory tagging can be used for probabilistically detecting many spatial and temporal safety bugs.

ARM recently added a memory tagging extension (MTE) to the ARM 8.5-A ISA [72] which employs a 16-byte granule size but is otherwise nearly identical to ADI. This makes it likely that hardware tagged memory will be widely available in the near future.

Pointer authentication

ARM pointer authentication (PAC) is a feature which stores a cryptographic MAC in the unused upper bits of each pointer. PAC is supported by recent ARM processors, and already used in today's iPhones [178]. With PAC, memory operations can be made to fail if the pointer being dereferenced does not have a MAC from the appropriate private key (e.g., for kernel pointers, the kernel's private key) and context (an additional input to the MAC that can be used to provide

compartmentalization). PAC instructions can be used to protect the integrity of data pointers, function pointers, and even stack pointers for CFI.

Bounds registers

Intel attempted to support memory safety with MPX *bounds registers* and *bounds tables*. With MPX, upper and lower bounds can be loaded into the bounds registers which are checked during loads and stores. Unfortunately, MPX failed to offer better performance than software-only solutions [166], leading to low adoption in practice, and even to GCC dropping support for MPX [66].

Hardware capabilities

The CHERI architecture [222] supports capabilities as an alternative to pointers. Capabilities are unforgeable references that contain both bounds data and access privileges, making them ideal for memory safety. While to date the CHERI architecture remains a research prototype, ARM recently announced plans to incorporate some of CHERI's ideas into future designs [74]. This represents a promising path forward for hardware memory safety.

6.2 Design goals

We propose to extend Wasm to provide the capability to efficiently enforce full memory-safety guarantees, even inside the Wasm sandbox. The design of our extension, MSWasm, has four major goals:

Strong safety guarantees

MSWasm seeks to provide abstractions that can be used to enforce memory safety, i.e., *spatial* and *temporal safety*, and *pointer integrity*. At the same time, these abstractions should also be sufficient to support weaker piecemeal mitigation and detection mechanisms.

Backwards compatibility

MSWasm must be a minimally invasive extension to Wasm. This includes making MSWasm backwards compatible with existing Wasm toolchains, and making all of its features opt-in. Thus, existing Wasm binaries should remain valid, with the same semantics as before. Likewise, existing source-to-Wasm compilers should continue to be valid.

Leveraging hardware

MSWasm backends should be able to leverage whatever memory-safety hardware features are available on a given hardware platform. Thus, the design of MSWasm should be general enough to accommodate different detection and enforcement mechanisms, and not be specific to any particular hardware mechanism, e.g., memory tagging or MPX.

Progressive enforcement

Enforcing full memory safety is the ideal, but doing this without sufficient hardware support is prohibitive in many use cases—and requiring it would discourage users from building and deploying their applications with MSWasm.

Instead, MSWasm should accommodate different design points that trade off security and performance, and leave it to backends to choose the best combination of software and hardware mechanisms to implement the desired guarantees. For this reason, MSWasm separates the memory-safety *abstraction*—the Wasm-level semantics needed to allow C/C++ compilers to encode sufficient information to efficiently enforce memory-safety guarantees—from the *enforcement policy*, i.e., how the backend actually implements whatever checks are necessary in order to meet the desired security and performance goals.

Different applications demand different points in the security-performance tradeoff space. For example, many applications would opt for mitigations over enforcement to stay within a reasonable performance budget (e.g., 5-10% CPU overhead). Security-critical applications, on the other hand, could demand full memory-safety guarantees, no matter the cost. Other applications might even request no enforcement at all—e.g., because performance is critical or, perhaps,

because memory safety is enforced statically or dynamically, with inline checks. Such a policy could ideally be implemented with no overhead, equivalent to existing Wasm without any checks.

As hardware support improves and the cost of memory safety decreases, MSWasm backends will be able to provide progressively stronger guarantees at lower overheads, transparently increasing safety without violating users' performance requirements.

6.3 Design

At the heart of MSWasm is a new *segment memory* that lives alongside the Wasm linear memory. Unlike the linear memory, the segment memory is well-structured; it consists of *segments* — linearly addressable, bounded regions of memory whose lifetimes are manually managed. Segments can only be accessed through *handles*, and not via Wasm's usual `load` and `store` instructions. This way, by placing certain restrictions on how handles are used, MSWasm can make strong guarantees about the memory safety of these segments.

In the rest of this section we describe MSWasm by showing how languages like C and C++ can be compiled to MSWasm, and how enforcing certain restrictions on MSWasm primitives can provide strong memory-safety guarantees.

6.3.1 Handles and segments

Handles are used to model pointers — specifically, pointers bounded to particular live allocations of memory. Abstractly, handles are described by the 4-tuple (`base`, `offset`, `bound`, `isCorrupted`). The `base` of the handle represents the address of the start of the segment (in segment memory) being pointed to. The `offset` is the offset *within* the segment, i.e., within the `bound`, that the handle points to. If we think of the `handle` as a pointer, the location it points to in the segment memory is the `base+offset`.

As handles are used to model pointers, we introduce new Wasm instructions for pointer arithmetic, including addition, subtraction, and comparisons on handles; and we also define a `NULL` handle. For example, the `handle.add` and `handle.sub` instructions modify the handle offset without changing the base or bound. Pointer arithmetic can give rise to *out-of-bounds* handles (i.e., when the offset is negative or larger than the bound). We don't prevent code from creating such handles; instead, memory-safe MSWasm backends will trap when out-of-bounds handles are used, i.e., when the pointer is dereferenced. Delaying this check until dereference is important both for performance — it eliminates unnecessary checks during pointer arithmetic — and compatibility — as pointers that temporarily point out of bounds are common [61] and benign behavior in C programs [149, 150].

MSWasm treats handles as opaque values and does not specify a byte-level representation for them. This means that individual backends can represent them in a way most suitable to each platform, which may include storing some of this data separately and not as part of the handle itself. Moreover, as discussed below, backends which do not provide certain guarantees need not keep track of some of this data at all.

Segments are linearly addressable, fixed sized, extents of memory. (In § 6.3.3 we detail how segments are created and released.) Wasm code can load and store values to the segment memory via handles:[2]

```
i32.segment_load(src: handle) -> i32
i64.segment_load(src: handle) -> i64
i32.segment_store(dst: handle, val: i32)
i64.segment_store(dst: handle, val: i64)
```

To enforce spatial safety, an MSWasm backend must ensure the following property:[3]

[2]On notation: When practical, we adopt the Wasm convention of prefixing instruction names with their return type — for example, `i32.add`. When additional clarity is necessary, we adopt the notation `new_segment(size: i32) -> handle` which shows the arguments and return type explicitly.

[3]To fully prevent all out-of-bounds access to the segment memory, MSWasm backends must also provide the *handle integrity* property defined later, which prevents out-of-bounds handles from being forged by an attacker.

Spatial safety for existing handles: For each segment load and store, the handle being dereferenced is *in-bounds*, i.e., the handle is not the `NULL` handle, and its offset is nonnegative and less than its bound.

On the other hand, if these bounds checks are omitted by the backend, bounds information for handles need not even be tracked, and the performance of the segment load and store instructions should be similar to Wasm's existing load and store.

6.3.2 Slicing handles

With the checks above, handles provide *inter-object* spatial safety, i.e., they restrict a pointer to accessing only the segment the handle points to. We also use handles to provide *intra-object* spatial safety through *slicing*. Wasm code can slice handles with:

```
segment_slice(parent: handle, base: i32, bound: i32) -> handle
```

This copies the parent handle and then grows the base, shrinks the bound, or both, to yield a smaller window into the segment. To illustrate this, consider the following code snippet.

```
struct A {char foo[4]; char bar[4];}
struct A * my_str = malloc(sizeof(struct A));
char * subfield = my_str->foo;
```

When we create a pointer to `foo` on the third line, the compiler can generate a new slice that includes only `foo`. Thus, if our code later tries to overflow `foo`, it will be contained to this slice by the spatial safety checks, and it will not be able to corrupt `bar`.

6.3.3 Segment allocation and deallocation

MSWasm code creates new segments and releases them with the new instructions:

```
new_segment(size: i32) -> handle
free_segment(h: handle)
```

The `new_segment` instruction returns a handle to a newly-allocated segment. New segments are initialized to all zeroes, much how Wasm zero-initializes its linear memory [215]. These segments are guaranteed to be live until released with `free_segment`.

MSWasm backends enforcing memory safety should ensure *temporal safety*. We considered two different semantics for this. A simple approach would require memory-safe implementations to trap immediately when a segment is accessed after it has been freed. However, this is often inefficient to implement — it adds overhead to the critical path of `free_segment` and forces synchronization between the allocator and embedding application thread. Instead, we propose to adopt the relaxed model of Kedia et al. [115]:

> *(Relaxed) temporal safety*: The backend guarantees a trap on any access to a segment *after it has been deallocated*. That is, the segment may remain accessible (and completely valid) for an unspecified amount of time after `free_segment` has been called, until the allocator reclaims the memory.

This definition allows backends to defer deallocation until the last possible moment, while still preserving temporal safety. Moreover, it allows us to efficiently support several different safe manual memory management systems [47, 115, 134] as further discussed later (§ 6.4.3).

6.3.4 Handle integrity

Since we model pointers with handles, code must be able to load and store handles from memory. Unlike C and C++, however, we do not provide a way to cast handles to integers (and back). This also means we cannot allow handles to be stored in the legacy Wasm linear memory

(we simply do not provide instructions to do so). Instead, MSWasm provides instructions for explicitly loading and storing handles from segment memory:

```
handle.segment_load(src: handle) -> handle
handle.segment_store(dst: handle, val: handle)
```

To ensure that a Wasm program cannot forge pointers (e.g., with the `i64.segment_store` and `handle.segment_load` instructions), MSWasm backends should enforce handle integrity:

> *Handle integrity*: The backend conceptually associates either the type `data` or the type `handle` to each handle-aligned location in the segment memory. (New segments are initialized to be entirely type `data`.) On each `segment_load` and `segment_store` instruction, it then preserves these types:
>
> - Storing data (handle) to a particular location updates the containing segment element's type to `data` (`handle`).
> - Loading a handle from a location of type `data` produces a *corrupted handle*, i.e., a handle with `isCorrupted=True`. Like the `NULL` handle, corrupted handles are *invalid* — loads and stores on corrupted handles are disallowed; the backend should (conceptually) check the `isCorrupted` bit on every segment load and store. However, corrupted handles can themselves be written to segments with `segment_store`; when storing such a handle, the value of the loaded element is preserved, as is the type — `data`. This allows us to efficiently support `memcpy`-like operations (see § 6.6.1).
> - Loading data from a location of type `handle` is also allowed, but results in an implementation-defined data value. Importantly, the restrictions above ensure that such a data value can never be used as (or turned back into) a valid handle.

An MSWasm backend that does not enforce handle integrity need not keep track of `data`/`handle` types in the segment memory, and likewise need not distinguish corrupted handles from valid

handles (need not keep track of `isCorrupted`); this should provide performance equivalent to existing Wasm. On the other hand, by enforcing handle integrity with the semantics described above, an MSWasm backend can ensure that *valid* handles can only be created during memory allocation or from existing valid handles with `segment_slice`, and furthermore that handles cannot be overwritten (even partially) in memory without becoming invalid.

6.4 Implementation strategies

MSWasm enables backend compilers and runtimes to enforce memory safety using a variety of hardware and software mechanisms. We review some of the promising current and future approaches.

6.4.1 Spatial safety

To ensure memory safety, MSWasm backends must ensure that all dereferenced handles are in-bounds (§ 6.3.1).

In software

Enforcing full spatial safety in software often imposes relatively high overheads. For instance, Baggy Bounds [3] reported average runtime overheads around 60% (highly varying by workload) and memory overheads around 15%. ManagedC [73] reports full spatial safety with runtime overheads around 15% on average, but is highly workload dependent and relies on an optimized just-in-time compiler (JIT), resulting in additional memory overheads.

Lower overheads can be achieved by relaxing certain safety properties. Delta Pointers [127] achieves around 35% average runtime overhead and negligible memory overhead, partly by ignoring buffer underflow errors and only detecting buffer overflow. Going further, Duck et al. [59] report overheads similar to Baggy Bounds for full spatial safety, but by omitting certain

checks (e.g., only checking writes), runtime overhead can be reduced below 10% [58]. However, even the fastest software schemes cannot match the efficiency possible with hardware support.

In hardware

The most promising and well-researched approach to strong spatial safety in hardware today is the Capability EnHanced RIsc (CHERI) system [222], which encodes spatial safety information in capability pointers — a fat pointer encoding that includes bounds information plus additional protection metadata. Current work suggests that overheads often in the low single digits are possible [50], and it seems likely that a production-quality processor could achieve even more impressive results. ARM recently announced plans to incorporate some of CHERI's ideas into future processors, and we feel optimistic about its prospects as the future of hardware-accelerated enforcement of full spatial safety.

Hardware tagged memory systems such as ARM MTE (§ 6.1.3) provide a weaker approach to spatial safety in the near term, but are very efficient. Using MTE, a MSWasm backend can ensure that the memory allocator assigns each allocation a different "color" (tag value) and ensure that adjacent allocations never share the same color. This is an incomplete solution, as many buffer overflows allow an attacker to address beyond adjacent objects. However, it does provide an efficient means of (probabilistically) detecting both spatial and temporal bugs. With 4-bit tags (as provided by ARM MTE), by randomly assigning a color to each allocation we can expect to detect both spatial and temporal bugs with a relatively high probability.

As MTE is only a mitigation, its security guarantees are not absolute, and it is unclear how much protection it provides against a determined attacker. For instance, an attacker may be able to brute-force the protection provided by randomly coloring allocations. In the end, the security benefits are highly dependent on details such as the particular type of vulnerability and how tags are assigned.

Finally, as discussed in § 6.1.3, Intel's MPX is already widely available, but is slower than comparable software solutions [166].

6.4.2 Handle integrity

As specified in § 6.3.4, full memory safety requires MSWasm backends to track the type of memory in segments, either `handle` or `data`. This can also be done in either software or hardware.

In software

Efficiently implementing handle integrity in software is challenging: the overheads of software tagged memory systems such as ASan [192] suggest that both memory and CPU overheads can easily be prohibitive. With enough type information, it seems possible to do better — e.g., ManagedC [73] achieves surprisingly modest overheads for full memory safety with the help of full C type information. Extending MSWasm with additional semantics to support a scheme like this might be worth exploring in future work.

In hardware

Hardware tagged memory is the most direct and efficient way to support handle integrity. CHERI uses tags to distinguish between pointers and data, as does lowRISC [138].

Unfortunately, hardware tagged memory implementations such as Sparc's ADI and ARM's MTE cannot easily be used to provide handle integrity because they provide tags for 16-byte (MTE) or 64-byte (ADI) regions of memory only. This means that each region of this size must have a single tag at any given point in time. Even if it were practical to ensure that every 16-byte or 64-byte region of memory contained either only handles or only data at all times — which is far from clear — this would require coordination between Wasm compilers and backends to, e.g., provide proper padding for structs in C (as a Wasm backend could not easily redo this padding on its own). Moreover, MSWasm would have to choose a granularity for this padding independent of backend, which would either exclude certain platforms (if it were too small) or incur unnecessary space overheads (if it were too large). For tagged-memory systems to be useful for pointer integrity, they must provide tags (of at least one bit) at the granularity of pointer-size or smaller. At present, no commercial hardware tagged memory implementation does this.

ARM PAC (see § 6.1.3) seems like a natural fit for providing handle integrity. Unfortunately, it has some limitations. First, the overhead of using PAC to protect all pointers (as MSWasm proposes) is around 20% [132], much worse than what memory tagging can provide. Further, storing a MAC in the upper bits of each pointer makes these bits unavailable for use in the fat pointer encodings required for many spatial safety approaches (see § 6.4.1). Thus, although PAC may be a promising mitigation for protecting function or vtable pointers, it is not well-suited for providing handle integrity.

6.4.3 Temporal safety

Temporal safety can be enforced in pure software, with the aid of virtual memory hardware, or using custom hardware designed for the purpose. MSWasm accommodates many recently proposed techniques by providing a separate segment memory and allocation interface (`new_segment` and `free_segment`) to support platform-specific allocators, and by slightly relaxing its temporal safety semantics (§ 6.3.3) to align with the guarantees these systems provide.

In pure software

Garbage collection is a traditional software-based solution for providing temporal safety. While an MSWasm implementation could employ garbage collection, recent systems provide substantially lower overheads while retaining manual memory management.

One approach, explored by DangSan [208] and other systems [129, 236], provides temporal safety by tracking all pointer aliases. When a pointer is freed, they rewrite its aliases to `NULL`. These systems still impose non-trivial overheads, e.g., DangSan incurs averages of 12%-41% runtime overhead, and 56%-140% memory overhead, on various single- and multi-threaded workloads (with results highly varying by workload, from 0% to over 700%). pSweeper [134] uses concurrent thread(s) to detect dangling pointers and avoids maintaining a precise points-to map; this results in lower runtime overheads than DangSan (12%-17% on average) with similar

memory overheads. More efficient approaches are possible with help from the virtual memory system.

Using the virtual memory system

Both OSCAR [47] and Project Snowflake from MSR [115, 172] leverage the virtual memory system to efficiently provide temporally safe manual memory management. Project Snowflake tracks when most of the objects on a page have been freed, then unmaps the page (so that future dereferences of dangling pointers will trap) while copying the remaining live objects to a new page and lazily patching references to them. This is conceptually similar to a copying GC, but compared to GC, they reduced peak working set size by $3\times$, and runtime overhead by $2\times$. OSCAR maps a unique virtual page for each allocation and unmaps on each deallocation. So long as no virtual addresses are re-used, accesses to freed objects will hit an unmapped page. This provides strong temporal safety with very low runtime overhead. pSweeper, OSCAR, and Project Snowflake all leverage the delayed-free semantics of MSWasm.

In pure hardware

Watchdog [160] shows that dedicated hardware support can provide both temporal and spatial safety efficiently. It incurs a runtime overhead of 18%, and memory overheads averaging 32%-56% (again highly workload-variant), while providing full spatial and temporal safety.

ARM MTE style tagging can also be used to enforce temporal safety. Specifically, each memory allocation can be assigned a random tag when allocated, and re-tagged with a new random tag when freed. This has the potential to detect use-after-free bugs efficiently enough to be used in production workloads. Unfortunately, this protection is probabilistic; an attacker could potentially easily brute-force this protection with, say, a 1/16 chance of success each time. Nonetheless, this presents an intriguing option in the security-performance tradeoff space which may be suitable for some applications.

6.5 Compiling to MSWasm

MSWasm, like Wasm, is intended to be a compilation target from higher level languages. We implement a compiler from C to MSWasm by extending the CHERI fork of Clang and LLVM [44]. CHERI modified LLVM to support fat pointers, which share many characteristics with MSWasm handles, and is thus a good starting point for MSWasm.

CHERI represents fat pointers at the LLVM IR level as pointers in a special, distinguished "address space"; pointers in this address space are lowered to CHERI capabilities in the appropriate LLVM backends. CHERI today only targets MIPS and RISC-V (with CHERI hardware extensions) backends; other backends, including the Wasm backend, are incompatible with CHERI's fat pointers. We modified the Wasm backend to emit MSWasm bytecode, lowering fat-pointer abstractions to MSWasm abstractions.

Global and Static Data

Our C-to-MSWasm compiler only emits handle-based load and store operations, resulting in MSWasm programs which do not use the linear memory at all. This provides additional safety guarantees (and implementation expediency) at the expense of some flexibility (e.g., we do not support integer-to-pointer casts, except for a few special cases like constant `0`). One consequence of this is that even global variables and static data need to be accessed via handles, and thus placed in the segment memory.[4] Our compiler emits instructions to allocate a segment for each LLVM global variable and store the corresponding handle in a Wasm global variable. When the target program needs a pointer to the global array, it simply retrieves the handle from the appropriate Wasm global variable.

Some global variables in C are themselves pointers, initialized via initialization expressions, and need to be pointing to valid, initialized memory at the beginning of the program.

[4]More precisely, global variables which the program never takes the address of, do not need this treatment, as we can compile them into Wasm globals; but global variables which the program does take the address of, such as global arrays, are accessed via pointers and thus must be located in the segment memory.

Our compiler generates the necessary information in the output `.wasm` file to instruct MSWasm compilers and runtimes to initialize certain segments at module initialization time.

C Stack

We compile part of the C stack to the segment memory. Specifically, stack variables whose address-of are taken and stack-allocated arrays cannot be placed on the (simple and safe) Wasm stack. Compilers from C to ordinary Wasm place these variables in the linear memory; our compiler places them in the segment memory.[5] We allocate a single large segment to represent stack memory for all of the variables which must be allocated in the segment memory; this means we have a single stack pointer, which we store in a dedicated Wasm global variable of type handle. Compared to using a separate segment for each stack allocation, our single-segment implementation is simpler (and faster) but trades-off some safety, e.g., we cannot prevent a stack buffer overflow from corrupting another stack-allocated buffer.

Standard library

Wasm programs which depend on `libc` need a Wasm-compatible implementation of `libc`. We modified WASI [158] to be compatible with MSWasm to the extent necessary for our benchmarks. Most importantly, we fully recompiled the WASI `libc` using our MSWasm compiler, in order to generate `libc` bytecode compatible with MSWasm. In our MSWasm version of the WASI `libc`, the implementations of `malloc` and `free` are completely replaced by trivial implementations consisting of the `segalloc` and `segfree` MSWasm instructions.

Implementation Effort

Our CHERI LLVM additions (in particular to its Wasm backend) and the WASI `libc`, amounted to approximately 1600 lines of code. While our compiler can target any MSWasm

[5]Stack variables which the program never takes the address of can be compiled to Wasm local variables, and data such as return addresses are never placed in the linear memory at all; Wasm implementations place them on a safe stack which is inaccessible to Wasm load and store instructions. The only stack variables which need to be placed in the linear memory, or for us the segment memory, are those we need pointers to.

backend, compiling general, real-world applications would likely require additional changes to WASI `libc`. We leave this to future work.

6.6 Discussion

In this section, we discuss some of the challenges with compiling MSWasm, how MSWasm can be used to further harden legacy code, alternative approaches to implementing memory safety for Wasm, and how future hardware mechanisms should shift to more efficiently support MSWasm and memory safety in general.

6.6.1 Compiling MSWasm

Handle sizing

We need to specify the size of handles at the Wasm level (independent of backends) so that compilers can target MSWasm. Handles are perhaps most naturally implemented as fat pointers, where bounds information is encoded directly in the pointer. We believe that 64-bit handles represent the optimal tradeoff: they are small enough to be efficient on modern architectures, while large enough to represent a handle (i.e., `base`, `bound`, `offset`) for Wasm's 32-bit address space when efficiently encoded [59, 127, 128]. Additionally, 64-bit handles are large enough to support most of the other schemes described above (for all safety properties).

Pointer semantics

The relationship between pointers and integer types in C is an important question for MSWasm which may impact compatibility with real-world C. The ISO C standard [108] (§6.2.3.2) is relatively strict as it defines casting integers to and from pointers as implementation-defined behavior or undefined behavior, with exception of `NULL` pointers (which MSWasm supports with the `NULL` handle). However, real-world C programs may rely on behavior beyond what is specified in the ISO standard.

With MSWasm the compiler can support some more liberal behaviors by modeling pointers with corrupted handles (see § 6.3.4) across casts in many situations, while MSWasm's strict rules on segment loads and stores ensure handle integrity is nevertheless always maintained. We believe our semantics maintains both broad compatibility with C programs and (for backends implementing the handle integrity checks) strong safety guarantees. An alternate semantics could allow more permissive casting between integers and handles while tracking provenance across integer types; however, this requires more invasive changes to Wasm's core semantics (e.g., to track `data`/`handle` types on the Wasm stack). Our concrete compiler discussed in Section 6.5 does not yet support most integer-to-pointer casts other than for `NULL` pointers.

Another challenge is supporting functions like `memcpy()` and `memmove()` that can copy both data and pointers (e.g., by casting pointers to integers). But, since our `handle.segment_load` and `handle.segment_store` preserve the data loaded even when operating on a non-handle, they can be used to implement `memcpy()` and preserve all data and type information when copying segments.

6.6.2 Beyond heap memory safety

Protecting the stack

MSWasm can be straightforwardly used to provide memory safety for heap allocations. However, the stack needs additional protection as well. C/C++-to-Wasm compilers like Emscripten [238, 239] currently *unsafely* store stack-allocated aggregate values like `struct` and `array` in linear memory, as Wasm stack and local variables can only hold scalars. To fully protect these values, we propose to store them in the segment memory. Much like Emscripten's current practice, the compiler could make a one-time large allocation for the entire stack. Then, when pointers to stack-allocated aggregate values are needed, the compiler can generate slices corresponding to those specific aggregates. These slices would not be temporally safe — they would remain valid after their target stack frame is popped — but would still be bounds checked

and provide pointer integrity. Previous studies suggest that this tradeoff is reasonable, as they have failed to find exploitable use-after-free vulnerabilities on the stack [28, 129]. Our concrete compiler discussed in Section 6.5 uses a single large allocation for the stack without slicing, providing weaker guarantees but still ensuring that stack pointers cannot be used to access the heap and vice versa.

Protecting Wasm function pointers

The Wasm spec makes it clear that Wasm has no function pointers. Instead, functions are accessed through a table, ensuring that one can only branch to a function entry point. However, the indexes to this table still live in memory, and are thus vulnerable to control flow bending attacks [33].

Extending MSWasm to protect function pointers is a relatively simple change. We could add a `function_index` type that is in other ways like a normal `i32` function index, but leverages segments for integrity protection similar to handles. Like for handles, MSWasm backends could trap on any attempt to use a `function_index` that has been overwritten by a non-`function_index` value.

PAC is an ideal fit for this task, as prior work puts its overhead for protecting both code pointers and return addresses at ($< 0.5\%$) on average [132].

6.6.3 Alternative paths to memory safety

Memory-safe languages

One way to ensure Wasm programs are memory-safe would be to write them in a memory-safe language like Rust, or safe variants of C (e.g., [61, 112]). This, unfortunately, is not realistic — developers want to deploy legacy C/C++ code to Wasm — and even memory-safe languages can benefit from the hardware-accelerated memory safety provided by MSWasm.

Leaving enforcement to compilers

Another option is to put the onus for enforcing memory safety solely on compilers or language runtimes, and leave Wasm agnostic to these concerns. This again is unrealistic: many state-of-the-art techniques for efficient memory safety rely on architecture- or OS-specific features, from temporal safety techniques that leverage page level protections [47, 115], to hardware features like ARM PAC and MTE. For Wasm backends to efficiently leverage these features, we need abstractions to express memory-safety properties directly in Wasm.

Object-based memory models

One way to encode memory-safety properties in Wasm would be to extend Wasm with a strongly typed, object-based memory model that relies on garbage collection, like those in the JVM and CLR. These kinds of memory models offer both strong spatial and temporal safety (through garbage collection). Such a model has been proposed as an extension to Wasm to support higher level GC'd languages [225]. However, mapping C/C++ to this model seems to be fundamentally inefficient. Garbage collection brings additional space and compute overheads that are unnecessary in languages with manual memory management [172]. Moreover, the type systems of these languages are also too restrictive to model real C/C++ code [73].

6.6.4 Future directions for hardware

Web platforms have a long history of waiting until security problems are out of control before starting to address them. Memory safety vulnerabilities are the most common and dangerous vulnerabilities of our time. It is absurd to assume these issues will not impact Wasm. Now is the time to start addressing this challenge.

In the future, Wasm standards bodies could benefit from engaging with the architecture community on how to best surface future memory safety capabilities in Wasm, and ensure the Wasm road map takes these features into account. Conversely, future hardware designs could benefit from considering how to add value through surfacing memory features in the Wasm ecosystem. Providing a standard IR for memory safety, such as MSWasm, provides a target for

hardware designers to work against and can play a critical role for enabling cooperation between these two communities.

Acknowledgments

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by an REMS EPSRC program grant (EP/K008528/1) and an EPSRC DTP award (EP/N509620/1).

Chapter 6, in part, is a reprint of the material as it appears in the Workshop on Hardware and Architectural Support for Security and Privacy (HASP), 2019. Disselkoen, Craig; Renner, John; Watt, Conrad; Garfinkel, Tal; Levy, Amit; Stefan, Deian. ACM, 2019. The dissertation author was the primary investigator and author of this material.

Chapter 6 also contains material currently under submission for publication. Michael, Alexandra; Gollamudi, Anitha; Disselkoen, Craig; Denlinger, Aidan; Bosamiya, Jay; Watt, Conrad; Parno, Bryan; Patrignani, Marco; Vassena, Marco; Stefan, Deian. The dissertation author was the primary investigator and author of the reprinted material.

Chapter 7

DMS: Deterministic Spatial Memory Safety with ARM MTE

As we have repeatedly seen in this dissertation, memory safety vulnerabilities are a serious problem for today's software. One obvious solution to this problem is to rewrite all of our security-critical code in memory-safe languages, such as Rust or Go. However, today's software systems are enormous, and rewriting them in a new language is an onerous proposition. For instance, Chromium alone contains over 14 million lines of C and C++ [201]; rewriting it would be prohibitively expensive and time-consuming.

A much more attractive solution is to automatically enforce memory safety directly for unmodified C and C++ code. Proposals such as CCured [163] and SoftBound [161] provide deterministic memory safety for C code using fully automated program transformations. By enforcing memory safety automatically and invisibly to the programmer, we reap the security benefits without incurring the tremendous cost of rewriting in a safe language.

But automatically enforcing memory safety for C and C++ incurs a large performance overhead — these languages weren't originally designed for memory safety enforcement. For

instance, SoftBound reports an average runtime overhead of 67% on selected SPEC2000 benchmarks written in C [161].

One way to improve the performance of these solutions is with dedicated hardware support. Hardware manufacturers have recently introduced or proposed many new features intended to help in the battle against memory safety vulnerabilities. The ARMv8.5-A specification includes a Memory Tagging Extension (MTE) [72], and ARM is also working on a prototype product called Morello which incorporates concepts from the CHERI capability architecture [11]. Similar to ARM MTE is Oracle's Application Data Integrity (ADI) feature introduced with the SPARC M7 [1]. Intel recently experimented with Memory Protection Extensions (MPX) [166] (now deprecated), has introduced Memory Protection Keys (MPK) [103], and even more recently proposed Cryptographic Capability Computing (C3) [130]. All of these features aim to facilitate low-overhead enforcement of (some aspect of) memory safety without requiring changes to application code.

Unfortunately, MTE, ADI, and C3 provide only incomplete protection from memory safety vulnerabilities — their security guarantees are only probabilistic. We propose DMS, a compiler-based defense which builds on top of MTE in order to achieve fully deterministic spatial memory safety enforcement. DMS identifies which memory accesses are provably (deterministically) safe, and which accesses may "slip through the cracks" of MTE's probabilistic enforcement. Then, DMS inserts software checks at each insecure memory access. Together, MTE's hardware enforcement and DMS's software checks provide fully deterministic spatial safety for legacy applications.

7.1 Background

We give a brief primer on memory safety concepts, on the new ARM MTE hardware extension, and on the natural way to use MTE to achieve (probabilistic) memory safety.

7.1.1 Memory safety

As we saw in Chapter 6, memory safety can be viewed as having three components: *spatial safety*, *temporal safety*, and *pointer integrity*. Spatial safety refers to the absence of out-of-bounds or buffer-overflow vulnerabilities; temporal safety refers to the absence of use-after-free or double-free vulnerabilities; and pointer integrity refers to the inability for attackers to forge valid pointers (e.g., by casting from integers).

Although all three components of memory safety are necessary for secure programs, in this work we follow many other previous efforts (e.g., [3, 127]) by focusing on mechanisms for providing spatial safety.

7.1.2 Probabilistic enforcement with ARM MTE

To help combat memory safety problems, ARM introduced its Memory Tagging Extension (MTE) [12]. MTE adds a 4-bit hardware tag to each 16-byte (aligned) *granule* of memory. As 4 bits provides 16 possible tag values, we refer to these tag values as *colors*; each granule of memory has one of the 16 possible colors. MTE also repurposes 4 otherwise-unused bits in the top byte of the pointer representation (virtual address) as a tag value representing the color of the pointer. Pointer arithmetic preserves the pointer's color — i.e., adding 4 to a pointer with tag value `0100` results in a pointer that also has the tag value `0100`. On every pointer dereference (load or store), the hardware compares the pointer's tag value (color) to the tag value (color) of the memory being accessed, and if they do not match, a fault is generated.

MTE hardware naturally lends itself to providing low-overhead memory-safety enforcement, as described in ARM's own whitepaper [12]. In the natural scheme, an MTE-aware memory allocator chooses a color (perhaps randomly) for each new memory allocation. The allocator colors all the bytes of the memory allocation with that color, and returns a pointer which also has that color. This alone is already sufficient to provide probabilistic spatial and temporal safety:

For instance, suppose a pointer with color `C` is incremented to point out-of-bounds of its intended memory allocation. With high probability, the out-of-bounds memory will have a color other than `C`, so when the pointer is dereferenced, the colors will mismatch, causing a fault. Or suppose a memory region with color `C` is freed and reallocated. If an old (dangling) pointer with color `C` attempts to access the now-reallocated memory, with high probability the memory now has a color other than `C`, so the colors will mismatch, causing a fault.

Unfortunately, the protection provided by this natural scheme is only probabilistic — it doesn't guarantee that it catches all memory safety violations. For instance, in both of the above examples, there remains a chance that the colors will still match, even for accesses that should be rejected. This is even more concerning in the presence of active adversaries, who may be able to engineer these color collisions, either by repeated attempts (brute force) or by carefully exploiting gleaned information about the colors of other (targeted) memory allocations.

With some small but careful adjustments to the allocator, we can catch some classes of memory safety problems deterministically:[1]

- ***Reserved free color***

  We can reserve one of MTE's 16 colors to mean "unallocated or freed"; during the `free` operation, the allocator can re-color the freed memory to this reserved color. This way, we deterministically catch any access to freed (but not-yet-reallocated) memory.

- ***Coloring adjacent allocations***

  We can also add checks in the memory allocator to ensure that it always chooses a color for new allocations that is different from the colors of allocations immediately adjacent on both sides. This way, we deterministically catch all "small overflows" which are out-of-bounds merely into the immediately adjacent allocation.

- ***Coloring reallocations***

[1]The MTE whitepaper alludes to at least one of these: "With careful software design, sequential safety violations where memory is accessed immediately before or after the true bounds can always be detected. 'Wild' violations to arbitrary locations in the address space can be detected probabilistically."

> By keeping some additional state in the memory allocator, we can ensure that each new allocation is always assigned a color which is different from the color that allocation was assigned immediately previously, i.e., before the most recent time it was freed. This way, we deterministically catch use-after-frees where the memory has been reallocated only once, i.e., use-after-frees where the dangling pointer is only one generation stale.

In all of these cases, although we deterministically catch some classes of memory safety problems, protection in the general case remains only probabilistic. These measures do not increase our protection against arbitrary out-of-bounds errors (e.g., completely attacker-controlled offsets into an array) or arbitrary use-after-frees (e.g., when the memory may be freed and reallocated many times). And this is important: according to Microsoft, 60% of today's spatial safety exploits involve out-of-bounds reads/writes to nonadjacent memory [151]. The goal of DMS is to provide deterministic spatial safety in *all* cases, not just cases representing low-hanging fruit.

7.2 Deterministic spatial safety with DMS

We present DMS, a system which builds on top of ARM MTE to provide fully deterministic spatial safety. DMS rests on the observation that in the presence of MTE, many pointer dereferences can be statically proved to be *spatially safe*. For this purpose, spatially safe includes both (1) pointers that are statically guaranteed to be in-bounds, but also (2) pointers for which we know MTE will deterministically catch the violation if they are out-of-bounds. As a simple example of the second category, consider a pointer known to be *at most* 16 bytes out-of-bounds. Assuming the carefully-coded allocator described in Section 7.1.2, MTE will deterministically catch out-of-bounds dereferences of this pointer: The pointer must be either still in-bounds, or point to the immediately adjacent allocation, which must be a different color.

DMS supplements the checks already performed by MTE with additional safety checks, implemented in software, for the cases where it cannot prove that a dereference is spatially safe —

i.e., the cases where MTE's protection is only probabilistic. The result is fully deterministic spatial safety through a combination of software and hardware checks. Compared to the natural use of MTE, this trades performance for deterministic security guarantees; but compared to previous spatial memory safety systems implemented purely in software, this greatly reduces the number of software checks required (thanks to DMS's partial reliance on MTE).

7.2.1 Pointer classifications

More precisely, DMS conceptually classifies all pointers into three categories (noting that the same pointer may have different classifications at different program points):

- ***Clean***

  These pointers are statically guaranteed to be in-bounds of the intended allocation. Clean pointers include, e.g., pointers which have not been modified since they were returned from `malloc` or similar; pointers to stack allocations or global variables which are known to be in-bounds; or pointers which have not been modified since they were last dereferenced. (Inductively, since DMS will deterministically cause a fault when an out-of-bounds pointer is dereferenced, if the program has continued executing after any dereference, we can assume the pointer is clean at all program points after the dereference.) Clean pointers may be safely dereferenced without any additional DMS safety checks.

- ***Blemished***

  These pointers are statically guaranteed to be modified by at most 16 bytes since they were last known to be clean. Blemished pointers arise when pointer arithmetic is done on clean pointers — specifically, when the program adds or subtracts small compile-time constants to clean pointers. Blemished pointers may be safely dereferenced without any additional DMS safety checks: If a blemished pointer is truly out-of-bounds, MTE will deterministically catch the violation.

► ***Dirty***

This category includes all pointers that can't be statically proved to be clean or blemished. Dirty pointers arise in a number of ways, such as pointer arithmetic with non-compile-time constants (e.g., dynamic array indices); pointers loaded from memory, in certain circumstances; or other cases where static analysis is insufficient to prove the pointer is clean or blemished.

Classifying pointers to struct elements

Accessing struct fields is one common purpose of pointer arithmetic, and often the constant offset from the struct base to the field is larger than DMS's threshold for blemished. However, given a valid (and clean) pointer to a struct, if DMS is allowed to assume that the pointed-to allocation is indeed at least the size of that struct, then all accesses to struct fields must be inbounds — i.e., pointers to fields of that struct can also be considered clean. The only way this could be potentially unsafe in practice, is if a program casts a pointer to a small allocation, into a pointer type implying it points to a struct larger than that allocation. DMS includes a configuration option, `trust_llvm_struct_types`, which determines whether DMS's analysis is allowed to assume that, given a clean pointer to a struct, pointers to that struct's fields are also clean.

Classifying pointers loaded from memory

One significant challenge with DMS's pointer classification is how to classify pointers which are loaded from memory. Conservatively, we could consider all such pointers to be dirty; unfortunately, this would be detrimental to DMS's performance, which relies on safely eliminating as many safety checks as possible (via classifying pointers clean or blemished). Instead, DMS uses a modified pointer representation for pointers residing in memory. DMS reserves two bits in the upper part of the pointer representation (virtual address) to represent whether the pointer value residing in memory was clean, blemished, or dirty at the time when it was stored to memory. Whenever a pointer is stored to memory, DMS sets those bits to indicate its status; and whenever

a pointer is loaded from memory, DMS checks those bits to learn its status. (DMS then obtains the correct pointer value by clearing these bits, since user-mode virtual addresses should contain all zeroes in the upper pointer bits.) This means that many pointers cannot be statically classified clean, blemished, or dirty, but instead have a *dynamic* classification. However, checking the dynamic classification of a pointer is much cheaper than performing the full SW safety check for the pointer: Merely checking the classification does not require an additional memory access, while the full safety check often requires an additional memory access, particularly for these pointers, as we'll discuss later. By checking the dynamic classification first, DMS saves significant time for dynamically clean or blemished pointers by avoiding the full SW bounds check.

Classifying pointers used as function arguments or return values

Another challenge is how to classify pointers which are used as function arguments or return values. A purely intraprocedural analysis cannot fully classify these pointers at compile time. One option is to conservatively classify all of these pointers as dirty, but this comes at a performance cost, as fewer safety checks could be eliminated. Another option is to use the same encoding as for pointers loaded from memory (discussed above). A third option would be to use link-time optimization (LTO) and interprocedural analysis to determine much more precise pointer classifications statically; but, the implementation would need to make careful tradeoffs to ensure the compile-time analysis still scales to large programs.

As of this writing, our DMS implementation is not yet complete in this respect.

Classifying pointers or array accesses inside loops

A naive application of the above pointer-classification rules would often lead to repetitive bounds checks inside loops, harming performance. Consider this simple example:

```
int sum;
for (int i = 0; i < 256; i++) {
    sum += arr[i];
}
```

Suppose the array `arr` is classified clean upon entry to the loop. The loop accesses `arr[i]`, which in C is a syntactic sugar for `arr + i*sizeof(int)`. Unfortunately, by the above rules this pointer value would be classified dirty, because it involves pointer arithmetic with non-compile-time constants, namely the index `i`.

However, DMS can prove via induction that none of the accesses in this loop need additional safety checks. The access in the first iteration, to `arr[0]`, is easily determined to be safe: Since `arr` is known to be clean upon entry to the loop, so must be the pointer `arr + 0*sizeof(int)`, which is the same as `arr`. In subsequent iterations, we know that the previous iteration dereferenced `arr[i-1]`, which means that `arr[i-1]` is clean according to the above rules. But `arr[i]` is only `sizeof(int)` bytes greater than `arr[i-1]` — so DMS can safely conclude that `arr[i]` is blemished and does not need an additional safety check. Thanks to this proof by induction, DMS can avoid adding any additional safety checks to the body of this loop.

DMS's induction reasoning is currently fairly unsophisticated, and only catches some simple (but important) cases. Our current implementation can reason about loops where a pointer is incremented by the same constant every iteration, and loops where an array index is incremented by the same constant every iteration. Importantly, we cannot apply this induction optimization unless we know that the pointer or array index in question *must be* dereferenced during *every* loop iteration; our current implementation again handles common cases of this check and when in doubt conservatively marks the pointer dirty.

7.2.2 Safety checks for dirty pointers

When dirty pointers are dereferenced, MTE provides only probabilistic spatial safety guarantees. Thus, DMS supplements MTE by inserting bounds checks in software to achieve full deterministic protection. This part of DMS's design is heavily inspired by SoftBound [161]. The main design question is how DMS obtains bounds information for each dirty pointer in order to perform standard software bounds checks.

Pointer creation and arithmetic

C pointers are created in two main ways: either via `malloc()`, or by taking the address of a stack or global variable. In both cases, DMS can trivially determine the pointer's bounds upon creation: For `malloc()`, the bounds of the resulting pointer correspond to the start and end of the allocation, while for stack and global variables, the size is known at compile time. Then, like SoftBound, DMS propagates the bounds information associated with a pointer during pointer assignment or pointer arithmetic. Pointer arithmetic does not change the bounds of the pointer, and also does not require any bounds checks; like many previous systems including SoftBound, DMS allows out-of-bounds pointers to exist (crucial for compatibility with some C idioms) and performs checks only when the pointer is dereferenced. In all of these cases, the bounds information is available statically, so the bounds checks are cheap; the limits can be checked with simple comparisons and no additional memory accesses.

Storing and loading pointers from memory

The more interesting design question is how to propagate bounds information for pointers which are stored and loaded from memory. Here, like SoftBound, DMS uses a (dynamic) global table which maps the memory location where a pointer is stored, to the bounds information for the pointer stored there. For each store operation that stores a pointer, DMS inserts instructions to store that pointer's bounds information to the global table; and if a bounds check is required for a pointer that was loaded from memory (or derived from a pointer loaded from memory), DMS likewise inserts instructions to load its bounds information from the global table. DMS only needs to store bounds information to this table for pointers which are actually stored to memory; for pointers which remain in registers, bounds information can be propagated statically without dynamic lookups.[2]

[2]In fact, DMS can also propagate bounds information statically for pointers which are merely spilled and unspilled from memory at the register allocation stage, due to register contention. At the LLVM IR level where DMS operates, these spills are invisible. Only non-spill memory operations with pointers require DMS to use the dynamic global table.

Pointers used as function arguments or return values

Similar to pointer classification, we again face a challenge with how to propagate bounds information for pointers used as function arguments or return values. As of this writing, our DMS implementation is not yet complete in this respect. However, we plan to follow SoftBound, keeping our analysis intraprocedural and adjusting function signatures to dynamically propagate this information at runtime via the C stack. This should be more efficient than reusing the dynamic global bounds table for this bounds information, which would require a table access both in the caller (to store the information) and in the callee (to retrieve the information).

7.3 Conclusion

Our implementation of DMS demonstrates how new hardware primitives such as ARM MTE can be fruitfully combined with software checks to provide strong security guarantees for legacy applications, without incurring the same performance overhead as previous purely-software proposals. As hardware memory tagging becomes available on more platforms and from more vendors, DMS represents a promising path to achieving deterministic memory safety for today's applications.

Acknowledgments

We thank Michael LeMay, Shravan Narayan, and Alexandra Michael for helpful discussions about this work.

Chapter 7, in part, is currently being prepared for submission for publication of the material. Disselkoen, Craig; Tullsen, Dean; Stefan, Deian. The dissertation author was the primary investigator and author of this material.

Conclusion

Memory safety vulnerabilities are an ongoing and serious problem for today's systems, and we can't simply wait around for new hardware or new programming languages to solve the problem for us. Instead, we need solutions that provide security for today's applications on today's hardware.

In this dissertation we first looked at memory safety vulnerabilities made possible by hardware features, namely Intel TSX (Chapter 1) and microarchitectural side-channels (Chapter 2 and Chapter 3). We demonstrated both the seriousness of these vulnerabilities, and the prevalence of these vulnerabilities in widely-used software. We presented tools capable of automatically finding vulnerabilities of several important kinds.

Then, we turned our attention from *finding* memory-safety vulnerabilities in existing code, to automatically *preventing* them in today's languages and on today's architectures. We presented a novel systematization of software defenses aginst Spectre attacks (Chapter 4), and went on to propose our own defense in this category (Chapter 5), incorporating many of the lessons learned from this systematization. Then, we proposed software solutions for providing greater memory safety for Wasm programs (Chapter 6), and for C/C++ programs with the help of new hardware features such as ARM MTE (Chapter 7). Our work demonstrates a promising path towards increased memory safety for today's applications on today's (and tomorrow's) architectures.

Bibliography

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, et al. M7: Oracle's next-generation SPARC processor. *IEEE Micro*, 35(2):36–45, 2015.

[2] Sam Ainsworth and Timothy M Jones. MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 2020.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. USENIX Security Symp.*, 2009.

[4] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2013.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2017.

[6] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Proc. USENIX Security Symp.*, 2016.

[7] AMD. Software techniques for managing speculation on AMD processors. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018.

[8] AMD. Security analysis of AMD predictive store forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf, 2020.

[9] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2015.

[10] Siena Anstis, Ron Deibert, Miles Kenyon, and John Scott-Railton. The dangerous effects of unregulated commercial spyware. https://citizenlab.ca/2019/06/the-dangerous-effects-of-unregulated-commercial-spyware/, 2019.

[11] ARM. Arm Morello program. https://www.arm.com/why-arm/architecture/cpu/morello.

[12] ARM. Armv8.5-A Memory Tagging Extension: White paper. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[13] ARM. Mbed Crypto. https://github.com/ARMmbed/mbed-crypto, 2019.

[14] ARM. Mbed TLS. https://github.com/armmbed/mbedtls, 2019.

[15] ARM. Straight-line speculation. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/downloads/straight-line-speculation, 2020.

[16] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2014.

[17] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM on Programming Languages*, 4(POPL), December 2019.

[18] Gilles Barthe, Sunjay Cauligi, Benjamin Gregoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the Spectre era. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2021.

[19] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *Proc. IEEE Computer Security Foundations Symp.*, CSF, 2018.

[20] Daniel J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[21] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. SpecROP: Speculative exploitation of ROP chains. In *Proc. Symp. on Research in Attacks, Intrusions, and Defenses*, RAID, 2020.

[22] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2019.

[23] Roderick Bloem, Swen Jacobs, and Yakir Vizel. Efficient information-flow verification under speculative execution. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 499–514, Cham, 2019. Springer.

[24] Google Security Blog. Mitigating Spectre with Site Isolation in Chrome. https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html, July 2010.

[25] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *Proc. USENIX Security Symp.*, 2020.

[26] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[27] Bytecode Alliance. Cranelift code generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift, 2020.

[28] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proc. ACM Intl. Symp. on Software Testing and Analysis*, ISSTA, 2012.

[29] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX Conf. on Operating Systems Design and Implementation*, OSDI, 2008.

[30] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2019.

[31] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of defenses against transient-execution attacks. In *Great Lakes Symposium on VLSI*, 2020.

[32] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proc. USENIX Security Symp.*, 2019.

[33] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. USENIX Security Symp.*, 2015.

[34] Chandler Carruth. RFC: Speculative load hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[35] Chandler Carruth. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2019.

[36] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2020.

[37] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Gregoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2019.

[38] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2012.

[39] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *Proc. IEEE Computer Security Foundations Symp.*, CSF, 2019.

[40] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[41] Chromium Security. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety.

[42] Robert J Colvin and Kirsten Winter. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In *Proc. Intl. Symp. on Formal Methods*, FM, 2019.

[43] Cryptography Coding Standard. Coding rules. https://cryptocoding.net/index.php/Coding_rules, 2016.

[44] CTSRD-CHERI. The CHERI LLVM compiler infrastructure. https://github.com/CTSRD-CHERI/llvm-project, 2022.

[45] CVE Details. CVE Details: The ultimate security vulnerability datasource. https://www.cvedetails.com.

[46] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 2007.

[47] Thurston H. Y. Dang, Petros Maniatis, and David A. Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *Proc. USENIX Security Symp.*, 2017.

[48] Lesly-Ann Daniel, Sebastian Bardin, and Tamara Rezk. Hunting the haunter — efficient relational symbolic execution for Spectre with Haunted RelSE. In *Proc. Network and Distributed System Security Symp.*, NDSS, 2021.

[49] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2020.

[50] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.

[51] Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. The meaning of memory safety. arXiv:1705.07354, 2017.

[52] Frank Denis. libsodium. https://github.com/jedisct1/libsodium.

[53] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[54] Alexis Deveria. Can I use WebAssembly? https://caniuse.com/#feat=wasm, 2019.

[55] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. The influence of malloc placement on TSX hardware transactional memory. https://arxiv.org/pdf/1504.04640.pdf, 2015.

[56] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. The code that never ran: Modeling attacks on speculative evaluation. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2019.

[57] Daniel Donenfeld. More Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/, 2020.

[58] Gregory J. Duck. LowFat. https://github.com/GJDuck/LowFat.

[59] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proc. Intl. Conf. on Compiler Construction*, CC, 2016.

[60] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proc. ACM Intl. Symp. on Software Testing and Analysis*, ISSTA, 2009.

[61] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *IEEE Cybersecurity Development*, SecDev, 2018.

[62] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *Proc. Design Automation Conf.*, DAC, 2020.

[63] Matt Fleming. A thorough introduction to eBPF. *Linux Weekly News*, 2017.

[64] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, USA, 2010.

[65] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An efficient data-centric defense mechanism against Spectre attacks. In *Proc. Design Automation Conf.*, DAC, 2019.

[66] GCC Wiki. Intel Memory Protection Extensions (Intel MPX) support in the GCC compiler. http://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler, 2018.

[67] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1), 2018.

[68] Jay L Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 1988.

[69] Michael Godfrey. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, 2013.

[70] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2020.

[71] Google. Google Chrome Native Client SDK release notes. https://developer.chrome.com/native-client/sdk/release-notes.

[72] Matthew Gretton-Dann. Arm A-Profile architecture developments 2018: Armv8.5-A, Sep 2018. https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a.

[73] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Memory-safe execution of C on a Java VM. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS, 2015.

[74] Richard Grisenthwaite. Supporting the UK in becoming a leading global player in cybersecurity. https://community.arm.com/blog/company/b/blog/posts/supporting-the-uk-in-becoming-a-leading-global-player-in-cybersecurity, 2019.

[75] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *Proc. Conf. on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA, 2016.

[76] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: automating attacks on inclusive last-level caches. In *Proc. USENIX Security Symp.*, 2015.

[77] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2015.

[78] Roberto Guanciale, Musard Balliu, and Mads Dam. InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2020.

[79] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: alias-driven attacks and verified countermeasures. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2016.

[80] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2020.

[81] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2021.

[82] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2011.

[83] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative symbolic execution for cache timing leak detection. In *Proc. Intl. Conf. on Software Engineering*, ICSE, 2020.

[84] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2017.

[85] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proc. Intl. Conf. on Internet of Things Design and Implementation*, IoTDI, 2019.

[86] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation Intel Core processor. *IEEE Micro*, 34(2):6–20, 2014.

[87] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *ACM SIGARCH Computer Architecture News*, 32(2):102, 2004.

[88] Brook Heisler and Jorge Aparicio. Criterion.rs: Statistics-driven microbenchmarking in rust. https://crates.io/crates/criterion, 2020.

[89] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 1993.

[90] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, Mar 2019.

[91] Jann Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[92] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4):233–254, 1992.

[93] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *Proc. Intl. Conf. on Cryptographic Hardware and Embedded Systems*, CHES, 2016.

[94] Intel. Desktop 4th generation Intel Core processor family, desktop Intel Pentium processor family, and desktop Intel Celeron processor family: specification update. Revision 036US, page 67.

[95] Intel. Improving real-time performance by utilizing Cache Allocation Technology. Technical report, Intel Corporation, 2015.

[96] Intel. ARK | your source for Intel product specifications, Jan 2017. https://ark.intel.com.

[97] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[98] Intel. Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, June 2018.

[99] Intel. Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass, 2018.

[100] Intel. Deep dive: Intel analysis of microarchitectural data sampling, 2019.

[101] Intel. An optimized mitigation approach for load value injection. https://software.intel.com/security-software-guidance/insights/optimized-mitigation-approach-load-value-injection, 2020.

[102] Intel. Side channel mitigation by product CPU model. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model, 2020.

[103] Intel. Intel 64 and IA-32 architectures software developer's manual, 2021.

[104] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: a shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2015.

[105] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *Euromicro Conf. on Digital System Design*, DSD, 2015.

[106] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2015.

[107] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. BranchSpec: information leakage attacks exploiting speculative branch instruction executions. In *Proc. Intl. Conf. on Computer Design*, ICCD, 2020.

[108] ISO. Information technology – programming languages – C. Standard, International Organization for Standardization, June 2018.

[109] Jan Jancar. Minerva: A ladder has no windows but can still leak. https://minerva.crocs.fi.muni.cz/, 2019.

[110] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2016.

[111] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J Peter Brady, Sergey Bratus, and Sean W Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *Proc. Symp. on Hot Topics in the Science of Security*, HotSos, 2020.

[112] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conf.*, ATC, 2002.

[113] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient tagged memory. In *Proc. Intl. Conf. on Computer Design*, ICCD, 2017.

[114] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proc. Design Automation Conf.*, DAC, 2016.

[115] Piyus Kedia, Manuel Costa, Dimitrios Vytiniotis, Matthew Parkinson, Kapil Vaswani, and Aaron Blankstein. Simple, fast and safe manual memory management. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, June 2017.

[116] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *Proc. Design Automation Conf.*, DAC, 2019.

[117] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proc. USENIX Security Symp.*, 2012.

[118] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *Proc. ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO, 2018.

[119] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv:1807.03757*, 2018.

[120] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *Proc. USENIX Security Symp.*, 2021.

[121] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*. Springer, 1996.

[122] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[123] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2019.

[124] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *Proc. USENIX Security Symp.*, 2016.

[125] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *Proc. USENIX Workshop on Offensive Technologies*, WOOT, 2018.

[126] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCFI: Mitigating Spectre attacks using CFI informed speculation. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2020.

[127] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proc. European Conf. on Computer Systems*, EuroSys, 2018.

[128] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2013.

[129] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proc. Network and Distributed System Security Symp.*, NDSS, 2015.

[130] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic capability computing. In *Proc. ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO, 2021.

[131] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against Spectre attacks. In *Proc. IEEE Symp. on High-Performance Computer Architecture*, 2019.

[132] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. http://arxiv.org/abs/1811.09189, 2018.

[133] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proc. USENIX Security Symp.*, 2018.

[134] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2018.

[135] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. IEEE Symp. on High-Performance Computer Architecture*, 2016.

[136] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2015.

[137] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *Proc. USENIX Security Symp.*, 2021.

[138] lowRISC. lowRISC: A fully open-sourced, linux-capable, system-on-a-chip. https://www.lowrisc.org/.

[139] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2010.

[140] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2018.

[141] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proc. Computer Security Applications Conf.*, ACSAC, 2019.

[142] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In *Proc. European Symp. on Research in Computer Security*, EuroS&P, 2021.

[143] Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. Security best practices for developing Windows Azure applications. Technical report, Microsoft Corp., 2010.

[144] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 2012.

[145] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Proc. Symp. on Research in Attacks, Intrusions, and Defenses*, RAID, 2015.

[146] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. WebAssembly: A new world of native exploits on the browser. In *Blackhat briefings 2018*, 2018.

[147] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.

[148] Tyler McMullen. Lucet: A compiler and runtime for high-concurrency low-latency sandboxing. Principles of Secure Compilation (PriSC), 2020.

[149] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. In *Proc. ACM Symp. on Principles of Programming Languages*, POPL, 2019.

[150] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2016.

[151] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://www.youtube.com/watch?v=PjbGojjnBZQ, 2019. BlueHat.

[152] Bodo Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. https://www.openssl.org/~bodo/tls-cbc.txt, 2004.

[153] Daniel Moghimi. Data sampling on MDS-resistant 10th Generation Intel Core (Ice Lake). *arXiv:2007.07428*, 2020.

[154] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *Proc. USENIX Security Symp.*, 2020.

[155] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *Proc. USENIX Security Symp.*, 2020.

[156] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2012.

[157] Mozilla. Network Security Services. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS, 2019.

[158] Mozilla. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 2019.

[159] Security/sandbox. https://wiki.mozilla.org/Security/Sandbox, 2018.

[160] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 2012.

[161] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2009.

[162] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *Proc. USENIX Security Symp.*, 2021.

[163] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. ACM Symp. on Principles of Programming Languages*, POPL, 2002.

[164] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector at the SMT competition 2019. In *Proc. Intl. Workshop on Satisfiability Modulo Theories*, SMT, 2019.

[165] Node.js Foundation. Node.js. https://nodejs.org/en/, 2019.

[166] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. *Proc. ACM on Measurement and Analysis of Computing Systems*, 2(2), June 2018.

[167] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *Proc. USENIX Security Symp.*, 2020.

[168] OpenSSL. Security policy. https://www.openssl.org/policies/secpolicy.html, 2019.

[169] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: practical cache attacks in JavaScript and their implications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2015.

[170] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proc. Cryptographers' Track at the RSA Conf. on Topics in Cryptology*, CT-RSA, 2006.

[171] Andrew Pardoe. Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.

[172] Matthew Parkinson, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Aaron Blankstein, Dylan McDermott, Jonathan Balkind, and Dimitrios Vytiniotis. Project Snowflake: Non-blocking safe manual memory management in .NET. Technical report, Microsoft, July 2017.

[173] Marco Patrignani and Marco Guarnieri. Exorcising Spectres with secure compilers. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2021.

[174] Colin Percival. Cache missing for fun and profit. BSDCan, 2005.

[175] Hernán Ponce de León and Johannes Kinder. Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2022.

[176] Thomas Pornin. Why constant-time crypto? https://www.bearssl.org/constanttime.html, 2016.

[177] Thomas Pornin. Constant-time toolkit. https://github.com/pornin/CTTK, 2018.

[178] Project Zero. Examining pointer authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.

[179] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic Web applications in WebAssembly. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2019.

[180] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering Spectre gadgets. In *Proc. Network and Distributed System Security Symp.*, NDSS, 2021.

[181] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2002.

[182] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proc. USENIX Security Symp.*, August 2015.

[183] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proc. USENIX Security Symp.*, 2015.

[184] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *Proc. USENIX Security Symp.*, 2019.

[185] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via Intel/AMD micro-op caches. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 2021.

[186] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1), 2003.

[187] Gururaj Saileshwar and Moinuddin K Qureshi. CleanupSpec: An “undo” approach to safe speculation. In *Proc. ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO, 2019.

[188] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating spectre. In *Proc. Network and Distributed System Security Symp.*, NDSS, 2020.

[189] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2019.

[190] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *Proc. European Symp. on Research in Computer Security*, EuroS&P, 2019.

[191] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. Specfuscator: Evaluating branch removal as a Spectre mitigation. In *Intl. Conf. on Financial Cryptography and Data Security*, FC, 2021.

[192] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proc. USENIX Annual Technical Conf.*, ATC, 2012.

[193] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves C/C++ memory safety. http://arxiv.org/abs/1802.09517, 2018.

[194] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Intl. Workshop on Hardware and Architectural Support for Security and Privacy*, HASP, 2019.

[195] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[196] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with Venkman. *arXiv:1903.10651*, 2019.

[197] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2016.

[198] Signal. Signal protocol C library. https://github.com/signalapp/libsignal-protocol-c, 2019.

[199] Juraj Somorovsky. Curious padding oracle in OpenSSL (CVE-2016-2107). https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html, 2016.

[200] Marius Sternberger. Spectre-NG: An avalanche of attacks. In *Wiesbaden Workshop on Advanced Microkernel Operating Systems*, WAMOS, 2018.

[201] Synopsys Black Duck Open Hub. Chromium (Google Chrome). https://www.openhub.net/p/chrome/analyses/latest/languages_summary.

[202] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.

[203] Vadim Tkachenko. 20-30% performance hit from the Spectre bug fix on Ubuntu. https://www.percona.com/blog/2018/01/23/20-30-performance-hit-spectre-bug-fix-ubuntu/, Jan 2018.

[204] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(1):37–71, January 2010.

[205] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *ACM SIGARCH Computer Architecture News*, 23(2):392–403, 1995.

[206] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. USENIX Security Symp.*, 2018.

[207] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2020.

[208] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable use-after-free detection. In *Proc. European Conf. on Computer Systems*, EuroSys, 2017.

[209] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, May 2019.

[210] Kenton Varda. WebAssembly on Cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, Dec 2018.

[211] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with Blade. In *Proc. ACM Symp. on Principles of Programming Languages*, POPL, 2021.

[212] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine-grained timers in Xen. In *Proc. ACM Cloud Computing Security Workshop*, CCSW, 2011.

[213] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. *SIGPLAN Not.*, 39(6):231–242, June 2004.

[214] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. BRB: Mitigating branch predictor side-channels. In *Proc. IEEE Symp. on High-Performance Computer Architecture*, 2019.

[215] W3C. WebAssembly core specification. https://webassembly.github.io/spec/core/bikeshed/index.html#data-segments%E2%91%A0, 2019.

[216] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. ACM Symp. on Operating System Principles*, SOSP, 1993.

[217] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(3):1–31, 2020.

[218] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against Spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[219] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proc. European Conf. on Computer Systems*, EuroSys, 2014.

[220] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proc. Intl. Symp. on Computer Architecture*, ISCA, 2007.

[221] Wasmer. Wasmer - universal WebAssembly runtime. https://wasmer.io/, 2019.

[222] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. IEEE Symp. on Security and Privacy*, IEEE S&P, 2015.

[223] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven secure cryptography for the Web ecosystem. *Proc. ACM on Programming Languages*, 3(POPL):77:1–77:29, January 2019.

[224] WebAssembly Community Group. WebAssembly. http://webassembly.org, 2018.

[225] WebAssembly Community Group. GC extension. https://github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md, 2019.

[226] WebAssembly Community Group. Semantics. https://github.com/WebAssembly/design/blob/master/Semantics.md, 2019.

[227] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *Proc. ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO, 2019.

[228] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In *Proc. ACM Conf. on Programming Language Design and Implementation*, PLDI, 2019.

[229] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Proc. European Symp. on Research in Computer Security*, EuroS&P, 2012.

[230] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 2021.

[231] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *Proc. ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO, 2018.

[232] Yuval Yarom. Mastik: a micro-architectural side-channel toolkit. http://cs.adelaide.edu.au/~yval/Mastik. Version 0.02.

[233] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high-resolution, low-noise, L3 cache side-channel attack. In *Proc. USENIX Security Symp.*, 2014.

[234] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the Intel last-level cache. https://eprint.iacr.org/2015/905.pdf, 2015.

[235] Luke Yen, Jayaram Bobba, Michael R Marty, Kevin E Moore, Haris Volos, Mark D Hill, Michael M Swift, and David A Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. IEEE Symp. on High-Performance Computer Architecture*, HPCA, 2007.

[236] Yves Younan. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *Proc. Network and Distributed System Security Symp.*, NDSS, 2015.

[237] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *Proc. ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO, 2019.

[238] Alon Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2011.

[239] Alon Zakai. Compiling to WebAssembly: It's Happening! https://hacks.mozilla.org/2015/12/compiling-to-webassembly-its-happening/, 2015.

[240] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2020.

[241] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: a real-time side-channel attack detection system in clouds. In *Proc. Symp. on Research in Attacks, Intrusions, and Defenses*, RAID, 2016.

[242] Lutan Zhao, Peinan Li, Rui Hou, Jiazhen Li, Michael C Huang, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors. In *Proc. Design Automation Conf.*, DAC, 2021.

[243] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, CCS, 2017.