

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Foundations for Speculative Side Channels

Permalink

<https://escholarship.org/uc/item/64n9f44x>

Author

Cauligi, Sunjay R

Publication Date

2021

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Foundations for Speculative Side Channels

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

in

Computer Science

by

Sunjay R Cauligi

Committee in charge:

Professor Deian Stefan, Chair
Professor Nadia Heninger
Professor Ranjit Jhala
Professor Farinaz Koushanfar
Professor Shachar Lovett

2021

Copyright
Sunjay R Cauligi, 2021
All rights reserved.

The dissertation of Sunjay R Cauligi is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

For Louisa.

EPIGRAPH

What?
Is it not a simple task?

Why, to someone like you,
it should be by no means be a
difficult task.

Except...

The one thing is...
I'm a very busy fellow...

And I must leave this place in
three days.

How grateful I would be if you
could bring it back to me before
my **time** here is up...

But, yes... You'll be fine.
I see you are young and have
tremendous courage.

I'm sure you'll find it right away.

Well then, I am counting on you...

—The Happy Mask Salesman
(The Legend of Zelda: Majora's Mask)

TABLE OF CONTENTS

Dissertation Approval Page	iii
Dedication	iv
Epigraph	v
Table of Contents	vi
List of Figures	ix
List of Tables	x
Acknowledgements	xi
Vita	xiii
Abstract of the Dissertation	xiv
Introduction	1
1 Timing side-channels	1
2 Spectre vulnerabilities	2
3 Principled and practical foundations	3
4 Outline	5
Chapter 1 FaCT: A DSL for Timing-Sensitive Computation	8
1.1 Background	10
1.2 FaCT	14
1.2.1 Core language	15
1.2.2 Type system	18
1.3 Front-end compiler	24
1.3.1 Return deferral	25
1.3.2 Branch removal	29
1.3.3 Compiler correctness and security	32
1.4 Implementation and evaluation	34
1.4.1 Case studies	35
1.4.2 User study	38
1.5 Limitations and future work	41
1.6 Related work	42

Chapter 2	Constant-Time Foundations for the New Spectre Era	46
	2.1 Motivating examples	48
	2.2 Speculative semantics and security	50
	2.2.1 Speculative constant-time	53
	2.2.2 Overview of the semantics	55
	2.2.3 Speculative execution	56
	2.2.4 Memory operations	59
	2.2.5 Aliasing prediction	66
	2.2.6 Speculation barriers	69
	2.2.7 Indirect jumps	70
	2.2.8 Function calls	72
	2.3 Detecting violations	77
	2.3.1 Evaluation procedure	78
	2.3.2 Detected violations	79
	2.4 Related work	81
	2.5 Conclusion	82
Chapter 3	Towards Verified Spectre-Resistant SFI Sandboxing	84
	3.1 Formal model	85
	3.1.1 Syntax	85
	3.1.2 Architectural semantics	86
	3.1.3 Attackers and observations	86
	3.1.4 Speculative semantics	89
	3.2 Formalizing SFI security	90
	3.2.1 Non-interference	90
	3.2.2 SFI security properties	92
	3.2.3 Establishing security	93
	3.2.4 Swivel-SFI	94
	3.2.5 Swivel-CET	96
	3.3 Conclusion	98
Chapter 4	Practical Foundations for Spectre Defenses	99
	4.1 Preliminaries	101
	4.1.1 Breaking cryptography with Spectre	101
	4.1.2 Breaking software isolation with Spectre	103
	4.1.3 Security properties and execution semantics	104
	4.2 Choices in semantics	106
	4.2.1 Leakage models	107
	4.2.2 Non-interference and policies	113
	4.2.3 Execution models	117
	4.2.4 Nondeterminism	120
	4.2.5 Higher-level abstractions	122
	4.2.6 Expressivity and microarchitectural features	125

4.3	Related Work	129
4.3.1	Systematization of Spectre attacks and defenses	129
4.3.2	Hardware-based Spectre defenses	130
4.3.3	Software-hardware co-design	130
4.3.4	Other transient execution attacks	131
4.4	Conclusion	132
	Conclusion	134
Appendix A	FaCT: Deferred definitions and proofs	136
A.1	Semantics	136
A.2	Return deferral	139
A.3	Branch removal	143
Appendix B	Pitchfork: Full proofs	151
B.1	Consistency	151
B.2	Security	154
B.3	Soundness of Pitchfork	156
	Bibliography	161

LIST OF FIGURES

Figure 1.1:	FaCT grammar, top-level constructs.	15
Figure 1.2:	FaCT grammar, statements, and expressions.	16
Figure 1.3:	FaCT types.	17
Figure 1.4:	FaCT expression typing rules (subset).	21
Figure 1.5:	FaCT statement type rules (subset).	22
Figure 1.6:	FaCT procedure typing rules (subset).	23
Figure 1.7:	Transformation rules for return deferral.	26
Figure 1.8:	Transformation rules for branch removal.	29
Figure 2.1:	Example demonstrating a Spectre v1 attack.	49
Figure 2.2:	Example demonstrating a hypothetical attack abusing an aliasing predictor.	52
Figure 2.3:	Definition of the register resolve function.	53
Figure 2.4:	Store hazard caused by late execution of store addresses.	64
Figure 2.5:	Example demonstrating a store-to-load Spectre v1.1 attack.	64
Figure 2.6:	Example demonstrating a v4 Spectre attack.	65
Figure 2.7:	Example demonstrating fencing mitigation against Spectre v1 attacks.	70
Figure 2.8:	Example demonstrating a Spectre v2 attack.	71
Figure 2.9:	Example demonstrating a ret2spec-style attack.	73
Figure 2.10:	Example demonstrating “retpoline” mitigation against Spectre v2 attack.	76
Figure 2.11:	Vulnerable snippet from <code>__libc__message()</code>	80
Figure 2.12:	Vulnerable snippet from the FaCT OpenSSL MEE implementation.	80
Figure 3.1:	Syntax of the $ZFI \Rightarrow \mathcal{Q}$ language.	87
Figure 3.2:	Architectural semantics for $ZFI \Rightarrow \mathcal{Q}$	88
Figure 3.3:	Speculative semantics for $ZFI \Rightarrow \mathcal{Q}$	89
Figure 4.1:	Code snippet which an attacker can exploit using Spectre.	102
Figure A.1:	Big-step semantics.	137
Figure A.2:	Type system \vdash_{rd} for return deferral.	139
Figure A.3:	Values equivalence.	141
Figure A.4:	Type system \vdash_{ct} for constant-time.	143
Figure A.5:	Type interpretation.	145

LIST OF TABLES

Table 1.1:	FaCT case study summary.	36
Table 1.2:	Overhead of FaCT ports compared to optimized C, for each benchmark. . .	37
Table 1.3:	Number of correct and constant-time solutions for each task.	40
Table 2.1:	Instructions and their transient instruction form.	51
Table 2.2:	Correct and incorrect branch prediction.	58
Table 2.3:	SCT violations found by Pitchfork.	79
Table 3.1:	Leakage models.	89
Table 4.1:	Comparison of various semantics and tools.	108
Table 4.2:	Speculative security properties in prior works.	115

ACKNOWLEDGEMENTS

I cannot begin this section without first thanking my partner, Louisa Fan. She has supported me through the pits of my graduate career, both emotionally and mentally; without her aid I would never have made it through my PhD. I am also truly blessed to have two wonderful parents, Raghothama and Pankaja Cauligi, who have given me nearly 30 years of love and encouragement despite my remaining a student for nearly 30 years.

I am incredibly thankful to my advisor, Deian Stefan, who decided to take a chance on a rather immature second-year and fostered him into the academic I am now. His myriad connections are what landed me with my frequent collaborator and quasi-advisor Gilles Barthe, who introduced me to the joys of semantics, and despite working with me first-hand still offered to put me up as a postdoc. And of course I must thank Geoff Voelker and Stefan Savage—my initial advisors—who were no doubt confused why I still showed up to their meetings for so many years.

To my colleagues and labmates, whose friendship gave me so much joy while we toiled away: Rob McGuinness, my perpetual roommate and brother-in-arms; Ariana Mirian, my twin sister and fellow jokester; Craig Disselkoen, who I all but conscripted into being my research assistant and also, somehow, my friend; and to the past and present members of the 3140 lunch crew, for their many, *many* interesting discussions over the years.

Finally, thank you to all my collaborators and classmates, and everyone I've interacted with during my research tenure at UCSD CSE and abroad—far too many to count and yet deeply impactful all the same.

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 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19). Cauligi, Sunjay; Soeller, Gary; Johannesmeyer, Brian; Brown, Fraser; Wahby, Riad S.; Renner, John; Grégoire, Benjamin; Barthe, Gilles; Jhala, Ranjit; Stefan, Deian, ACM, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). 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 this paper.

Chapter 3, in part, is currently being prepared for submission for publication of the material. Cauligi, Sunjay; Guarnieri, Marco; Mehta, Aastha; Moghimi, Daniel; Narayan, Shrvan; Stefan, Deian; Vahldiek-Oberwagner, Anjo; Vassena, Marco. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been submitted for publication of the material as it may appear in 43rd IEEE Symposium on Security and Privacy (Oakland '22), Cauligi, Sunjay; Disselkoen, Craig; Moghimi, Daniel; Barthe, Gilles; Stefan, Deian. The dissertation author was the primary investigator and author of this material.

VITA

2015	Bachelor of Science, Mathematics, <i>magna cum laude</i> University of Washington, Seattle
2015	Bachelor of Science, Computer Engineering, <i>magna cum laude</i> University of Washington, Seattle
2015-2021	Research Assistant, Computer Science University of California San Diego
2018	Master of Science, Computer Science University of California San Diego
2021	Doctor of Philosophy, Computer Science University of California San Diego

PUBLICATIONS

C. Watt, J. Renner, N. Popescu, S. Cauligi, D. Stefan. “CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem.” S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Gregoire, G. Barthe, R. Jhala, and D. Stefan. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), January 2019.

“FaCT: A DSL for Timing-Sensitive Computation.” 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2019.

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

M. Vassena, C. Disselkoen, K. v Gleissenthall, S. Cauligi, R. K1c1, 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.

G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, P. Schwabe. “High-Assurance Cryptography in the Spectre Era.” 42nd IEEE Symposium on Security and Privacy (Oakland), May 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.” 30th USENIX Security Symposium (USENIX), August 2021.

S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, D. Stefan. “SoK: Practical Foundations for Spectre Defenses.” In submission.

S. Cauligi, M. Guarnieri, A. Mehta, D. Moghimi, S. Narayan, D. Stefan, A. Vahldiek-Oberwagner, M. Vassena. “Formal Guarantees for Spectre-resistant SFI Sandboxing.” Unpublished.

ABSTRACT OF THE DISSERTATION

Foundations for Speculative Side Channels

by

Sunjay R Cauligi

Doctor of Philosophy in Computer Science

University of California San Diego, 2021

Professor Deian Stefan, Chair

Developers of high-security systems (e.g., cryptographic libraries, web browsers) must not allow sensitive information (e.g., encryption keys, browser cookies) to make its way to an attacker. However, clever attackers can make use of unintentional *side-channels*—such as timing information or other hardware resource metrics—to infer or *leak* the values of these secrets. Even worse, attackers can exploit hardware features such as *speculative execution* to create *new* vectors for side-channel leakage even where none existed before.

Side-channels are not typically captured in formal program semantics—information from a side-channel is leaked to an attacker purely as a side-effect of execution, rather than any explicit data flow. Furthermore, speculative execution fundamentally destroys security properties like

memory or type safety, as they implicitly assume a standard sequential execution model. Without formal models to rely on, developers find themselves manually applying *ad-hoc* mitigations as a best-effort solution to prevent timing side-channels and speculative attacks. Unfortunately, these ad-hoc mitigations often lead to obfuscated code—and yet give no guarantee of a sound or complete defense.

This dissertation seeks to remedy this. We explore several formal frameworks that make side-channel effects explicit, both with and without the threat of speculative execution. Along the way, we introduce FaCT, a language and compiler for writing code free from side-channels; Pitchfork, a semantics and tool for detecting speculative side-channels in binaries; and ZFI \Rightarrow , a framework for validating sandbox protections against speculative attacks. In addition to the systems presented in this dissertation, the research community writ large has developed several program analysis and defense tools backed by formal models, whether these models are explicit or implicit. We round out this dissertation by surveying these systems, examining various design choices and identifying areas of open research.

Ultimately, this dissertation demonstrates the power of practical, formal foundations when dealing with speculative side-channel security. By relying on sound, formal frameworks, high-security developers can write programs that verifiably do not leak sensitive information.

Introduction

Protecting secrets in software is hard. Security and cryptography engineers must write programs that protect secrets both at the source level and when executed on real hardware. Unfortunately, hardware too easily divulges information about a program’s execution via *side-channels*—e.g., an attacker can learn program secrets by observing the side-effects of the program on the hardware [59]. More alarmingly, modern hardware features such as *speculative execution* give rise to attacks such as *Spectre*, in which an attacker can exploit architecturally invalid execution paths to create new side-channels. Indeed, these issues destabilize the ground upon which standard notions of security are built. And accordingly, developers of secure software require *sound structural support*: Tools with *sound, formal backing* that can ensure that programs are free from such vulnerabilities, even in the face of speculative execution.

1 Timing side-channels

We first give a background on *timing side-channels*, wherein code executes for observably different amounts of time depending on the value of secret information. For example, a textbook implementation of RSA decryption takes a different amount of time depending on the individual key bits [84]—each ‘1’ bit requires an additional multiplication and thus more time. The cumulative effects of these operations on the running time is large enough for the attacker to reconstruct the value of the secret key. Timing vulnerabilities like these are not merely of academic

interest: They have been found in implementations of both RSA [32] and AES [20, 115], where they allowed even remote network attackers to divine the values of secret keys.

The most robust way to deal with timing side-channels is via *constant-time programming*—the paradigm used to implement almost all modern cryptography [13, 46, 50, 118, 119]. Constant-time programs can neither branch on secrets nor access memory based on secret data.¹ The first class of vulnerability, from control flow, arises when the value of a secret influences control flow, as attackers can often observe the path of execution through a program: For example, if conditional branch targets take different amounts of time to execute [118] or if different program paths use different amounts of hardware resources [29]. The second class of vulnerability, from memory accesses, arises when memory access patterns depend on secret data. An attacker co-located on the same machine as a victim process, for example, can easily infer secret memory access patterns by observing their own cache hits and misses [59, 115]; alarmingly, attackers might even learn such information across a datacenter—or even over the Internet [32, 126].

The constant-time paradigm implicitly assumes that each instruction in a program is executed in order. However, modern processors do not execute sequentially—instead, they *speculatively* execute (potentially incorrect) program instructions ahead of time before prior instructions are fully resolved. Standard constant-time guarantees are therefore insufficient for most modern hardware.

2 Spectre vulnerabilities

We next give an overview of Spectre attacks [9, 12, 69, 82, 86, 87, 97, 169], a recently discovered family of vulnerabilities caused by *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

¹Constant-time programs must also not use secret data as input to any variable-time operation—e.g., floating-point multiplication [10].

or value forwarding). When the processor learns that its prediction was wrong, 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.

The following code gives an example of a vulnerable function; an attacker can exploit branch misprediction to leak arbitrary memory via the data cache:

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

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`.

3 Principled and practical foundations

Many developers rely on community best-practices and recipes to manually write constant-time code [104, 118]. Developers apply these recipes in an *ad-hoc* manner, leaving overlooked vulnerabilities open to attack. Even then, it can be tricky for developers to *correctly* apply the

recipes. For example, an attempt to use a recipe to fix a timing attack vulnerability in TLS [104] led to the Lucky13 timing vulnerability in OpenSSL [3]—and the purported fix for Lucky13 opened the door to yet another vulnerability [140].

Spectre mitigations, even when inserted automatically via tooling, have fared no better: The MSVC compiler’s `/Qspectre` flag—one of the first compiler defenses [102]—inserts mitigations by searching for code patterns. Since these patterns are not based in any rigorous analysis, the compiler easily misses similarly vulnerable code patterns [113]. Chrome adopted process isolation as its core defense mechanism against Spectre attacks [123], but this is also unsound: [35] shows that Spectre attacks can be performed across the process boundary, and [128] shows how to read cross-origin data in the browser. Like constant-time recipes, Spectre defense mechanisms are applied ad-hoc and incompletely.

For targeted, flexible, *sound* defenses, we must turn to formal methods. Formal security analysis is rooted in *program semantics*, which provides rigorous models of program behavior and serves as the basis for *formal security policies*. These policies help us carefully and explicitly spell out our assumptions about the attacker’s strength and to gain confidence that our tools are sound with respect to this class of attackers—that timing side-channel defenses indeed enforce a constant-time policy, or that Spectre detection tools find the vulnerabilities they claim.

Formal foundations not only ensure constant-time and Spectre defenses are secure, but also help improve the performance of practical tools. Without formalizations, manual defenses cannot be assured sound, and automatic defenses are usually either overly conservative (unnecessarily flagging code as vulnerable, which ultimately leads to unnecessary and slow mitigations) or overly aggressive (and thus vulnerable). Developing proper foundations allows us to craft defenses that are instead *targeted* while still being provably secure [7, 65, 151].

4 Outline

This dissertation lays *principled* and *practical* foundations for rebuilding side-channel defenses in the speculative domain.

Chapter 1 presents FaCT, a compiler and domain-specific language for writing *sequentially* constant-time code. Although FaCT does not analyze speculative effects, it gives us a blueprint for what *security-aware* compilation can achieve. At the core of FaCT is a set of formal *compiler transformations* that describe how to soundly replace leaky program behavior: The results of transformation are programs with equivalent behavior, but that don't leak secrets. The FaCT language itself is a C-like language augmented with *secrecy annotations*, which allow the developer to explicitly specify which program variables are indeed secret. The FaCT compiler tracks these annotations through the compilation pipeline, allowing it to apply the transformation rules only when necessary to produce constant-time code.

Chapter 2 presents a structural foundation for speculative analysis: A formal instruction-level semantics that models the speculative behavior—such as branch predictions and value forwarding—of modern processors. On top of this execution model, we apply the secrecy annotations from FaCT and define the notion of *speculative constant-time* (SCT): A speculative side-channel leak (such as a Spectre attack) is a violation of SCT. This semantics is expressive enough to capture all known Spectre attacks, including a variant of Spectre that was unrealized at the time. This semantics has been used to show the soundness of several tools that detect Spectre vulnerabilities, including our own verification tool, Pitchfork.

Chapter 3 builds upon this foundation, constructing a framework to analyze *software isolation* (or *sandboxing*) in the speculative context. Current systems that prevent speculative sandbox attacks are implemented as collections of ad-hoc mitigations, without any formal backing. We rectify this, expanding our speculative properties and semantics to capture speculative sandbox

security in addition to SCT. Our formal model shows that existing systems are not sound and make several implicit assumptions about the underlying hardware.

Finally, Chapter 4 surveys the current state of Spectre analysis and defense tools, both with and without associated formal models. We examine and categorize these systems by the different choices they make in their stated (or implied) semantics and security properties. Our analysis provides practical suggestions and considerations both for developers of analysis and mitigation tools and for researchers of speculative security.

Acknowledgements

Introduction, in part, uses the following material:

Material as it appears in 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19). Cauligi, Sunjay; Soeller, Gary; Johannesmeyer, Brian; Brown, Fraser; Wahby, Riad S.; Renner, John; Grégoire, Benjamin; Barthe, Gilles; Jhala, Ranjit; Stefan, Deian, ACM, 2019. The dissertation author was the primary investigator and author of this paper.

Material as it appears in 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). 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 this paper.

Material currently being prepared for submission for publication. Cauligi, Sunjay; Guarnieri, Marco; Mehta, Aastha; Moghimi, Daniel; Narayan, Shravan; Stefan, Deian; Vahldiek-Oberwagner, Anjo; Vassena, Marco. The dissertation author was the primary investigator and author of this paper.

Material that has been submitted for publication as it may appear in 43rd IEEE Symposium on Security and Privacy (Oakland '22), Cauligi, Sunjay; Disselkoen, Craig; Moghimi, Daniel;

Barthe, Gilles; Stefan, Deian. The dissertation author was the primary investigator and author of this material.

Chapter 1

FaCT: A DSL for Timing-Sensitive Computation

Or, a sketch of the tower.

Despite many strides in language design over the past half-century, modern cryptographic routines are still typically written in C. This is good for speed but bad for keeping secrets. Like most general-purpose languages, C gives the programmer no way to denote which data is sensitive—and therefore gives the programmer no warnings about code that inadvertently divulges secrets.

The only recourse developers have to avoid timing vulnerabilities is to make their code ugly. Specifically, they use a selection of *recipes* to turn dangerous but readable code into safe but obfuscated code: they re-write potentially secret-revealing constructs like branches into low level sequences of assignments that operate in *constant-time* regardless of the values of secret data. For example, the readable

```
if (secret) x = e
```

which branches on a `secret` bit is replaced by

```
x = (-secret & e) | (secret - 1) & x
```

which, unlike the branch, executes in the same amount of time no matter the value of `secret`.

This is a sorry state of affairs. First, developers apply the recipes in an *ad-hoc* way, and any untransformed computation is left vulnerable to attack. Second, the recipes *obfuscate* the code, making it harder to determine whether the routine is even computing the desired value. Third, it can be tricky for developers to *correctly* apply the recipes. For example, an attempt to use a recipe to fix a timing attack vulnerability in TLS [104] led to the Lucky13 timing vulnerability in OpenSSL [3], and the purported fix for Lucky13 opened the door to yet another vulnerability [140]!

In this chapter, we introduce FaCT, a domain-specific language and compiler for writing *readable* and *timing-secure* cryptographic routines. FaCT lets developers write readable code using high-level control-flow constructs like branches and procedural abstractions, but then automatically compiles this code into efficient, constant-time executables. We develop FaCT via four contributions:

1. Language design. Our first contribution is the design of a language for writing cryptographic code. The language allows programmers to use standard control-flow constructs like `if` and `return` statements. However, the language is equipped with an *information-flow* type system that programmers can use to specify which data are `secret`. The type system prevents leaks by ensuring that `secrets` do not explicitly or implicitly influence the `public-visible` outputs (§1.2).

2. Public safety. Our second contribution is the observation that not all programs are amenable to constant-time compilation. Specifically, we show that naive application of constant-time recipes can mangle otherwise safe programs, causing memory errors or undefined behavior. We address this problem by introducing a notion called *public safety* that characterizes the source programs that can be compiled to constant-time without introducing errors (§1.2.2.3).

3. Constant-time compilation. Our third contribution is a compiler that automatically converts (public safe) source programs into constant-time executables. The FaCT compiler is based on the key insight that we can exploit the `secret` types to *automatically* apply the

recipes that developers have hitherto applied by hand, and can do so *systematically*, i.e., exactly where needed to prevent the exposure of secrets via timing. We formalize the compiler with two transformations, *return deferral* and *branch removal*, and prove that compilation yields constant-time executables with source-equivalent semantics (§1.3).

4. Implementation and evaluation. Our final contribution is an implementation of FaCT that produces LLVM IR from high-level sources, and uses LLVM’s `clang` to generate the final object or assembly file. We evaluate FaCT’s *usability* with a user study, surveying students in an upper-level, undergraduate programming languages course at a U.S. university, where 57% of the participants found FaCT easier to write than C (compared to 15% who found C easier). We evaluate FaCT’s *expressiveness* and *performance* by using our implementation to port 7 cryptographic routines from 3 widely used libraries: OpenSSL, libsodium, and curve25519-donna, totaling about 2400 lines of C source. The unoptimized FaCT code—which we *formally* guaranteed to be constant-time—is between 16–346% slower than the C equivalent. The `clang`-optimized FaCT code—which we *empirically* check to be constant-time using `dudect` [125]—is between 5% slower to 21% *faster* than the C equivalent, showing that FaCT yields readable constant-time code whose performance is competitive with C (§1.4).

We make all source and data available under an open source license at: <https://fact.programming.systems>.

1.1 Background

Some common C constructs—branches, returns, and array updates—can reveal secrets via timing channels. In this section, for each potentially dangerous construct, we explain: (1) how that construct could introduce bugs in real-world projects; (2) how developers must use recipes to avoid the dangerous construct; and, (3) how FaCT allows programmers to forgo recipes and write readable yet safe code.

Branching on secret values. A first class of vulnerability arises from directly branching on the value of a secret—attackers can often reconstruct control flow through a program, and thus secret condition values (e.g., because the `true` branch takes orders of magnitude longer to execute than the `false` branch) [118]. To avoid this type of vulnerability, developers manually translate branching code to straight-line code by replacing `if`-statements with constant-time bitmasks. Consider the following example from OpenSSL (edited slightly for brevity), which formats a message before computing a message authentication code (MAC):

```
for (j = 0; j < md_block_size; j++, k++) {
    b = data[k - header_length];
    b = constant_time_select_8(is_past_c, 0x80, b);
    b = b & ~is_past_cp1;
    b &= ~is_block_b | is_block_a;
    block[j] = b;
}
```

It's hard to tell, but this snippet (1) iterates over plaintext message `data`, (2) terminates the message with standard-defined `0x80`, and (3) pads the terminated message to fill a hash block—all while keeping `data` secret. To this end, even the selection operator `constant_time_select_8(mask, a, b)` is a series of bitmasks: $(\text{mask} \ \& \ a) \ | \ (\sim\text{mask} \ \& \ b)$.

Translating each line of this OpenSSL code to FaCT leads to drastically more readable code:

```
for (uint64 j from 0 to md_block_size) {
    k += 1;
    b = is_past_c ? 0x80 : data[k - (len header)];
    if (is_past_cp1 || (is_block_b && !is_block_a)) {
        b = 0;
    }
}
```

```
    block[j] = b;
}
```

With FaCT, the programmer declares the sensitive variables as used in the conditions as `secret`. After doing so, they are free to use plain conditional expressions and ternary operators to compute the final value of `b`. The FaCT compiler automatically uses the type annotations to generate machine code equivalent to the C example.

Early termination. Both loops and procedures can terminate early depending on the value of a secret, thereby leaking the secret. A well-known padding oracle attack in older versions of OpenSSL exploits an early function return [152]: a packet processing function would decrypt a packet and then check that the padding was valid, and, in the case of invalid padding, would return immediately. An attacker could exploit this to recover the SSL session key by sending specially crafted packets and use timing measurements to determine whether or not the padding of the decrypted packet was valid. Similarly, if the number of loop iterations in a program depends on a secret, attackers can use timing to uncover the value of that secret (e.g., in the Lucky13 attack [3]).

C programmers again use special recipes, turning idiomatic programs into hard-to-read constant-time code. Consider the following buffer comparison code from the libsodium cryptographic library:

```
for (i = 0; i < n; i++)
    d |= x[i] ^ y[i];
return (1 & ((d - 1) >> 8)) - 1;
```

This snippet compares the first `n` bytes of the arrays `x` and `y`, returning `0` if they are the same, and `-1` otherwise. To avoid leaking information about the contents of the arrays, though, the loop *cannot* simply return early when it detects differing values; instead, the programmer must maintain a “flag” (`d`), and update it at each iteration to signal success or failure. While iterating

inside the loop, if the values $x[i]$ and $y[i]$ are the same, then $x[i] \oplus y[i]$ will be 0, leaving d unchanged. However, if $x[i]$ and $y[i]$ are different, then their XOR will have at least one bit set, causing d to also have a non-zero value. After the loop, the code uses a complex shift-and-mask dance to collapse d into the value -1 if any bits are set, and 0 otherwise.

FaCT lets programmers avoid the “flag” contortions:

```
for (uint64 i from 0 to n)
    if (x[i] != y[i])
        return -1;
return 0;
```

With FaCT, the programmer can readily express returning -1 in the case of failure as the compiler automatically generates a special variable for the return value, and uses the `secret` type to translate returns-under-secret conditions into (constant-time) updates to this variable, producing machine code roughly equivalent to the C recipe above.

Memory access. Memory access patterns that depend on secret data can also inadvertently leak that secret data. An attacker co-located on the same machine as a victim process, for example, can easily infer secret memory access patterns by observing their own cache hits and misses [59, 115]; alarmingly, attackers might even learn such information across a datacenter—or even over the Internet [32, 126].

To avoid leaking information via memory access patterns, developers rely on recipes that avoid accessing memory based on secrets. The following C code (from the “donna” Curve25519 implementation), for example, swaps the values of array a with array b based on the value of a secret (`swap`):

```
for (i = 0; i < 5; ++i) {
    const limb x = swap & (a[i] ^ b[i]);
    a[i] ^= x;
```

```
    b[i] ^= x;
}
```

To avoid leaking the value of the secret `swap`, the code *always* accesses both `a[i]` and `b[i]` at each loop iteration, and uses bitmask operations that only change them if `swap` is a mask of all 1-bits.

FaCT, again, makes such subterfuge unnecessary:

```
if (swap != 0) {
    for (uint64 i from 0 to 5) {
        secret uint64 tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }
}
```

Once the programmer has marked `swap` as `secret`, the compiler will automatically synthesize masked array reads similar to those from the original Curve25519 code.

1.2 FaCT

FaCT is a high-level, strongly-typed C-like DSL, designed for writing constant-time crypto code. In this section, we describe the DSL and its type system, one that both disallows certain unsafe programs and specifies how the compiler should transform code to run in constant-time.¹ We describe the type-directed transformations in §1.3.

```

PROGRAM
program ::= [ fdef | sdef ] ...

STRUCTURE DEFINITION
sdef ::= struct name {  $\beta$  x; ... }

PROCEDURE DEFINITIONS
fdef ::=
    |  $f(\vec{x} : \vec{\beta}) \{ S \} : \beta$           internal procedure
    | export  $f(\vec{x} : \vec{\beta}) \{ S \} : \beta$   exported procedure
    | extern  $f(\vec{x} : \vec{\beta}) : \beta$         external procedure

```

Figure 1.1: FaCT grammar, top-level constructs.

1.2.1 Core language

FaCT is designed to be embedded into existing crypto projects (e.g., OpenSSL), and not to be used as a standalone language. As such, FaCT “programs” are organized as collections of procedures. As shown in Figure 1.1, developers can export these procedures as C functions to the embedding environment. They can also *import* trusted procedures. This is especially useful when using FaCT to implement error-prone glue code around already known-safe C crypto primitives (e.g., building a block cipher mode that calls an AES primitive).

FaCT procedures are composed of a sequence of statements (e.g., *if* statements, *for* loops, etc.), which are themselves composed of expressions. Both statements and expressions are mostly standard. We only remark on the more notable language constructs we add to make writing cryptographic code more natural.

First, FaCT includes a number of *array primitives* to capture common idioms in cryptographic routines, and to replace unsafe pointer arithmetic. The operation `len e` returns the length of an array *e*; `zeros(β , e)` creates an array of zeros of type β of length *e*; `clone(e)` copies the

¹The *surface* language as used by developers is slightly less verbose than the *core* language presented in this section. For example, our surface syntax allows procedures to be called in expressions; FaCT desugars such expressions into core language procedure-call statements. We refer to both the surface and core languages as FaCT.

LABELS	SIZE	ARRAY SIZE
$\ell ::= \text{PUB} \mid \text{SEC}$	$s ::= 8 \mid 16 \mid \dots \mid 128$	$\text{sz} ::= * \mid 0 \mid 1 \mid \dots$
MUTABILITY	BASE TYPES	
$m ::= \text{R} \mid \text{RW}$	$\beta ::= \text{BOOL}_\ell \mid (\text{U})\text{INT}_\ell^s \mid \text{REF}_m[\beta] \mid \text{ARR}^{\text{sz}}[\beta] \mid \{\vec{x} : \vec{\beta}\}$	

Figure 1.3: FaCT types.

array e ; and $\text{view}(e_1, e_2, e_{len})$ returns a *slice* of array e_1 starting at position e_2 and with length e_{len} . We introduce `views` to make up for the lack of pointers: `views` allow developers to efficiently compute on small pieces of large buffers.

Second, we provide *vector primitives*: parallel vector arithmetic and vector shuffle instructions. These instructions allow developers to implement crypto algorithms that leverage fast SIMD instructions (e.g., SSE4 in `x86_64`) without resorting to architecture-specific inline assembly or compiler intrinsics.

Third, we expose `ctselect`, a constant-time selection primitive. The operation $\text{ctselect}(e_1, e_2, e_3)$ evaluates to either e_2 or e_3 , depending on whether e_1 is `true` or `false`, respectively. The compiler guarantees that `ctselect` compiles to constant-time code (e.g., as a series of bitmasks or the `CMOV` instruction on `x86_64`). Developers need not use `ctselect` directly; instead, they can use our higher-level `if`-statements, which our compiler transforms to such `ctselects` (§1.3).

Lastly, FaCT includes a `declassify` primitive that takes a secret expression as input and returns a public value. Developers can use this primitive to bypass FaCT’s typing restrictions (described below) and explicitly make information public. This is useful, e.g., for implementing encryption: a buffer containing a secret message must be treated with care, but if the buffer is encrypted in-place, it is thereafter safe to `declassify` because it contains ciphertext.

1.2.2 Type system

The most important feature of the FaCT language is its static information-flow type system. We rely on this type system to: (1) provide a way for developers to demarcate the sensitivity of data—whether it is secret or public; (2) reject unsafe programs, i.e., programs that are not information-flow secure or cannot be safely transformed to constant-time code; and (3) direct the compiler when applying transformations. Below, we give an overview of our type system and explain how it fulfills the first two roles; we leave the third for §1.3.

Like previous information-flow type systems [109, 110, 129, 156], FaCT decorates each base type with a `secret` or `public` *secrecy label*². Figure 1.3 summarizes our base types; they are largely standard. Reference types wrap another base type and inherit its secrecy label; they are also access controlled, i.e., they can be read-only or read-write. In the FaCT surface syntax, we disallow recursively-typed references—only references of integer and boolean types are expressible. Array types, like references, inherit the secrecy of their base type; arrays have a size which is either a statically known constant or dynamically determined (*). Struct types *do not* carry a secrecy label; instead, each struct field is individually labeled.

Developers explicitly specify labels when they declare variables and procedures. FaCT’s type system, in turn, uses these labels to reject unsafe programs and specify how the compiler should transform high-level code that uses seemingly unsafe constructs (e.g., `secret if`-statements) to constant-time code. Below, we walk through our typing rules for expressions, statements, and procedures.

²Labels are partially ordered according to \sqsubseteq as usual: $\text{PUB} \sqsubseteq \ell$ and $\ell \sqsubseteq \text{SEC}$ holds true for any label ℓ . The join of two labels is similarly standard: $\ell_1 \sqcup \ell_2$ is SEC if either label is SEC, and PUB otherwise. For brevity, we also use these operators on types (operating on the underlying label), much like previous work (e.g., [109, 110]).

1.2.2.1 Expression typing

FaCT’s expression typing judgment $\Gamma \vdash e : \beta$ states that under the type context Γ , which maps variables to their declared types, the expression e has the type β . We write $x : \beta \in \Gamma$ when variable x maps to type β in the context Γ .

Figure 1.4 gives the typing rules for the most interesting expressions. The rule for `ctselect`, for example, ensures that (1) the result is at least as secret as all the arguments to `ctselect` and (2) all the arguments can be cast to integers—since, internally, `ctselect` may be implemented as a series of constant-time bitmasks. The typing rules for other constructs similarly preserve secrecy.

The type system also disallows certain kinds of unsafe computations. For example, we reject programs that index memory based on secrets: the rules for T-ARR-GET and T-ARR-VIEW ensure that array indices are `public` and in-bounds. The in-bounds checks are `highlighted`, and detailed in §1.2.2.3.

1.2.2.2 Statement and procedure typing

FaCT allows developers to write code whose control flow depends on sensitive data. Unfortunately, not all such code can be safely or efficiently transformed. For example, to safely allow writes to arrays using a secret-dependent index we must (transform the code to) write to *all* indices [103, 117, 122]; such a transformation would be expensive, and FaCT instead disallows such computations. As such, typing rules for statements and procedures rely on a *secrecy context*, which comprises a pair of secrecy labels pc, rc called the *path* and *return* context, respectively.

The path context label pc for a statement is `secret` if the statement is contained within—i.e., is control-dependent upon—a `secret` branch. Since a procedure caller’s path context must persist through to the callee’s path context, the initial path context label of a procedure is `secret` if it is ever called from a `secret` context; otherwise the initial path context label is `public`. We use ω to map procedures to their initial path context labels.

The return context label rc for a statement is `secret` if the statement *may* be preceded by a `return` statement that is itself control-dependent on a `secret` value. A procedure’s return context label is always initially `public`. Thus, the *secrecy context* ($pc \sqcup rc$) for a statement represents whether the flow of control (to get to the statement) can be influenced by `secret` values. For example, if the conditional expression of an `if` statement is `secret`, then the statements of each branch are judged with $pc = \text{SEC}$, and are thus typed under a `secret` context.

Statement typing. FaCT’s statement typing judgment is of the form $\omega, pc, \beta_r \vdash S : \Gamma, rc \rightarrow \Gamma', rc'$, where β_r is the return type of the procedure containing the statement S . This judgment states that, given a type- and security- context defined by ω, pc, β_r and initial Γ, rc , the statement S : (1) can be safely compiled into constant-time code, and (2) yields a new updated type context Γ' and return context rc' . This typing judgment accounts for new variable declarations and ensures that the secrecy context influences subsequent statements. For example, if a `return` statement resides within a `secret` branch, then all statements executed after that branch must also be typed under a `secret` context, since their execution now depends on the `return`.

Figure 1.5 shows the most interesting statement typing rules. For example, (T-ASGN) checks that when updating a reference, the current secrecy context does not exceed the secrecy label of the value e_2 being assigned. This ensures that `secret` data cannot be leaked via control flow. Rules (T-IF) and (T-RET) account for such secret contexts; the latter additionally ensures that the procedure cannot return a value more sensitive than specified by the procedure return type.

Rule (T-FOR) is more restricting: it ensures that `secrets` do not influence the running time of `for` loops by requiring that the loop bounds—and therefore the number of iterations—be `public`. The updated return context rc' must both be a fixpoint of the loop, and must be no lower than the original return context rc . In practice, our type checker only assigns rc' to be `secret` if it cannot assign it to be `public`.

$$\begin{array}{c}
\text{T-CT-SEL} \\
\frac{\Gamma \vdash e_1 : \text{BOOL}_\ell \quad \beta \text{ is numeric or } \text{BOOL} \quad \Gamma \vdash e_2 : \beta \quad \Gamma \vdash e_3 : \beta}{\Gamma \vdash \text{ctselect}(e_1, e_2, e_3) : \beta \sqcup \ell} \\
\\
\text{T-ARR-GET} \\
\frac{\Gamma \vdash e_1 : \text{ARR}^{sz}[\beta] \quad \Gamma \vdash e_2 : \text{UINT}_{\text{PUB}}^s \quad \Gamma \Rightarrow e_2 < \text{len } e_1}{\Gamma \vdash e_1[e_2] : \beta} \\
\\
\text{T-ARR-VIEW} \\
\frac{\Gamma \vdash e_1 : \text{ARR}^{sz}[\beta] \quad \Gamma \vdash e_2 : \text{UINT}_{\text{PUB}}^s \quad \Gamma \vdash e_{len} : \text{UINT}_{\text{PUB}}^s \quad sz' = szOfExpr(e_{len})}{\Gamma \vdash \text{view}(e_1, e_2, e_{len}) : \text{ARR}^{sz'}[\beta]}
\end{array}$$

Figure 1.4: FaCT expression typing rules (subset).

The typing for procedure calls given by (T-CALL) is slightly more complex. In particular, this rule ensures that procedures can only be called with suitable inputs and checks that the output type is *compatible* with the variable being assigned. To this end, we ensure that if the procedure f has visible effects, then its initial path context $\omega(f)$ must be at least the label of the calling context. This, in effect, ensures that in a `secret` context we *cannot* call procedures that (1) modify public parameters, i.e., take mutable public references as input parameters; (2) are `externally` defined and so possibly have `publicly` visible side-effects; or (3) are `exported` (top-level) procedures.

Procedure typing. Figure 1.6 shows rules for typing procedure definitions. FaCT's procedure typing judgment is of the form $\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r$, which states that under ω , the procedure f with named parameters \vec{x} of types $\vec{\beta}$ has return type β_r . Procedures in FaCT may only return simple types (i.e., boolean values or integers), but there are no such restrictions on the types of parameters. When typing procedures, the initial type context Γ is formed from the procedure's parameters, and the initial path context pc is given by $\omega(f)$. The return context rc always starts as `PUB`, as the procedure body S (vacuously) has no preceding `secret`-dependent `return` statements. The return type β_r is taken from the procedure definition. If the body S is well-typed under these initial contexts, then the procedure itself is considered well-typed.

$$\begin{array}{c}
\text{T-CALL} \\
\frac{\omega \vdash f(\vec{\beta}) : \beta \quad \text{hasEffects}(f) \Rightarrow pc \sqcup rc \sqsubseteq \omega(f) \quad \Gamma \vdash e_i : \beta_i \quad \Gamma' = \Gamma, x : \beta}{\omega, pc, \beta_r \vdash \beta \ x = f(\vec{e}) : \Gamma, rc \rightarrow \Gamma', rc} \\
\\
\text{T-ASGN} \\
\frac{\Gamma \vdash e_1 : \text{REF}_W[\beta] \quad \Gamma \vdash e_2 : \beta \quad pc \sqcup rc \sqsubseteq \beta}{\omega, pc, \beta_r \vdash e_1 := e_2 : \Gamma, rc \rightarrow \Gamma, rc} \\
\\
\text{T-IF} \\
\frac{\Gamma \vdash e : \text{BOOL}_\ell \quad \omega, pc \sqcup \ell, \beta_r \vdash S_1 : \Gamma \ \wedge e, rc \rightarrow \Gamma_1, rc_1 \quad \omega, pc \sqcup \ell, \beta_r \vdash S_2 : \Gamma \ \wedge \neg e, rc \rightarrow \Gamma_2, rc_2}{\omega, pc, \beta_r \vdash \text{if}(e) \{ S_1 \} \text{ else } \{ S_2 \} : \Gamma, rc \rightarrow \Gamma, rc_1 \sqcup rc_2} \\
\\
\text{T-FOR} \\
\frac{\Gamma \vdash e_1 : \text{UINT}_{\text{PUB}} \quad \Gamma \vdash e_2 : \text{UINT}_{\text{PUB}} \quad \Gamma' = \Gamma, x : \text{UINT}_{\text{PUB}} \ \wedge e_1 \leq x < e_2 \quad rc \sqsubseteq rc' \quad \omega, pc, \beta_r \vdash S : \Gamma', rc' \rightarrow \Gamma'', rc'}{\omega, pc, \beta_r \vdash \text{for}(x \text{ from } e_1 \text{ to } e_2) \{ S \} : \Gamma, rc \rightarrow \Gamma, rc'} \\
\\
\text{T-RET} \quad \frac{\Gamma \vdash e : \beta_r \quad pc \sqcup rc \sqsubseteq \beta_r}{\omega, pc, \beta_r \vdash \text{return } e : \Gamma, rc \rightarrow \Gamma, pc \sqcup rc} \quad \text{T-ASSUME} \quad \frac{\Gamma \vdash e : \text{BOOL}_\ell \quad \Gamma' = \Gamma \ \wedge e}{\omega, pc, \beta_r \vdash \text{assume}(e) : \Gamma, rc \rightarrow \Gamma', rc}
\end{array}$$

Figure 1.5: FaCT statement type rules (subset).

1.2.2.3 Public safety

The FaCT type system ensures that procedures can be transformed using constant-time recipes without giving up safety. Naively applying recipes can inadvertently *introduce* safety and security vulnerabilities while making the code constant-time. Consider the following procedure:

```

void potential_oob( secret mut uint32[] buf
                  , public uint64 i
                  , secret uint64 secret_index ) {
    assume(secret_index <= len buf);
    if (i < secret_index)

```

$$\begin{array}{c}
\text{T-FN} \\
\frac{
\begin{array}{l}
pc = \omega(f) \\
\Gamma = \{\vec{x} : \vec{\beta}\} \\
\beta_r \text{ is numeric or BOOL} \\
\omega, pc, \beta_r \vdash S : \Gamma, \text{PUB} \rightarrow \Gamma', rc'
\end{array}
}{
\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r
}
\end{array}
\qquad
\begin{array}{c}
\text{T-FN-EXTERN} \\
\frac{
\begin{array}{l}
\omega(f) = \text{PUB} \\
\beta_r \text{ is numeric or BOOL}
\end{array}
}{
\omega \vdash \text{extern } f(\vec{x} : \vec{\beta}) : \beta_r
}
\end{array}$$

Figure 1.6: FaCT procedure typing rules (subset).

```

buf[i] = 0;
...
}

```

This code is memory safe as the branch condition ensures that we only update `buf[i]` when `i` is within bounds. However, the update is predicated upon a `secret` condition. To make the above code constant-time, we must ensure that the access to `buf[i]` happens regardless of that condition, or else the memory access pattern will reveal the secret. Consequently, the constant-time recipes—that we discuss in §1.3—would compile the code into:

```

cond = (i < secret_index);
buf[i] = ctselect(cond, 0, buf[i]);

```

Such a naive transformation introduces a potential *out-of-bounds* access. In other cases it can introduce yet different kinds of undefined behavior.

Public safety. We avoid the above problem with the key observation that for a program to be amenable to constant-time compilation, the source must be *publicly safe*: It must be memory-safe and free from buffer overflows and undefined behavior using only `public`-visible information, i.e., the code must be safe even after removal of `secret`-dependent control-flow. We formalize the notion of public safety in FaCT’s type system by extending the type environment Γ to track `public`-visible path conditions, using these conditions to check safety. In Figures 1.4 and 1.5 these public safety extensions are **highlighted**.

Public views. We first define the judgment $\Gamma \vdash_i e$ to mean that e is *immutable* in Γ ; that is, e is only composed of constants, immutable variables, array lengths, or operations thereon. Next, we define the operation $\Gamma \wedge e$, which *conjoins* Γ with a *public view* of the condition e : if e is a `public bool` ($\Gamma \vdash e : \text{BOOL}_{\text{PUB}}$) and e is immutable ($\Gamma \vdash_i e$), then $\Gamma \wedge e$ represents the environment Γ with the additional assumption that e is true. Otherwise, $\Gamma \wedge e = \Gamma$, i.e., conjoining Γ with a `secret` condition does not add any new assumptions to Γ . Rules T-IF and T-FOR in Figure 1.5 show how we propagate public views, tracking (`public`) conditions and loop ranges to use when type checking statements.

For cases where the public safety checker is incomplete, we allow developers to add assumptions directly to the environment Γ with the `assume` primitive (Figure 1.2). This is useful for aiding the checker by, e.g., adding preconditions to a procedure.

Checking public safety. Finally, we define $\Gamma \Rightarrow e$ to mean that the conditions in Γ *imply* e . This is checked via an SMT solver. We use this predicate in the expression typing rules T-ARR-GET and T-ARR-VIEW (Figure 1.4) to check that memory accesses are never out of bounds. In the example program given earlier, since the expression `i < secret_index` is of type `BOOLSEC`, it is not added to Γ ; thus the predicate $\Gamma \Rightarrow i < \text{len } \text{buf}$ does not hold when typing the expression `buf[i]`, and the program (correctly) does not type check.

The FaCT type system also prevents undefined behavior from invalid operand values (not shown in Figure 1.4). For example, integer division has the additional requirement $\Gamma \Rightarrow e_2 \neq 0$, and the left- and right-shift operators have the requirement $\Gamma \Rightarrow 0 \leq e_2 < s$ where s is the bitwidth of e_1 .

1.3 Front-end compiler

The FaCT compiler consists of two passes. The first pass is a source to source transformation—it compiles well-typed code into semantically equivalent FaCT constant-time

code whose observable timing is `secret`-independent. The second pass is straightforward—it takes the `secret`-independent code and generates LLVM bitcode. In the rest of the section, we thus only describe and formalize FaCT’s transformation pass.

Since our type checker (§1.2.2) already ensures that memory accesses, loop iterations, and variable-time instructions are `secret`-independent, the transformations need only make procedure returns and branches `secret`-independent. FaCT does this in two steps, *return deferral* and *branch removal*.

The first step replaces `secret`-dependent `return` statements by (1) creating a boolean that represents whether the procedure has returned and (2) conditioning all later code on that boolean to prevent statements from executing after the original procedure would have terminated. That is, return deferral converts control flow in terms of `secret` returns into control flow in terms of `secret ifs`.

The second step turns all `secret`-dependent conditional branches into straight-line code. This includes both `secret if` statements in the original source as well as those generated by return deferral. Thus, by eliminating `secret ifs`—the last source of `secret`-dependent timing—this transformation yields constant-time code.

1.3.1 Return deferral

As previously mentioned, early returns that depend on secrets often leak information. Consider the following snippet:

```
if (sec) { return 1; }  
// long-running computation ...
```

Here, an attacker can determine whether `sec` is `true` by observing a quick computation, or `false` by observing a slow computation.

FaCT prevents code from leaking such secrets by *deferring* returns to the end of each procedure. For example, the compiler transforms the above code to:

$$\begin{array}{c}
\text{TR-RET-DEC} \\
\frac{\Phi = (\omega, \{\vec{x} : \vec{\beta}\}, \beta_r) \quad \Phi, \omega(f), \text{PUB} \vdash S \rightarrow S'}{\omega \vdash f(\vec{\beta}) \{ S \} : \beta_r \rightarrow} \\
f(\vec{\beta}) \{ \text{REFRW}[\beta_r] \text{ rval} = \text{init}(\beta_r); \\
\text{REFRW}[\text{BOOL}_{\text{SEC}}] \text{ notRet} = \text{true}; \\
S'; \text{ return rval} \} : \beta_r \\
\\
\text{TR-RET-GUARD-PUB} \quad \text{TR-RET-GUARD-SEC} \\
\frac{\Phi, pc, \text{PUB} \vdash S \rightarrow S'}{\Phi, pc, \text{PUB} \vdash S \rightsquigarrow S'} \quad \frac{\Phi, pc, \text{SEC} \vdash S \rightarrow S'}{\Phi, pc, \text{SEC} \vdash S \rightsquigarrow \text{if}(\text{notRet}) \{ S' \}} \\
\\
\text{TR-RET} \\
\frac{pc \sqcup rc = \text{SEC}}{\Phi, pc, rc \vdash \text{return } e \rightarrow \text{rval} := e; \text{notRet} := \text{false}} \\
\\
\text{TR-RET-SEQ} \\
\frac{\Phi = (\omega, \Gamma, \beta_r) \quad \omega, pc, \beta_r \vdash S_1 : \Gamma, rc \rightarrow \Gamma', rc' \quad \Phi, pc, rc \vdash S_1 \rightarrow S'_1 \quad \Phi, pc, rc' \vdash S_2 \rightsquigarrow S'_2}{\Phi, pc, rc \vdash S_1; S_2 \rightarrow S'_1; S'_2} \\
\\
\text{TR-RET-FOR} \\
\frac{\Phi = (\omega, \Gamma, \beta_r) \quad rc \sqsubseteq rc' \quad \omega, pc, \beta_r \vdash S : \Gamma, rc' \rightarrow \Gamma', rc' \quad \Phi, pc, rc' \vdash S \rightsquigarrow S'}{\Phi, pc, rc \vdash \text{for}(x \text{ from } e_1 \text{ to } e_2) \{ S \} \rightarrow \text{for}(x \text{ from } e_1 \text{ to } e_2) \{ S' \}}
\end{array}$$

Figure 1.7: Transformation rules for return deferral.

```

secret mut uint32 rval = 0;
secret mut bool notRet = true;
if (sec) { rval = 1; notRet = false; }
if (notRet) {
    // long-running computation ...
}
return rval;

```

The new `notRet` variable tracks whether or not the procedure *would have* returned, and any statement that could be executed after the `return` is guarded by the `notRet` variable. Finally, the actual `return` occurs at the very end of each procedure, returning the value stored in `rval`.

Transformation rules. We formalize return deferral using three kinds of rewrite rules, shown in Figure 1.7. The first *procedure-transformation* rule $\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r \rightarrow f(\vec{x} : \vec{\beta}) \{ S' \} : \beta_r$ is used to rewrite the body S of a procedure f into a `secret`-independent body S' . (This is accomplished using the other two rewrite rules.) The second *guarded-execution* rule $\Phi, pc, rc \vdash S \rightsquigarrow S'$ transforms a statement S , given a secrecy context pc, rc , into S' by making implicit control flow (due to secret returns) explicit. Finally, the *return-elimination* rule $\Phi, pc, rc \vdash S \rightarrow S'$ transforms S into S' by replacing all secret returns with assignments. Below, we walk through some of these rules in detail.

1. Procedure transformation. The TR-RET-DEC rule declares two special (mutable) variables `notRet` and `rval` that respectively hold the `secret`-dependent return state and the value to be returned. The return state `notRet` is set to `true`, while the return value `rval` is initialized to a default value for its type. The rule then eliminates all secret returns from S and inserts a (deferred) `return` after, as the very last statement of the transformed body S' .

2. Guarded execution. Rules TR-RET-GUARD-PUB and TR-RET-GUARD-SEC are used to transform statements that appear *after* any secret returns. Both of these rules first eliminate secret returns from S to obtain S' . If the original statement S is typed with $rc = \text{SEC}$, i.e., S may be preceded by a secret return, then the rule TR-RET-GUARD-SEC additionally *guards* the execution of S' with the condition `notRet`.

3. Return elimination. The bulk of the transformation is done by the remaining rules in Figure 1.7. We omit rules where we either do not transform the statement, or simply recursively transform any sub-statements. Rule TR-RET replaces secret returns by updating `rval` with the (deferred) return value and setting `notRet` to `false`, to signal that subsequent code should *not* be executed.

Rule TR-RET-SEQ handles sequenced statements $S_1; S_2$ by guarding the execution of instructions in S_2 against possible `secret` returns in S_1 . The rule first eliminates the secret returns from the *first* block to get S'_1 . Next, it extracts the secrecy context rc' produced by type checking S_1 . Finally, the rule uses rc' to derive a guarded version of the *second* statement S'_2 .

The TR-RET-FOR rule handles secret returns inside loops. As control flow can jump back to the beginning of a loop, a secret return inside a loop body S can affect the execution of the entire body, as in the following example:

```
for (uint32 i from 0 to 5) {
  b[i] = 1;
  if (i == sec) { return i; }
  a[i] = 2;
}
```

Here, if `i == sec` becomes `true`, the program must stop overwriting the elements in both `a` and `b`. The rule accounts for returns in the body S by using the secrecy context rc' from type checking the body, and in turn, uses this to derive the guarded form of the body S' . In our example, the `secret-dependent` return makes the return context $rc' = \text{SEC}$, and so the entire body is guarded by `notRet`, to obtain the transformed program:

```
for (uint32 i from 0 to 5) {
  // for-loop rule
  if (notRet) {
    b[i] = 1;
    if (i == sec) { rval = i; notRet = false; }
    // sequencing rule
    if (notRet) { a[i] = 2; }
  }
}
```

$$\begin{array}{c}
\text{TR-BR-DEC} \\
\frac{\Phi = (\omega, \{\vec{x} : \vec{\beta}\}, \beta_r) \quad \omega(f) = \text{PUB} \quad \Phi, \text{true} \vdash S \rightarrow S'}{\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r \rightarrow f(\vec{x} : \vec{\beta}) \{ S' \} : \beta_r} \\
\\
\text{TR-BR-DEC-SEC} \\
\frac{\Phi = (\omega, \{\vec{x} : \vec{\beta}\}, \beta_r) \quad \omega(f) = \text{SEC} \quad \Phi, \text{callCtx} \vdash S \rightarrow S'}{\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r \rightarrow f(\vec{x} : \vec{\beta}, \text{callCtx} : \text{BOOL}_{\text{SEC}}) \{ S' \} : \beta_r} \\
\\
\text{TR-BR-IF} \\
\frac{\Gamma \vdash e : \text{BOOL}_{\text{SEC}} \quad \text{FRESH } m_t, m_f \quad \Phi = (\omega, \Gamma, \beta_r) \quad \Phi, (p \& m_t) \vdash S_1 \rightarrow S'_1 \quad \Phi, (p \& m_f) \vdash S_2 \rightarrow S'_2}{\Phi, p \vdash \text{if}(e) \{ S_1 \} \text{ else } \{ S_2 \} \rightarrow \left\{ \begin{array}{l} \text{BOOL}_{\text{SEC}} m_t = e; \\ \text{BOOL}_{\text{SEC}} m_f = \neg m_t; \\ S'_1; S'_2 \end{array} \right\}} \\
\\
\begin{array}{cc}
\text{TR-BR-ASSIGN} & \text{TR-BR-CALL} \\
\frac{p \neq \text{true}}{\Phi, p \vdash e_1 := e_2 \rightarrow e_1 := \text{ctselect}(p, e_2, e_1)} & \frac{\omega(f) = \text{SEC}}{\Phi, p \vdash \beta x = f(\vec{e}) \rightarrow \beta x = f(\vec{e}, p)}
\end{array}
\end{array}$$

Figure 1.8: Transformation rules for branch removal.

1.3.2 Branch removal

Return deferral eliminates `secret` returns by introducing `secret`-dependent branches. In this section we eliminate `secret`-dependent control flow as the final step towards producing constant-time code.

To this end, FaCT replaces `secret` branches with constant-time selections. Consider the following snippet:

```

if      (sec1) { a[1] = 3; }
else if (sec2) { a[2] = 4; }

```

The updates to `a[1]` and `a[2]` are guarded by the `secret` values `sec1` and `sec2` and, therefore, produce memory access patterns that can reveal the values of those secrets when left untransformed—this is the classic *implicit flows* problem [129]. We eliminate the implicit flow in two steps. First,

we track the *control predicates* that correspond to (the conjunction of) the `secret`-conditions.

Then, we perform both memory writes, but use `ctselect` to preserve conditional semantics:

```
a[1] = ctselect( sec1      , 3, a[1]);
a[2] = ctselect(~sec1 & sec2, 4, a[2]);
```

Our general strategy is to transform each conditional array assignment into a re-assignment to a conditional (`ctselect`).

Transforming code that calls procedures is less straightforward: if a procedure takes a mutable parameter, the procedure may update that parameter's value in a way that is visible to the caller. For example:

```
void foo(secret mut uint32 x) { x = 5; }
...
if (sec) {
    foo(x);
    // x is now 5
}
```

The transformation of this code must ensure that updates to `x` only occur if `sec` is `true`. We do so using a *call-context* parameter passed to callee `foo`; this parameter is the caller control predicate—in this case, `sec`—which we use to guard updates in `foo`. Our compiler converts the above into semantically equivalent constant-time code:

```
void foo(secret mut uint32 x,
         secret bool callCtx) {
    x = ctselect(callCtx, 5, x);
}
...
foo(x, sec);
// x is 5 only if sec is true
```

Transformation rules. We formalize branch removal using two kinds of rules, shown in Figure 1.8. The *procedure transformation* rule $\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r \rightarrow f(\vec{x}' : \vec{\beta}') \{ S' \} : \beta_r$ transforms the body S of the procedure f to S' , much like for `secret`-return removals. This rule, however, additionally extends f 's set of parameters \vec{x} to include the extra *call-context* parameter $callCtx$. The *statement transformation* rule $\Phi, p \vdash S \rightarrow S'$, transforms S to S' given context Φ and control predicate p . We walk through some of the rules below.

1. Procedure transformation rule. Both TR-BR-DEC and TR-BR-DEC-SEC remove branches from procedures. TR-BR-DEC transforms procedures that do not depend on `secret` contexts by transforming each procedure's body S into S' using the initial control predicate `true`. TR-BR-DEC-SEC, on the other hand, transforms a procedure f if $\omega(f) = \text{SEC}$, i.e., where f depends on the caller's `secret` context. The rule adds a new parameter `secret bool callCtx` that holds the control predicate at each call-site, and then transforms the body S starting with the initial control predicate `callCtx`.

2. Branch elimination. The remaining rules in Figure 1.8 remove branches from statements. Rule TR-BR-IF, for example, eliminates `secret`-dependent conditional branches by saving the condition (resp. its negation) in the variable m_t (resp. m_f). The “then” statement S_1 (resp. “else” statement S_2) is then transformed after conjoining m_t (resp. m_f) to the control predicate p . To prevent name collision when transforming nested conditionals, the FRESH metafunction guarantees that all m_t and m_f variables have unique names. The declarations of m_t , m_f and transformed branches S'_1, S'_2 are sequenced to obtain the final result.

Rule TR-BR-ASSIGN handles side-effecting assignment statements, using the control predicate to `ctselect` the old or new values. But, if the assignment occurs under the trivial control predicate (i.e., the literal `true`), the assignment is left unchanged.

Finally, rule TR-BR-CALL handles calls to ω -SEC procedures f by explicitly passing the control predicate p as the call-context parameter. This ensures that updates within f only occur according to the caller's control flow.

1.3.3 Compiler correctness and security

In this section, we prove that our compiler preserves semantics and outputs constant-time procedures. To formalize these claims, we define an instrumented semantics that describes procedure behavior and *leakage*, i.e., the sequence of branches taken, the memory addresses accessed, and the operands to variable-time instructions. Intuitively, a procedure is constant-time if its leakage is not influenced by any secret values [15].

In particular, we consider a big-step semantics of the form $F : (\vec{v}, h) \xrightarrow{\psi} (v, h')$ where F is shorthand for a procedure $f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r$, the term \vec{v} represents the values of parameters, h and h' are *heaps* mapping pointers to values, v is the final value of the procedure, and ψ is the leakage. The semantics is parametrized by an allocation function, and the proofs of the claims below rely on several (minor) assumptions on this function. We give these assumptions, formal definition, and complete proofs in Appendix A.

We first prove the *correctness* of our compiler, using the notation $\omega \vdash F \rightarrow F'$ to denote the combined return deferral and branch removal transformations. Compiler correctness states that the compiler preserves the meaning of well-typed statements. To account for new references and variables that are introduced by the compiler pass itself, we show *equivalence* of the final heaps h' and h'' , i.e., for any pointer p' in h' , there is an equivalent pointer p'' in h'' such that $h'(p')$ and $h''(p'')$ are either equal values, or are themselves equivalent pointers.

Theorem 1.3.1 (Compiler correctness). *If $\omega \vdash F \rightarrow F'$ and F' is well-typed, then $F : (\vec{v}, h) \xrightarrow{\psi} (v, h')$ implies that $F' : (\vec{v}, h) \xrightarrow{\psi'} (v, h'')$ and h' and h'' are equivalent.*

Proof sketch. By induction on the derivation. □

Note that our compiler correctness theorem does not make any claim about leakage. We separately prove that the compiler produces constant-time procedures. To this end, we first define the notion of a constant-time procedure.

Definition 1.3.2. A procedure F where $\omega \vdash F$ is constant-time iff for every pair of executions $F : (\vec{v}_1, h_1) \xrightarrow{\psi_1} (v_1, h'_1)$ and $F : (\vec{v}_2, h_2) \xrightarrow{\psi_2} (v_2, h'_2)$, we have $\vec{v}_1, h_1 \equiv \vec{v}_2, h_2$ implies $\psi_1 = \psi_2$, where \equiv is a suitably parametrized notion of equivalence (e.g., *public* or “low” equivalence [7, 15, 156]).

Much like CT-Wasm [156], we cannot prove that *all* FaCT procedures are constant-time—FaCT allows procedures to declassify secret data and call external procedures over which it has no control. We can, however, provide guarantees for a safe subset of *declassify-free* procedures, i.e., procedures that do not contain any `declassify` statements nor call other procedures unless they too are declassify-free (and not `extern`).

Theorem 1.3.3 (Compiler security). *If F is declassify-free and $\omega \vdash F \rightarrow F'$, then F' is constant-time.*

Proof sketch. We define two additional type systems that impose stricter constraints on programs, and prove type-preservation for return deferral and branch removal. We then conclude by proving that the final type system guarantees that programs are constant-time. It is important to note that these type systems are merely proof artifacts, i.e., type checking is not performed again after transformations.

Informally, the two type systems are incremental restrictions on the FaCT type system. The first type system, which we denote by \vdash_{rd} , rejects programs that contain secret returns; the second type system, denoted \vdash_{ct} , rejects programs that branch on secrets.

We then establish type-preservation for return deferral and branch removal:

- ▶ If $\omega \vdash F$ and $\omega \vdash_{\text{rd}} F \rightarrow F'$ then $\omega \vdash_{\text{rd}} F'$.
- ▶ If $\omega \vdash_{\text{rd}} F$ and $\omega \vdash_{\text{ct}} F \rightarrow F'$ then $\omega \vdash_{\text{ct}} F'$.

Both are proved by induction on derivations, using adequate ancillary statements for the induction to go through.

We conclude by proving that \vdash_{ct} guarantees that programs are constant-time. The proof follows from a “locally preserves” unwinding lemma, stating that equivalent states yield equivalent final configurations and equal leakage. \square

1.4 Implementation and evaluation

We implement a prototype compiler for FaCT in ~ 6000 lines of OCaml. The compiler transforms FaCT source code into LLVM IR, which it passes to `clang` (version 6.0.1) to generate assembly or object code. The compiler uses the Z3 SMT solver [49] to check public safety assertions (§1.2.2.3).

We evaluate FaCT by asking the following questions:

- ▶ Is FaCT expressive enough to implement real-world cryptographic algorithms?
- ▶ Does FaCT produce constant-time code?
- ▶ What is FaCT’s performance overhead?
- ▶ Compared to C, does FaCT improve non-experts’ experience reading and writing constant-time code?

We answer the first three questions with case studies in which we integrate FaCT into real-world projects (§1.4.1). We find that FaCT is expressive enough to implement a range of cryptographic primitives. We use `dudect` [125] to empirically check that our implementations, including compiler optimizations, are constant-time. We find that, compared to optimized C code, unoptimized FaCT code runs 16–346% more slowly, while optimized FaCT code ranges from 5% slower to 21% faster.

We answer the fourth question with a study comparing user experiences reading and writing FaCT and C (§1.4.2). In sum, a plurality of participants found FaCT easier to read than C, and a majority found FaCT easier to write.

1.4.1 Case studies

We integrate FaCT into three popular open source libraries by porting pieces of these libraries from C to FaCT:

- ▶ The NaCl `secretbox` API for symmetric-key authenticated encryption and decryption. We port the entire libsodium (version 1.0.16) [50] `secretbox` API, including the two underlying primitives, the Poly1305 message authentication code (MAC) and the XSalsa20 stream cipher.
- ▶ The Curve25519 Elliptic-Curve Diffie-Hellman (ECDH) primitive for asymmetric key exchange. We port Adam Langley’s `curve25519-donna` library [90] in whole.
- ▶ The OpenSSL [114] `ssl3_cbc_digest_record` function used to verify decrypted SSLv3 messages. At its core, this function computes the MAC of a padded message without revealing the padding length. Our implementation invokes OpenSSL’s SHA-1 hash primitive as an `extern` (§1.2.1).
- ▶ The OpenSSL `aesni_cbc_hmac_sha1_cipher` function used in the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) construction. This function performs AES-CBC decryption and then verifies the MAC and padding of the decrypted message. Our implementation invokes OpenSSL’s AES and SHA-1 primitives as `externs`.

We choose these functions because they (1) are complex enough to exercise all of the FaCT language features; (2) implement a range of algorithms; and (3) demonstrate that FaCT can be used in different settings, from implementing individual procedures to large portions of libraries.

Method. We port in three steps. First, we port the C code to FaCT by translating C constructs to their corresponding FaCT counterparts. During this translation process, we label sensitive messages, keys, etc. as `secret`, and add `assume` and `declassify` statements as appropriate to ensure the code typechecks (§1.4.1.1); we also replace “bit hacks” (§1.1) with high-level FaCT constructs (e.g., `if`). Second, we check the correctness of our ports using each

Table 1.1: FaCT case study summary: lines of code (per `cloc`) and uses of `assume` (#A), `declassify` (#D), and `extern` (#E).

Case study	Lines of code		#A	#D	#E
	C	FaCT			
<code>libsodium secretbox</code>	984	1068	16	1	0
<code>curve25519-donna</code>	310	342	0	0	0
<code>OpenSSL record validate</code>	92	91	3	0	2
<code>OpenSSL MEE-CBC</code>	201	219	10	1	4

library’s test harness, and we empirically check that the ports are constant-time using `dudect` (§1.4.1.2). Finally, we use each library’s benchmarking suite to compare our ports to the C implementations (§1.4.1.3).

1.4.1.1 Expressiveness

Table 1.1 summarizes our ports. FaCT implementations are at worst $\sim 10\%$ longer than the corresponding C code. Much of the extra length is because FaCT does not have a macro system; instead, we translated macro definitions and then manually expanded them. (We note that it would be straightforward to instead use the C preprocessor with FaCT.) FaCT code is also more verbose than C when processing buffers: since FaCT has no pointer arithmetic, FaCT code must use extra variables to track offsets into arrays.

Our ports make sparing use of `extern`, `declassify`, and `assume`. For example, our ports use `assume` to help the public safety verifier track values through memory and reason across procedure and language boundaries. We `declassify` in two cases: in `libsodium secretbox` decryption and in `OpenSSL MEE-CBC` verification; these declassifications are permitted by the libraries’ respective attacker models [26, 45, 91]. Finally, we use `extern` to invoke existing primitives (e.g., `OpenSSL`’s SHA-1 implementation).

Table 1.2: Overhead of FaCT ports compared to optimized C, for each benchmark. `secretbox` results are for encryption and decryption overhead, respectively.

Benchmark	% Overhead of FaCT	
	Unoptimized	Optimized
<code>secretbox</code> (reference)	345.57/373.49%	-20.92/-14.56%
<code>secretbox</code> (vectorized)	427.21/427.09%	-6.54/-4.99%
<code>curve25519-donna</code>	144.42%	2.21%
OpenSSL record validate	30.13–35.16%	0.64–4.62%
OpenSSL MEE-CBC	16.15–31.97%	-2.56–4.16%

1.4.1.2 Security

We prove that FaCT’s transformations produce constant-time code (§1.3.3), but this applies only to the *unoptimized* LLVM IR produced by the FaCT compiler.³ Since we use `clang` to generate optimized object code, an LLVM optimization pass might break FaCT’s constant-time guarantees.

To empirically check that our case study implementations run in constant-time, even after optimization, we use the `dudect` [125] analysis tool. At a high level, `dudect` tests for constant-time execution by running the code under test for a large number of iterations and collecting timing information using the CPU’s cycle counters. It then tests the collected timing information for statistically significant variation in execution time that are correlated with changes to secret inputs. In our evaluation, we configure `dudect` to collect 50 million measurements for each benchmark. It finds no statistically significant timing variation.

Several other works concerned with constant-time crypto implementation [14, 125, 139, 156] have reported using `dudect`. In our testing, we found the tool to quickly and reliably find timing differences in buggy code. We note, however, that `dudect` is only a check—not a proof—of constant-time behavior; we discuss further in Section 1.5.

1.4.1.3 Performance

Table 1.2 shows the performance cost of porting C to FaCT. We benchmark each implementation on an Intel i7-6700K at 4GHz with 64GB of RAM using `clang` 6.0.1. We compare both unoptimized and optimized FaCT implementations with C implementations that are compiled at the corresponding project’s default optimization level.⁴ Our optimized FaCT code uses the same optimization flags as the C code.

For `libsodium` and `curve25519-donna`, we use the library’s benchmarking suites. We measure the mean of $\sim 2^{24}$ and $\sim 2^{17}$ iterations, respectively, and report the median of five such measurements. For the OpenSSL implementations, we use OpenSSL’s `s_server` and `s_client` commands to measure throughput when transferring 256MB, 1GB, and 4GB files. We compute the median throughput of five transfers at each file size, and report the minimum and maximum result; overhead was uncorrelated with file size.

For most benchmarks, we find that optimized FaCT is comparable to C: the overhead is never more than 5%. Notably, the FaCT implementation of `libsodium secretbox` is 15-20% *faster* than the C reference implementation. We attribute this speedup to vectorization: inspecting the XSalsa20 assembly code, we find that `clang` generates vector instructions for the FaCT implementation, but not for C. To explore this discrepancy, we measure performance of `secretbox` with XSalsa20 explicitly vectorized (using vectors in FaCT, intrinsics in C). In this case, FaCT is still 5-6% faster than C, but this speedup appears to be an artifact of LLVM’s applying different optimizations to different code.

1.4.2 User study

We evaluate the usability of FaCT by conducting a user study as part of an upper-level, undergraduate programming languages course at UC San Diego.⁵ Prior to the study, we dedicated

³And to procedures that do not use `declassify`.

⁴For OpenSSL, `-O3`; for other projects, `-O2`.

⁵Our study was reviewed and exempted by the IRB.

three lectures to timing side-channels, constant-time programming in general, and constant-time programming specifically in C and FaCT. As an optional assignment, students were asked to (1) *explain* the behavior of constant-time code written in C and FaCT, and (2) *implement* constant-time algorithms in both C and FaCT. Of the 129 enrolled students, 77 completed the study over a nine-day period. We describe methods and conclusions below; in our extended paper [39], we give further lessons from the study, e.g., compilation errors participants ran into frequently.

Method. The user study is a sequence of web-based tasks. For each task, the participant is first given a warm-up code comprehension question, whose answer is subsequently revealed. The participant is then given a second, related question. This question is repeated twice, in C and in FaCT; we randomize the order of the languages per participant, i.e., half the participants' tasks are in C and then FaCT, and vice-versa. On a given question, participants can repeatedly check partial answers for correctness; once finished, the participant *submits* a final answer, which can no longer be viewed or revised. A task is *complete* if the participant submits a final answer for both C and FaCT; we discard incomplete tasks.

The user study was built on an earlier version of FaCT which did not enforce public memory safety. Nevertheless, we believe the results largely translate to the version presented in this chapter, because the surface language did not change significantly.

1.4.2.1 Understanding constant-time code

To evaluate participants' understanding of C and FaCT code, we asked them to describe the behavior of two functions. The first function takes two input buffers—a header and a message—and copies the header and message to an output buffer and adds padding up to a fixed size. The second function implements long division: it computes a quotient and remainder, writes each to an output buffer, and returns a status code indicating success or failure.

We graded participants on their ability to correctly describe each function's behavior. In both cases, we find that participants showed slightly better understanding of FaCT than of C: for

Table 1.3: Number of correct and constant-time solutions for each task: Number of participants (out of 77) that submitted correct and constant-time solutions for each task. The `check_pkcs7_padding` task was misconfigured, and marked variable-time code as constant-time (16 submissions); we report these numbers for completeness (§1.4.2.2).

Programming task	FaCT	C
<code>remove_secret_padding</code>	62	49
<code>check_pkcs7_padding</code>	35	32 (16)
<code>remove_pkcs7_padding</code>	34	24

the first function, the mean score was 57% for FaCT and 53% for C; for the second, it was 40% for FaCT and 32% for C. Participants also reported a slight preference for FaCT; specifically, 31 participants found FaCT easier to understand compared to 10 that found C easier and 28 that reported similar difficulty.

1.4.2.2 Writing constant-time code

To evaluate participants’ ability to write constant-time code in FaCT and C, we had them implement three functions:

- ▶ `remove_secret_padding`: given a buffer and secret length, this function removes any secret padding, i.e., sets every element of the buffer past the length to zero.
- ▶ `check_pkcs7_padding`: this function checks whether a supplied buffer contains a valid PKCS#7 [79] message.
- ▶ `remove_pkcs7_padding`: this function removes padding from a supplied buffer, if it contains a valid message.

Participants could compile their code, run a test suite, and, for C code, check constant-time correctness with `ct-verif` [7]. They could also give up on a task and move to the next one.

Table 1.3 summarizes our findings. Of the 68 participants that completed the first task, 62 submitted correct and constant-time FaCT code, and 49 submitted correct and constant-time C code. For the third task, 34 participants submitted correct, constant-time FaCT code compared

to 24 participants for C. In the survey, 40 participants reported finding FaCT easier to write, 11 found C easier, and 18 found them similar.

We cannot draw conclusions from `check_pkcs7_padding`, because the task had a bug that incorrectly marked variable-time code as constant-time; only 16 of the 32 C submissions marked “correct” were constant-time. The bug was limited to this task, but because `check_pkcs7_padding` is required for `remove_pkcs7_padding`, some participants needed to correct their code to pass the third task.

1.5 Limitations and future work

FaCT makes it easier to write constant-time code, but it is not perfect. Limitations and future work include:

The type system. The type system lacks polymorphism and flow sensitivity [110, 129], which reduces both expressivity and performance. For example, our type system cannot express a program that branches on a buffer’s `public` contents and then decrypts the buffer in-place, upgrading its label to `secret`. We leave such extensions to future work.

The public safety checker. FaCT’s public safety checker does not reason about mutable variables or properties across function calls. For example, indexing an array based on a mutable variable requires `assume`-ing the index is in bounds.

The brittleness of constant-time behavior. FaCT’s compiler only guarantees constant-time behavior for the LLVM IR that it produces. Crucially, this means that LLVM’s optimization passes and lowering to assembly can introduce variable-time behavior. Though many optimizations *do* preserve constant-time property [17], FaCT relies on `dudect` to empirically check that a piece of code is constant-time.

Sound, *symbolic* verification of constant-time behavior using `ct-verif` [7] would give much stronger guarantees. Unfortunately, `ct-verif` currently has limited support for declassification and

vector instructions. Extending `ct-verify` to support these primitives and applying it to optimized FaCT code is future work.

The evaluation. Our evaluation of FaCT is preliminary and thus incomplete. For example, we relied on `extern` versions of SHA-1 and AES (§1.4.1) because we preferred to focus on porting higher-level OpenSSL functions with a history of timing attacks. Moreover, some of the low-level primitives we ported (XSalsa20, Poly1305, and Curve25519) were explicitly designed for ease of constant-time implementation [21,22,24]. Future work is expanding FaCT’s repertoire with potentially more challenging algorithms.

Finally, our user study has limited scope and involves only non-expert users; remedying these issues is also future work.

1.6 Related work

This work supersedes an initial design we previously described in [38]. In particular, we present a design and implementation of a DSL for writing constant-time crypto, provide a formal semantics and security guarantees for FaCT, and evaluate FaCT on several dimensions; in [38] we outlined the vision for such a DSL. Our implementation and formalization efforts revealed insights previously missed in [38]—e.g., the need for *public safety* (§1.2.2.3) and challenges with using `ct-verify` [7] to verify code with inline declassifications. At the same time, in this chapter, we did not explore parts of the design space outlined in [38]—e.g., we do not expose some hardware-specific instructions like `add-with-carry`, which could simplify asymmetric-key crypto implementations.

Domain-specific languages. There are several efforts designing DSLs for implementing cryptographic primitives and protocols. Bernstein’s `qasm` is a low-level portable assembly for writing high-speed crypto routines [23]; it does not distinguish `secret` data from `public` data, so does not prevent information leaks by construction.

Vale [30] and Jasmin [5] are DSLs for writing and verifying high-performance assembly code. Vale developers write platform-independent assembly code and specify the target architecture; the Vale compiler uses Dafny to verify semantics and non-interference. Jasmin allows developers to use architecture-specific instructions alongside higher-level code, and the verified Jasmin compiler rejects non-constant-time programs. Low* is a higher-level, embedded (in F*) DSL that compiles to verified constant-time C [120]. The verified NaCl [25] library, HAACL* [172], is written in Low*. CT-Wasm [156] is a formally verified extension to the WebAssembly language [157] for writing crypto code in the browser. CT-Wasm uses a strict label-based type system to enforce its constant-time policy. These languages provide support for high-level control flow constructs and procedures, but they require developers to manually write constant-time code.

Constant-Time Toolkit (CTTK) is a C library [119] that follows recipes in [46, 118] to provide functions—including low-level constant-time primitives—for crypto libraries, but developers must compose these low-level blocks.

Verification. There is a growing body of work on both building verified cryptographic implementations and verifying existing libraries. Bhargavan et. al verify an implementation of TLS, including low-level cryptographic primitives [27]. Barthe et. al [15] verify constant-time properties of various PolarSSL implementations. Ye et. al [165] verify the mbedTLS implementation of HMAC-DRBG. Appel [11] and Beringer et. al [19] respectively verify OpenSSL’s implementation of SHA-256 and HMAC. Tsai et. al [147] verify core parts of X25519. Almeida et. al [6] verify AWS Lab’s s2n MEE-CBC implementation (after identifying a vulnerability); they also verify security properties of NaCl libraries [8]. Erbsen et. al [53] synthesize and verify elliptic curve implementations from high-level descriptions. Almeida et. al develop ct-verif [7] and verify constant-time properties of several cryptographic algorithms. Many of these verification efforts are specific to the projects being analyzed. Additionally, developers still bear the burden of manually writing constant-time code, which FaCT aims to alleviate.

General techniques for eliminating timing channels. FaCT uses an information flow control type system to eliminate programs that may introduce information leaks or are otherwise inefficient (or impossible) to transform to constant-time. Our label-based type system is a standard IFC type system [129] that borrows explicit mutability from ownership-based systems [43]. Previous solutions have also relied on type- and static-analysis techniques (e.g., [15, 52, 127, 143, 168]) to address timing leaks. FaCT automatically transforms secret sub-computations into constant-time straight-line code. Our approach follows several previous efforts on eliminating timing channels via source code transformations [1, 18, 108, 112, 117, 122]. Most similar in ethos is SC-Eliminator [160]. This system takes as input a program and a list of secrets, and uses tag propagation to transform LLVM IR into its constant-time equivalent. Though both projects perform transformations, they use orthogonal approaches: SC-Eliminator repairs already-existing code, while FaCT is a language for writing such code from the start. Finally, many other efforts employ system-level techniques to eliminate and detect timing-channels [31, 59, 94, 125, 141, 171].

Acknowledgements

We thank the anonymous PLDI and PLDI AEC reviewers and our shepherd Limin Jia for their suggestions and insightful comments. We thank the participants of the Dagstuhl Seminar on Secure Compilation for early feedback on this work, especially Tamara Rezk. We thank Ariana Mirian for handling the IRB for our user study, Shravan Narayan for his help in understanding the subtleties of LLVM, and Joseph Jaeger and Jess Sorrell for helping us understand elliptic curve implementations. We also thank the CSE 130 TAs for their help in testing our user study, and the CSE 130 students for participating in the user study. This work was supported in part by gifts from Fujitsu and Cisco, by the National Science Foundation under Grant Number CNS-1514435, by ONR Grant N000141512750, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapter 1, in part, is a reprint of the material as it appears in 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19). Cauligi, Sunjay; Soeller, Gary; Johannesmeyer, Brian; Brown, Fraser; Wahby, Riad S.; Renner, John; Grégoire, Benjamin; Barthe, Gilles; Jhala, Ranjit; Stefan, Deian, ACM, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Constant-Time Foundations for the New Spectre Era

In which we shore up the earth.

The previous chapter demonstrated how we can compile a high-level language, FaCT, all the way down to low-level code, while enforcing sound constant-time guarantees—as long as we assume a standard sequential execution model. As we will see, *microarchitectural* features—and in particular, *speculative execution*—break many of our constant-time techniques. To reclaim constant-time properties even when accounting for such features, we must develop a new formal strategy for analyzing programs.

In this chapter, we lay the foundations for 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 [15, 40]. These semantics consider very powerful attackers—e.g., attackers in [15] 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. We first define a semantics for an abstract, three-stage (fetch, execute, and retire) machine. Our machine supports out-of-order and speculative execution by modeling *reorder buffers* and *transient instructions*, respectively. We assume that attackers have complete control over microarchitectural features (e.g., the branch target predictor) when executing a victim program and model the attacker’s control over predictors using *directives*. This keeps our semantics simple yet powerful: our semantics abstracts over all predictors when proving security—of course, assuming that predictors themselves do not leak secrets. We further show how our semantics can be extended to capture new predictors—e.g., a hypothetical *memory aliasing* predictor.

We then 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. [35], for example, manifest as violations of our constant-time property.

We further use our semantics as the basis for a prototype analysis tool, Pitchfork, built on top of the `angr` symbolic execution engine [138]. 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 multiple Spectre bugs in real code. We use Pitchfork to detect leaks in the well-known Kocher test cases [85] for Spectre v1, as well as our more extensive test suite which includes Spectre v1.1 variants. More significantly, we use 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>.

2.1 Motivating examples

In this section, we show why classical constant-time programming is insufficient when attackers can exploit microarchitectural features. We do this via two example attacks and show how these attacks are captured by our semantics.

Classical constant time is not enough. Our first example consists of 3 lines of code, shown in Figure 2.1 (top right). The program, a variant of the classical Spectre v1 attack [86], branches on the value of register r_a (line 1). If r_a 's value is smaller than 4, the program jumps to program location 2, where it uses r_a to index into a public array A , saves the value into register r_b , and uses r_b to index into another public array B . If r_a is larger than or equal to 4 (i.e., the index is out of bounds), the program skips the two load instructions and jumps to location 4. 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 at line 1 evaluates to true, even though r_a contains value 9. It will therefore continue down the “true” branch speculatively. In hardware, such guesses are made by a branch prediction unit, which may have been mistrained by an adversary.

These guesses, as well as additional choices such as execution order, are directly supplied by the adversary in our semantics. We model this through a series of *directives*, as shown on the bottom left of Figure 2.1. The directive `fetch: true` instructs our model to speculatively follow the true branch and to place the fetched instruction at index $\bar{1}$ in the *reorder buffer*. Similarly, the two following fetch directives place the loads at indices $\bar{2}$ and $\bar{3}$ in the buffer. The instructions in the reorder buffer, called *transient instructions*, do not necessarily match the original instructions,

Registers		Program	
r	$\rho(r)$	n	$\mu(n)$
r_a	9_{pub}	$\underline{1}$	$\text{br}(>, (4, r_a), \underline{2}, \underline{4})$
	Memory	$\underline{2}$	$(r_b = \text{load}([40, r_a], \underline{3}))$
a	$\mu(a)$	$\underline{3}$	$(r_c = \text{load}([44, r_b], \underline{4}))$
40..43	array A_{pub}	$\underline{4}$...
44..47	array B_{pub}		
48..4B	array Key_{sec}		

Speculative execution:		
Directive	Effect on reorder buffer	Leakage
fetch: true	$\bar{1} \mapsto \text{br}(>, (4, r_a), \underline{2}, (\underline{2}, \underline{4}))$	
fetch	$\bar{2} \mapsto (r_b = \text{load}([40, r_a]))$	
fetch	$\bar{3} \mapsto (r_c = \text{load}([44, r_b]))$	
execute $\bar{2}$	$\bar{2} \mapsto (r_b = \text{Key}[1]_{\text{sec}})$	read 49_{pub}
execute $\bar{3}$	$\bar{3} \mapsto (r_c = X)$	read a_{sec}
	where $a = \text{Key}[1]_{\text{sec}} + 44$	

Figure 2.1: Example demonstrating a Spectre v1 attack. The branch at $\underline{1}$ acts as bounds check for array A . The execution speculatively ignores the bounds check, and leaks a byte of the secret Key .

but can contain additional information (see Table 2.1). For instance, the transient version of the branch instruction records which branch has been speculatively taken.

In our example, the attacker next instructs the model to execute the first load, using the directive execute $\bar{2}$. Because the bounds check has not yet been executed, the load reads from the secret element $Key[1]$, placing the value in r_b . The attacker then issues directive execute $\bar{3}$ to execute the following load; this load’s address is calculated as $44 + Key[1]$. Accessing this address affects externally visible cache state, allowing the attacker to recover $Key[1]$ through a cache side-channel attack [59]. This is encoded by the leakage observation shown in bold on the bottom right. Though this secret leakage cannot happen under sequential execution, our semantics clearly highlights the possible leak when we account for microarchitectural features.

Modeling hypothetical attacks. Next, we give an example of a hypothetical class of Spectre attack captured by our extended semantics. The attack is based on a microarchitectural

feature which would allow processors to speculate whether a store and load pair might operate on the same address, and forward values between them [75, 131].

We demonstrate this attack in Figure 2.2. The reorder buffer, after all instructions have been fetched, is shown in the top right. The program stores the value of register r_b into the $secretKey_{sec}$ array and eventually loads two values from public arrays. The attacker first issues the directive $execute \bar{2} : value$; this results in a buffer where the store instruction at $\bar{2}$ has been modified to record the resolved value x_{sec} . Next, the attacker issues the directive $execute \bar{7} : fwd \bar{2}$, which causes the model to mispredict that the load at $\bar{7}$ aliases with the store at $\bar{2}$, and thus to forward the value x_{sec} to the load. The forwarded value x_{sec} is then used in the address $a = 48 + x_{sec}$ of the load instruction at index $\bar{8}$. There, the loaded value X is irrelevant, but the address a is leaked to the attacker, allowing them to recover the secret value x_{sec} . The speculative execution continues and rolls back when the misprediction is detected (details on this are given in Section 2.2), but at this point, the secret has already been leaked.

As with the example in Figure 2.1, the program in this example follows the (sequential) constant-time discipline, yet leaks during speculative execution. But, both examples are insecure under our new notion of *speculative constant-time* as we discuss next.

2.2 Speculative semantics and security

In this section we define the notion of speculative constant time, and propose a speculative semantics that models execution on modern processors. We start by laying the groundwork for our definitions and semantics.

Configurations. A configuration $C \in \text{Confs}$ represents the state of execution at a given step. It is defined as a tuple (ρ, μ, n, buf) where:

- ▶ $\rho : \mathcal{R} \rightarrow \mathcal{V}$ is a map from a finite set of register names \mathcal{R} to values;
- ▶ $\mu : \mathcal{V} \rightarrow \mathcal{V}$ is a memory;

Table 2.1: Instructions and their transient instruction form.

Instruction	Transient form(s)
arithmetic operation (<i>op</i> specifies opcode)	$(r = \text{op}(op, \vec{n}, n'))$ $(r = v_\ell)$ (unresolved <i>op</i>) (resolved value)
conditional branch	$\text{br}(op, \vec{n}, n^{\text{true}}, n^{\text{false}})$ $\text{jump } n_0$ (unresolved conditional) (resolved conditional)
memory load (at program point <i>n</i>)	$(r = \text{load}(\vec{n}))^n$ (unresolved load) $(r = \text{load}(\vec{n}, (v_\ell, j)))^n$ (partially resolved load with dependencies) $(r = v_\ell\{_, a\})^n$ (resolved load without dependencies) $(r = v_\ell\{j, a\})^n$ (resolved load with dependency on <i>j</i>)
memory store	$\text{store}(n, \vec{n})$ (unresolved store) $\text{store}(v_\ell, a_\ell)$ (resolved store)
indirect jump	$\text{jmpi}(\vec{n})$ (unresolved jump predicted to n_0)
function calls	$\text{call}(n_f, n_{ret})$ (unresolved call) ret (unresolved return)
speculation fence	$\text{fence } n$ (no resolution step)

Registers		Reorder buffer	
r	$\rho(r)$	i	$buf(i)$
r_a	2_{pub}	$\bar{2}$	$\text{store}(r_b, [40, r_a])$
r_b	x_{sec}		\dots
Memory		$\bar{7}$	$(r_c = \text{load}([45]))$
a	$\mu(a)$	$\bar{8}$	$(r_c = \text{load}([48, r_c]))$
40..43	$\text{secretKey}_{\text{sec}}$		
44..47	$\text{pubArrA}_{\text{pub}}$		
48..4B	$\text{pubArrB}_{\text{pub}}$		

Directive	Speculative execution Effect on buf	Leakage
execute $\bar{2}$: value	$\bar{2} \mapsto \text{store}(x_{\text{sec}}, [40, r_a])$	
execute $\bar{7}$: fwd $\bar{2}$	$\bar{7} \mapsto (r_c = \text{load}([45], x_{\text{sec}}, \bar{2}))$	
execute $\bar{8}$	$\bar{8} \mapsto (r_c = X\{\perp, a\})$	read a_{sec}
execute $\bar{2}$: addr	$\bar{2} \mapsto \text{store}(r_b, 42_{\text{pub}})$	fwd 42_{pub}
execute $\bar{7}$	$\{\bar{7}, \bar{8}\} \notin buf$	rollback, fwd 45_{pub}
	where $a = x_{\text{sec}} + 48$	

Figure 2.2: Example demonstrating a hypothetical attack abusing an aliasing predictor. This attack differs from prior speculative data forwarding attacks in that branch misprediction is not needed.

- $n : \mathcal{V}$ is the current program point;
- $buf : \mathbb{N} \rightarrow \text{TransInstr}$ is the reorder buffer.

Values and labels. As a convention, we use n for memory addresses that map to instructions, and a for addresses that map to data. Each value is annotated with a label from a lattice of security labels with join operator \sqcup . For brevity, we sometimes omit public label annotation on values.

Using labels, we define an equivalence \simeq_{pub} on configurations. We say that two configurations are equivalent if they coincide on public values in registers and memories.

Reorder buffer. The *reorder buffer* maps buffer indices (natural numbers) to transient instructions. We write $buf(i)$ to denote the instruction at index i in buffer buf , if i is in buf 's domain. We write $buf[i \mapsto \underline{\text{instr}}]$ to denote the result of extending buf with the mapping from

$$(buf +_i \rho)(r) = \begin{cases} v_\ell & \text{if } \max(j) < i : buf(j) = (r = _) \wedge \\ & \quad buf(j) = (r = v_\ell) \\ \rho(r) & \text{if } \forall j < i : buf(j) \neq (r = _) \\ \perp & \text{otherwise} \end{cases}$$

Figure 2.3: Definition of the register resolve function.

i to instr , and $buf \setminus buf(i)$ for the function formed by removing i from buf 's domain. We write $buf[j : j < i]$ to denote the restriction of buf 's domain to all indices j , s.t. $j < i$ (i.e., removing all mappings at indices i and greater). Our rules add and remove indices in a way that ensures that buf 's domain will always be contiguous.

Notation. We let $\text{MIN}(M)$ (resp. $\text{MAX}(M)$) denote the minimum (maximum) index in the domain of a mapping M . We denote the empty mapping as \emptyset and let $\text{MIN}(\emptyset) = \text{MAX}(\emptyset) = 0$.

For a formula φ , we may discuss the bounded highest (lowest) index for which a formula holds. We write $\max(j) < i : \varphi(j)$ to mean that j is the highest index less than i for which φ holds, and define $\min(j) > i : \varphi(j)$ analogously.

Register resolve function. In Figure 2.3, we define the *register resolve function*, which we use to determine the value of a register in the presence of transient instructions in the reorder buffer. For index i and register r , the function may **(1)** return the latest assignment to r prior to position i in the buffer, if the corresponding operation is already resolved; **(2)** return the value from the register map ρ , if there are no pending assignments to r in the buffer; or **(3)** be undefined. Note that if the latest assignment to r is yet unresolved then $(buf +_i \rho)(r) = \perp$. We extend this definition to values by defining $(buf +_i \rho)(v_\ell) = v_\ell$ for all $v_\ell \in \mathcal{V}$, and lift it to lists of registers or values using a pointwise lifting.

2.2.1 Speculative constant-time

We present our new notion of constant-time security in terms of a small-step semantics, which relates program configurations, observations, and attacker directives.

Our semantics does not directly model caches, nor any of the predictors used by speculative semantics. Rather, we model externally visible effects—memory accesses and control flow—by producing a sequence of *observations*. We can thus reason about *any* possible cache implementation, as any cache eviction policy can be expressed as a function of the sequence of observations. Furthermore, exposing control flow observations directly in our semantics 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 [86], they only serve to leak the path taken through the code—and thus modeling these observations separately would be redundant. For the same reason, we do not model a particular branch prediction strategy; we instead let the attacker resolve scheduling non-determinism by supplying a series of *directives*.

This approach has two important consequences. First, the use of observations and directives 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.

Given an attacker directive d , we use $C \xrightarrow{d} C'$ to denote the execution step from configuration C to configuration C' that produces observation o . Program execution is defined from the small-step semantics in the usual style. We use $C \Downarrow_D^N C'$ to denote a sequence of execution steps from C to C' . Here D and O are the concatenation of the single-step directives and leakages, respectively; N is the number of retired instructions, i.e., $N = \#\{d \in D \mid d = \text{retire}\}$. When such a big step from C to C' is possible, we say D is a *well-formed* schedule of directives for C . We omit D , N , or O when not used.

Definition 2.2.1 (Speculative constant-time). *We say a configuration C with schedule D satisfies speculative constant-time (SCT) with respect to a low-equivalence relation \simeq_{pub} iff for every C'*

such that $C \simeq_{pub} C'$:

$$C \Downarrow_O C_1 \text{ iff } C' \Downarrow_{O'} C'_1 \text{ and } C_1 \simeq_{pub} C'_1 \text{ and } O = O'.$$

A program satisfies SCT iff every initial configuration satisfies SCT under any schedule.

Aside, on sequential execution. Processors work hard to create the illusion that assembly instructions are executed sequentially. We validate our semantics by proving equivalence with respect to sequential execution. Formally, we define *sequential schedules* as schedules that execute and retire instructions immediately upon fetching them. We attach to each program a canonical sequential schedule and write $C \Downarrow_{seq}^N C'$ to model execution under this canonical schedule. Our sequential validation is defined relative to an equivalence \approx on configurations. Informally, two configurations are equivalent if their memories and register files are equal, even if their speculative states may be different.

Theorem 2.2.2 (Sequential equivalence). *Let C be an initial configuration and D a well-formed schedule for C . If $C \Downarrow_D^N C_1$, then $C \Downarrow_{seq}^N C_2$ and $C_1 \approx C_2$.*

Complete definitions, more properties, and proofs are given in Appendix B.

2.2.2 Overview of the semantics

As shown in Table 2.1, each instruction has a *physical* form and one or more *transient* forms. Our semantics operates on these instructions similar to a multi-stage processor pipeline. Physical instructions are *fetched* from memory and become transient instructions in the reorder buffer. They are then *executed* until they are fully resolved. Finally they are *retired*, updating the non-speculative state in the configuration.

In the rest of this section, we show how we model speculative execution (Section 2.2.3), memory operations (Section 2.2.4), aliasing prediction (Section 2.2.5), and fence instructions

(Section 2.2.6). We also extend our semantics with indirect jumps (Section 2.2.7) and function calls (Section 2.2.8).

Our semantics captures a variety of existing Spectre variants, including v1 (Figure 2.1), v1.1 (Figure 2.5), and v4 (Figure 2.6), as well as a new hypothetical variant (Figure 2.2). Additional variants (e.g., v2 and *ret2spec*) can be expressed with the extended semantics given in Sections 2.2.7 and 2.2.8. Our semantics shows that these attacks violate SCT by producing observations depending on secrets.

2.2.3 Speculative execution

We start with the semantics for *conditional branches* which introduce speculative execution.

Conditional branching. The physical instruction for conditional branches has the form $\text{br}(op, \vec{r}, n^{\text{true}}, n^{\text{false}})$, where op is a Boolean operator whose result determines whether or not to execute the jump, \vec{r} are the operands to op , and n^{true} and n^{false} are the program points for the *true* and *false* branches, respectively.

We show br 's transient counterparts in Table 2.1. The unresolved form extends the physical instruction with a program point n_0 , which is used to record the branch that is executed (n^{true} or n^{false}) speculatively, and may or may not correspond to the branch that is actually taken once op is resolved. The resolved form contains the final jump target.

Fetch. We give the rule for the fetch stage below.

$$\begin{array}{c}
 \text{COND-FETCH} \\
 \mu(n) = \text{br}(op, \vec{r}, n^{\text{true}}, n^{\text{false}}) \quad i = \text{MAX}(buf) + 1 \\
 buf' = buf[i \mapsto \text{br}(op, \vec{r}, n^{\text{true}}, (n^{\text{true}}, n^{\text{false}}))] \\
 \hline
 (\rho, \mu, n, buf) \xrightarrow{\text{fetch: true}} (\rho, \mu, n^{\text{true}}, buf')
 \end{array}$$

The COND-FETCH rule speculatively executes the branch determined by a Boolean value b given by the directive. We show the case for $b = \text{true}$; the case for false is analogous. The rule updates the current program point n , allowing execution to continue along the specified branch. The rule then records the chosen branch n^{true} (resp. n^{false}) in the transient jump instruction.

This semantics models the behavior of most modern processors. Since the target of the branch cannot be resolved in the fetch stage, speculation allows execution to continue and not stall until the branch target is resolved. In hardware, a branch predictor chooses which branch to execute; in our semantics, the directives `fetch: true` and `fetch: false` determine which of the rules to execute. This allows us to abstract over all possible predictor implementations.

Execute. Next, we describe the rules for the execute stage.

COND-EXECUTE-CORRECT

$$\begin{array}{c}
\text{buf}(i) = \text{br}(op, \vec{n}, n_0, (n^{\text{true}}, n^{\text{false}})) \quad \forall j < i : \text{buf}(j) \neq \text{fence} \\
\text{(buf} +_i \rho)(\vec{n}) = \vec{v}_\ell \quad \llbracket op(\vec{v}_\ell) \rrbracket = \text{true}_\ell \quad n^{\text{true}} = n_0 \quad \text{buf}' = \text{buf}[i \mapsto \text{jump } n^{\text{true}}] \\
\hline
(\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i]{\text{jump } n_\ell^{\text{true}}} (\rho, \mu, n, \text{buf}')
\end{array}$$

COND-EXECUTE-INCORRECT

$$\begin{array}{c}
\text{buf}(i) = \text{br}(op, \vec{n}, n_0, (n^{\text{true}}, n^{\text{false}})) \quad \forall j < i : \text{buf}(j) \neq \text{fence} \quad (\text{buf} +_i \rho)(\vec{n}) = \vec{v}_\ell \\
\llbracket op(\vec{v}_\ell) \rrbracket = \text{true}_\ell \quad n^{\text{true}} \neq n_0 \quad \text{buf}' = \text{buf}[j : j < i][i \mapsto \text{jump } n^{\text{true}}] \\
\hline
(\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i]{\text{rollback, jump } n_\ell^{\text{true}}} (\rho, \mu, n^{\text{true}}, \text{buf}')
\end{array}$$

Both rules evaluate the condition op via an evaluation function $\llbracket \cdot \rrbracket$. In both, the function produces `true`; but the `false` rules are analogous. The rules then compare the actual branch target n_{true} against the speculatively chosen target n_0 from the fetch stage.

If the *correct* path was chosen during speculation, i.e., n_0 agrees with the correct branch n^{true} , rule COND-EXECUTE-CORRECT updates buf with the fully resolved jump instruction and emits an observation: `jump n_ℓ^{true}` . This models an attacker that can observe control flow, e.g., by

Table 2.2: Correct and incorrect branch prediction. Initially, $r_a = 3$. In (b), the misprediction causes a rollback to $\bar{4}$.

(a) Predicted correctly			(b) Predicted incorrectly		
i	Initial $buf(i)$	$buf(i)$ after exe $\bar{4}$	i	Initial $buf(i)$	$buf(i)$ after exe $\bar{4}$
$\bar{3}$	$(r_b = 4)$	$(r_b = 4)$	$\bar{3}$	$(r_b = 4)$	$(r_b = 4)$
$\bar{4}$	$br(<, (2, r_a), \underline{9}, (\underline{9}, \underline{12}))$	jump $\underline{9}$	$\bar{4}$	$br(<, (2, r_a), \underline{12}, (\underline{9}, \underline{12}))$	jump $\underline{9}$
$\bar{5}$	$(r_c = op(+, (1, r_b)))$	$(r_c = op(+, (1, r_b)))$	$\bar{5}$	$(r_d = op(*, (r_g, r_h)))$	-

timing executions along different paths. The leaked observation n^{true} has label ℓ , propagated from the evaluation of the condition.

In case the *wrong* path was taken during speculation, i.e., the calculated branch n^{true} *disagrees* with n_0 , the semantics must roll back all execution steps along the erroneous path. For this, rule COND-EXECUTE-INCORRECT removes all entries in buf that are newer than the current instruction (i.e., all entries $j \geq i$), sets the program point n to the correct branch, and updates buf at index i with correct value for the resolved jump instruction. Since an attacker can observe misspeculation through instruction timing [86], the rule issues a `rollback` observation in addition to the `jump` observation.

Retire. The rule for the retire stage is shown below; its only effect is to remove the jump instruction from the buffer.

$$\begin{array}{c}
 \text{JUMP-RETIRE} \\
 \hline
 \text{MIN}(buf) = i \quad buf(i) = \text{jump } n_0 \quad buf' = buf \setminus buf(i) \\
 \hline
 (\rho, \mu, n, buf) \xrightarrow{\text{retire}} (\rho, \mu, n, buf')
 \end{array}$$

Examples. Table 2.2 shows how branch prediction affects the reorder buffer. In part (a), the branch at index $\bar{4}$ is predicted correctly. The jump instruction is resolved, and execution proceeds as normal. In part (b), the branch at index $\bar{4}$ is incorrectly predicted. Upon executing the branch, the misprediction is detected, and buf is rolled back to index $\bar{4}$.

2.2.4 Memory operations

The physical instruction for loads is $(r = \text{load}(\vec{n}, n'))$, while the form for stores is $\text{store}(rv, \vec{n}, n')$. As before, n' is the program point of the next instruction. For loads, r is the register receiving the result; for stores, rv is the register or value to be stored. For both loads and stores, \vec{n} is a list of operands (registers and values) which are used to calculate the operation's target address.

Transient counterparts of load and store are given in Table 2.1. We annotate unresolved load instructions with the program point of the physical instruction that generated them; we omit annotations whenever not used. Unresolved and resolved store instructions share the same syntax, but for resolved stores, both address and operand are required to be single values.

Address calculation. We assume an arithmetic operator $addr$ which calculates target addresses for stores and loads from its operands. We leave this operation abstract in order to model a large variety of architectures. For example, in a simple addressing mode, $\llbracket addr(\vec{v}) \rrbracket$ might compute the sum of its operands; in an x86-style address mode, $\llbracket addr([v_1, v_2, v_3]) \rrbracket$ might instead compute $v_1 + v_2 \cdot v_3$.

Store forwarding. Multiple transient load and store instructions may exist concurrently in the reorder buffer. In particular, there may be unresolved loads and stores that will read or write to the same address in memory. Under a naive model, we must wait to execute load instructions until all prior store instructions have been retired, in case they write to the address we will load from. Indeed, some real-world processors behave exactly this way [42].

For performance, most modern processors implement *store-forwarding* for memory operations: if a load reads from the same address as a prior store and the store has already been resolved, the processor can *forward* the resolved value to the load. The load can then proceed without waiting for the store to commit to memory [159].

To model these store forwarding semantics, we use annotations to recall if a load was resolved from memory or forwarding. A resolved load has the form $(r = v_\ell\{j, a\})^n$, where the

index j records either the buffer index of the store instruction that forwarded its value to the load, or \perp if the value was taken from memory. We also record the memory address a associated with the data, and retain the program point n of the load instruction that generated the value instruction. The resolved load otherwise behaves as a resolved value instruction (e.g., for the register resolve function).

Fetch. We now discuss the inference rules for memory operations, starting with the fetch stage.

SIMPLE-FETCH

$$\frac{\begin{array}{l} \mu(n) \in \{\text{op}, \text{load}, \text{store}, \text{fence}\} \\ n' = \text{next}(\mu(n)) \quad i = \text{MAX}(\text{buf}) + 1 \quad \text{buf}' = \text{buf}[i \mapsto \text{transient}(\mu(n))] \end{array}}{(\rho, \mu, n, \text{buf}) \xrightarrow[\text{fetch}]{} (\rho, \mu, n', \text{buf}')}$$

Given a fetch directive, rule SIMPLE-FETCH extends the reorder buffer buf with a new transient instruction (see Table 2.1). Other than load and store, the rule also applies to op and fence instructions. The $\text{transient}(\cdot)$ function simply translates the physical instruction at $\mu(n)$ to its unresolved transient form. It inserts the new, transient instruction at the first empty index in buf , and sets the current program point to the next instruction n' . Note that $\text{transient}(\cdot)$ annotates the transient load instruction with its program point.

Load execution. Next, we cover the rules for the load execute stage.

LOAD-EXECUTE-NODEP

$$\frac{\begin{array}{l} \text{buf}(i) = (r = \text{load}(\vec{r}))^n \quad \forall j < i : \text{buf}(j) \neq \text{fence} \\ (\text{buf} +_i \rho)(\vec{r}) = \vec{v}_\ell \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \\ \ell_a = \sqcup \vec{\ell} \quad \forall j < i : \text{buf}(j) \neq \text{store}(_, a) \quad \mu(a) = v_\ell \quad \text{buf}' = \text{buf}[i \mapsto (r = v_\ell\{\perp, a\})^n] \end{array}}{(\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i]{\text{read } a_{\ell_a}} (\rho, \mu, n, \text{buf}')}$$

LOAD-EXECUTE-FORWARD

$$\begin{array}{c}
\text{buf}(i) = (r = \text{load}(\vec{rv}))^n \quad \forall j < i : \text{buf}(j) \neq \text{fence} \\
(\text{buf} +_i \rho)(\vec{rv}) = \vec{v}_\ell \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \quad \ell_a = \perp \vec{\ell} \\
\text{max}(j) < i : \text{buf}(j) = \text{store}(_, a) \wedge \text{buf}(j) = \text{store}(v_\ell, a,) \quad \text{buf}' = \text{buf}[i \mapsto (r = v_\ell\{j, a\})^n] \\
\hline
(\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i]{\text{fwd } a_{\ell_a}} (\rho, \mu, n, \text{buf}')
\end{array}$$

Given an execute directive for buffer index i , under the condition that i points to an unresolved load, rule LOAD-EXECUTE-NODEP applies if there are no prior store instructions in buf that have a resolved, matching address. The rule first resolves the operand list \vec{rv} into a list of values \vec{v}_ℓ , and then uses \vec{v}_ℓ to calculate the target address a . It then retrieves the current value v_ℓ at address a from memory, and finally adds to the buffer a resolved value instruction assigning v_ℓ to the target register r . We annotate the value instruction with the address a and \perp , signifying that the value comes from memory. Finally, the rule produces the observation $\text{read } a_{\ell_a}$, which renders the memory read at address a with label ℓ_a visible to an attacker.

Rule LOAD-EXECUTE-FORWARD applies if the most recent store instruction in buf with a resolved, matching address has a resolved data value. Instead of accessing memory, the rule forwards the value from the store instruction, annotating the new value instruction with the calculated address a and the index j of the originating store instruction. The rule produces a fwd observation with the labeled address a_{ℓ_a} . This observation captures that the attacker can determine (e.g., by observing the *absence* of memory access using a cache timing attack) that a forwarded value from address a was found in the buffer instead of loaded from memory.

Importantly, neither of the rules has to wait for prior stores to be resolved and can proceed speculatively. This can lead to memory hazards when a more recent store to the load's address has not been resolved yet; we show how to deal with hazards in the rules for the store instruction.

Store execution. We show the rules for stores below.

STORE-EXECUTE-VALUE

$$\begin{array}{c}
 \text{buf}(i) = \text{store}(rv, \vec{rv}) \\
 \hline
 \forall j < i : \text{buf}(j) \neq \text{fence} \quad (\text{buf} +_i \rho)(rv) = v_\ell \quad \text{buf}' = \text{buf}[i \mapsto \text{store}(v_\ell, \vec{rv})] \\
 \hline
 (\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i : \text{value}]{} (\rho, \mu, n, \text{buf}')
 \end{array}$$

STORE-EXECUTE-ADDR-OK

$$\begin{array}{c}
 \text{buf}(i) = \text{store}(rv, \vec{rv}) \\
 \hline
 \forall j < i : \text{buf}(j) \neq \text{fence} \quad (\text{buf} +_i \rho)(\vec{rv}) = \vec{v}_\ell \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \quad \ell_a = \sqcup \vec{\ell} \\
 \forall k > i : \text{buf}(k) = (r = \dots \{j_k, a_k\}) : \quad (a_k = a \Rightarrow j_k \geq i) \wedge (j_k = i \Rightarrow a_k = a) \\
 \text{buf}' = \text{buf}[i \mapsto \text{store}(rv, a_{\ell_a})] \\
 \hline
 (\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i : \text{addr}]{\text{fwd } a_{\ell_a}} (\rho, \mu, n, \text{buf}')
 \end{array}$$

STORE-EXECUTE-ADDR-HAZARD

$$\begin{array}{c}
 \text{buf}(i) = \text{store}(rv, \vec{rv}) \\
 \hline
 \forall j < i : \text{buf}(j) \neq \text{fence} \quad (\text{buf} +_i \rho)(\vec{rv}) = \vec{v}_\ell \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \quad \ell_a = \sqcup \vec{\ell} \\
 \min(k) > i : \text{buf}(k) = (r = \dots \{j_k, a_k\})^{n_k} : \quad (a_k = a \wedge j_k < i) \vee (j_k = i \wedge a_k \neq a) \\
 \text{buf}' = \text{buf}[j : j < k][i \mapsto \text{store}(rv, a_{\ell_a})] \\
 \hline
 (\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i : \text{addr}]{\text{rollback, fwd } a_{\ell_a}} (\rho, \mu, n_k, \text{buf}')
 \end{array}$$

The execution of store is split into two steps: value resolution, represented by the directive `execute i : value`, and address resolution, represented by the directive `execute i : addr`; a schedule may have either step first. Either step may be skipped if data or address are already in immediate form.

Rule STORE-EXECUTE-ADDR-OK applies if no misprediction has been detected, i.e., if no load instruction forwarded data from an outdated store. We check this by requiring that all value instructions *after* the current index (indices $k > i$) with an address a matching the current store must be using a value forwarded from a store *at least as recent* as this one ($a_k = a \Rightarrow j_k \geq i$). We

define $\perp < n$ for any index n —that is, if a future load matches the address of the current store but loaded its value from memory, we consider this a hazard.

If there is indeed a hazard, i.e., if there was a resolved load with an outdated value, the rule STORE-EXECUTE-ADDR-HAZARD picks the *earliest* such instruction (index k) and restarts execution by resetting the instruction pointer to the program point n_k of this instruction. It then discards all transient instructions at indices at least k from the reorder buffer. As in the case of misspeculation, the rule issues a `rollback` observation.

Retire. Resolved loads are retired using the following rule.

VALUE-RETIRE

$$\frac{\text{MIN}(buf) = i \quad buf(i) = (r = v_\ell) \quad \rho' = \rho[r \mapsto v_\ell] \quad buf' = buf \setminus buf(i)}{(\rho, \mu, n, buf) \xrightarrow[\text{retire}]{} (\rho', \mu, n, buf')}$$

This is the same retire rule used for simple value instructions (e.g., resolved op instructions). The rule updates the register map ρ with the new value, and removes the instruction from the reorder buffer.

Stores are retired using the rule below.

STORE-RETIRE

$$\frac{\text{MIN}(buf) = i \quad buf(i) = \text{store}(v_\ell, a_{\ell_a}) \quad \mu' = \mu[a \mapsto v_\ell] \quad buf' = buf \setminus buf(i)}{(\rho, \mu, n, buf) \xrightarrow[\text{retire}]{\text{write } a_{\ell_a}} (\rho, \mu', n, buf')}$$

A fully resolved store instruction retires similarly to a value instruction. However, instead of updating the register map ρ , rule STORE-RETIRE updates the memory μ . Since an attacker can observe memory writes, the rule produces the observation `write` a_{ℓ_a} with the labeled address of the store.

Example. Figure 2.4 gives an example of store-to-load forwarding. In the starting configuration, the store at index $\bar{2}$ is fully resolved, while the store at index $\bar{3}$ has an unresolved

Registers	$\rho(r_a) = 40_{\text{pub}}$	
Directives	D= execute $\bar{4}$; execute $\bar{3}$: addr	
Leakage for D	fwd 43_{pub} ; rollback, fwd 43_{pub}	
starting <i>buf</i>	<i>buf</i> after execute $\bar{4}$	<i>buf</i> after D
$\bar{2}$ store(12, 43_{pub})	$\bar{2}$ store(12, 43_{pub})	$\bar{2}$ store(12, 43_{pub})
$\bar{3}$ store(20, $[3, r_a]$)	$\bar{3}$ store(20, $[3, r_a]$)	$\bar{3}$ store(20, 43_{pub})
$\bar{4}$ ($r_c = \text{load}([43])$)	$\bar{4}$ ($r_c = 12\{\bar{2}, 43\}$)	

Figure 2.4: Store hazard caused by late execution of store addresses. The store address for $\bar{3}$ is resolved too late, causing the later load instruction to forward from the wrong store. When $\bar{3}$'s address is resolved, the execution must be rolled back. In this example, $\llbracket \text{addr}(\cdot) \rrbracket$ adds its arguments.

Registers		Reorder buffer	
r	$\rho(r)$	i	$\text{buf}(i)$
r_a	5_{pub}	$\bar{1}$	br(>, (4, r_a), $\bar{2}$, ($\bar{2}$, $\bar{4}$))
r_b	x_{sec}	$\bar{2}$	store(r_b , $[40, r_a]$)
Memory			...
a	$\mu(a)$	$\bar{7}$	($r_c = \text{load}([45])$)
40..43	$\text{secretKey}_{\text{sec}}$	$\bar{8}$	($r_c = \text{load}([48, r_c])$)
44..47	$\text{pubArrA}_{\text{pub}}$		
48..4B	$\text{pubArrB}_{\text{pub}}$		
Directive	Effect on <i>buf</i>	Leakage	
execute $\bar{2}$: addr	$\bar{2} \mapsto \text{store}(r_b, 45_{\text{pub}})$	fwd 45_{pub}	
execute $\bar{2}$: value	$\bar{2} \mapsto \text{store}(x_{\text{sec}}, 45_{\text{pub}})$		
execute $\bar{7}$	$\bar{7} \mapsto (r_c = x_{\text{sec}}\{\bar{2}, 45\})$	fwd 45_{pub}	
execute $\bar{8}$	$\bar{8} \mapsto (r_c = X\{\perp, a\})$	read a_{sec}	
where $a = x_{\text{sec}} + 48$			

Figure 2.5: Example demonstrating a store-to-load Spectre v1.1 attack. A speculatively stored value is forwarded and then leaked using a subsequent load instruction.

Registers		Reorder buffer	
r	$\rho(r)$	i	$buf(i)$
r_a	40_{pub}	$\bar{2}$	$store(0, [3, r_a])$
	Memory	$\bar{3}$	$(r_c = load([43]))$
a	$\mu(a)$	$\bar{4}$	$(r_c = load([44, r_c]))$
40..43	$secretKey_{sec}$		
44..47	$pubArrA_{pub}$		

Directive	Effect on buf	Leakage
execute $\bar{3}$	$\bar{3} \mapsto (r_c = secretKey[3]\{\perp, 43\})$	read 43_{pub}
execute $\bar{4}$	$\bar{4} \mapsto (r_c = X\{\perp, a\})$	read a_{sec}
execute $\bar{2} : addr$	$\{\bar{3}, \bar{4}\} \notin buf$ $\bar{2} \mapsto store(0, 43_{pub})$	rollback, fwd 43_{pub}

where $a = secretKey[3]_{sec} + 44$

Figure 2.6: Example demonstrating a v4 Spectre attack. The store is executed too late, causing later load instructions to use outdated values.

address. The first directive executes the load at $\bar{4}$. This load accesses address 43, which matches the store at index $\bar{2}$. Since this is the most recent such store and has a resolved value, the load gets the value 12 from this store. The following directive resolves the address of the store at index $\bar{3}$. This store also matches address 43. As this store is more recent than store $\bar{2}$, this directive triggers a hazard for the load at $\bar{4}$, leading to the rollback of the load from the reorder buffer.

Capturing Spectre. We now have enough machinery to capture several variants of Spectre attacks.

We discussed how our semantics model Spectre v1 in Section 2.1 (Figure 2.1). Figure 2.5 shows a simple disclosure gadget using forwarding from an out-of-bounds write. In this example, a secret value x_{sec} is supposed to be written to $secretKey$ at an index r_a as long as r_a is within bounds. However, due to branch misprediction, the store instruction is executed despite r_a being too large. The load instruction at index $\bar{7}$, normally benign, now aliases with the store at index $\bar{2}$, and receives the secret x_{sec} instead of a public value from $pubArrA$. This value is then used as the address of another load instruction, causing x_{sec} to leak.

Figure 2.6 shows a Spectre v4 vulnerability caused when a store *fails* to forward to a future load. In this example, the load at index $\bar{3}$ executes before the store at $\bar{2}$ calculates its

address. As a result, this execution loads the outdated secret value at address 43 and leaks it, instead of using the public zeroed-out value that would be written.

2.2.5 Aliasing prediction

We extend the memory semantics from the previous section to model aliasing prediction by introducing a new transient instruction $(r = \text{load}(\vec{r}, (v_\ell, j)))^n$. This instruction represents a *partially resolved* load with speculatively forwarded data. As before, r is the target register, \vec{r} is the list of arguments for address calculation, and n is the program point of the physical load instruction. The new parameters are v_ℓ , the forwarded data, and j , the index of the originating store.

Forwarding via prediction.

$$\begin{array}{c}
 \text{LOAD-EXECUTE-FORWARDED-GUESSED} \\
 \text{buf}(i) = (r = \text{load}(\vec{r}))^n \quad j < i \quad \forall k < i : \text{buf}(k) \neq \text{fence} \\
 \text{buf}(j) = \text{store}(v_\ell, \vec{r}_j) \quad \text{buf}' = \text{buf}[i \mapsto (r = \text{load}(\vec{r}, (v_\ell, j)))^n] \\
 \hline
 (\rho, \mu, n, \text{buf}) \xrightarrow{\text{execute } i: \text{fwd } j} (\rho, \mu, n, \text{buf}')
 \end{array}$$

Rule LOAD-EXECUTE-FORWARDED-GUESSED implements forwarding in the presence of unresolved target addresses. Instead of forwarding the value from a store with a matching address, as in Section 2.2.4, the attacker can now freely choose to forward from *any* store with a resolved value—even if its target address is not known yet. Given a choice of which store j to forward from—supplied via directive—the rule updates the reorder buffer with the new partially resolved load and records both the forwarded value v_ℓ and the buffer index j of the store instruction.

Register resolve function. We extend the register resolve function $(\text{buf} +_i \rho)$ to allow using values from partially resolved loads. In particular, whenever the register resolve function computes the latest resolved assignment to some register r , it now considers not only fully resolved

value instructions, but also our new partially resolved load: whenever the latest assignment in the buffer is a partially resolved load, the register resolve function returns its value.

We now discuss the execution rules, where partially resolved loads may fully resolve against either the originating store or against memory.

Resolving when originating store is in the reorder buffer.

LOAD-EXECUTE-ADDR-OK

$$\begin{array}{c}
\text{buf}(i) = (r = \text{load}(\vec{rv}, (v_\ell, j)))^n \quad (\text{buf} +_i \rho)(\vec{rv}) = \vec{v}_\ell \\
\llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \quad \ell_a = \sqcup \vec{\ell} \quad \text{buf}(j) = \text{store}(v_\ell, \vec{rv}_j) \wedge (\vec{rv}_j = a' \Rightarrow a' = a) \\
\forall k : (j < k < i) : \text{buf}(k) \neq \text{store}(_, a) \quad \text{buf}' = \text{buf}[i \mapsto (r = v_\ell\{j, a\})^n] \\
\hline
(\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i]{\text{fwd } a_{\ell_a}} (\rho, \mu, n, \text{buf}')
\end{array}$$

LOAD-EXECUTE-ADDR-HAZARD

$$\begin{array}{c}
\text{buf}(i) = (r = \text{load}(\vec{rv}, (v_\ell, j)))^{n'} \\
(\text{buf} +_i \rho)(\vec{rv}) = \vec{v}_\ell \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \quad \ell_a = \sqcup \vec{\ell} \quad (\text{buf}(j) = \text{store}(v_\ell, a') \wedge a' \neq a) \vee \\
(\exists k : j < k < i \wedge \text{buf}(k) = \text{store}(_, a)) \quad \text{buf}' = \text{buf}[j : j < i] \\
\hline
(\rho, \mu, n, \text{buf}) \xrightarrow[\text{execute } i]{\text{rollback, fwd } a_{\ell_a}} (\rho, \mu, n', \text{buf}')
\end{array}$$

To resolve $(r = \text{load}(\vec{rv}, (v_\ell, j)))^n$ when its originating store is still in buf , we calculate the load's actual target address a and compare it against the target address of the originating store at $\text{buf}(j)$. If the store is not followed by later stores to a , and either **(1)** the store's address is resolved and its address is indeed a , or **(2)** the store's address is still unresolved, we update the reorder buffer with an annotated value instruction (rule LOAD-EXECUTE-ADDR-OK).

If, however, either the originating store resolved to a *different* address (mispredicted aliasing) or a later store resolved to the same address (hazard), we roll back our execution to just before the load (rule LOAD-EXECUTE-ADDR-HAZARD).

We allow the load to execute even if the originating store has not yet resolved its address. When the store does finally resolve its address, it must check that the addresses match and that the

forwarding was correct. The gray formulas in STORE-EXECUTE-ADDR-OK and STORE-EXECUTE-ADDR-HAZARD (Section 2.2.4) perform these checks: For forwarding to be correct, all values forwarded from a store at $buf(i)$ must have a matching annotated address ($\forall k > i : j_k = i \Rightarrow a_k = a$). Otherwise, if any value annotation has a mismatched address, then the instruction is rolled back ($j_k = i \wedge a_k \neq a$).

Resolving when originating store is not in the buffer. We must also consider the case where we have delayed resolving the load address to the point where the originating store has already retired, and is no longer available in buf . If this is the case, and no other prior store instructions have a matching address, then we must check the forwarded data against memory.

LOAD-EXECUTE-ADDR-MEM-MATCH

$$\begin{array}{c}
buf(i) = (r = \text{load}(\vec{r}v, v_\ell, j))^n \\
j \notin buf \quad (buf +_i \rho)(\vec{r}v) = \vec{v}_\ell \quad \ell_a = \sqcup \vec{\ell} \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \\
\forall k < i : buf(k) \neq \text{store}(_, a) \quad \mu(a) = v_\ell \quad buf' = buf[i \mapsto (r = v_\ell\{\perp, a\})^n] \\
\hline
(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{read } a_{\ell_a}} (\rho, \mu, n, buf')
\end{array}$$

LOAD-EXECUTE-ADDR-MEM-HAZARD

$$\begin{array}{c}
buf(i) = (r = \text{load}(\vec{r}v, v_\ell, j))^{n'} \\
j \notin buf \quad (buf +_i \rho)(\vec{r}v) = \vec{v}_\ell \quad \ell_a = \sqcup \vec{\ell} \quad \llbracket \text{addr}(\vec{v}_\ell) \rrbracket = a \\
\forall k < i : buf(k) \neq \text{store}(_, a) \quad \mu(a) = v'_{\ell'} \quad v'_{\ell'} \neq v_\ell \quad buf' = buf[j : j < i] \\
\hline
(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{rollback, read } a_{\ell_a}} (\rho, \mu, n', buf')
\end{array}$$

If the originating store has retired, and no intervening stores match the same address, we must load the value from memory to ensure we were originally forwarded the correct value. If the value loaded from memory matches the value we were forwarded, we update the reorder buffer with a resolved load annotated as if it had been loaded from memory (rule LOAD-EXECUTE-ADDR-MEM-MATCH).

If a store *different* from the originating store overwrote the originally forwarded value, the value loaded from memory may not match the value we were originally forwarded. In this case we roll back execution to just before the load (rule LOAD-EXECUTE-ADDR-MEM-HAZARD).

We demonstrate these semantics in the attack shown in Figure 2.2. An earlier draft of this work [37] incorrectly claimed to have a proof-of-concept exploit for this attack on real hardware.

2.2.6 Speculation barriers

We extend our semantics with a *speculation barrier* instruction, fence n , that prevents further speculative execution until all prior instructions have been retired.

$$\begin{array}{c}
 \text{FENCE-RETIRE} \\
 \text{MIN}(buf) = i \quad buf(i) = \text{fence} \quad buf' = buf \setminus buf(i) \\
 \hline
 (\rho, \mu, n, buf) \xrightarrow[\text{retire}]{} (\rho, \mu, n, buf')
 \end{array}$$

The fence instruction uses SIMPLE-FETCH as its fetch rule, and its rule for retire only removes the instruction from the buffer. It does not have an execute rule. However, fence instructions affect the execution of all instructions in the reorder buffer that come *after* them. In prior sections, execute rules have the highlighted condition $\forall j < i : buf(j) \neq \text{fence}$. This condition ensures that as long as a fence instruction remains in buf , any instructions fetched after the fence cannot be executed.

We use fence instructions to restrict out-of-order execution in our semantics. Notably, we can use it to prevent attacks of the forms shown in Figures 2.1, 2.5 and 2.6.

Example. The example in Figure 2.7 shows how placing a fence instruction just after the br instruction prevents the Spectre v1 attack from Figure 2.1. The fence in this example prevents the load instructions at $\bar{2}$ and $\bar{3}$ from executing and forces the br to be resolved first. Evaluating the br exposes the misprediction and causes the two loads (as well as the fence) to be rolled back.

Before executing $\bar{1}$		After	
i	$buf[i]$	i	$buf[i]$
$\bar{1}$	$br(>, (4, r_a), \underline{2}, (2, \underline{5}))$	$\bar{1}$	$jump \underline{5}$
$\bar{2}$	$fence$		
$\bar{3}$	$(r_b = load([40, r_a]))$		
$\bar{4}$	$(r_c = load([44, r_b]))$		

Figure 2.7: Example demonstrating fencing mitigation against Spectre v1 attacks. The fence instruction prevents the load instructions from executing before the br.

2.2.7 Indirect jumps

We introduce a new form of control flow to our semantics, *indirect jumps*, which allow the program to dynamically jump to arbitrary locations. The physical instruction for an indirect jump is $jmp_i(\vec{n})$, where \vec{n} is a list of operands used to calculate the jump target. The semantics for jmp_i are given below:

JMPI-FETCH

$$\frac{\mu(n) = jmp_i(\vec{n}) \quad i = MAX(buf) + 1 \quad buf' = buf[i \mapsto jmp_i(\vec{n}, n')]}{(\rho, \mu, n, buf) \xrightarrow[\text{fetch: } n']{} (\rho, \mu, n', buf')}$$

JMPI-EXECUTE-CORRECT

$$\frac{\begin{array}{l} buf(i) = jmp_i(\vec{n}, n_0) \quad \forall j < i : buf(j) \neq fence \\ (buf +_i \rho)(\vec{n}) = \vec{v}_\ell \quad \ell = \sqcup \vec{\ell} \quad \llbracket addr(\vec{v}_\ell) \rrbracket = n_0 \quad buf' = buf[i \mapsto jump n_0] \end{array}}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{jump } n_{0\ell}} (\rho, \mu, n, buf')}$$

JMPI-EXECUTE-INCORRECT

$$\frac{\begin{array}{l} buf(i) = jmp_i(\vec{n}, n_0) \quad \forall j < i : buf[j] \neq fence \quad (buf +_i \rho)(\vec{n}) = \vec{v}_\ell \\ \ell = \sqcup \vec{\ell} \quad \llbracket addr(\vec{v}_\ell) \rrbracket = n' \neq n_0 \quad buf' = buf[j : j < i][i \mapsto jump n'] \end{array}}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{rollback, jump } n'_\ell} (\rho, \mu, n', buf')}$$

When fetching a jmp_i instruction, the schedule guesses the jump target n' . The rule records the operands and the guessed program point in a new buffer entry. In a real processor, the jump

Registers		Program	
r	$\rho(r)$	n	$\mu(n)$
r_a	1_{pub}	$\underline{1}$	$(r_c = \text{load}([48, r_a], \underline{2}))$
r_b	8_{pub}	$\underline{2}$	fence $\underline{3}$
	Memory	$\underline{3}$	jmp _i ([12, r_b])
a	$\mu(a)$...
44..47	array B_{pub}	$\underline{16}$	fence $\underline{17}$
48..4B	array Key_{sec}	$\underline{17}$	$(r_d = \text{load}([44, r_c], \underline{18}))$

Directive	Effect on buf	Leakage
fetch	$\bar{1} \mapsto r_c = \text{load}(48 + r_a)$	
fetch	$\bar{2} \mapsto \text{fence}$	
execute $\bar{1}$	$\bar{1} \mapsto r_c = Key[1]_{\text{sec}}$	read 49_{pub}
fetch: $\underline{17}$	$\bar{3} \mapsto \text{jmp}_i([12, r_b], \underline{17})$	
fetch	$\bar{4} \mapsto r_d = \text{load}([44, r_c])$	
retire	$\bar{1} \notin buf$	
retire	$\bar{2} \notin buf$	
execute $\bar{4}$	$\bar{4} \mapsto r_d = X$	read a_{sec}
	where $a = Key[1]_{\text{sec}} + 40$	

Figure 2.8: Example demonstrating a Spectre v2 attack from a mistrained indirect branch predictor. Speculation barriers are not a useful defense against this style of attack.

target guess is supplied by an indirect branch predictor; as branch predictors can be arbitrarily influenced by an adversary [54], we model the guess as an attacker directive.

In the execute stage, we calculate the actual jump target and compare it to the guess. If the actual target and the guess match, we update the entry in the reorder buffer to the resolved jump instruction $\text{jump } n_0$. If actual target and the guess do not match, we roll back the execution by removing all buffer entries larger or equal to i , update the buffer with the resolved jump to the correct address, and set the next instruction.

Like conditional branch instructions, indirect jumps leak the calculated jump target.

Examples. The example in Figure 2.8 shows how a mistrained indirect branch predictor can lead to disclosure vulnerabilities. After loading a secret value into r_c at program point $\underline{1}$, the program makes an indirect jump. An adversary can mistrain the predictor to send execution to $\underline{17}$ instead of the intended branch target, where the secret value in r_c is immediately leaked. Because indirect jumps can have arbitrary branch target locations, fence instructions do not prevent these

kinds of attacks; an adversary can simply retarget the indirect jump to the instruction after the fence, as is seen in this example.

2.2.8 Function calls

Finally, we present how our semantics models function calls. The physical instructions are $\text{call}(n_f, n_{ret})$, where n_f is the target program point of the call and n_{ret} is the return program point; and the return instruction ret . We “decode” calls and returns into multiple instructions, leaving their respective transient forms simply as markers call and ret .

Call stack. To track control flow in the presence of function calls, our semantics explicitly maintains a call stack in memory. For this, we use a dedicated register r_{sp} which points to the top of the call stack, and which we call the *stack pointer register*. On fetching a call instruction, we update r_{sp} to point to the address of the next element of the stack using an abstract operation succ . It then saves the return address to the newly computed address. On returning from a function call, our semantics transfers control to the return address at r_{sp} , and then updates r_{sp} to point to the address of the previous element using a function pred . This step makes use of a temporary register r_{imp} .

We use abstract operations succ and pred to manipulate r_{sp} . On a 32-bit x86 processor with a downward-growing stack, $\text{op}(\text{succ}, r_{sp})$ would be implemented as $r_{sp} - 4$, while $\text{op}(\text{pred}, r_{sp})$ would be implemented as $r_{sp} + 4$; on an upward growing system, the reverse would be true. Note that the stack register r_{sp} is not protected from illegal access and can be updated freely.

Return stack buffer. For performance, modern processors speculatively predict return addresses. To model this, we extend configurations with a new piece of state called the *return stack buffer* (RSB), written as σ . The return stack buffer contains the expected return address at any execution point. Its implementation is simple: for a call instruction, the semantics pushes the return address to the RSB, while for a ret instruction, the semantics pops the address at the top of

	Program	n	$\underline{1}$	$\underline{2}$	$\underline{3}$
		$\mu(n)$	$\text{call}(\underline{3}, \underline{2})$	ret	ret
Directive	n	buf	σ		
fetch	$\underline{1} \rightarrow \underline{3}$	$\bar{1} \mapsto \text{call}$	$\bar{1} \mapsto \text{push } \underline{2}$		
		$\bar{2} \mapsto r_{sp} = \text{op}(\text{succ}, r_{sp})$			
		$\bar{3} \mapsto \text{store}(2, [r_{sp}])$			
fetch	$\underline{3} \rightarrow \underline{2}$	$\bar{4} \mapsto \text{ret}$	$\bar{4} \mapsto \text{pop}$		
		$\bar{5} \mapsto r_{tmp} = \text{load}([r_{sp}])$			
		$\bar{6} \mapsto r_{sp} = \text{op}(\text{pred}, r_{sp})$			
		$\bar{7} \mapsto \text{jmpi}([r_{tmp}], \underline{2})$			
fetch: \underline{n}	$\underline{2} \rightarrow \underline{n}$	$\bar{8} \mapsto \text{ret}$	$\bar{8} \mapsto \text{pop}$		
		$\bar{9} \mapsto r_{tmp} = \text{load}([r_{sp}])$			
		$\bar{10} \mapsto r_{sp} = \text{op}(\text{pred}, r_{sp})$			
		$\bar{11} \mapsto \text{jmpi}([r_{tmp}], \underline{n})$			

Figure 2.9: Example demonstrating a ret2spec-style attack [97]. The attacker is able to send (speculative) execution to an arbitrary program point, shown in bold.

the RSB. Similar to the reorder buffer, we address the RSB through indices and roll it back on misspeculation or memory hazards.

We model return prediction directly through the return stack buffer rather than relying on attacker directives, as most processors follow this simple strategy, and the predictions therefore cannot be (directly) controlled by an attacker.

Calling.

CALL-DIRECT-FETCH

$$\mu(n) = \text{call}(n_f, n_{ret})$$

$$i = \text{MAX}(buf) + 1 \quad buf_1 = buf[i \mapsto \text{call}][i + 1 \mapsto (r_{sp} = \text{op}(\text{succ}, r_{sp}))]$$

$$buf' = buf_1[i + 2 \mapsto \text{store}(n_{ret}, [r_{sp}])] \quad \sigma' = \sigma[i \mapsto \text{push } n_{ret}] \quad n' = n_f$$

$$\frac{}{(\rho, \mu, n, buf, \sigma) \xrightarrow{\text{fetch}} (\rho, \mu, n', buf', \sigma')}$$

CALL-RETIRE

$$\begin{array}{c}
\text{MIN}(buf) = i \quad buf(i) = \text{call} \quad buf(i+1) = (r_{sp} = v_\ell) \quad buf(i+2) = \text{store}(n_{ret}, a_{\ell_a}) \\
\rho' = \rho[r_{sp} \mapsto v_\ell] \quad \mu' = \mu[a \mapsto n_{ret}] \quad buf' = buf[j : j > i+2] \\
\hline
(\rho, \mu, n, buf, \sigma) \xrightleftharpoons[\text{retire}]{\text{write } a_{\ell_a}} (\rho', \mu', n, buf', \sigma)
\end{array}$$

On fetching a call instruction, we add three transient instructions to the reorder buffer to model pushing the return address to the in-memory stack. The first transient instruction, call, simply serves as an indication that the following two instructions come from fetching a call instruction. The remaining two instructions advance r_{sp} to point to a new stack entry, then store the return address n_{ret} in the new entry. Neither of these transient instructions are fully resolved—they will need to be executed in later steps. We next add a new entry to the RSB, signifying a push of the return address n_{ret} to the RSB. Finally, we set our program point to the target of the call n_f .

When retiring a call, all three instructions generated during the fetch are retired together. The register file is updated with the new value of r_{sp} , and the return address is written to physical memory, producing the corresponding leakage.

The semantics for direct calls can be extended to cover indirect calls in a straightforward manner by imitating the semantics for indirect jumps. We omit them for brevity.

Evaluating the RSB. We define a function $top(\sigma)$ that retrieves the value at the top of the RSB stack. For this, we let $\llbracket \sigma \rrbracket$ be a function that transforms the RSB stack σ into a stack in the form of a partial map $(st : \mathcal{N} \rightarrow \mathcal{V})$ from the natural numbers to program points, as follows: the function $\llbracket \cdot \rrbracket$ applies the commands for each value in the domain of σ , in the order of the indices. For a *push* n it adds n to the lowest empty index of st . For *pop*, it and removes the value with the highest index in st , if it exists. We then define $top(\sigma)$ as $st(\text{MAX}(st))$, where $st = \llbracket \sigma \rrbracket$, and \perp , if the domain of st is empty. For example, if σ is given as $\emptyset[1 \mapsto \text{push } 4][2 \mapsto \text{push } 5][3 \mapsto \text{pop}]$, then $\llbracket \sigma \rrbracket = \emptyset[1 \mapsto 4]$, and $top(\sigma) = 4$.

Returning.

RET-FETCH-RSB

$$\begin{array}{l}
\mu(n) = \text{ret} \quad \text{top}(\sigma) = n' \quad i = \text{MAX}(\text{buf}) + 1 \quad \text{buf}_1 = \text{buf}[i \mapsto \text{ret}] \\
\text{buf}_2 = \text{buf}_1[i + 1 \mapsto (r_{\text{imp}} = \text{load}([r_{\text{sp}}]))] \quad \text{buf}_3 = \text{buf}_2[i + 2 \mapsto (r_{\text{sp}} = \text{op}(\text{pred}, r_{\text{sp}}))] \\
\text{buf}_4 = \text{buf}_3[i + 3 \mapsto \text{jmpi}([r_{\text{imp}}], n')] \quad \sigma' = \sigma[i \mapsto \text{pop}] \\
\hline
(\rho, \mu, n, \text{buf}, \sigma) \xrightarrow[\text{fetch}]{} (\rho, \mu, n', \text{buf}_4, \sigma')
\end{array}$$

RET-FETCH-RSB-EMPTY

$$\begin{array}{l}
\mu(n) = \text{ret} \quad \text{top}(\sigma) = \perp \quad i = \text{MAX}(\text{buf}) + 1 \quad \text{buf}_1 = \text{buf}[i \mapsto \text{ret}] \\
\text{buf}_2 = \text{buf}_1[i + 1 \mapsto (r_{\text{imp}} = \text{load}([r_{\text{sp}}]))] \quad \text{buf}_3 = \text{buf}_2[i + 2 \mapsto (r_{\text{sp}} = \text{op}(\text{pred}, r_{\text{sp}}))] \\
\text{buf}_4 = \text{buf}_3[i + 3 \mapsto \text{jmpi}([r_{\text{imp}}], n')] \quad \sigma' = \sigma[i \mapsto \text{pop}] \\
\hline
(\rho, \mu, n, \text{buf}, \sigma) \xrightarrow[\text{fetch: } n']{} (\rho, \mu, n', \text{buf}_4, \sigma')
\end{array}$$

RET-RETIRE

$$\begin{array}{l}
\text{MIN}(\text{buf}) = i \quad \text{buf}(i) = \text{ret} \quad \text{buf}(i + 1) = (r_{\text{imp}} = v_{1\ell_1}) \quad \text{buf}(i + 2) = (r_{\text{sp}} = v_{2\ell_2}) \\
\text{buf}(i + 3) = \text{jump } n' \quad \rho' = \rho[r_{\text{sp}} \mapsto v_{2\ell_2}] \quad \text{buf}' = \text{buf}[j : j > i + 3] \\
\hline
(\rho, \mu, n, \text{buf}, \sigma) \xrightarrow[\text{retire}]{} (\rho', \mu, n, \text{buf}', \sigma)
\end{array}$$

On a fetch of `ret`, the next program point is set to the predicted return address, i.e., the top value of the RSB, $\text{top}(\sigma)$. Just as with `call`, we add the transient `ret` instruction to the reorder buffer, followed by the following (unresolved) instructions: we load the value at address r_{sp} into a temporary register r_{imp} , we “pop” r_{sp} to point back to the previous stack entry, and then add an indirect jump to the program point given by r_{imp} . Finally, we add a `pop` entry to the RSB. As with `call` instructions, the set of instructions generated by a `ret` fetch are retired all at once.

When the RSB is empty, the attacker can supply a speculative return address via the directive `fetch: n'` . This is consistent with the behavior of existing processors. In practice, there are several variants on what processors actually do when the RSB is empty [97]:

ret instruction is fetched, since $top(\sigma) = \perp$, the program point n is no longer set by the RSB, and is instead set by the (attacker-controlled) schedule.

Retpoline mitigation. A mitigation for Spectre v2 attacks is to replace indirect jumps with *retpolines* [148]. Figure 2.10 shows a retpoline construction that would replace the indirect jump in Figure 2.8. The call sends execution to program point $\underline{5}$, while adding $\underline{4}$ to the RSB. The next two instructions at $\underline{5}$ and $\underline{6}$ calculate the same target as the indirect jump in Figure 2.8 and overwrite the return address in memory with this jump target. When executed speculatively, the ret at $\underline{7}$ will pop the top value off the RSB, $\underline{4}$, and jump there, landing on a fence instruction that loops back on itself. Thus speculative execution cannot proceed beyond this point. When the transient instructions in the ret sequence finally execute, the indirect jump target $\underline{20}$ is loaded from memory, causing a roll back. However, execution is then directed to the proper jump target. Notably, at no point is an attacker able to hijack the jump target via misprediction.

2.3 Detecting violations

We develop a tool Pitchfork based on our semantics to check for SCT violations. Pitchfork only exercises a subset of our semantics; it only detects SCT violations stemming from branch misprediction or basic store-forwarding errors (Sections 2.2.3 and 2.2.4). Regardless, Pitchfork still soundly exposes Spectre-PHT and Spectre-STL vulnerabilities.

Pitchfork constructs worst-case schedules to maximize speculation, parametrized by a *speculation bound* which limits the depth of speculation. When encountering conditional branches, Pitchfork examines both possible path outcomes as if they were (mis)predicted), delaying the execution of the branch condition itself as late as possible. To account for load-store forwarding hazards, Pitchfork similarly examines all possible forwarding outcomes for each load instruction. All other instructions are executed eagerly and in order. We formalize the soundness of Pitchfork’s schedule construction in more detail in Appendix B.3.



We implement Pitchfork on top of the `angr` binary-analysis tool [138]. 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, though we can increase this bound to 250 instructions when we disable checks 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, as well as SCT violations in real-world crypto code.






2.3.1 Evaluation procedure

To evaluate Pitchfork on real-world crypto implementations, we use the same case studies as FaCT [40], a domain-specific language and compiler for constant-time crypto code. We use FaCT's case studies for two reasons: these implementations have been verified to be (sequentially) constant-time, and their inputs have already been annotated by the FaCT authors with secrecy labels.¹

We analyzed both the FaCT-generated binaries and the corresponding C binaries for the case studies. For each binary, we ran Pitchfork without forwarding hazard detection—only looking for Spectre v1 and v1.1 violations—and with a speculation bound of 250 instructions. If Pitchfork did not flag any violations, we re-enabled forwarding hazard detection—looking for Spectre v4 violations—and ran Pitchfork with a reduced bound of 20 instructions. The reduced bound ensured that the analysis was tractable.

¹<https://github.com/PLSysSec/fact-eval>

Table 2.3: SCT violations found by Pitchfork. A  indicates Pitchfork found an SCT violation. A  indicates the violation was found only with forwarding hazard detection.

Case Study	C	FaCT
curve25519-donna	✓	✓
libsodium secretbox		✓
OpenSSL ssl3 record validate		
OpenSSL MEE-CBC		

2.3.2 Detected violations

Table 2.3 shows our results. Pitchfork did not flag any SCT violations in the curve25519-donna implementations; this is not surprising, as the curve25519-donna library is a straightforward implementation of crypto primitives. Pitchfork did, however, find SCT violations (without forwarding hazard detection) in both the libsodium and OpenSSL codebases. Specifically, Pitchfork found violations in the C implementations of these libraries, in code ancillary to the core crypto routines. This aligns with our intuition that crypto primitives will not themselves be vulnerable to Spectre attacks, but higher-level code that interfaces with these primitives may still leak secrets. Such higher-level code is not present in the corresponding FaCT implementations, and Pitchfork did not find any violations in the FaCT code with these settings. However, with forwarding hazard detection, Pitchfork was able to find vulnerabilities even in the FaCT versions of the OpenSSL implementations. We describe two of the violations Pitchfork flagged next.

C libsodium secretbox. The libsodium codebase compiles with stack protection [58] turned on by default. This means that, for certain functions (e.g., functions with stack allocated `char` buffers), the compiler inserts code in the function epilogue to check if the stack was “smashed”. If so, the program displays an error message and aborts. As part of printing the error message, the program calls a function `__libc_message`, which does `printf`-style string formatting.

²Code snippet taken from https://github.com/lattera/glibc/blob/895ef79e04a953cac1493863bcae29ad85657ee1/sysdeps/posix/libc_fatal.c

```

1  for (int cnt = nlist - 1; cnt >= 0; --cnt) {
2      iov[cnt].iov_base = (char *) list->str;
3      // ...
4      list = list->next;
5  }

```

Figure 2.11: Vulnerable snippet from `__libc_message()`.²

```

1  aesni_cbc_encrypt(/* ... */);
2  // (len _out) is in %r14
3  secret mut uint32 pad = _out[len _out - 1];
4  public uint32 maxpad = tmppad > 255 ? 255 : tmppad;
5  if (pad > maxpad) {
6      pad = maxpad;
7      ret = 0; // overwrites %r14
8  }
9  // ...
10 _shal_update(/* ... */); // can return to line 3

```

Figure 2.12: Vulnerable snippet from the FaCT OpenSSL MEE implementation.³

Figure 2.11 shows a snippet from this function which traverses a linked list. When running the C `secretbox` implementation speculatively, the processor may misspeculate on the stack tampering check and jump into the error handling code, eventually calling `__libc_message`. Again due to misspeculation, the processor may incorrectly proceed through the loop extra times, traversing non-existent links, eventually causing secret data to be stored into `list` instead of a valid address (line 4). On the following iteration of the loop, dereferencing `list` (line 2) causes a secret-dependent memory access.

FaCT OpenSSL MEE. In Figure 2.12, we show the code from the FaCT port of OpenSSL’s authenticated encryption implementation. The FaCT compiler transforms the branch at lines 5-7 into straight-line constant-time code, since the variable `pad` is considered `secret`.

Initially, register `%r14` holds the length of the array `_out`. The processor leaks this value due to the array access on line 3; this is not a security violation, as the length is public. On line 7,

³Code snippet taken from https://github.com/PLSysSec/fact-eval/blob/888bc6c6898a06cef54170ea273de91868ea621e/openssl-mee/20170717_latest.fact

the value of `%r14` is overwritten with 0 if `pad > maxpad`, or 1 (the initial value of `ret`) otherwise. Afterwards, the processor calls `_sha1_update`.

To return from `_sha1_update`, the processor must first load the return address from memory. When forwarding hazard detection is enabled, Pitchfork allows this load to speculatively receive data from stores *older* than the most recent store to that address (see Section 2.2.4). Specifically, it may receive the prior value that was stored at that location: the return address for the call to `aesni_cbc_encrypt`.

After the speculative return, the processor executes line 3 a second time. This time, `%r14` does not hold the public value `len _out`; it instead holds the value of `ret`, which was derived from the secret condition `pad > maxpad`. The processor thus accesses either `_out[0]` or `_out[-1]`, leaking information about the secret value of `pad` via cache state.

2.4 Related work

Prior work on modeling speculative or out-of-order execution is concerned with correctness rather than security [4, 89]. We instead focus on security and model side-channel leakage explicitly. Moreover, we abstract away the specifics of microarchitectural features, considering them to be adversarially controlled.

Disselkoen et al. [51] explore speculation and out-of-order effects through a relaxed memory model. Their semantics sits at a higher level, and is orthogonal to our approach. They do not define a semantic notion of security that prevents Spectre-like attacks, and do not provide support for verification.

Mcilroy et al. [100] reason about micro-architectural attacks using a multi-stage pipeline semantics (though they do not define a formal security property). Their semantics models branch predictor and cache state explicitly. However, they do not model the effects of speculative barriers,

nor other microarchitecture features such as store-forwarding. Thus, their semantics can only capture Spectre v1 attacks.

Both Guarnieri et al. [64] and Cheang et al. [41] define speculative semantics that are supported by tools. Their semantics handle speculation through branch prediction—where the predictor is left abstract—but do not capture more general out-of-order execution nor other types of speculation. These works also propose new semantic notions of security (different from SCT); both essentially require that the speculative execution of a program not leak more than its sequential execution. If a program is sequentially constant-time, this additional security property is equivalent to our notion of speculative constant-time. Though our property is stronger, it is also simpler to verify: we can directly check SCT without first checking if a program is sequentially constant-time. And since we focus on cryptographic code, we directly require the stronger SCT property.

Balliu et al. [63] define a semantics in a style similar to ours. Their semantics captures various Spectre attacks, including an attack similar to our alias prediction example (Figure 2.2), and a new attack based on their memory ordering semantics, which our semantics cannot capture.

Finally, several tools detect Spectre vulnerabilities, but do not present semantics. The oo7 static analysis tool [155], for example, uses taint tracking to find Spectre attacks and automatically insert mitigations for several variants. Wu and Wang [161], on the other hand, perform cache analysis of LLVM programs under speculative execution, capturing Spectre v1 attacks.

2.5 Conclusion

We introduced a semantics for reasoning about side-channels under adversarially controlled out-of-order and speculative execution. Our semantics capture existing transient execution attacks—namely Spectre—but can be extended to future hardware predictors and potential attacks. We also defined a new notion of constant-time code under speculation—speculative constant-time

(SCT)—and implemented a prototype tool to check if code is SCT. Our prototype, Pitchfork, discovered new vulnerabilities in real-world crypto libraries.

There are several directions for future work. Our immediate plan is to use our semantics to prove the effectiveness of existing countermeasures (e.g., retpolines) and to justify new countermeasures.

Acknowledgements

We thank the anonymous PLDI and PLDI AEC reviewers and our shepherd James Bornholt for their suggestions and insightful comments. We thank David Kaplan from AMD for his detailed analysis of our proof-of-concept exploit that we incorrectly thought to be abusing an aliasing predictor. We also thank Natalie Popescu for her aid in editing and formatting the original published paper. This work was supported in part by gifts from Cisco and Fastly, by the NSF under Grant Number CCF-1918573, by ONR Grant N000141512750, 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 appears in 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). Cauligi, Sunjay; Disselkoe, Craig; v. Gleissenthall, Klaus; Tullsen, Dean; Stefan, Deian; Rezk, Tamara; Barthe, Gilles, ACM, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Towards Verified Spectre-Resistant SFI Sandboxing

In which we build to new heights.

Speculative constant-time (SCT) is difficult to achieve without some additional structure. One way we can approach SCT is by placing programs inside a “speculation sandbox”: We prevent them from speculatively accessing any data that they would not otherwise be able to touch. If a given program is already sequentially constant-time—perhaps having been compiled by FaCT—then it will certainly be SCT after being sandboxed.

One popular technique for sandboxing untrusted code is *Software-based Fault Isolation* (SFI) [144]. Web browsers and cloud providers, for example, rely on SFI-based sandboxes to prevent buggy or malicious code from corrupting the memory of the host and other sandboxes [68, 101, 166]. Unfortunately, untrusted code can leverage speculative execution to break out of the sandbox and access trusted memory regions, thus making existing SFI implementations vulnerable to Spectre attacks [77, 86].

Researchers have proposed different approaches to mitigate Spectre attacks in SFI-style sandboxes [78, 111, 137]. However, these are best-effort proposals: They rely on carefully combining several intricate software protections and hardware extensions to prevent unsafe

speculative behaviors. It is unclear whether the combination of these countermeasures work as intended and so, in practice, these approaches may fail to provide the expected security guarantees against Spectre attacks.

In this chapter, we develop principled foundations to build reliable sandboxing mechanisms against Spectre attacks. Towards this goal, we have formulated security properties to formally capture the essence of Spectre SFI attacks, and have already uncovered bugs in the implementation of the *Swivel* SFI system [111]. We investigate Swivel’s security claims and show which Spectre attacks it can soundly mitigate and for which it falls short.

3.1 Formal model

To study SFI in the context of speculative execution attacks, we focus on a simple assembly-style language, $ZFI=\mathcal{L}$. We present the syntax of $ZFI=\mathcal{L}$, then formalize its architectural and speculative semantics.

3.1.1 Syntax

The syntax of $ZFI=\mathcal{L}$ programs is given in Figure 3.1. In $ZFI=\mathcal{L}$, expressions are constructed by combining immediate values v and registers r using basic arithmetic operations \oplus . $ZFI=\mathcal{L}$ supports standard control-flow instructions (direct and indirect jumps, function calls and returns), register assignments ($r := e$), and memory loads ($r' := *(r + e)$) and stores ($*(r + e) := e'$). Memory instructions always access an offset e from a base register r . $ZFI=\mathcal{L}$ also supports dedicated instructions *flush* (e.g., clearing predictor state) and *endbranch* (e.g., control-flow integrity checks), which we use to model countermeasures against Spectre attacks.

3.1.2 Architectural semantics

We first cover the *architectural semantics* of $ZFI \stackrel{=}{=} \mathcal{Q}$, which models the execution of our basic assembly programs *without* any speculative behavior. The semantics is defined in terms of architectural configurations Ψ . Each configuration Ψ is a quadruple consisting of a program P mapping values to instructions, a program counter $pc \in \mathbb{V}$, a register file $Reg : \mathbb{R} \rightarrow \mathbb{V}$ mapping registers to values, and a memory $Mem : \mathbb{V} \rightarrow \mathbb{V}$ that maps memory addresses to values. We use dot-notation to access a context's elements, e.g., $\Psi.Mem$ denotes the memory associated with Ψ . We use bracket-notation to update an element within a context, e.g., $\Psi\{Reg := Reg'\}$ denotes the context obtained by updating the register file to Reg' . Furthermore, $\Psi[s]$ denotes that s is the instruction pointed by the current program counter and Ψ^{++} denotes the context obtained by incrementing Ψ 's program counter by 1.

The architectural semantics is formalized by the \rightarrow relation in Figure 3.2, which describes how architectural contexts are modified during the computation. In the rules, $\llbracket e \rrbracket_{\Psi}$ denotes the value of expression e in the context of Ψ , and r_{Stk} and r_{Heap} represent the unique *stack pointer* and *heap pointer* registers.

3.1.3 Attackers and observations

To represent the power of attackers to observe and exfiltrate secret data, we have our semantics emit *leakage observations* that represent side-channel information an attacker can glean. The observations emitted by different instructions depends on the *leakage model* we wish to consider. We consider the following three leakage models, each giving increasing power to an attacker:

- *dmem*, where attackers can observe the state of the data cache,
- *ct*, where attackers can observe leaks considered by the *constant-time* paradigm [36], and
- *arch*, where attackers can observe all values retrieved from memory [65].

Basic types																							
(Values)	$v \in \mathbb{V}$																						
(Registers)	$r \in \mathbb{R}$																						
(Operators)	$\oplus \in \oplus$																						
Syntax																							
(Expressions)	$e \in v \mid r \mid e \oplus e$																						
(Instructions)	$s \in$ <table style="display: inline-table; vertical-align: middle; border-collapse: collapse;"> <tr> <td style="padding-right: 5px;">$r := e$</td> <td style="padding-left: 10px;">(assignments)</td> </tr> <tr> <td style="padding-right: 5px;">$r := *(r + e)$</td> <td style="padding-left: 10px;">(memory load)</td> </tr> <tr> <td style="padding-right: 5px;">$*(r + e) := e$</td> <td style="padding-left: 10px;">(memory store)</td> </tr> <tr> <td style="padding-right: 5px;">$jmp \pm v$</td> <td style="padding-left: 10px;">(unconditional jump)</td> </tr> <tr> <td style="padding-right: 5px;">$jmp \pm v \text{ if } e$</td> <td style="padding-left: 10px;">(conditional jump)</td> </tr> <tr> <td style="padding-right: 5px;">$jmp r$</td> <td style="padding-left: 10px;">(indirect jump)</td> </tr> <tr> <td style="padding-right: 5px;">$call \pm v$</td> <td style="padding-left: 10px;">(direct call)</td> </tr> <tr> <td style="padding-right: 5px;">$call r$</td> <td style="padding-left: 10px;">(indirect call)</td> </tr> <tr> <td style="padding-right: 5px;">ret</td> <td style="padding-left: 10px;">(return)</td> </tr> <tr> <td style="padding-right: 5px;">$flush$</td> <td style="padding-left: 10px;">(BTB state flush)</td> </tr> <tr> <td style="padding-right: 5px;">$endbranch$</td> <td style="padding-left: 10px;">(CET “endbranch”)</td> </tr> </table>	$r := e$	(assignments)	$ r := *(r + e)$	(memory load)	$ *(r + e) := e$	(memory store)	$ jmp \pm v$	(unconditional jump)	$ jmp \pm v \text{ if } e$	(conditional jump)	$ jmp r$	(indirect jump)	$ call \pm v$	(direct call)	$ call r$	(indirect call)	$ ret$	(return)	$ flush$	(BTB state flush)	$ endbranch$	(CET “endbranch”)
$r := e$	(assignments)																						
$ r := *(r + e)$	(memory load)																						
$ *(r + e) := e$	(memory store)																						
$ jmp \pm v$	(unconditional jump)																						
$ jmp \pm v \text{ if } e$	(conditional jump)																						
$ jmp r$	(indirect jump)																						
$ call \pm v$	(direct call)																						
$ call r$	(indirect call)																						
$ ret$	(return)																						
$ flush$	(BTB state flush)																						
$ endbranch$	(CET “endbranch”)																						

Figure 3.1: Syntax of the $ZFI \stackrel{=}{=} \mathcal{L}$ language.

The *dmem* model is the weakest of the three models, and considers the data cache as the only viable leakage channel. In this model, an attacker can observe cache state—specifically, the data cache—using attacks such as PRIME+PROBE [146], but cannot determine the control flow trace of a program. In the *ct* model, we consider an attacker that can observe the standard *constant-time* leakages [36] via timing or other microarchitectural leaks [62, 105, 164]. The data cache as well as the control flow trace are visible to the attacker in this model. Finally, in the *arch* model, we assume the attacker observes all values loaded from memory. Since the initial memory is the source of all values in the program, this is equivalent to an attacker seeing the full trace of all values during execution [65].

We formalize each leakage model by a function $LEAKS(\Psi)$ that takes as input a configuration $\Psi[insn]$ and outputs a sequence of observations for each *jump*, *load*, or *store* operation that occurs during the semantic execution rule for *insn*; Table 3.1 informally illustrates our leakage models. For example, $LEAKS(\Psi[ret])$ under the *ct* model produces two observations: v_{Stk} , for loading the return address; and $Mem[v_{Stk}]$, for jumping to that location (see RETURN in Figure 3.2).

$$\begin{array}{c}
\text{ASSIGNMENT} \\
\frac{v = \llbracket e \rrbracket_{\Psi}}{\Psi[r := e] \rightarrow \Psi^{++} \{ \text{Reg}[r] := v \}} \\
\\
\text{STORE} \\
\frac{v_{\text{addr}} = \llbracket r_{\text{base}} + e_{\text{off}} \rrbracket_{\Psi} \quad v = \llbracket e \rrbracket_{\Psi}}{\Psi[* (r_{\text{base}} + e_{\text{off}}) := e] \rightarrow \Psi^{++} \{ \text{Mem}[v_{\text{addr}}] := v \}} \\
\\
\text{LOAD} \\
\frac{v_{\text{addr}} = \llbracket r_{\text{base}} + e_{\text{off}} \rrbracket_{\Psi} \quad v = \Psi.\text{Mem}[v_{\text{addr}}]}{\Psi[r := *(r_{\text{base}} + e_{\text{off}})] \rightarrow \Psi^{++} \{ \text{Reg}[r] := v \}} \\
\\
\text{JUMP} \\
\frac{}{\Psi[\text{jmp } +i] \rightarrow \Psi \{ pc := pc + i \}} \\
\\
\text{JUMP-COND-TAKEN} \\
\frac{\llbracket e \rrbracket_{\Psi}}{\Psi[\text{jmp } +i \text{ if } e] \rightarrow \Psi \{ pc := pc + i \}} \\
\\
\text{JUMP-COND-NOT-TAKEN} \\
\frac{\neg \llbracket e \rrbracket_{\Psi}}{\Psi[\text{jmp } +i \text{ if } e] \rightarrow \Psi^{++}} \\
\\
\text{JUMP-INDIRECT} \\
\frac{v_{\text{addr}} = \llbracket r \rrbracket_{\Psi}}{\Psi[\text{jmp } r] \rightarrow \Psi \{ pc := v_{\text{addr}} \}} \\
\\
\text{CALL} \\
\frac{v_{\text{Stk}} = \llbracket r_{\text{Stk}} - 1 \rrbracket_{\Psi}}{\Psi[\text{call } r] \rightarrow \Psi \left\{ \begin{array}{l} \text{Mem}[v_{\text{Stk}}] := pc + 1 \\ \text{Reg}[r_{\text{Stk}}] := v_{\text{Stk}} \\ pc := pc + i \end{array} \right\}} \\
\\
\text{CALL-INDIRECT} \\
\frac{v_{\text{addr}} = \llbracket r \rrbracket_{\Psi} \quad v_{\text{Stk}} = \llbracket r_{\text{Stk}} - 1 \rrbracket_{\Psi}}{\Psi[\text{call } r] \rightarrow \Psi \left\{ \begin{array}{l} \text{Mem}[v_{\text{Stk}}] := pc + 1 \\ \text{Reg}[r_{\text{Stk}}] := v_{\text{Stk}} \\ pc := v_{\text{addr}} \end{array} \right\}} \\
\\
\text{RETURN} \\
\frac{v_{\text{Stk}} = \llbracket r_{\text{Stk}} \rrbracket_{\Psi}}{\Psi[\text{ret}] \rightarrow \Psi \left\{ \begin{array}{l} \text{Reg}[r_{\text{Stk}}] := v_{\text{Stk}} + 1 \\ pc := \text{Mem}[v_{\text{Stk}}] \end{array} \right\}}
\end{array}$$

Figure 3.2: Architectural semantics for ZFI $\stackrel{?}{\approx}$.

Finally, we include a structure *Obs* in our configuration to collect the sequence of leakage observations during execution. We update *Obs* with each architectural step using the relation $\rightarrow_{\text{trace}}$ induced by the following rule:

$$\begin{array}{c}
\text{TRACE} \\
\frac{\Psi \rightarrow \Psi' \quad \text{Obs}' = \Psi.\text{Obs} ++ \text{LEAKS}(\Psi)}{\Psi[\text{insn}] \rightarrow_{\text{trace}} \Psi' \{ \text{Obs}' \}}
\end{array}$$

We refer to this extended relation as \rightarrow for brevity, as it merely adds bookkeeping to the semantics.

Table 3.1: Leakage models.

LEAKS(\cdot)	$dmem$	ct	$arch$
any jump $pc := v$	—	v	v
any load $Mem[v_{addr}] = v$	v_{addr}	v_{addr}	v_{addr}, v
any store $Mem[v_{addr}] := v$	v_{addr}	v_{addr}	v_{addr}

SPEC-PREDICT

$$\frac{\text{ISCONTROLFLOW}(insn) \quad \Psi \rightarrow \Psi' \quad correct = (pc' = \Psi'.pc)}{pc', \mu state' = \text{Oracle}(insn, \Psi.pc, \Psi.Reg, \Psi.\mu state) \quad \Psi[insn] \rightsquigarrow \Psi' \{ pc', \mu state', \text{mispredicted} := \Psi.mispredicted \vee \neg correct \}}$$

SPEC-STEP

$$\frac{\neg \text{ISCONTROLFLOW}(insn) \quad insn \neq flush \quad \Psi \rightarrow \Psi'}{\Psi[insn] \rightsquigarrow \Psi'}$$

Figure 3.3: Speculative semantics for $ZFI \stackrel{?}{=} \perp$.

3.1.4 Speculative semantics

To reason about speculative leaks, we equip $ZFI \stackrel{?}{=} \perp$ with a speculative semantics that captures the effects of speculatively executed instructions.

We model microarchitectural predictors using a *prediction oracle* which abstracts away from the microarchitectural prediction details. The oracle is defined in terms of a set of oracle states $\mu state$ (which contains a designated initial state \perp) and a function

$$\text{Oracle}(insn, pc, Reg, \mu state)$$

that, given the current instruction, the current pc , the register file Reg , and the current oracle state, produces the prediction pc and an updated oracle state $\mu state'$ (which is then used in following predictions).

The speculative semantics is formalized by the relation \rightsquigarrow given in Figure 3.3. In the rules defining \rightsquigarrow configurations, Ψ is extended to store the $\mu state$ of the prediction oracle (which is

updated throughout the computation) as well as a simple flag *mispredicted* that is set as soon as an oracle prediction is incorrect.

Our speculative semantics consists of three rules: The SPEC-PREDICT rule describes the execution of control-flow statements, where the prediction oracle is invoked to obtain the new program counter and predictor state; the correctness of the prediction is recorded in the flag *mispredicted*. The SPEC-FLUSH rule (described in Section 3.2.4) models the execution of *flush* instructions, which reset the predictor state to \perp . Finally, the SPEC-STEP handles the remaining statements by updating the configuration according to the architectural semantics \rightarrow .

Unlike prior semantics [36, 64], our language does not have any form of speculative rollback. Instead, we track the speculative state through the *mispredicted* flag, which persists in the configuration for the duration of the program.

3.2 Formalizing SFI security

Building atop the semantics for $ZFI \stackrel{?}{=} \perp$, we define what it means for a program to be *speculatively secure*. We examine the security properties that Swivel claims to provide, formalizing them in terms of our formal security property and investigate whether Swivel can soundly uphold these properties.

3.2.1 Non-interference

We define the security of $ZFI \stackrel{?}{=} \perp$ programs as a form of *non-interference property*. A program is *non-interferent* if an attacker cannot distinguish between two executions that differ only in their secret values. Formally, we define an *equivalence relation* on configurations: Two configurations are equivalent if and only if they differ only in their secret values. Then, if two equivalent configurations produce identical leakage observations, they are indistinguishable to an attacker.

Definition 3.2.1 (Speculative leakage security). *A program P is speculatively secure (up to n steps) with respect to an equivalence \approx and a given leakage model m if:*

$$\begin{aligned} \Psi_1 \approx \Psi_2 \text{ and } \Psi_1 \rightsquigarrow^n \Psi'_1 \\ \text{and } \Psi_2 \rightsquigarrow^n \Psi'_2 \\ \implies \Psi'_1.Obs = \Psi'_2.Obs. \end{aligned}$$

When dealing with speculative execution, we can define what is secret (and thus our equivalence relation) in two different ways: If we already have an idea of which values in memory should not be leaked to an attacker, we can define an explicit *secrecy policy* that states which addresses are public and secret. Alternatively, we can define secrets to be any values that are not *already* observable by an attacker during architectural execution; that is, that speculative execution leaks *no additional* information to an attacker.

Definition 3.2.2 (Policy equivalence, \approx_π). *Ψ_1 and Ψ_2 are equivalent with respect to a secrecy policy π iff:*

$$\forall v_{addr} \in \pi : \Psi_1.Mem[v_{addr}] = \Psi_2.Mem[v_{addr}]$$

and all other structures in Ψ_1 and Ψ_2 are syntactically equal. We write this as $\Psi_1 \approx_\pi \Psi_2$.

Definition 3.2.3 (Architectural equivalence, \approx_m). *Ψ_1 and Ψ_2 are architecturally equivalent (up to n steps) with respect to a leakage model m if:*

$$\begin{aligned} \Psi_1 \rightarrow^n \Psi_1^* \text{ and } \Psi_2 \rightarrow^n \Psi_2^* \\ \implies \Psi_1^*.Obs = \Psi_2^*.Obs. \end{aligned}$$

and all structures other than Mem in Ψ_1 and Ψ_2 are syntactically equal. We write this as $\Psi_1 \approx_m^n \Psi_2$.

3.2.2 SFI security properties

Swivel enforces two distinct notions of security. First, the host application does not trust the individual sandboxes: Swivel must prevent *breakout attacks*, where a sandbox accesses data outside of its defined memory regions. Second, Swivel’s sandboxes are *mutually distrusting*: Swivel must prevent *poisoning attacks*, where an attacker is able to leak secrets from a victim sandbox. We can formalize both of these properties in terms of speculative leakage security.

The first property we formalize captures *sandbox breakout* attacks. A sandbox breakout occurs when a malicious sandbox is able to directly access the contents of memory outside of its own memory segments, e.g., from the host application or from another sandbox. As an example, the following program has a possible breakout attack:

```

jmp +5 if  $e_{check}$       ; if  $e_{check}$ 
 $* (r_{Stk} + 4) := r_{Heap}$   ; spill  $r_{Heap}$  to the stack
 $r_{Heap} := r_A$           ; and replace its contents with  $r_A$ 
jmp +2 if  $\neg e_{check}$    ; else
 $r_B := *(r_{Heap} + 24)$  ; load a value from the heap

```

Even though *architecturally* the final load is safe, as the two conditions are mutually exclusive, *speculatively* we might (mis)predict and enter both conditional blocks anyway. An attacker can exploit this if it can control r_A , as under these conditions the value in r_A is incorrectly used as the heap base address.

Formally, to prevent breakout attacks, we want non-interference under the *arch* leakage model. We use equivalence with respect to a policy π that only defines the sandbox’s own memory segments to be public. By using the *arch* model, we consider even *accessing* a secret value to be a successful attack; since our policy π only considers the sandbox memory itself to be public, our property fully captures sandbox breakout attacks.

The second property we formalize captures what we term *poisoning attacks*. Even if a sandbox protects its own secrets from leaking architecturally, it may be speculatively *poisoned*

and still leak these secrets on mispredicted execution paths. We present the following simple example, where X and Y are arrays of length 64 in the sandbox’s heap and r_A is an index into X .

```

jmp +3 if  $r_A \geq 64$            ; check bound for heap array  $X$ 
 $r_B := *(r_{Heap} + X + r_A)$       ; out of bounds if mispredicted
 $r_C := *(r_{Heap} + Y + r_B)$       ; leak  $r_B$  via memory address

```

Under architectural execution, any value within X may be leaked due to the final memory access, but values outside of X are not leaked due to the initial conditional check. However, during speculative execution, we may incorrectly predict that the branch should fall through even when r_A is out-of-bounds for X . If an attacker is able to control the value of r_A , it can then leak any value in the victim sandbox’s heap.

Since we do not know which memory locations a sandbox developer considers secret within their sandbox, we assume that sandboxed programs are architecturally constant-time, and impose non-interference using architectural equivalence under the ct leakage model. This way we can be certain that the sandbox, at the very least, leaks no more information than its architectural execution would.

Swivel offers two different implementations to mitigate these attacks: The first approach, Swivel-SFI, is intended for current x86 processors and relies heavily on rewriting control flow constructs. The second approach, Swivel-CET, relies on the Control-flow Enforcement Technology (CET) extensions developed by Intel in their latest hardware [136].

We cover some useful properties common to both implementations, then examine whether each implementation in turn can soundly prevent breakout and poisoning attacks.

3.2.3 Establishing security

Since Swivel only operates on valid WebAssembly programs, we can make certain assumptions about the structure of our initial programs. For example, the stack region (represented in $ZFI \stackrel{=}{\sim} \mathcal{L}$ as $Mem[r_{Stk} + e_{off}]$) is only used for local variables and register spills; all stack loads

and stores use constant (immediate) offsets from the stack pointer (i.e., e_{off} for r_{Stk} is always a simple value).

Furthermore, Swivel modifies the WebAssembly compiler to ensure the security of memory segment registers: The heap pointer (r_{Heap} in ZFI= \mathcal{L}) is never spilled to the stack, and the stack pointer (r_{Stk}) is only modified when establishing function stack frames.

Linear blocks. A fundamental building block of Swivel’s mitigations is *linear blocks*. A linear block is a sequence of instructions ending in *any* control flow instruction. In our execution model, even during speculative execution, we can assume that all instructions within a linear block are executed sequentially in order. Thus linear blocks allow us to establish local invariants: E.g., if a heap offset is truncated to the size of the heap (e.g., via an arithmetic masking operation) at the beginning of a linear block, we can assume it will still be safe to use for the rest of the block.

Chaining linear blocks. If we can show that a program, upon leaving any linear block, will always land on the start of a new linear block, then we can inductively extend certain local block invariants to cover the whole program—in particular, invariants about memory safety. For example, if we show that within any linear block, all heap offsets are masked before they are used, then we can inductively show that all heap offsets in the program are safe.

3.2.4 Swivel-SFI

Swivel-SFI provides security, somewhat counterintuitively, by replacing all non-trivial control flow with indirect jumps. Conditional jumps are emulated by selecting the target block’s address based on the relevant condition; calls and returns are replaced with instructions that save return addresses to a *separate stack*, distinct from the existing stack memory region and with its own dedicated stack pointer register.

By converting all control flow to indirect jumps, Swivel-SFI can protect speculative control flow by flushing the indirect jump predictor (or *BTB* for *Branch Target Buffer* [35]) state at the start of the program:

$$\frac{}{\Psi[\textit{flush}] \rightsquigarrow \Psi^{++} \{\mu\textit{state} := \perp\}}$$

Since the only relevant predictor in Swivel-SFI is the BTB, we treat *flush* as clearing the entire $\mu\textit{state}$ to the empty state \perp . Flushing $\mu\textit{state}$ will not prevent misprediction—it does, however, limit an attacker attempting to mistrain victim predictors. Since BTB predictions have no state to rely on beyond the program itself, any given jump instruction can only be trained to architecturally valid targets for that instruction.

Breakout security. Swivel masks all memory operations within a linear block, we need only show that Swivel-SFI executes programs as a sequence of linear blocks. Since we flush the predictor state at the start of the program, we assume that the BTB can only be trained on valid jump targets; thus $\text{Oracle}(\cdot)$ will only provide values for pc that were *correct* at least once. Since all valid jump targets are linear blocks, we can be sure that predicted jump targets always land on linear blocks.

Poisoning security. Unfortunately, even if we assume that the BTB always predicts valid targets, we cannot prove security from poisoning attacks. As a trivial example, consider the program demonstrating a poisoning attack in Section 3.2.2. Even after it is converted to use an indirect jump to replace the conditional branch, it may still mispredict and execute the vulnerable loads—flushing the BTB does not prevent mispredictions from happening. However, by flushing the BTB, Swivel-SFI claims to prevent an attacker from *actively* mistraining a predictor—i.e., an attacker cannot force the victim sandbox to mispredict [111]. Our framework does not (yet) distinguish active attackers in its security model, and so cannot verify this claim.

3.2.5 Swivel-CET

The Swivel-CET implementation makes use of two features from Intel’s CET hardware extensions: The *endbranch* instruction and the *shadow stack*. We formalize the CET hardware extensions as an augmented step relation \rightsquigarrow_{cet} built on top of our prior speculative relation \rightsquigarrow :

$$\frac{\text{SPEC-CET-STEP} \quad \neg \text{ISCONTROLFLOW}(insn) \quad insn \notin \{call \cdot, ret\} \quad \Psi \rightsquigarrow \Psi'}{\Psi[insn] \rightsquigarrow_{cet} \Psi'}$$

Breakout security. The *endbranch* instruction provides *forward-edge control-flow integrity* (CFI): Every control flow instruction (except *ret*) must land on an *endbranch* instruction, even when executing speculatively. For most control flow instructions, the augmented semantics simply ensures that the following instruction is indeed *endbranch*.

$$\frac{\text{SPEC-CET-ENDBRANCH} \quad \text{ISCONTROLFLOW}(insn) \quad insn \notin \{call \cdot, ret\} \quad \Psi \rightsquigarrow \Psi' \quad \Psi'[endbranch]}{\Psi[insn] \rightsquigarrow_{cet} \Psi'}$$

For call and return instructions, CET provides a *shadow stack*: All calls and returns, in addition to pushing and popping return addresses off the regular stack, also push and pop return addresses on a separate protected memory region. On a *ret*, the processor will only jump to a predicted return location if it agrees with the address popped from the shadow stack [136].

$$\frac{\text{SPEC-CET-CALL} \quad \Psi \rightsquigarrow \Psi' \quad \Psi'[endbranch] \quad v_{SStk} = \llbracket r_{SStk} - 1 \rrbracket_{\Psi}}{\Psi[call \cdot] \rightsquigarrow_{cet} \Psi' \{ \text{Mem}[v_{SStk}] := pc + 1 \ , \ \text{Reg}[r_{SStk}] := v_{SStk} \}}$$

SPEC-CET-RETURN

$$\frac{\Psi \rightsquigarrow \Psi' \quad v_{SStk} = \llbracket r_{SStk} \rrbracket_{\Psi} \quad \Psi'.pc = \Psi.Mem[v_{SStk}]}{\Psi[ret] \rightsquigarrow_{cet} \Psi' \{Reg[r_{SStk}] := v_{SStk} + 1\}}$$

The register r_{SStk} is the protected pointer to the latest shadow stack entry.

As with Swivel-SFI, Swivel-CET masks all memory operations within a linear block. By placing *endbranches* only at the tops of linear blocks and by relying on the CET shadow stack, Swivel-CET ensures that programs always execute as a chain of linear blocks.

Poisoning security. To mitigate poisoning attacks, Swivel-CET constructs a *register interlock* at every linear block transition. The register interlock detects whether speculative control flow has been mispredicted, and if so, clears all the memory base registers (i.e., r_{Heap} and r_{Stk}). By doing so, all memory operations following a misprediction are directed to invalid addresses near the address 0. Memory accesses to this faulting page will not leak the address as they will not create a cache entry—we treat this behavior as a special exception to our established leakage models.

The interlock is implemented as follows: We first give each linear block in the program a unique label. At the end of each block we dynamically calculate the label of the target block without branching. For example, at a conditional branch, we use the same condition expression to select between the two target block labels. When we arrive at the new block, we compare the calculated label to the label of executing block. If the labels do not match, we set r_{Heap} and r_{Stk} to \perp .

With register interlocks in place, we have the following lemma: If $\Psi.mispredicted$ is true, then all following memory operations will fail without leaking (per our earlier assumption about near-zero addresses).

However, while this prevents leaking via memory operations, this does not stop leakages via control flow. For example, if a sandbox secret is already in a register before we mispredict, then a later linear block may branch on this register, leaking the secret value. Thus we can only

prove poisoning security for Swivel-CET with respect to the weaker *dmem* leakage model instead of the stronger *ct* leakage model.

3.3 Conclusion

We present the first formal framework for SFI security in the face of Spectre attacks. Our language, $ZFI \stackrel{=}{=} \mathcal{L}$, is expressive enough to verify the security claims of the Swivel SFI system; by formalizing Swivel’s security properties, we reveal which of its security claims it soundly upholds, as well as the explicit assumptions about hardware execution that Swivel relies on.

Acknowledgements

This work was supported in part by gifts from Cisco and Intel; by the NSF under Grant Numbers CNS-1514435, CCF-1918573, and CAREER CNS-2048262; by the Community of Madrid under the project S2018/TCS-4339 BLOQUES; by the Spanish Ministry of Science, Innovation, and University under the project RTI2018-102043-B-I00 SCUM and the Juan de la Cierva-Formación grant FJC2018-036513-I; by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPANet-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 3, in part, is currently being prepared for submission for publication of the material. Cauligi, Sunjay; Guarnieri, Marco; Mehta, Aastha; Moghimi, Daniel; Narayan, Shravan; Stefan, Deian; Vahldiek-Oberwagner, Anjo; Vassena, Marco. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Practical Foundations for Spectre Defenses

Or, a view from the sky.

As we have seen throughout this dissertation, *program semantics* and *formal security policies* can help us achieve *provable security guarantees*. 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 mitigate and find.

Alas, not all foundations are equally practical. The systems presented here, as well as other similar frameworks in the field, all explore different design choices—many of which have important ramifications on defense tools and the software they produce or analyze. 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 specific *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 [15] and sandbox isolation [65] models are the most pragmatic; leakage models that only consider the data cache trade off security for no clear benefits (e.g., of 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 verification and mitigation of software. Other models, which capture microarchitectural details like cache structures, make the analysis unnecessarily complicated: They do not fundamentally capture additional attacks, and they give up on portability.

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 [34, 35, 162] 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.

Contributions. We systematize knowledge of software Spectre defenses and their associated formalizations, by studying the choices available to developers of Spectre analysis and mitigation tools. Specifically, we:

- ▶ 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 of systematization. In our systematization, we focus on software-only defenses against Spectre attacks. We focus on *Spectre* because most other transient attacks (e.g., Melt-down [93], LVI [150], MDS [71], or Foreshadow [149]) 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. [35], 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 Breaking cryptography with Spectre

High-assurance cryptography has long relied on *constant-time programming* [15] in order to create software which is secure from timing side-channel attacks. Constant-time programming

```

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.

ensures that program execution does not depend on secrets. It does this via three rules of thumb [15, 17]: 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 [29].

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 [124], based on which instructions were (speculatively) accessed; or (4) through port contention [29], branch predictor state [76], or other microarchitectural resources that differ among the branches.

4.1.2 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 [56]. Spectre attacks can break the memory safety and isolation mechanisms commonly used in these settings [78, 98, 111, 137].

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.

4.1.3 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. [65], writing $\llbracket \cdot \rrbracket^{\text{seq}}$ to represent the execution model for standard sequential execution. We notate other

execution models similarly; for example, $\llbracket \cdot \rrbracket^{\text{pht}}$ 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, we 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. [65], which defines the leakage models $\llbracket \cdot \rrbracket_{\text{ct}}$ and $\llbracket \cdot \rrbracket_{\text{arch}}$. The $\llbracket \cdot \rrbracket_{\text{ct}}$ 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 $\llbracket \cdot \rrbracket_{\text{arch}}$ 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) [65]. 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 $\llbracket \cdot \rrbracket_{\text{arch}}$ are a strict superset of those in $\llbracket \cdot \rrbracket_{\text{ct}}$, we say that $\llbracket \cdot \rrbracket_{\text{arch}}$ is *stronger* than $\llbracket \cdot \rrbracket_{\text{ct}}$ (i.e., it models a more powerful attacker). These properties make $\llbracket \cdot \rrbracket_{\text{arch}}$ most useful for software isolation, as any out-of-bounds accesses will immediately show up in an $\llbracket \cdot \rrbracket_{\text{arch}}$ leakage trace.

Surprisingly, the $\llbracket \cdot \rrbracket_{\text{ct}}$ and $\llbracket \cdot \rrbracket_{\text{arch}}$ models both 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 $\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}}$, while our hypothetical Spectre-PHT semantics would provide the contract $\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}}$. Formally, the contract governs the attacker-visible information produced when executing a program: Given

¹Like Guarnieri et al. [65], we omit variable-latency instructions from our formal model for simplicity.

a program p , a semantics with contract $\llbracket \cdot \rrbracket_\ell^\alpha$, and an initial state σ , we write $\llbracket p \rrbracket_\ell^\alpha(\sigma)$ 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_π 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 $\llbracket \cdot \rrbracket_\ell$ and security policies π . Sections 4.2.3 and 4.2.4 discuss tradeoffs for different execution models $\llbracket \cdot \rrbracket^\alpha$ 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 this respect, formalisms for constant-time have been successful in the non-speculative world: 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 4.1.1, high-assurance cryptography implementations have long relied on the constant-time programming model; thus, semantics intended for cryptographic programs naturally choose the $\llbracket \cdot \rrbracket_{ct}$ leakage model. Like the constant-time programming model in the non-speculative world, the $\llbracket \cdot \rrbracket_{ct}$ leakage model is strong and hardware-agnostic, making it a solid foundation for security guarantees. The

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.

¹Extension to other variants is discussed, but not performed. ²Semantics captures indirect jump effects, but cannot mispredict indirect jump targets. ³“Weak” variants of semantics leak loaded values during non-speculative execution. ⁴Detects only “speculative type confusion vulnerabilities”, a specific subset of Spectre-PHT. ⁵Mitigates Spectre-PHT without inserting fences. ⁶Defends by effectively preventing speculation, so leakage model is irrelevant. ⁷Specifically, $\llbracket \cdot \rrbracket_{\text{mem}}$ for loads, but closer to $\llbracket \cdot \rrbracket_{\text{arch}}$ for stores. ⁸Operates on WebAssembly, which does not have fences. However, can insert fences in 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	P – Spectre-PHT
Medium	Structured control flow such as if-then-else	B – Speculation rollbacks	B – Spectre-BTB
High	In the style of weak memory models	M – Adrs. of memory operations	R – Spectre-RSB
—	The work has no associated formal semantics	C – Cache lines / cache state	S – Spectre-STL
		T – Step counter / timer	
Fence	Does it reason about speculation fences?	Tool – Does the paper include a tool?	
✓	Fully reasons about fences in the target/input code	Det	Tool detects insecure programs or verifies secure programs
∩	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)	Mit	Tool modifies programs to ensure they are secure
×	Does not reason about, or insert, fences	Val	Tool is only used to validate the semantics, does not automatically perform any security analysis
		*	Tool’s connection to the semantics is incomplete/unclear (e.g., tool does not implement the full semantics)
		—	Does not include a tool
Hijack	Can it model or mitigate speculative hijack?	Implementation – How does the tool detect or mitigate vulnerabilities? (Section 4.2.4)	
✓	Models/mitigates speculative hijack attacks	Taint	Taint tracking (abstract execution) Manual effort
→	Models/mitigates forward-edge (jimp) hijack only	Safety	Memory safety (abstract execution) Fuzz
∩	Models/mitigates hijack only via speculative stores	SelfC	Self composition (abstract execution) Flow
×	Does not model/mitigate speculative hijack attacks	Cache	Cache must-hit analysis (abstract execution) Struct
		+	Includes additional work or constraints to remove sequential trace (Section 4.2.2)
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)		

Semantics or tool name	Level	Leakage	Variants	Nondet.	Fence	OOO	Win.	Hij.	Tool	Impl.
Cauligi et al. [36] (Pitchfork)	Low	[·]_et P,B,M	P,B,R,S	Directives	✓	✓	✓	✓	Det*	Taint
Cheang et al. [41]	Low	[·]_arch P,M,S,R	P	Oracle	✓	×	✓	×	Det/Mit	SelfC+
Daniel et al. [47] (Binsec/Haunted)	Low	[·]_et P,M	P,S	Mispredict	×	×	✓	×	Det	SelfC
Guanciale et al. [63] (InSpectre)	Low	[·]_et P,M	P,B,R,S	—	✓	×	×	✓	—	—
Guarnieri et al. [64] (Spectector)	Low	[·]_et P,B,M	P	Oracle	✓	×	✓	→	Det	SelfC+
Guarnieri et al. [65]	Low	(parametrized)	P ¹	Oracle	✓	✓	✓	×	Det	SelfC+
McIlroy et al. [100]	Low	[·]_cache T	P ²	Oracle	~	×	✓	→	Mit*	Manual
Barthe et al. [16] (Jasmin)	Medium	[·]_et P,B,M	P,S	Directives	✓	×	×	×	Det	Safety
Patrignani and Guarnieri [116]	Medium	[·]_et P,B,M,L ³	P ¹	Mispredict	✓	×	✓	×	—	—
Vassena et al. [151] (Blade)	Medium	[·]_et B,M	P	Directives	✓	✓	×	×	Mit	Flow
Colvin and Winter [44]	High	[·]_mem M	P	Weak-mem	✓	✓	×	×	Val	Val
Disselkoen et al. [51]	High	[·]_mem M	P	Weak-mem	✓	✓	×	×	—	—
AISE [161]	—	[·]_cache C	P	Mispredict	×	×	✓	×	Det	Cache+
ASTCVW [83]	—	[·]_arch L	P ⁴	—	×	×	×	×	Det	Taint
ELFbac [78]	—	[·]_arch L	P	—	×	×	×	✓	Mit	Struct
KLEESpectre [154]	(w/ cache)	[·]_cache C	P	Mispredict	✓	×	✓	×	Det	Cache
	(w/o cache)	[·]_mem M	P	Mispredict	✓	×	✓	×	Det	Taint
oo7 [155]	(v1 pattern)	[·]_mem M	P	—	~	×	✓	×	Det/Mit	Flow
	(“weak” and v1.1 patterns)	[·]_arch L	P	—	~	×	✓	~	Det/Mit	Flow
Specfuser [135]	—	— ⁶	P,B,R	—	×	×	×	✓	Mit	Struct
SpecFuzz [113]	—	[·]_arch L	P	Mispredict	—	—	—	✓	Det	Fuzz
SpecTaint [121]	—	[·]_mem ⁷ M	P	Mispredict	✓	×	✓	~	Det	Taint
SpecuSym [66]	—	[·]_cache C	P	Mispredict	×	×	✓	×	Det	SelfC+
Swivel [111]	(poisoning protection)	[·]_mem M	P,B,R	—	~ ⁸	×	×	✓	Mit	Struct
	(breakout protection)	[·]_arch L	P,B,R	—	~ ⁸	×	×	✓	Mit	Struct
Venkman [137]	—	[·]_arch L	P,B,R	—	~	×	×	✓	Mit	Struct

$\llbracket \cdot \rrbracket_{\text{ct}}$ 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 $\llbracket \cdot \rrbracket_{\text{ct}}$ leakage model (or an equivalent) [16, 36, 47, 63, 64, 116, 151]. Guarnieri et al. [65] leave the leakage model abstract, allowing the semantics to be used with several different leakage models, including $\llbracket \cdot \rrbracket_{\text{ct}}$.

Leakage models for isolation. Sections 4.1.2 and 4.1.3 describe the $\llbracket \cdot \rrbracket_{\text{arch}}$ leakage model, which is a better fit for modeling speculative isolation, e.g., for a WebAssembly runtime executing untrusted code [111] or a kernel defending against memory region probing [61]. Under $\llbracket \cdot \rrbracket_{\text{arch}}$, *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 $\llbracket \cdot \rrbracket_{\text{arch}}$ and this π .

The $\llbracket \cdot \rrbracket_{\text{arch}}$ leakage model appears less-frequently than $\llbracket \cdot \rrbracket_{\text{ct}}$ in formal models: Only two of the semantics in Table 4.1 ([41, 65]) use the $\llbracket \cdot \rrbracket_{\text{arch}}$ leakage model. On the other hand, Spectre sandbox isolation frameworks such as Swivel [111], Venkman [137], and ELFbac [78] implicitly use the $\llbracket \cdot \rrbracket_{\text{arch}}$ model, as do SpecFuzz [113], ASTCVW [83], SpecTaint [121], and certain modes of oo7 [155]. The three isolation frameworks all explicitly prevent memory reads or writes to any locations outside of isolation boundary—i.e., enforcing non-interference under $\llbracket \cdot \rrbracket_{\text{arch}}$. The four detection tools, SpecFuzz, ASTCVW, SpecTaint, and oo7 (in “weak” or “v1.1” mode), more generally 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 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 $\llbracket \cdot \rrbracket_{\text{mem}}$ leakage model, like $\llbracket \cdot \rrbracket_{\text{ct}}$, allows an attacker to observe the sequence of memory accesses during the execution of the program. The $\llbracket \cdot \rrbracket_{\text{cache}}$ leakage model instead 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 leakage models even weaker than these—for instance, oo7 only emits leakages that it considers 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 once the program terminates.

All of these models, including $\llbracket \cdot \rrbracket_{\text{mem}}$ and $\llbracket \cdot \rrbracket_{\text{cache}}$, are weaker than $\llbracket \cdot \rrbracket_{\text{ct}}$ —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 influences a memory address. For instance, in the following code, even though `cond` does not directly influence a memory address, an attacker could infer the value of `cond` based on whether `arr[a]` is cached or not:

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

Because the $\llbracket \cdot \rrbracket_{\text{mem}}$ and $\llbracket \cdot \rrbracket_{\text{cache}}$ 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 choose a strong leakage model appropriate for their application domain, such as $\llbracket \cdot \rrbracket_{\text{ct}}$ or $\llbracket \cdot \rrbracket_{\text{arch}}$.

That said, weaker leakage models are still useful in certain settings: Tools which are interested in only a certain vulnerability class 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—even some data cache leakage, as discussed above—some detection tools still use the $\llbracket \cdot \rrbracket_{\text{cache}}$ or $\llbracket \cdot \rrbracket_{\text{mem}}$ 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 [66, 154] 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 [44] and Disselkoe et al. [51] have a more structured view of program execution, for instance, using pomsets [60]. Both of these semantics define leakages in a way equivalent to the $\llbracket \cdot \rrbracket_{\text{mem}}$ leakage model; it remains an open problem to explore how to define $\llbracket \cdot \rrbracket_{\text{ct}}$ or $\llbracket \cdot \rrbracket_{\text{arch}}$ leakage in this more structured execution model—in particular, what it means for such a semantics to allow an attacker to observe control-flow leakage.

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 [67] and object capabilities [96] to language-based isolation techniques like information flow control [129]. 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 simply enforce the same leakage model (e.g., $\llbracket \cdot \rrbracket_{\text{ct}}$) with the same security policy—no secrets should be leaked whether in normal or speculative execution. We refer to this straightforward extension as a *direct* non-interference property, or direct NI.

Definition 4.2.1 (Direct non-interference). *Program p satisfies direct non-interference with respect to a given contract $\llbracket \cdot \rrbracket$ 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, $p \vdash NI(\pi, \llbracket \cdot \rrbracket)$ is defined as*

$$\forall \sigma, \sigma' : \sigma \simeq_{\pi} \sigma' \Rightarrow \llbracket p \rrbracket(\sigma) = \llbracket p \rrbracket(\sigma').$$

We elide writing π for brevity—e.g., $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$ 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 4.2.2 (Relative non-interference). *Program p satisfies relative non-interference from contract $\llbracket \cdot \rrbracket_a^{\text{seq}}$ to $\llbracket \cdot \rrbracket_b^{\beta}$ and with policy π if: For all pairs of π -equivalent initial states σ and σ' ,*

if executing p under $\llbracket \cdot \rrbracket_a^{seq}$ produces equal traces, then executing p under $\llbracket \cdot \rrbracket_b^\beta$ produces equal traces. That is, $p \vdash NI(\pi, \llbracket \cdot \rrbracket_a^{seq} \Rightarrow \llbracket \cdot \rrbracket_b^\beta)$ is defined as

$$\begin{aligned} \forall \sigma, \sigma' : \sigma \simeq_\pi \sigma' \wedge \llbracket p \rrbracket_a^{seq}(\sigma) = \llbracket p \rrbracket_a^{seq}(\sigma') \\ \implies \llbracket p \rrbracket_b^\beta(\sigma) = \llbracket p \rrbracket_b^\beta(\sigma'). \end{aligned}$$

For non-terminating programs, we can compare finite prefixes of $\llbracket p \rrbracket^\beta$ against their sequential projections to $\llbracket p \rrbracket^{seq}$ —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 $NI(\pi, \llbracket \cdot \rrbracket_a^{seq} \Rightarrow \llbracket \cdot \rrbracket_b^\beta)$ for a program p can be expressed equivalently as a direct property $NI(\pi', \llbracket \cdot \rrbracket_b^\beta)$, where $\pi' = \pi \setminus canLeak(p, \llbracket \cdot \rrbracket_a^{seq})$. That is, we treat anything that could possibly leak under contract $\llbracket \cdot \rrbracket_a^{seq}$ as public. Relative NI is thus a weaker property than direct NI, as it implicitly declassifies anything that might leak during sequential execution.

However, relative NI is a stronger property than a conventional implication. For example, the property $NI(\llbracket \cdot \rrbracket_{ct}^{seq} \Rightarrow \llbracket \cdot \rrbracket_{ct}^{pht})$ makes no guarantees at all about a program that is not sequentially constant-time. Conversely, the relative NI property $NI(\llbracket \cdot \rrbracket_{ct}^{seq} \Rightarrow \llbracket \cdot \rrbracket_{ct}^{pht})$ 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 speculative 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

Table 4.2: Speculative security properties in prior works and their equivalent non-interference statements (on following page; legend appears here). We write $\approx NI(\dots)$ for unsound approximations of non-interference properties. ¹Tracks taint of *attacker influence* rather than value sensitivity. ²These works all derive their property from the definition given in [36] and share the same property name despite differences in execution mode. ³The analysis tool of [36], Pitchfork, only verifies the weaker property $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht-st}})$. ⁴The definitions of SNI and wSNI are parameterized over the target leakage model. ⁵The definition of wSNI in [65] does not require that the initial states be π -equivalent.

Property or tool name	Non-interference prop.	Precision
McIlroy et al. [100]	$\approx NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$	hyper
oo7 [155] $\Phi_{\text{spectre}}^{\text{weak}}, \Phi_{\text{spectre}}^{\text{p1,1}}$	$\approx NI(\llbracket \cdot \rrbracket_{\text{mem}}^{\text{pht}})$ $\approx NI(\llbracket \cdot \rrbracket_{\text{arch}}^{\text{pht}})$	taint ¹
Cache analysis [66, 161] [154]	$NI(\llbracket \cdot \rrbracket_{\text{etche}}^{\text{pht}})$	hyper taint
Weak memory modeling [44, 51]	$NI(\llbracket \cdot \rrbracket_{\text{mem}}^{\text{pht}})$	hyper
[151]	$NI(\llbracket \cdot \rrbracket_{\text{et}}^{\text{pht}})$	taint
Speculative constant-time (SCT) ² [16, 47] [36]	$NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht-st}})$ $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pbrs}})$ ³	hyper
Speculative non-interference (SNI) [64, 65]	$NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$ ⁴	hyper
Robust speculative non-interference (RSNI) [116]	$NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$	hyper taint
Robust speculative safety (RSS) [116]		
Conditional noninterference [63]	$NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{ct}}^{\text{pbrs}})$	hyper
Weak speculative non-interference (wSNI) [65]	$NI(\llbracket \cdot \rrbracket_{\text{arch}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{arch}}^{\text{pht}})$ ^{4,5}	hyper
Weak robust speculative non-interference (RSNI ⁻) [116]		hyper
Trace property-dependent observational determinism (TPOD) [41]	$NI(\llbracket \cdot \rrbracket_{\text{arch}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$	hyper hyper taint
Weak robust speculative safety (RSS ⁻) [116]		

Execution models (Section 4.2.3)	Precision of the defined security property
$\llbracket \cdot \rrbracket^{\text{seq}}$ Sequential execution	hyper
$\llbracket \cdot \rrbracket^{\text{pht}}$ Captures Spectre-PHT	taint
$\llbracket \cdot \rrbracket^{\text{pht-st}}$ Captures Spectre-PHT/-STL	Non-interference hyperproperty, requires two π -equivalent executions
$\llbracket \cdot \rrbracket^{\text{pbrs}}$ Captures Spectre-PHT/-BTB/-RSB/-STL	Sound approximation using taint tracking, requires only one execution

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 ([16, 36, 47, 151]) all verify that programs satisfy the direct *speculative constant-time* (SCT) property.

Additionally, we find that current tools that verify relative NI [41, 64] 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 $NI(\llbracket \cdot \rrbracket_{ct}^{seq})$, then a compiler that enforces $NI(\llbracket \cdot \rrbracket_{ct}^{seq} \Rightarrow \llbracket \cdot \rrbracket_{ct}^{pht})$ will compile p to a program that is *speculatively* constant-time $NI(\llbracket \cdot \rrbracket_{ct}^{pht})$. Similarly, if a program is properly sandboxed under sequential execution $NI(\llbracket \cdot \rrbracket_{arch}^{seq})$, and is compiled with a compiler that introduces no new *arch* leakage, the resulting program will remain sandboxed even speculatively. Indeed, these propositions are proven by Guarnieri et al. [65].

Similarly, Patrignani and Guarnieri [116] 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)—indeed, most other security properties listed in Table 4.2 are implicitly robust. 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 [16], Binsec/Haunted [47], and Pitchfork [36] 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) [87] or BTB (for indirect jumps) [86]. 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) [82, 99, 142]. Formally, these

²Including, on x86-family processors, into the *middle* of an instruction [28].

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, our semantics needs to be able to model these behaviors in some fashion.

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) [155] and SpecTaint [121] conservatively assume that writes to transient addresses can overwrite *anything*, and thus immediately flag this behavior as vulnerable.

The InSpectre semantics [63] proceeds in the opposite direction—it allows the processor to (mis)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.

Speculation windows. As shown in Table 4.1, several semantics and tools 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. And 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 [47], 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 [151] and Jasmin [16] are able to prove security without reasoning about speculation windows, modeling a speculation window would reduce 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 tricks to sidestep particularly problematic Spectre variants. For example, even though Jasmin [16] 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 [70]. After eliminating complex or otherwise troublesome speculative behavior, a tool only needs to consider those that remain.

Cross-address-space attacks. Previous systematizations of Spectre attacks [35] differentiate between *same-address-space* and *cross-address-space* attacks. Same-address-space attacks are generally simpler to perform, as they rely on repeatedly executing the victim code itself 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 [155], 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. [154].

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. The semantics in Table 4.1 all allow these nondeterministic choices to be actively adversarial—for instance, given by attacker-specified directives [36, 151], or, equivalently, by consulting an abstract oracle [41, 64, 65, 100]. 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, they 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 [66], KLEESpectre [154], and AISE [161] handle 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 of course only handles the conditional branch predictor; i.e., Spectre-PHT attacks. Pitchfork [36] and Binsec/Haunted [47] adapt the always-mispredict strategy to additionally account for out-of-order execution and Spectre-STL. Although it may not be immediately clear that these always-mispredict strategies are sufficient to prove security, especially when the attacker can make any number of antagonistic prediction choices, these strategies do indeed form a sound analysis [36, 47, 64].

Unfortunately, simulating execution only works for semantics where the nondeterminism is relatively constrained: Conditional branches are a simple boolean choice, and store-to-load

predictions are limited to prior memory operations within the speculation window. If we pursue other Spectre variants, we will quickly become overwhelmed—again, an unconstrained hijack gadget can be exploited to land 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 [151] and oo7 [155] 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 [111], Venkman [137], and ELFbac [78] 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 of memory—regardless of how a program may misspeculate, it will not be able to read (and thus 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 proposes to recompile all programs on the system to mask addresses before jumping. Both systems also enforce speculative control-flow integrity checks to prevent speculative hijacking, whether by relying on hardware features [74] 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 of the tools in Table 4.1. Perhaps the most complete comparison is by Daniel et al. [47], 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 would help us 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, 47, 48, 64, 154]. 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 [17, 47, 48]. 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 [16], Patrignani and Guarnieri [116], and Blade [151] 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. [151] 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. [16] 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 can only access 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 can combine with other sequential security properties to give corresponding speculative guarantees, such as for sandboxing, information flow, or rich type systems.

Weak-memory-style semantics. Colvin and Winter [44] and Disselkoen et al. [51] both present a further abstracted semantics in the style of weak memory models. These semantics represent a fundamentally different approach: Rather than creating operational models of speculative hardware, these authors lift the concept of speculative execution directly to a higher level and reason about it there.

These works provide interesting insights about the relation between Spectre attacks and the weak memory models which characterize modern hardware. They also open the door to adapting techniques from that community to defend against Spectre attacks in software. However, as these models are abstracted away from microarchitectural details, they are only suited for analyzing particular Spectre variants—both [44, 51] focus only on Spectre-PHT—and are difficult to adapt to other attacks. In addition, it remains an open problem to translate a semantics of this style into a concrete analysis tool: Neither of these works present a tool which can automatically perform a security analysis of a target program.³ That said, this 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 [116], they are overly conservative and incur a significant performance cost. Other compiler mitigations been

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

shown unsound [113]—or worse, even introduce new Spectre vulnerabilities [47]—further reinforcing that these techniques must be grounded in a formal semantics.

Open problems: Formalization of new compilation techniques. Swivel [111], Venkman [137], and ELFbac [78] show how the structure of code itself can provide security guarantees at a reduced performance cost. For instance, Venkman [137] and Swivel [111] 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 [40], 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. [151] 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: Compiling standard C code with 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 is too expressive—allows too many possible behaviors—makes many analyses intractable. One fundamental purpose of semantics is to provide

a reasonable abstraction (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. We discussed these tradeoffs in Section 4.2.3. But beyond speculative execution itself, there are many other microarchitectural features which could be 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 out-of-order execution. 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. Further, as a result of including out-of-order execution in their respective semantics, Disselkoen et al. [51] and Guanciale et al. [63] propose to abuse out-of-order execution to conduct (at least theoretical) novel side-channel attacks.⁵

Although modeling out-of-order execution might make the semantics simpler, the additional expressivity definitely 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

⁴Or, perhaps because out-of-order execution is often discussed alongside, or even confused with, speculative execution

⁵Disselkoen et al. [51] propose to abuse compile-time instruction reordering, which is different from microarchitectural out-of-order execution, but related.

intractable analysis. Indeed, while Guarnieri et al. [65] and Colvin and Winter [44] present analysis tools based on their respective out-of-order semantics, they only analyze very simple Spectre gadgets, 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 argue that this reduction does not affect the soundness of Pitchfork’s analysis.

Caches and TLBs. Some speculative semantics and tools [66, 100, 154, 161] 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 $\llbracket \cdot \rrbracket_{\text{cache}}$ leakage model, rather than the $\llbracket \cdot \rrbracket_{\text{ct}}$ or $\llbracket \cdot \rrbracket_{\text{arch}}$ leakage models which provide stronger defensive guarantees. Following in the tradition 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 could result in leakages, beyond the ones we’ve been focusing on in this chapter. For instance, in the presence of multithreading, port contention in the processor’s execution units can reveal sensitive information [29]; and many processor instructions, e.g., floating-point or SIMD instructions, can reveal information about their operands through timing side channels [10]. Most existing semantics do not model these specific effects. However, the commonly-used $\llbracket \cdot \rrbracket_{\text{ct}}$ and $\llbracket \cdot \rrbracket_{\text{arch}}$ 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 $\llbracket \cdot \rrbracket_{ct}$ leakage model would have already considered the sensitive data leaked once it influenced control flow. For variable-time instructions, most works’ definitions of $\llbracket \cdot \rrbracket_{ct}$ do not capture this leakage, but extending those definitions to cover it is straightforward [7]. In both of these examples, the $\llbracket \cdot \rrbracket_{arch}$ leakage model would capture all of the leaks, because it (even more conservatively) would already consider the sensitive data leaked once it reached a register, long before it could influence control-flow or be used in a variable-time instruction. Although modeling any of these effects more precisely could 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 $\llbracket \cdot \rrbracket_{ct}$ and $\llbracket \cdot \rrbracket_{arch}$ 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 presented by Colvin and Winter [44] and Disselkoen et al. [51]—multiple threads are an inherent feature for this style of semantics. These semantics may be a promising vehicle for further exploring the interaction between speculation and concurrency. For other semantics, adding detailed models of multithreading is probably not worth the increased analysis complexity.

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 [123]. This shifts the burden to OS engineers from application and runtime system engineers. Developing Spectre foundations to model the process abstraction would elucidate the security guarantees of such systems. This would be especially useful since there is evidence showing that the process boundary does not keep an attacker from performing

out-of-place training of the conditional branch predictor, or from leaking secrets via the cache state [35].

4.3 Related Work

There has been a lot of interest in Spectre and other transient execution attacks, both in industry and in academia. 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. [35] 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); then they 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 for each. 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. [34] 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. [162] 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 [81], cleaning up the cache state after misprediction [130], or making the cache invisible to speculation by incorporating some separate internal state [2, 80, 163]. Unfortunately, attackers can still use side channels other than the cache to exploit speculative execution [29, 134]. NDA [158], DOLMA [95], and Speculative Taint Tracking (STT) [167] block additional speculative covert channels by analyzing and classifying instructions that can leak information.

Fadiheh et al. [55] 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 $NI(\pi, \llbracket \cdot \rrbracket_{\text{arch}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{arch}}^{\text{pht}})$.

The insights and recommendations from our work can guide future hardware mitigations; properties like $\llbracket \cdot \rrbracket_{\text{ct}}$ or $\llbracket \cdot \rrbracket_{\text{arch}}$ can serve as contracts of what software expects from hardware [65] (or how defenses need to bridge the gap in software when hardware only offers partial mitigations).

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. [145] propose context-sensitive fencing, making various speculative barriers available to software. Li et al. [92] propose memory instructions with a conditional speculation flag. Context [132] and SpectreGuard [57] allow software to mark secrets in memory. This information is propagated through the microarchitecture to block speculative access to the marked regions. SpecCFI [88] suggests a hardware extension similar to Intel CET [74] that provides target label instructions

with speculative guarantees. Finally, several recent proposals allow partitioning branch predictors based on context provided by the software [153, 170]. As these approaches require both software and hardware changes, we will need a formal semantics to apply them correctly; this represents valuable future work.

4.3.4 Other transient execution attacks

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

Meltdown variants. The Meltdown attack [93] 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 [149] and microarchitectural data sampling (MDS) [33, 71] 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 [106, 107, 133], but most recent Intel CPUs have hardware-level mitigations for all these vulnerabilities in the form of microcode patches or proprietary hardware fixes [73].

Load value injection. Load value injection (LVI) [150] exploits the same root cause as Meltdown, Foreshadow, and MDS. But LVI 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 [72, 150], Intel only suggests applying them to legacy enclave software. Like Meltdown, LVI does not need software-based mitigation on recent Intel CPUs, and our systematization does not apply.

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 $\llbracket \cdot \rrbracket_{\text{arch}}$ or $\llbracket \cdot \rrbracket_{\text{ct}}$. 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 can increase 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 $\llbracket \cdot \rrbracket_{\text{arch}}$ and $\llbracket \cdot \rrbracket_{\text{ct}}$ models; details like cache structures or port contention introduce complexity and give up on 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 atop formal frameworks and to develop sound Spectre defenses with strong, precise security guarantees.

Acknowledgements

We thank Matthew Kolosick for helping us understand some of the formal systems and in organizing our presentation. This work was supported in part by gifts from Cisco; by the NSF under Grant Numbers CNS-1514435, CCF-1918573, and CAREER CNS-2048262; and, by the

CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Work by Gilles Barthe was supported by the Office of Naval Research (ONR) under project N00014-15-1-2750.

Chapter 4, in part, has been submitted for publication of the material as it may appear in 43rd IEEE Symposium on Security and Privacy (Oakland '22), Cauligi, Sunjay; Disselkoen, Craig; Moghimi, Daniel; Barthe, Gilles; Stefan, Deian. The dissertation author was the primary investigator and author of this material.

Conclusion

We see time and time again that timing side-channels thoroughly erode our mental models of how programs execute and how we can keep data confidential—even worse, the effects of speculative execution topple all semblance of basic security properties such as memory safety, type safety, or even simply basic control flow. These problems, however, are not insurmountable: With the proper groundwork, we can yet reclaim these security properties in the face of speculative execution. To that end, this dissertation has laid the foundation for rebuilding formal security atop the shaky ground of speculative execution.

We started with FaCT, a DSL for writing sequential constant-time code. Although FaCT doesn't consider speculative effects, it gives us a blueprint for automatic, sound, and secure compilation in the speculative domain. We introduced a formal type system for constant-time, allowing us to capture timing side-channels as a violation of typing judgements. We then demonstrated various compilation techniques that *automatically* transform potentially insecure, high-level FaCT programs all the way down to low-level constant-time bitcode.

We then developed the foundations of *speculative* constant-time with Pitchfork: We defined a formal semantics that captures the effects of microarchitectural predictors and speculative execution. Through this semantics, we were able to extend the traditional definition of constant-time to the speculative domain and show that Spectre attacks are simply a violation of this new property. We also showed that our formal foundation was indeed solid and practical: Our verification tool, Pitchfork, was able to find subtle Spectre vulnerabilities in real code.

We built upon Pitchfork’s foundations, adapting its semantics for tackling speculative security in the higher-level context of SFI and software sandboxing. We generalized the notion of speculative constant-time to formally capture the SFI protections against both sandbox breakout and poisoning attacks. We demonstrated the structural soundness of our framework, showing how mitigations from existing tools serve (or fail) to uphold our speculative SFI properties.

Finally, we gave a bird’s eye view of speculative software semantics at the time of writing: We categorized and systematized various design choices made in our and others’ semantics and identified open areas that have yet to be filled in. We examined how each design choice taken either builds towards or works against our eventual goals; how each open problem solved is one less impediment in our pursuit of high-level software security.

Ultimately, we want to allow developers to program in high-level languages while being verifiably free from Spectre attacks. This dissertation presented formal foundations and frameworks for reclaiming these security goals: We defined type systems, semantics, and formal techniques for verifying and enforcing constant-time and sandbox properties even in the face of speculation; and we implemented these techniques in practical tools to detect and defend against constant-time and Spectre attacks.

Appendix A

FaCT: Deferred definitions and proofs

A.1 Semantics

We define the behavior of expressions, statements and functions using an instrumented big-step semantics. Informally, the big-step semantics relates initial configurations, final configurations, and leakages. Initial configurations are triples of the form (C, ρ, h) where C is an expression, a statement or a function, ρ is an environment mapping variables to values, and h is a heap mapping pointers to values.

Definition A.1.1 (Values). *The set of values is defined by the following syntax:*

$$\begin{array}{ll} v ::= n & \textit{integer} \\ | b & \textit{boolean} \\ | p & \textit{pointer} \\ | [v_1; \dots; v_n] & \textit{array of size } n \\ | \{x_1 = v_1, \dots, x_n = v_n\} & \textit{structure} \end{array}$$

An environment is defined as a partial mapping from variables to values, and a heap is defined as a partial mapping from pointers to values. We say that a pointer p is allocated in a heap h , written $p \in h$, if $h(p)$ is defined. If $p \in h$ then the associated value to p can be updated: $h[p \leftarrow v]$. The associated values of other pointers are unchanged. We assume we are given a

$$\begin{array}{c}
\text{EXPR-STEP} \\
(e, \rho, h) \xrightarrow{\psi} (v, h)
\end{array}
\qquad
\begin{array}{c}
\text{STMT-STEP} \\
(S, \rho, h) \xrightarrow{\psi} (v, h)
\end{array}$$

$$\begin{array}{c}
\text{REF} \\
\frac{(e, \rho, h) \xrightarrow{\psi} (v, h')}{\text{FRESH}(h', v) = (p, h'')} \\
\frac{}{(\text{ref } e, \rho, h) \xrightarrow{\psi} (p, \rho, h'')}
\end{array}
\qquad
\begin{array}{c}
\text{ARRAY-GET} \\
\frac{(e_1, \rho, h) \xrightarrow{\psi_1} ([v_0, \dots, v_k], h')}{(e_2, \rho, h') \xrightarrow{\psi_2} (n, h'')} \quad 0 \leq n < k \\
\frac{}{(e_1[e_2], \rho, h) \xrightarrow{\psi_1 + \psi_2 + \text{Arr}(n)} (v_n, h'')}
\end{array}
\qquad
\begin{array}{c}
\text{SEQ-RET} \\
\frac{(i, \rho, h) \xrightarrow{\psi} (v, h')}{(i; S, \rho, h) \xrightarrow{\psi} (v, h')}
\end{array}$$

$$\begin{array}{c}
\text{SEQ-NORET} \\
\frac{(i, \rho, h) \xrightarrow{\psi_1} (\rho', h')}{(S, \rho', h') \xrightarrow{\psi_2} (v, h'')} \\
\frac{}{(i; S, \rho, h) \xrightarrow{\psi_1 + \psi_2} (v, h'')}
\end{array}
\qquad
\begin{array}{c}
\text{VARDEC} \\
\frac{(e, \rho, h) \xrightarrow{\psi} (v, h')}{(x = e, \rho, h) \xrightarrow{\psi} (\rho[x \leftarrow v], h')}
\end{array}$$

$$\begin{array}{c}
\text{ASSIGN} \\
\frac{(e_1, \rho, h) \xrightarrow{\psi_1} (p, h')}{(e_2, \rho, h') \xrightarrow{\psi_2} (v, h'')} \\
\frac{}{(e_1 := e_2, \rho, h) \xrightarrow{\psi_1 + \psi_2} (\rho, h''[p \leftarrow v])}
\end{array}
\qquad
\begin{array}{c}
\text{RETURN} \\
\frac{(e, \rho, h) \xrightarrow{\psi} (v, h')}{(\text{return } e, \rho, h) \xrightarrow{\psi} (v, h')}
\end{array}$$

$$\begin{array}{c}
\text{BLOCK} \\
\frac{(S, \rho, h) \xrightarrow{\psi} (v, h')}{(\{S\}, \rho, h) \xrightarrow{\psi} (v, h')}
\end{array}
\qquad
\begin{array}{c}
\text{FN-CALL} \\
\frac{(e_1, \rho, h) \xrightarrow{\psi_1} (v_1, h_1) \quad \dots \quad (e_n, \rho, h_{n-1}) \xrightarrow{\psi_n} (v_n, h_n)}{(F, \vec{v}, h_n) \xrightarrow{\psi} (v, h')} \\
\frac{}{(x = f(\vec{e}), \rho, h) \xrightarrow{\sum \psi_i + \psi} (\rho[x \leftarrow v], h')}
\end{array}$$

$$\begin{array}{c}
\text{FN} \\
\frac{(F.S, [F.\vec{x} \leftarrow \vec{v}], h) \xrightarrow{\psi} (v, h')}{(F, \vec{v}, h) \xrightarrow{\psi} (v, h')}
\end{array}
\qquad
\begin{array}{c}
\text{IF} \\
\frac{(e, \rho, h) \xrightarrow{\psi} (b, h')}{(S_b, \rho, h') \xrightarrow{\psi_b} (v, h'')} \\
\frac{}{(\text{if } *_{\ell} e \text{ then } S_T \text{ else } S_F, \rho, h) \xrightarrow{\psi + \text{if}*(\ell, b) + \psi_b} (v, h'')}
\end{array}$$

$$\begin{array}{c}
\text{FOR} \\
\frac{(e_1, \rho, h) \xrightarrow{\psi_1} (n_1, h_1) \quad (e_2, \rho, h_1) \xrightarrow{\psi_2} (n_2, h_2)}{(\text{if } *_{\ell} n_1 < n_2 \text{ then } \{\{i = n_1; S\}; \\ \text{for } *_{\ell} i = n_1 + 1 \text{ to } n_2 \text{ DO } S\}, \rho, h_2) \xrightarrow{\psi} (v, h')} \\
\frac{}{(\text{for } *_{\ell} i = n_1 \text{ to } n_2 \text{ DO } S\}, \rho, h) \xrightarrow{\psi_1 + \psi_2 + \psi} (v, h')}
\end{array}$$

Figure A.1: Big-step semantics.

deterministic operator FRESH for creating and initializing a fresh pointer: $\text{FRESH}(h, v) = (p, h')$.

This operator satisfies:

- ▶ p is a fresh pointer, i.e., $p \notin h$
- ▶ The associated value of p is v , i.e., $h'(p) = v$
- ▶ Other pointers are unchanged, i.e., $\forall p', h(p') = h'(p')$

We further assume the existence of an equivalence relation \approx on heaps such that:

- ▶ \approx is stable by allocation: If $\text{FRESH}(h_1, v_1) = (p_1, h'_1)$ and $\text{FRESH}(h_2, v_2) = (p_2, h'_2)$ and $h_1 \approx h_2$ then $p_1 = p_2$ and $h'_1 \approx h'_2$.
- ▶ \approx is stable by update: if $h_1 \approx h_2$ then $h_1[p \leftarrow v_1] \approx h_2[p \leftarrow v_2]$.

A final configuration is either a pair consisting of a value and a heap, or of an environment and a heap. In particular, the semantics of expressions $(e, \rho, h) \xrightarrow{\psi} (v, h')$ returns a value and a new heap (creation of fresh reference). Here, ψ corresponds to the leakage of the evaluation of e . The semantics of statements is given by two judgments of similar form: $(S, \rho, h) \xrightarrow{\psi} (\rho', h')$ and $(S, \rho, h) \xrightarrow{\psi} (v, h')$. These judgments correspond to statements that do not and do return values, respectively. Again, ψ is the leakage produced by the evaluation of the statement. Finally, the semantics of a function is modelled by a judgment of the form $(f, \vec{v}, h) \xrightarrow{\psi} (v', h')$, where \vec{v} denotes the values of the parameters of the function, and v' is the return value (we only consider functions that return a value). Figure A.1 presents the semantics. Rules are standard, with the exception of leakage. Primarily, array accesses leak the index at which they are accessed, conditionals leak their control flow, and other rules combine leakage of sub-computations according to evaluation order. Note that in the rules for conditionals and for loops we assume that the guard of the statement is identified by a unique label, which we record in the leakage.

$$\begin{array}{c}
\text{RULES} \\
\Gamma \vdash e : \beta \\
pc, \beta_r \vdash S : \Gamma \rightarrow \Gamma' \\
\omega \vdash \beta_r f(\vec{x} : \vec{\beta}) \{ S \}
\end{array}$$

$$\begin{array}{c}
\text{SEQ} \\
\frac{pc, \beta_r \vdash S_1 : \Gamma \rightarrow \Gamma' \quad pc, \beta_r \vdash S_2 : \Gamma' \rightarrow \Gamma''}{pc, \beta_r \vdash S_1; S_2 : \Gamma \rightarrow \Gamma''}
\end{array}$$

$$\begin{array}{c}
\text{VAR-DEC} \\
\frac{\Gamma \vdash e : \beta \quad \Gamma' = \Gamma, x : \beta}{pc, \beta_r \vdash \beta x = e : \Gamma \rightarrow \Gamma'}
\end{array}$$

$$\begin{array}{c}
\text{VAR-DEC-FN-CALL} \\
\frac{f : (\vec{\beta}) \rightarrow \beta \quad hasMut(f) \Rightarrow pc \sqsubseteq \omega(f) \quad \Gamma \vdash e_i : \beta_i \quad \Gamma' = \Gamma, x : \beta}{pc, \beta_r \vdash \beta x = f(\vec{e}) : \Gamma \rightarrow \Gamma'}
\end{array}$$

$$\begin{array}{c}
\text{ASSIGN} \\
\frac{\Gamma \vdash e_1 : \text{REF}_W[\beta] \quad \Gamma \vdash e_2 : \beta \quad pc \sqsubseteq \beta}{pc, \beta_r \vdash e_1 := e_2 : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{IF} \\
\frac{\Gamma \vdash e : \text{BOOL}_\ell \quad pc \sqcup \ell, \beta_r \vdash S_1 : \Gamma \rightarrow \Gamma_1 \quad pc \sqcup \ell, \beta_r \vdash S_2 : \Gamma \rightarrow \Gamma_2}{pc, \beta_r \vdash \text{if}(e) \{ S_1 \} \text{ else } \{ S_2 \} : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{FOR-RANGE} \\
\frac{\Gamma \vdash e_1 : \text{UINT}_{\text{SEC}} \quad \Gamma \vdash e_2 : \text{UINT}_{\text{SEC}} \quad \Gamma' = \Gamma, x : \text{UINT}_{\text{SEC}} \quad pc, \beta_r \vdash S : \Gamma' \rightarrow \Gamma''}{pc, \beta_r \vdash \text{for}(x \text{ from } e_1 \text{ to } e_2) \{ S \} : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{RETURN} \\
\frac{\Gamma \vdash e : \beta_r \quad pc \sqsubseteq \beta_r}{pc, \beta_r \vdash \text{return } e : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{FN-DEC} \\
\frac{pc = \omega(f) \quad \Gamma = \{\vec{x} : \vec{\beta}\} \quad pc, \beta_r \vdash S : \Gamma, \text{PUB} \rightarrow \Gamma'}{\omega \vdash \beta_r f(\vec{x} : \vec{\beta}) \{ S \}}
\end{array}$$

Figure A.2: Type system \vdash_{rd} for return deferral.

A.2 Return deferral

We prove that return deferral is correct, i.e., preserves the behavior of programs; and secure, which we formalize as a type-preservation result.

Type system and type-preservation. We define a variant of the type system that only allows `return` statements in `PUB` contexts. The judgments are thus of the form $pc, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$ or $\omega \vdash \beta_r f(\vec{x} : \vec{\beta}) \{ S \}$, i.e., the return context label is omitted. The typing rules for statements are given in Figure A.2; rules for expressions do not change.

We prove that return deferral transforms typeable expressions (resp. statements and procedures) of the source type system into typeable expressions (resp. statements and procedures) of the \vdash_{rd} type system.

First, we prove preliminary lemmas.

Lemma A.2.1. *If $\omega, pc, \beta_r \vdash S : \Gamma, rc \rightarrow \Gamma', rc'$ then $rc \sqsubseteq rc'$.*

Lemma A.2.2 (PC subtyping type system for return deferral). *For all $pc_1 \sqsubseteq pc_2$, if $pc_2, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$ then $pc_1, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$.*

Proof. By induction on $pc_2, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$. □

Lemma A.2.3 (Type preservation for return deferral).

If $\omega, pc, \beta_r \vdash S : \Gamma, rc \rightarrow \Gamma', rc'$:

1. $\Phi, pc, rc \vdash S \rightarrow S'$ then $pc \sqcup rc, \beta_r \vdash S' : \bar{\Gamma} \rightarrow \bar{\Gamma}'$
2. $\Phi, pc, rc \vdash S \rightsquigarrow S'$ then $pc \sqcup rc, \beta_r \vdash S' : \bar{\Gamma} \rightarrow \bar{\Gamma}'$

where $\bar{\Gamma} = \Gamma[\text{notRet}, \text{rval} \leftarrow \text{REF}_{RW}[\text{BOOL}], \text{REF}_{RW}[\beta_r]]$.

Proof. We start by proving (2). Assuming that (1) holds for a given S , we prove that (2) holds for S . By case on rc :

- If rc is PUB then $\Phi, pc, rc \vdash S \rightsquigarrow S'$ is $\Phi, pc, rc \vdash S \rightarrow S'$, and we can trivially conclude using (1).
- If rc is SEC we should prove $pc \sqcup \text{SEC}, \beta_r \vdash \text{if}(\text{deref notRet})\{S'\} : \bar{\Gamma} \rightarrow \bar{\Gamma}'$. By hypothesis we have $pc \sqcup \text{SEC}, \beta_r \vdash S' : \bar{\Gamma} \rightarrow \bar{\Gamma}'$ and we can apply the IF rule of type system 2 to conclude (where l is SEC).

We now prove (1) by induction on S . The cases for (VAR-DEC, ASSIGN, IF, FOR-RANGE, RETURN) are trivial.

$$\begin{array}{c}
\text{BOOL} \\
b \simeq_m b \\
\\
\text{INT} \\
i \simeq_m i \\
\\
\text{REF} \\
\frac{p_2 = m(p_1)}{p_1 \simeq_m p_2} \\
\\
\text{ARR} \\
\frac{v_i \simeq_m w_i}{[v_0; \dots; v_n] \simeq_m [w_0; \dots; w_n]} \\
\\
\text{STRUCT} \\
\frac{v_i \simeq_m w_i}{\{x_1 = v_1, \dots, x_n = v_n\} \simeq_m \{x_1 = w_1, \dots, x_n = w_n\}} \\
\\
\text{HEAP} \\
\frac{\forall p_1 p_2, m(p_1) = p_2 \Rightarrow h_1(p_1) \simeq_m h_2(p_2)}{h_1 \simeq_m h_2} \\
\\
\text{ENV} \\
\frac{\forall x, \text{Defined } \rho(x) \Rightarrow \text{Defined } \rho'(x) \text{ and } \rho(x) \simeq_m \rho'(x)}{\rho \simeq_m \rho'}
\end{array}$$

Figure A.3: Values equivalence.

- If $S = S_1; S_2$ then we have $S' = S'_1; S'_2$ where

$$\begin{array}{l}
\omega, pc, rc \vdash S_1 : \Gamma, rc \rightarrow \Gamma', rc' \\
\Phi, pc, rc \vdash S_1 \rightarrow S'_1 \\
\omega', pc, rc' \vdash S_2 : \Gamma', rc' \rightarrow \Gamma'', rc'' \\
\Phi', pc, rc' \vdash S_2 \rightsquigarrow S'_2
\end{array}$$

By induction hypothesis, we have $pc \sqcup rc, \beta_r \vdash S'_1 : \bar{\Gamma} \rightarrow \bar{\Gamma}'$ and by (2) (using the induction hypothesis on S_2) we have $pc \sqcup rc', \beta_r \vdash S'_2 : \bar{\Gamma}' \rightarrow \bar{\Gamma}''$. Since $rc \sqsubseteq rc'$ (by lemma A.2.1), we can apply lemma A.2.2 to obtain $pc \sqcup rc, \beta_r \vdash S'_2 : \bar{\Gamma}' \rightarrow \bar{\Gamma}''$ and conclude.

- If $S = \beta x = f(\vec{e})$, we can conclude by induction hypothesis ($f.S$ can be seen as a sub-statement of S since there is no recursion).

□

Preservation of semantics. We now prove the preservation of semantics for return deferral. Since the compilation introduces references and variables, the correctness lemmas should take this into account. Given a partial mapping m from pointers to pointers, we say that two values v and v' are in relation for m , $v \simeq_m v'$ if they are equal up to pointers. Figure A.3 defines this relation. The relation is extended to heaps $h \simeq_m h'$ (rule HEAP), if for all pointers p in m we have $h(p) \simeq_m h'(m(p))$. The relation is extended to environments (rule ENV): for all defined variables x in ρ , x should be defined in ρ' and the associated values should be in relation for m .

Lemma A.2.4 (Preservation of semantics for return deferral). *Let $\rho_1 \simeq_m \rho'_1$ and $\rho'_1(\text{notRet}) = p_r$ and $\rho'_1(\text{rval}) = p_v$ and $h_1 \simeq_m h'_1$ and $h'_1(p_r) = \text{true}$ and $h'_1(p_v) = \text{init}(\beta_r)$. If $\Phi, pc, rc \vdash S \rightarrow S'$ and $(S, \rho_1, h_1) \rightarrow (v, h_2)$, then there exists v', m', h'_2 such that $m \sqsubseteq m'$ and $(S', \rho'_1, h'_1) \rightarrow (v', h'_2)$ and $h_2 \simeq_{m'} h'_2$:*

- *If $v = \rho_2$ then there exists ρ'_2 such that $v' = \rho'_2$ and $\rho_2 \simeq_{m'} \rho'_2$ and $\rho'_2(\text{notRet}) = p_r$ and $\rho'_2(\text{rval}) = p_v$ and $h'_2(p_r) = \text{true}$ and $h'_2(p_v) = \text{init}(\beta_r)$.*
- *If $v = v$, there exists v' such that $v \simeq_{m'} v'$ and $v' = v'$, or there exists ρ'_2 such that $v' = \rho'_2$ and $\rho'_2(\text{notRet}) = p_r$ and $\rho'_2(\text{notRet}) = p_v$ and $h'_2(p_r) = \text{false}$ and $h'_2(p_v) = v'$ and $v \simeq_{m'} v'$.*

Furthermore, if $h_1 \simeq_m h'_1$ and $\vec{v} \simeq_m \vec{v}'$ and $\omega \vdash F \rightarrow F'$ and $(F, \vec{v}, h_1) \rightarrow (v, h_2)$ then there exists v', m', h'_2 such that $v \simeq_{m'} v'$ and $h_2 \simeq_{m'} h'_2$ and $(F', \vec{v}', h'_1) \rightarrow (v', h'_2)$.

Proof. The proof is done by mutual induction on $\Phi, pc, rc \vdash S \rightarrow S'$ and $(F, \vec{v}, h_1) \rightarrow (v, h_2)$. The case for functions is a direct consequence of the case for statements. For statements, the interesting case is the one for sequencing, i.e., $S = S_1; S_2$. If S_1 returns in a SEC context then S'_1 will not immediately return, but after its execution notRet will be `false`. So $S'_2 = \text{if } (\text{notRet}) \{ S''_2 \}$ will immediately terminate. \square

$$\begin{array}{c}
\text{RULES} \\
\Gamma \vdash e : \beta \\
\beta_r \vdash S : \Gamma \rightarrow \Gamma' \\
\vdash \beta_r f(\vec{x} : \vec{\beta}) \{ S \}
\end{array}$$

$$\begin{array}{c}
\text{SEQ} \\
\frac{\beta_r \vdash S_1 : \Gamma \rightarrow \Gamma' \quad \beta_r \vdash S_2 : \Gamma' \rightarrow \Gamma''}{\beta_r \vdash S_1; S_2 : \Gamma \rightarrow \Gamma''}
\end{array}$$

$$\begin{array}{c}
\text{VAR-DEC} \\
\frac{\Gamma \vdash e : \beta \quad \Gamma' = \Gamma, x : \beta}{\beta_r \vdash \beta x = e : \Gamma \rightarrow \Gamma'}
\end{array}$$

$$\begin{array}{c}
\text{VAR-DEC-FN-CALL} \\
\frac{f : (\vec{\beta}) \rightarrow \beta \quad \Gamma \vdash e_i : \beta_i \quad \Gamma' = \Gamma, x : \beta}{\beta_r \vdash \beta x = f(\vec{e}) : \Gamma \rightarrow \Gamma'}
\end{array}$$

$$\begin{array}{c}
\text{ASSIGN} \\
\frac{\Gamma \vdash e_1 : \text{REF}_W[\beta] \quad \Gamma \vdash e_2 : \beta}{\beta_r \vdash e_1 := e_2 : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{IF} \\
\frac{\Gamma \vdash e : \text{BOOL}_{\text{PUB}} \quad \beta_r \vdash S_1 : \Gamma \rightarrow \Gamma_1 \quad \beta_r \vdash S_2 : \Gamma \rightarrow \Gamma_2}{\beta_r \vdash \text{if}(e) \{ S_1 \} \text{ else } \{ S_2 \} : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{FOR-RANGE} \\
\frac{\Gamma \vdash e_1 : \text{UINT}_{\text{PUB}} \quad \Gamma \vdash e_2 : \text{UINT}_{\text{PUB}} \quad \Gamma' = \Gamma, x : \text{UINT}_{\text{PUB}} \quad \beta_r \vdash S : \Gamma' \rightarrow \Gamma''}{\beta_r \vdash \text{for}(x \text{ from } e_1 \text{ to } e_2) \{ S \} : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{RETURN} \\
\frac{\Gamma \vdash e : \beta_r}{\beta_r \vdash \text{return } e : \Gamma \rightarrow \Gamma}
\end{array}$$

$$\begin{array}{c}
\text{FN-DEC} \\
\frac{\Gamma = \{\vec{x} : \vec{\beta}\} \quad \beta_r \vdash S : \Gamma \rightarrow \Gamma'}{\vdash \beta_r f(\vec{x} : \vec{\beta}) \{ S \}}
\end{array}$$

Figure A.4: Type system \vdash_{ct} for constant-time.

A.3 Branch removal

We prove that branch removal is correct, i.e., preserves the behavior of programs, and secure. For the latter, we define a new type system \vdash_{ct} , show that branch removal returns programs that are typeable with respect to \vdash_{ct} , and that typeable programs are constant-time.

Type system and type-preservation. The type system manipulates judgments of the form $\beta_r \vdash S : \Gamma \rightarrow \Gamma'$ and $\vdash \beta_r f(\vec{x} : \vec{\beta}) \{ S \}$. Notably, the path context label is omitted. Since we require that statements no longer branch on secrets, we can assume that the path context label is public throughout execution.

Figure A.4 presents the typing rules for statements in \vdash_{ct} . Rules for expressions do not change.

We prove that branch removal transforms expressions (resp. statements and procedures) typeable in \vdash_{rd} into expressions (resp. statements and procedures) typeable in \vdash_{ct} .

Lemma A.3.1. *If $\Phi, p \vdash S \rightarrow S'$ and $\bar{p}, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$ then $\beta_r \vdash S' : \bar{\Gamma}_p \rightarrow \bar{\Gamma}'_p$, where \bar{p} is PUB if $p = \text{true}$, SEC otherwise and $\bar{\Gamma}_p = \Gamma[\text{vars}(p) \leftarrow \text{BOOL}_{\text{SEC}}]$ and $\text{vars}(p)$ is the set of variables in p .*

Proof. By induction on S . □

Typeable programs are constant-time. We start by defining an equivalence between heaps. We index equivalence by a partial mapping t from pointers to types. Note that such partial mappings are naturally equipped with a partial order relation: we write that $t_1 \sqsubseteq t_2$ if for all p, β such that $t_1(p) = \beta$ we have $t_2(p) = \beta$.

We define a relation $v_1 \equiv_{\beta, t} v_2$ between values saying that the two values v_1 and v_2 are in relation with respect to the type β and the partial mapping t . The relation imposes that the values have type β and are equal according to the security level. For example, base values (booleans and integers) must be equal if their level is PUB but can be arbitrary otherwise. For pointers, the relation imposes that the two pointers are equal and the mapping t should associate a type β' such that $\beta' \sqsubseteq \beta$. The relation $h_1 \equiv_t h_2$ is extended to heaps in the following way: the two heaps should be in relation for \approx , and for all pointers p such that $m(p) = \beta$, the associated values should be in relation with respect to t and β : $h_1(p) \equiv_{\beta, t} h_2(p)$. The relation is extended to environments naturally: $\rho_1, h_1 \equiv_{\Gamma, t} \rho_2, h_2$. The relation is extended to final configurations in a straightforward manner. The formal definition is given in Figure A.5.

We prove some preliminary lemmas.

Lemma A.3.2 (Stability of type interpretation). *For all partial maps t and t' such that $t \sqsubseteq t'$ the following properties hold:*

1. *For all v_1, v_2 , if $v_1 \equiv_{\beta, t} v_2$ then $v_1 \equiv_{\beta, t'} v_2$*

$$\begin{array}{c}
\text{RULE} \\
\frac{v_1 \equiv_{\beta,t} v_2 \quad h_1 \equiv_m h_2}{\rho_1, h_1 \equiv_{\Gamma,t} \rho_2, h_2}
\end{array}
\quad
\begin{array}{c}
\text{BOOL} \\
\frac{\ell = \text{PUB} \Rightarrow b_1 = b_2}{b_1 \equiv_{\text{BOOL}_{\ell,t}} b_2}
\end{array}
\quad
\begin{array}{c}
\text{INT} \\
\frac{\ell = \text{PUB} \Rightarrow i_1^s = i_2^s}{i_1^s \equiv_{\text{INT}_{\ell,t}^s} i_2^s}
\end{array}
\quad
\begin{array}{c}
\text{UINT} \\
\frac{\ell = \text{PUB} \Rightarrow i_1^s = i_2^s}{i_1^s \equiv_{\text{UINT}_{\ell,t}^s} i_2^s}
\end{array}$$

$$\begin{array}{c}
\text{REF} \\
\frac{m(p) = \beta' \quad \beta' \sqsubseteq \beta}{p \equiv_{\text{REF}_x[\beta],t} p}
\end{array}
\quad
\begin{array}{c}
\text{ARR} \\
\frac{v_i \equiv_{\beta,t} w_i}{[v_0; \dots; v_n] \equiv_{\text{ARR}[\beta,*],t} [w_0; \dots; w_n]}
\end{array}$$

$$\begin{array}{c}
\text{STRUCT} \\
\frac{v_i \equiv_{\beta_i,t} w_i}{\{x_1 = v_1, \dots, x_n = v_n\} \equiv_{\{x_1:\beta_1, \dots, x_n:\beta_n\},t} \{x_1 = w_1, \dots, x_n = w_n\}}
\end{array}$$

$$\begin{array}{c}
\text{HEAP} \\
\frac{\forall p \beta, m(p) = \beta \Rightarrow h_1(p) \equiv_{\beta,t} h_2(p)}{h_1 \equiv_t h_2}
\end{array}
\quad
\begin{array}{c}
\text{ENV} \\
\frac{\forall x, x \in \Gamma \Rightarrow \rho_1(x) \equiv_{\Gamma(x),t} \rho_2(x)}{\rho_1, h_1 \equiv_{\Gamma,t} \rho_2, h_2}
\end{array}$$

$$\begin{array}{c}
\text{NORET} \\
\frac{\rho_1, h_1 \equiv_{\Gamma,t} \rho_2, h_2}{\rho_1, h_1 \equiv_{\Gamma,\beta_r,t} \rho_2, h_2}
\end{array}
\quad
\begin{array}{c}
\text{RET} \\
\frac{v_1, h_1 \equiv_{\beta_r,t} v_2, h_2}{v_1, h_1 \equiv_{\Gamma,\beta_r,t} v_2, h_2}
\end{array}$$

Figure A.5: Type interpretation.

2. For all heaps $h_1 h_2 h'_1 h'_2$ such that $h'_1 \equiv_{t'} h'_2$, $\rho_1, h_1 \equiv_{\beta, t} \rho_2, h_2 \Rightarrow \rho_1, h'_1 \equiv_{\beta, t} \rho_2, h'_2$

Proof. We prove (1) by induction on $v_1, h_1 \equiv_{\beta, t} v_2, h_2$. The only interesting case is for REF, which follows directly from definitions of $t \sqsubseteq t'$. (2) is a direct consequence of (1). \square

Lemma A.3.3 (Reference creation). *If $h_1 \equiv_t h_2$ and*

FRESH(h_1, v_1) = (p_1, h'_1) and FRESH(h_2, v_2) = (p_2, h'_2) and $v_1 \equiv_{b, t} v_2$ then $p_1 = p_2$ and $h'_1 \equiv_{m[p_1 \leftarrow \beta]} h'_2$.

Proof. $h_1 \equiv_t h_2$ implies $h_1 \approx h_2$, so $p_1 = p_2$ and $h'_1 \approx h'_2$. It remains to prove $\forall p \beta', m[p_1 \leftarrow \beta](p) = \beta' \Rightarrow h'_1(p) \equiv_{\beta', t} h'_2(p)$. If $p = p_1$ then $m[p_1 \leftarrow \beta](p) = \beta$ and $h'_i(p) = v_i$ and we have $v_1 \equiv_{\beta, t} v_2$ by hypothesis. Else $p \neq p_1$ and $m[p_1 \leftarrow \beta](p) = m(p)$ and $h'_i(p) = h_i(p)$ and the property follows from $h_1 \equiv_t h_2$. \square

We now prove that typeable expressions and statements are constant-time.

Lemma A.3.4 (Typing constant-time, expressions).

$$\left. \begin{array}{l} \rho_1, h_1 \equiv_{\Gamma, t} \rho_2, h_2 \\ \Gamma \vdash e : \beta \\ (e, \rho_1, h_1) \xrightarrow{\psi_1} (v_1, h'_1) \\ (e, \rho_2, h_2) \xrightarrow{\psi_2} (v_2, h'_2) \end{array} \right\} \Rightarrow \exists t', \left\{ \begin{array}{l} t \sqsubseteq t' \\ h'_1 \equiv_{t'} h'_2 \\ \psi_1 = \psi_2 \\ v_1 \equiv_{\beta, t'} v_2 \end{array} \right.$$

Proof. By induction on $\Gamma \vdash e : \beta$. We do only the interesting cases:

► Case $e = e_1[e_2]$, we have

$$\begin{aligned}
(e_1, \rho_1, h_1) &\xrightarrow{\psi'_1} ([w_1; \dots; w_{k_1}], h''_1) \\
(e_1, \rho_2, h_2) &\xrightarrow{\psi'_2} ([w'_1; \dots; w'_{k_2}], h''_2) \\
(e_2, \rho_1, h'_1) &\xrightarrow{\psi''_1} (n_1, h'_1) \\
(e_2, \rho_2, h'_2) &\xrightarrow{\psi''_2} (n_2, h'_2) \\
v_1 = w_{n_1} &\quad v_2 = w'_{n_2} \\
\psi_1 &= \psi'_1 + \psi''_1 + \text{ARR}[n_1] \\
\psi_2 &= \psi'_2 + \psi''_2 + \text{ARR}[n_2] \\
\Gamma \vdash e_1 &: \text{ARR}[\beta, e_{len}] \\
\Gamma \vdash e_2 &: \text{UINT}^{\text{PUB}}
\end{aligned}$$

By induction hypothesis on e_1 there exists t' such that

$$\begin{aligned}
t \sqsubseteq t'' \quad h''_1 \equiv_{t''} h''_2 \quad \psi'_1 = \psi'_2 \\
[w_1; \dots; w_{k_1}] \equiv_{\text{ARR}[\beta, e_{len}], t''} [w'_1; \dots; w'_{k_2}]
\end{aligned}$$

By lemma A.3.2, we have $\rho_1, h''_1 \equiv_{\Gamma, t''} \rho_2, h''_2$ and we can apply the induction hypothesis on e_2 to get:

$$\begin{aligned}
t'' \sqsubseteq t' \quad h'_1 \equiv_{t'} h'_2 \quad \psi''_1 = \psi''_2 \\
n_1 \equiv_{\text{UINT}^{\text{PUB}}, t'} n_2
\end{aligned}$$

So $n_1 = n_2$ and by lemma A.3.2 we get

$$[w_1; \dots; w_{k_1}] \equiv_{\text{ARR}[\beta, e_{len}], t'} [w'_1; \dots; w'_{k_2}]$$

which allows to conclude $v_1 \equiv_{\beta, t'} v_2$. We conclude by using t' as witness.

► Case $e = \text{REF}[e']$, we have

$$\begin{aligned}
(e', \rho_1, h_1) &\xrightarrow{\psi'_1} (v'_1, h''_1) \\
(e', \rho_2, h_2) &\xrightarrow{\psi'_2} (v'_2, h''_2) \\
\text{FRESH}(h''_1, v'_1) &= (p_1, h'_1) \\
\text{FRESH}(h''_2, v'_2) &= (p_2, h'_2) \\
v_1 = p_1 \quad v_2 = p_2 \\
\Gamma \vdash e' : \beta' \quad \beta &= \text{REF}_{\text{RW}}[\beta']
\end{aligned}$$

By induction hypothesis on e' we get

$$\begin{aligned}
t \sqsubseteq t'' \quad h''_1 \equiv_{t''} h''_2 \quad \psi'_1 = \psi'_2 \\
v'_1 \equiv_{\beta', t''} v'_2
\end{aligned}$$

We can conclude the proof by using

$$t' = t''[p_1 \leftarrow \beta']$$

and use lemma A.3.3.

□

Lemma A.3.5 (Typing constant-time: statements).

$$\left. \begin{array}{l}
\rho_1, h_1 \equiv_{\Gamma, t} \rho_2, h_2 \\
\beta_r \vdash S : \Gamma \rightarrow \Gamma' \\
(S, \rho_1, h_1) \xrightarrow{\psi_1} (v_1, h'_1) \\
(S, \rho_2, h_2) \xrightarrow{\psi_2} (v_2, h'_2)
\end{array} \right\} \Rightarrow \exists t', \left\{ \begin{array}{l}
t \sqsubseteq t' \\
\psi_1 = \psi_2 \\
v_1, h_1 \equiv_{\Gamma, \beta_r, t} v_2, h_2
\end{array} \right.$$

Proof. By induction on $(S, \rho_1, h_1) \xrightarrow{\psi_1} (v_1, h'_1)$.

- ▶ Cases SEQ-RET, SEQ-NORET, and BLOCK are trivial.
- ▶ Cases VARDEC, ASSIGN, RETURN, IF and FN-CALL follow from lemmas A.3.4 and A.3.2.
- ▶ The last case, FOR, is almost a direct consequence of the induction hypothesis, the only difficulty being to prove that the statement is well-typed:

$$\beta_r \vdash \begin{array}{l} \text{if } *_{\ell} n_1 < n_2 \text{ then } \{i = n_1; S\}; \\ \text{for } *_{\ell} i = n_1 + 1 \text{ to } n_2 \text{ DO } S \} : \Gamma \rightarrow \Gamma \end{array}$$

□

Preservation of semantics. Finally, we prove that branch removal preserves the semantics of programs. The proof is performed in two steps. First, we show that if the value of the control predicate is `false` then the code does not modify the initial heap; it can only create fresh references.

Lemma A.3.6. *Let m a partial mapping on pointers. Assume that p is not trivially true (i.e., p is not the literal `true`) and $\Phi, p \vdash S \rightarrow S'$ and $h \simeq_m h'_1$ and $\rho \simeq_m \rho'_1$ and $\text{SEC}, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$ and $(S', \rho'_1, h'_1) \rightarrow (v', h'_2)$ (i.e., S' is safe). If $(p, \rho', h'_1) \rightarrow \text{false}$ then $h \simeq_m h'_2$ and there exists ρ'_2 such that $v' = \rho'_2$ and $\rho \simeq_m \rho'_2$.*

Furthermore, assume that p is not trivially true and $\omega \vdash F \rightarrow F'$ and $\omega(f) = \text{SEC}$ and $h \simeq_m h'_1$ and F is well typed and $(F', (\vec{v}', \text{false}), h'_1) \rightarrow (v', h'_2)$. Then $h \simeq_m h'_2$.

Proof. By mutual induction on S and F . The key point of the proof is to notice that if p is not trivially true then the pc used for type-checking is necessarily `SEC`, so there is no return statement in S' . □

We now prove that if the control predicate evaluates to `true` then the semantics of statements and functions are preserved.

Lemma A.3.7. *Let m be a partial mapping on pointers. Assume $\Phi, p \vdash S \rightarrow S'$ and $h_1 \simeq_m h'_1$ and $\rho_1 \simeq_m \rho'_1$ and $pc, \beta_r \vdash S : \Gamma \rightarrow \Gamma'$ and $pc = (\text{if } p = \text{true} \text{ then SEC else PUB})$ and $(S, \rho_1, h_1) \rightarrow (v, h_2)$ and $(S', \rho'_1, h'_1) \rightarrow (v', h'_2)$ (i.e., S' is safe). If $(p, \rho', h'_1) \rightarrow \text{true}$ then there exists m' such that $m \sqsubseteq m'$ and $h_2 \simeq_{m'} h'_2$ and $v \simeq_{m'} v'$.*

Furthermore, assume that $\omega \vdash F \rightarrow F'$ and F is well typed and $(F, \vec{v}, h_1) \rightarrow (v, h_2)$ and $h_1 \simeq_m h'_1$ and $\vec{v} \simeq_m \vec{v}'$. If $\omega(f) = \text{PUB}$ and $(F', \vec{v}', h'_1) \rightarrow (v', h'_2)$ then there exists m' such that $m \sqsubseteq m'$ and $h_2 \simeq_{m'} h'_2$ and $v \simeq_{m'} v'$. Else, if $\omega(f) = \text{SEC}$ and $(F', (\vec{v}', \text{true}), h'_1) \rightarrow (v', h'_2)$ then there exists m' such that $m \sqsubseteq m'$ and $h_2 \simeq_{m'} h'_2$ and $v \simeq_{m'} v'$.

Proof. By mutual induction on S and F . □

Appendix B

Pitchfork: Full proofs

B.1 Consistency

Lemma B.1.1 (Determinism). *If $C \xrightarrow[d]{o'} C'$ and $C \xrightarrow[d]{o''} C''$ then $C' = C''$ and $o' = o''$.*

Proof. The tuple (C, d) fully determines which rule of the semantics can be executed. □

Definition B.1.2 (Initial/terminal configuration). *A configuration C is an initial (or terminal) configuration if $|C.buf| = 0$.*

Definition B.1.3 (Sequential schedule). *Given a configuration C , we say a schedule D is sequential if every instruction that is fetched is executed and retired before further instructions are fetched.*

Definition B.1.4 (Sequential execution). *$C \Downarrow_D^N C'$ is a sequential execution if C is an initial configuration, D is a sequential schedule for C , and C' is a terminal configuration.*

We write $C \Downarrow_{seq}^N C'$ if we execute sequentially.

Lemma B.1.5 (Sequential equivalence). *If $C \Downarrow_{D_1}^N C_1$ is sequential and $C \Downarrow_{D_2}^N C_2$ is sequential, then $C_1 = C_2$.*

Proof. Suppose $N = 0$. Then neither D_1 nor D_2 may contain any retire directives. Since we assume that both $C_1.buf$ and $C_2.buf$ have size 0, neither D_1 nor D_2 may contain any fetch directives. Therefore, both D_1 and D_2 are empty; both C_1 and C_2 are equal to C .

We proceed by induction on N .

Let D'_1 be a sequential prefix of D_1 up to the $N - 1$ th retire, and let D''_1 be the remainder of D_1 . That is, $\#\{d \in D'_1 \mid d = \text{retire}\} = N - 1$ and $D'_1 \parallel D''_1 = D_1$. Let D'_2 and D''_2 be similarly defined.

By our induction hypothesis, we know $C \xrightarrow{O'_1 \downarrow_{D'_1}^{N-1}} C'$ and $C \xrightarrow{O'_2 \downarrow_{D'_2}^{N-1}} C'$ for some C' . Since D'_1 (resp. D'_2) is sequential and $|C'.buf| = 0$, the first directive in D''_1 (resp. D''_2) must be a fetch directive. Furthermore, $C' \xrightarrow{O''_1 \downarrow_{D''_1}^1} C_1$ and $C' \xrightarrow{O''_2 \downarrow_{D''_2}^1} C_2$.

We can now proceed by cases on $C'.\mu[C'.n]$, the final instruction to be fetched.

- ▶ For `op`, the only valid sequence of directives is (fetch, execute i , retire) where i is the sole valid index in the buffer. Similarly for `fence`, with the sequence {fetch, retire}.
- ▶ For `load`, alias prediction is not possible, as no prior stores exist in the buffer. Therefore, just as with `op`, the only valid sequence of directives is (fetch, execute i , retire).
- ▶ For `store`, the only possible difference between D''_1 and D''_2 is the ordering of the execute i : value and execute i : addr directives. However, both orderings will result in the same configuration since they independently resolve the components of the store.
- ▶ For `br`, D''_1 and D''_2 may have different guesses for their initial fetch directives. However, both COND-EXECUTE-CORRECT and COND-EXECUTE-INCORRECT will result in the same configuration regardless of the initial guess, as the `br` is the only instruction in the buffer. Similarly for `jmp`.
- ▶ For `call` and `ret`, the ordering of execution of the resulting transient instructions does not affect the final configuration.

Thus for all cases we have $C_1 = C_2$. □

To make our discussion easier, we will say that a directive d *applies to* a buffer index i if when executing a step $C \xrightarrow[d]{o} C'$:

- ▶ d is a fetch directive, and would fetch an instruction into index i in *buf*.
- ▶ d is an execute directive, and would execute the instruction at index i in *buf*.
- ▶ d is a retire directive, and would retire the instruction at index i in *buf*.

We would like to reason about schedules that do not contain *misspeculated steps*, i.e., directives that are superfluous due to their effects getting wiped away by rollbacks.

Definition B.1.6 (Misspeculated steps). *Given an execution $C \Downarrow_D^N C'$, we say that D contains misspeculated steps if there exists $d \in D$ such that $D' = D \setminus d$ and $C \Downarrow_{D'}^N C'' = C'$.*

Given an execution $C \Downarrow_D^N C'$ that may contain rollbacks, we can create an alternate schedule D^* without any rollbacks by removing all misspeculated steps. Note that sequential schedules have no misspeculated steps¹ as defined in Definition B.1.6.

Theorem B.1.7 (Equivalence to sequential execution). *Let C be an initial configuration and D a well-formed schedule for C . If $C \Downarrow_D^N C_1$, then $C \Downarrow_{seq}^N C_2$ and $C_1 \approx C_2$. Furthermore, if C_1 is terminal then $C_1 = C_2$.*

Proof. Since we can always remove all misspeculated steps from any well-formed execution without affecting the final configuration, we assume D_1 has no misspeculated steps.

Suppose $N = 0$. Then the theorem is trivially true. We proceed by induction on N .

Let D'_1 be the subsequence of D_1 containing the first $N - 1$ retire directives and the directives that apply to the same indices of the first $N - 1$ retire directives. Let D''_1 be the complement of D'_1 with respect to D_1 . All directives in D''_1 apply to indices later than any directive

¹Sequential schedules may still misspeculate on conditional branches but the rollback does not imply removal of any reorder buffer instructions as defined in Definition B.1.6.

in D'_1 , and thus cannot affect the execution of directives in D'_1 . Thus D'_1 is a well-formed schedule and produces execution $C \downarrow_{O'_1 D'_1}^{N-1} C'_1$.

Since D_1 contains no misspeculated steps, the directives in D'_1 can be reordered after the directives in D'_1 . Thus D''_1 is a well-formed schedule for C'_1 , producing execution $C'_1 \downarrow_{O''_1 D''_1} C''_1$ with $C''_1 \approx C_1$. If C_1 is terminal, then C''_1 is also terminal and $C''_1 = C_1$.

By our induction hypothesis, we know there exists D'_{seq} such that $C \downarrow_{O'_2 D'_{seq}}^{N-1} C'_2$. Since D'_1 contains equal numbers of fetch and retire directives, ends with a retire, and contains no misspeculated steps, C'_1 is terminal. Thus $C'_1 = C'_2$.

Let D''_{seq} be the subsequence of D''_1 containing the retire directive in D''_1 and the directives that apply to the same index. D''_{seq} is sequential with respect to C'_1 and produces execution $C'_1 \downarrow_{O''_2 D''_{seq}} C''_2$ with $C''_2 \approx C''_1 \approx C_1$. If C''_1 is terminal, then $D''_{seq} = D''_1$ and thus $C''_2 = C''_1 = C_1$.

Let $D_{seq} = D'_{seq} \parallel D''_{seq}$. D_{seq} is thus itself sequential and produces execution $C \downarrow_{(O'_2 \parallel O''_2) D_{seq}}^N C''_2$, completing our proof. \square

Corollary B.1.8 (General consistency). *Let C be an initial configuration. If $C \downarrow_{O_1 D_1}^N C_1$ and $C \downarrow_{O_2 D_2}^N C_2$, then $C_1 \approx C_2$. Furthermore, if C_1 and C_2 are both terminal then $C_1 = C_2$.*

Proof. By Theorem B.1.7, there exists D'_{seq} such that executing with C produces $C'_1 \approx C_1$ (resp. $C'_1 = C_1$). Similarly, there exists D''_{seq} that produces $C'_2 \approx C_2$ (resp. $C'_2 = C_2$). By Lemma B.1.5, we have $C'_1 = C'_2$. Thus $C_1 \approx C_2$ (resp. $C_1 = C_2$). \square

B.2 Security

Theorem B.2.1 (Label stability). *Let ℓ be a label in the lattice \mathcal{L} . If $C \downarrow_{O_1 D_1}^N C_1$ and $\forall o \in O_1 : \ell \notin o$, then $C \downarrow_{O_2 D_2}^N C_2$ and $\forall o \in O_2 : \ell \notin o$.*

Proof. Let D_1^* be the schedule given by removing all misspeculated steps from D_1 . The corresponding trace O_1^* is a subsequence of O_1 , and hence $\forall o \in O_1^* : \ell \notin o$. We thus proceed assuming that execution of D_1 contains no misspeculated steps.

Our proof closely follows that of Theorem B.1.7. When constructing D'_1 and D''_1 from D_1 in the inductive step, we know that all directives in D''_1 apply to indices later than any directive in D'_1 , and cannot affect execution of any directive in D'_1 . This implies that O'_1 is the subsequence of O_1 that corresponds to the mapping of D'_1 to D_1 .

Reordering the directives in D''_1 after D'_1 do not affect the observations produced by most directives. The exceptions to this are execute directives for load instructions that would have received a forwarded value: after reordering, the store instruction they forwarded from may have been retired, and they must fetch their value from memory. However, even in this case, the address a_{ℓ_a} attached to the observation does not change. Thus $\forall o \in O''_2 : \ell \notin o$.

Continuing the proof as in Theorem B.1.7, we create schedule D'_{seq} (with trace O'_2) from the induction hypothesis and D''_{seq} (with trace O''_2) as the subsequence of D'_1 of directives applying to the remaining instruction to be retired. As noted before, executing the subsequence of a schedule produces the corresponding subsequence of the original trace; hence $\forall o \in O''_2 : \ell \notin o$.

The trace of the final (sequential) schedule $D_{seq} = D'_{seq} \parallel D''_{seq}$ is $O'_2 \parallel O''_2$. Since O'_2 satisfies the label stability property via the induction hypothesis, we have $\forall o \in O'_2 \parallel O''_2 : \ell \notin o$.

□

By letting ℓ be the label secret, we get the following corollary:

Corollary B.2.2 (Secrecy). *If speculative execution of C under schedule D produces a trace O that contains no **secret** labels, then sequential execution of C will never produce a trace that contains any **secret** labels.*

With this, we can prove the following proposition:

Proposition B.2.3. *For a given initial configuration C and well-formed schedule D , if C is SCT with respect to D , and execution of C with D results in a terminal configuration C_1 , then C is also sequentially constant-time.*

Proof. Since C is SCT, we know that for all $C' \simeq_{\text{pub}} C$, we have $C \Downarrow_D^N C_1$ and $C' \Downarrow_D^N C'_1$ where $C_1 \simeq_{\text{pub}} C'_1$ and $O = O'$. By Theorem B.1.7, we know there exist sequential executions such that $C \Downarrow_{O_{seq}}^N C_2$ and $C' \Downarrow_{O'_{seq}}^N C'_2$. Note that the two sequential schedules need not be the same.

C_1 is terminal by hypothesis. Execution of C' uses the same schedule D , so C'_1 is also terminal. Since we have $C_1 = C_2$ and $C'_1 = C'_2$, we can lift $C_1 \simeq_{\text{pub}} C'_1$ to get $C_2 \simeq_{\text{pub}} C'_2$.

To prove the trace property $O_{seq} = O'_{seq}$, we note that if $O_{seq} \neq O'_{seq}$, then since $C_2 \simeq_{\text{pub}} C'_2$, it must be the case that there exists some $o \in O_{seq}$ such that $\text{secret} \in O_{seq}$. Since this is also true for O and O' , we know that there exist no observations in either O or O' that contain secret labels. By Corollary B.2.2, it follows that no secret labels appear in either O_{seq} or O'_{seq} , and thus $O_{seq} = O'_{seq}$. \square

B.3 Soundness of Pitchfork

Definition B.3.1 (Affecting an index). *We say a directive d affects an index i if:*

- ▶ *d is a fetch-type directive and would produce a new mapping in buf at index i .*
- ▶ *d is an execute-type directive and specifies index i directly (e.g., execute i).*
- ▶ *d is a retire directive and would cause the instruction at i in buf to be removed.*

Definition B.3.2 (Path function). *The function $\text{Path}(C, D)$ produces the sequence of branch choice (from fetching br instructions) and store-forwarding information (when executing load instructions) when executing D with initial configuration C . That is, for a schedule D without*

misspeculated steps:

$$\begin{aligned}
 \text{Path}(C, \emptyset) &= [] \\
 \text{Path}(C, D \| d) &= \begin{cases} \text{Path}(C, D); (i, b), & d = \text{fetch}: b \\ \text{Path}(C, D); (i, j), & d \text{ produces } v_\ell\{j, a\} \\ \text{Path}(C, D); (i, \perp), & d \text{ produces } v_\ell\{\perp, a\} \\ \text{Path}(C, D), & \text{otherwise} \end{cases}
 \end{aligned}$$

where d affects index i . If D has misspeculated steps, then $\text{Path}(C, D) = \text{Path}(C, D^*)$ where D^* is the subset of D with misspeculated steps removed. We write simply $\text{Path}(D)$ when C is obvious.

For the Lemmas B.3.3, B.3.5 and B.3.6, we start with the following shared assumptions:

- ▶ C is an initial configuration.
- ▶ D_1 and D_2 are nonempty schedules.
- ▶ $C_{D_1} \Downarrow_{O_1} C_1$ and $C_{D_2} \Downarrow_{O_2} C_2$.
- ▶ $\text{Path}(C, D_1) = \text{Path}(C, D_2)$.
- ▶ $D_1 = D'_1 \| d_1$ and $D_2 = D'_2 \| d_2$ and $d_1 = d_2$.
- ▶ d_1 and d_2 affect the same index i in the their respective reorder buffers.

Let o_1 (resp. o_2) be the observation produced during execution of d_1 (resp. d_2).

Lemma B.3.3 (Fetch). *If d_1 and d_2 are both fetch-type directives, then $C_1.n = C_2.n$ and $C_1.\text{buf}[i] = C_2.\text{buf}[i]$.*

Proof. Since fetches happen in-order, the index i of a given physical instruction along a control flow path is deterministic. Both D_1 and D_2 both have the same (control flow) path. Since by

hypothesis both d_1 and d_2 affect the same index i , d_1 and d_2 must necessarily both be fetching the same physical instruction. Furthermore, since $Path(D_1) = Path(D_2)$, if the fetched instruction is a br instruction, then both d_1 and d_2 must have made the same guess. The lemma statements all hold accordingly. \square

Corollary B.3.4. *If D_1^* and D_2^* are nonempty schedules such that $C_{D_1^*} \Downarrow C_1^*$ and $C_{D_2^*} \Downarrow C_2^*$ and $Path(C, D_1^*) = Path(C, D_2^*)$, then: For any $i \in C_1^*.buf$, if $i \in C_2^*.buf$, then both $C_1^*.buf[i]$ and $C_2^*.buf[i]$ were derived from the same physical instruction.*

Proof. Let D_1 be the prefix of D_1^* such that the final directive in D_1 is the latest fetch that affects i . Let D_2 be similarly defined w.r.t. D_2^* . Then by Lemma B.3.3, D_1 and D_2 both fetch the same physical instruction to index i . \square

Lemma B.3.5. *If d_1 and d_2 are both execute-type directives, then $C_1.buf[i] = C_2.buf[i]$ and $o_1 = o_2$.*

Proof. We proceed by full induction on the size of D_1 .

For the base case: if $|D_1| = 1$, then the lemma statements are trivial regardless of the directive d_1 .

We know from Corollary B.3.4 that since d_1 and d_2 both affect the same index i , the two transient instruction must be derived from the same physical instruction, and thus has the same register dependencies. For each register dependency r , if the register was calculated by a transient instruction at a prior index j , we can create prefixes $D_{1,j}$ and $D_{2,j}$ of D_1 and D_2 respectively that end at the execute directive that resolves r at buffer index j . By our induction hypothesis, both $D_{1,j}$ and $D_{2,j}$ calculate the same value v_ℓ for r .

We now proceed by cases on the transient instruction being executed.

Op, Store (value). Since all dependencies calculate the same values, both instructions calculate the same value.

Store (address). Both instructions calculate the same address. Since $Path(D_1) = Path(D_2)$, both schedules have the same pattern of store-forwarding behavior. Thus execution of d_1 causes a hazard if and only if d_2 causes a hazard.

Load. Both instructions calculate the same address, producing the same observations o_1 and o_2 . Since $Path(D_1) = Path(D_2)$, either d_1 and d_2 cause the values to be retrieved from the same prior stores, or they both load values from the same address in memory. By our induction hypothesis, these values will be the same, so both instructions will resolve to the same value.

Branch. Both instructions calculate the same branch condition, producing the same observations o_1 and o_2 . Since $Path(D_1) = Path(D_2)$, execution of d_1 causes a misspeculation hazard if and only if d_2 also causes misspeculation hazard. \square

Lemma B.3.6. *If d_1 and d_2 are both retire directives, then $o_1 = o_2$.*

Proof. From Lemmas B.3.3 and B.3.5 we know that for both d_1 and d_2 , the transient instructions to be retired are the same. Thus the produced observations o_1 and o_2 are also the same. \square

We now formally define the set of schedules examined by Pitchfork:

Definition B.3.7 (Tool schedules). *Given an initial configuration C and a speculative window size n , we define the set of tool schedules $D_T(n)$ recursively as follows: The empty schedule \emptyset is in $D_T(n)$. If $D_0 \in D_T(n)$ and $C_{D_0} \Downarrow C_0$ and $|C_0.buf| < n$, then based on the next instruction to be fetched (and where i is the index of the fetched instruction):*

- ▶ *op: $D_0 \parallel \text{fetch; execute } i \in D_T(n)$.*
- ▶ *load: $D_0 \parallel \text{fetch; execute } i \in D_T(n)$.*
- ▶ *store: $D_0 \parallel \text{fetch; execute } i : \text{value} \in D_T(n)$ and $D_0 \parallel \text{fetch; execute } i : \text{value; execute } i : \text{addr} \in D_T(n)$.*

- *br*: Let b be the “correct” path for the branch condition. Then $D_0 \parallel \text{fetch: } b; \text{execute } i \in D_T(n)$ and $D_0 \parallel \text{fetch: } \neg b \in D_T(n)$.

Otherwise, if $|C_0.\text{buf}| = n$, then we instead extend based on the oldest instruction in the reorder buffer. If the oldest instruction is a store with an unresolved address, and will not cause a hazard, then $D_0 \parallel \text{execute } i : \text{addr}; \text{retire} \in D_T(n)$. Otherwise, if the oldest instruction is fully resolved, then $D_0 \parallel \text{retire} \in D_T(n)$.

Proposition B.3.8 (Path coverage). *If D_1 is a well-formed schedule for C whose reorder buffer never grows beyond size n , then $\exists D_2 : \text{Path}(D_1) = \text{Path}(D_2) \wedge D_2 \in D_T(n)$.*

Proof. The proof stems directly from the definition of $D_T(n)$; at every branch, both branches are added to the set of schedules, and every load is able to “skip” any combination of prior stores. \square

Theorem B.3.9 (Soundness of tool). *If speculative execution of C under a schedule D with speculation bound n produces a trace O that contains at least one secret label, then there exists a schedule $D_t \in D_T(n)$ that produces a trace O_t that also contains at least one secret label.*

Proof. We can truncate D to a schedule D^* that ends at the first directive to produce a secret observation. By Proposition B.3.8 there exists a schedule $D_0 \in D_T(n)$ such that $\text{Path}(D_t) = \text{Path}(D^*)$. By following construction of tool schedules as given in Definition B.3.7, we can find a schedule $D_t \in D_T(n)$ that satisfies the preconditions for Lemma B.3.5. Then by that same lemma, D_t produces the same final observation as D^* , which contains a secret label. \square

Bibliography

- [1] Johan Agat. Transforming out timing leaks. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2000.
- [2] Sam Ainsworth and Timothy M Jones. MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *47th Annual International Symposium on Computer Architecture*. ACM/IEEE, 2020.
- [3] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *34th IEEE Symposium on Security and Privacy*. IEEE, 2013.
- [4] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and ARM multiprocessor machine code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, 2009.
- [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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.
- [6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In *Fast Software Encryption*. Springer, 2016.
- [7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium*. USENIX Association, 2016.
- [8] José Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 2013.
- [9] AMD. Security analysis of AMD predictive store forwarding. <https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf>, 2020.

- [10] Marc Andryscio, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *36th IEEE Symposium on Security and Privacy*. IEEE, 2015.
- [11] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 2015.
- [12] ARM. Straight-line speculation. <https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/downloads/straight-line-speculation>, 2020.
- [13] Arm Mbed. mbed TLS. <https://github.com/armmbed/mbedtls>, 2018.
- [14] Jean-Philippe Aumasson and Yolán Romainier. Automated testing of crypto software using differential fuzzing. *Black Hat USA*, 2017.
- [15] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.
- [16] 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 *IEEE S&P*, 2021.
- [17] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”. In *Computer Security Foundations Symposium*, 2018.
- [18] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 2006.
- [19] Lennart Beringer, Adam Petcher, Q. Ye Katherine, and Andrew W. Appel. Verified correctness and security of openssl hmac. In *24th USENIX Security Symposium*, 2015.
- [20] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005. <https://cr.yp.to/antiforgery/cachetiming-20050414.pdf>.
- [21] Daniel J. Bernstein. The Poly1305-AES message-authentication code. In *Fast Software Encryption*. IACR, 2005.
- [22] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*. Springer, 2006.
- [23] Daniel J. Bernstein. qhasm: Tools to help write high-speed software. <https://cr.yp.to/qhasm.html>, 2007.
- [24] Daniel J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*. Springer, 2008.

- [25] Daniel J. Bernstein. Cryptography in NaCl. Technical report, 2009. <http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>.
- [26] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012.
- [27] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013.
- [28] Atri Bhattacharyya, Andrés Sánchez, Esmail M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. SpecROP: Speculative exploitation of ROP chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.
- [29] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMOtherSpectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
- [30] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium*. USENIX Association, 2017.
- [31] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. <https://arxiv.org/abs/1506.00189>, 2015.
- [32] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.
- [33] 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019.
- [34] 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.
- [35] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium*. USENIX Association, 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 *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
- [37] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Towards constant-time foundations for the new Spectre era. <https://arxiv.org/pdf/1910.01755v2.pdf>, 2019.
- [38] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannismeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: A flexible, constant-time programming language. In *Secure Development Conference*. IEEE, 2017.
- [39] Sunjay Cauligi, Gary Soeller, Brian Johannismeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. https://fact.programming.systems/FaCT_extended.pdf, 2019.
- [40] Sunjay Cauligi, Gary Soeller, Brian Johannismeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Gregoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019.
- [41] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium*, 2019.
- [42] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *5th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1992.
- [43] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 1998.
- [44] Robert J Colvin and Kirsten Winter. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In *International Symposium on Formal Methods*, 2019.
- [45] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy*,. IEEE, 2009.
- [46] Cryptography Coding Standard. Coding rules. https://cryptocoding.net/index.php/Coding_rules, 2016.

- [47] Lesly-Ann Daniel, Sebastian Bardin, and Tamara Rezk. Hunting the haunter — efficient relational symbolic execution for Spectre with Haunted RelSE. In *Network and Distributed Systems Security Symposium 2021*. Internet Society, 2021.
- [48] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *41st IEEE Symposium on Security and Privacy*. IEEE, 2020.
- [49] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [50] Frank Denis. libsodium. <https://github.com/jedisct1/libsodium>, 2019.
- [51] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. The code that never ran: Modeling attacks on speculative evaluation. In *40th IEEE Symposium on Security and Privacy*. IEEE, 2019.
- [52] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, 2015.
- [53] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Systematic generation of fast elliptic curve cryptography implementations. Technical report, 2018. <https://people.csail.mit.edu/jgross/personal-website/papers/2018-fiat-crypto-pldi-draft.pdf>.
- [54] Dmitry Evtushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018.
- [55] 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 *57th ACM/IEEE Design Automation Conference*. ACM/IEEE, 2020.
- [56] Matt Fleming. A thorough introduction to eBPF. <https://lwn.net/Articles/740157/>, 2017.
- [57] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An efficient data-centric defense mechanism against Spectre attacks. In *56th ACM/IEEE Design Automation Conference*, 2019.
- [58] GCC Team. Using the gnu compiler collection (gcc): Instrumentation options. <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>, 2019.
- [59] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2018.

- [60] Jay L Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 1988.
- [61] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [62] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium*, 2018.
- [63] Roberto Guanciale, Musard Balliu, and Mads Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *CCS*, 2020.
- [64] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. In *41st IEEE Symposium on Security and Privacy*. IEEE, 2020.
- [65] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *42nd IEEE Symposium on Security and Privacy*. IEEE, 2021.
- [66] 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 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.
- [67] 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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.
- [68] Pat Hickey. Announcing Lucet: Fastly’s native WebAssembly compiler and runtime. <https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime>, 2019.
- [69] Jann Horn. Speculative execution, variant 4: speculative store bypass. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1528>, 2018.
- [70] Intel. Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115. <https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass>, 2018.
- [71] Intel. Deep dive: Intel analysis of microarchitectural data sampling. <https://software.intel.com/security-software-guidance/software-guidance/microarchitectural-data-sampling>, 2019.
- [72] Intel. An Optimized Mitigation Approach for Load Value Injection. <https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injection>, 2020.

- [73] 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.
- [74] Intel 64 and IA-32 architectures software developer’s manual, 2021.
- [75] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *28th USENIX Security Symposium*. USENIX Association, 2019.
- [76] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. BranchSpec: Information leakage attacks exploiting speculative branch instruction executions. In *38th International Conference on Computer Design*. IEEE, 2020.
- [77] Jann Horn. Reading privileged memory with a side-channel. <https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>, 2018.
- [78] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J. Peter Brady, Sergey Bratus, and Sean W. Smith. Ghostbusting: Mitigating spectre with intraprocess memory isolation. In *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*. ACM, 2020.
- [79] Burt Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, 1998.
- [80] Khaled N Khasawneh, Esmail Mohammadian Koruyeh, Chengyu Song, Dmitry Evtushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *56th ACM/IEEE Design Automation Conference*. ACM/IEEE, 2019.
- [81] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2018.
- [82] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. <https://arxiv.org/pdf/1807.03757.pdf>, 2018.
- [83] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *30th USENIX Security Symposium*, 2021.
- [84] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*. Springer, 1996.
- [85] Paul Kocher. Spectre mitigations in Microsoft’s C/C++ compiler. <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>, 2018.
- [86] 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 *40th IEEE Symposium on Security and Privacy*. IEEE, 2019.

- [87] Esmail Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, 2018.
- [88] Esmail Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCEFI: Mitigating Spectre attacks using CFI informed speculation. In *41st IEEE Symposium on Security and Privacy*, 2020.
- [89] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors in uclid. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 2002.
- [90] Adam Langley. curve25519-donna. <https://github.com/agl/curve25519-donna>.
- [91] Adam Langley. ImperialViolet - lucky thirteen attack on tls cbc. <https://www.imperialviolet.org/2013/02/04/luckythirteen.html>, 2013.
- [92] 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 *IEEE International Symposium on High Performance Computer Architecture*, 2019.
- [93] 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 *27th USENIX Security Symposium*. USENIX Association, 2018.
- [94] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *IEEE 26th Computer Security Foundations Symposium*. IEEE, 2013.
- [95] 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 *30th USENIX Security Symposium*, 2021.
- [96] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy*, 2010.
- [97] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.
- [98] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.
- [99] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. <https://arxiv.org/pdf/2003.05503.pdf>, 2020.

- [100] 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. <https://arxiv.org/pdf/1902.05178>, 2019.
- [101] Tyler McMullen. Lucet: A compiler and runtime for high-concurrency low-latency sandboxing. *Principles of Secure Compilation*, 2020.
- [102] Microsoft. Spectre mitigations in MSVC. <https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/>, 2018.
- [103] John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium*. IEEE, 2012.
- [104] Bodo Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. <https://www.openssl.org/~bodo/tls-cbc.txt>, 2004.
- [105] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 2019.
- [106] Daniel Moghimi. Data sampling on MDS-resistant 10th Generation Intel Core (Ice Lake). *arXiv:2007.07428*, 2020.
- [107] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium*, 2020.
- [108] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology*. Springer, 2006.
- [109] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1999.
- [110] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow, 2006. <http://www.cs.cornell.edu/jif>.
- [111] 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 *30th USENIX Security Symposium*, 2021.
- [112] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *38th IEEE Symposium on Security and Privacy*. IEEE, 2017.

- [113] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium*, 2020.
- [114] The OpenSSL Project. OpenSSL. <https://github.com/openssl/openssl>.
- [115] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 2006.
- [116] Marco Patrignani and Marco Guarnieri. Exorcising Spectres with secure compilers. <https://arxiv.org/pdf/1910.08607>, 2020.
- [117] Jérémy Planul and John C. Mitchell. Oblivious program execution and path-sensitive non-interference. In *26th IEEE Computer Security Foundations Symposium*. IEEE, 2013.
- [118] Thomas Pornin. Why constant-time crypto? <https://www.bearssl.org/constanttime.html>, 2016.
- [119] Thomas Pornin. Constant-time toolkit. <https://github.com/pornin/CTTK>, 2018.
- [120] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramanandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hrițcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 2017.
- [121] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering Spectre gadgets. In *Network and Distributed Systems Security Symposium 2021*, 2021.
- [122] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*. USENIX Association, 2015.
- [123] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium*, 2019.
- [124] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μ ops: Leaking secrets via Intel/AMD micro-op caches. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2021.
- [125] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *2017 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2017.
- [126] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

- [127] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *25th International Conference on Compiler Construction*. ACM, 2016.
- [128] Stephen Röttger and Artur Janc. A Spectre proof-of-concept for a Spectre-proof web. <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>, 2021.
- [129] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
- [130] Gururaj Saileshwar and Moinuddin K Qureshi. CleanupSpec: An “undo” approach to safe speculation. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.
- [131] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. <https://arxiv.org/pdf/1905.05725>, 2019.
- [132] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTeXT: A generic approach for mitigating Spectre. In *NDSS*, 2020.
- [133] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019.
- [134] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, 2019.
- [135] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. Specfuscator: Evaluating branch removal as a Spectre mitigation. In *Financial Cryptography and Data Security*, 2021.
- [136] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.
- [137] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with Venkman. <https://arxiv.org/pdf/1903.10651>, 2019.
- [138] 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 *37th IEEE Symposium on Security and Privacy*. IEEE, 2016.

- [139] Laurent Simon, David Chisnall, and Ross J. Anderson. What you get is what you C: controlling side effects in mainstream C compilers. In *3rd IEEE European Symposium on Security and Privacy*. IEEE, 2018.
- [140] 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.
- [141] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security*. Springer, 2013.
- [142] Marius Sternberger. Spectre-ng: An avalanche of attacks. In *Wiesbaden Workshop on Advanced Microkernel Operating Systems*, 2018.
- [143] Josef Svenningsson and David Sands. Specification and verification of side channel declassification. In *International Workshop on Formal Aspects in Security and Trust*. Springer, 2009.
- [144] Gang Tan. *Principles and Implementation Techniques of Software-Based Fault Isolation*. Now Publishers Inc., 2017.
- [145] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
- [146] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 2010.
- [147] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.
- [148] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. <https://support.google.com/faqs/answer/7625886>, 2019.
- [149] 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 *27th USENIX Security Symposium*, 2018.
- [150] 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 *41st IEEE Symposium on Security and Privacy*, 2020.

- [151] 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 *Proceedings of the ACM on Programming Languages*, 2021.
- [152] Serge Vaudenay. Security flaws induced by CBC padding — applications to SSL, IPSEC, WTLS. . . . In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2002.
- [153] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. BRB: Mitigating branch predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture*, 2019.
- [154] 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*, 2020.
- [155] 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.
- [156] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 2019.
- [157] WebAssembly Community Group. Webassembly. <http://webassembly.org>, 2018.
- [158] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.
- [159] Henry Wong. Store-to-load forwarding and memory disambiguation in x86 processors. <https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/>, 2014.
- [160] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018.
- [161] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In *40th SIGPLAN ACM Conference on Programming Language Design and Implementation*. ACM, 2019.
- [162] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 2021.

- [163] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.
- [164] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on openssl constant-time RSA. *Journal of Cryptographic Engineering*, 2017.
- [165] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.
- [166] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, 2009.
- [167] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.
- [168] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. *ACM SIGPLAN Notices*, 2015.
- [169] Tao Zhang, Kenneth Koltermann, and Dmitry Evtvushkin. Exploring branch predictors for constructing transient execution trojans. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.
- [170] 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. <https://arxiv.org/pdf/2005.08183>, 2020.
- [171] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.
- [172] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACl*: a verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.