UNIVERSITY OF CALIFORNIA SAN DIEGO

Security at the Boundary: Formally Securing Transitions in Component Isolation

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

in

Computer Science

by

Matthew Kolosick

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel Buss
Professor Nadia Polikarpova
Professor Deian Stefan

2025

Copyright

Matthew Kolosick, 2025

All rights reserved.

The Dissertation of Matthew Kolosick is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2025

DEDICATION

This thesis is dedicated to all of those who have supported and cared for me throughout grad school: every one of you is a part of this work.

To the UCSD ProgSys group, for the support and positive energy.

To the third floor of CSE, for being the social heart of my grad school experience.

To Michael, whose friendship has ranged all over the map.

To Yousef and Emily: we all owe you a better world.

To Andi and Val, for reminding me to look at the world in awe.

To Anish, my first co-author. I couldn't have done it without you.

To Elizabeth, for letting a near stranger sleep on your couch post surgery and for always staying optimistic with me.

To Ariana, who taught me how to voluntell.

To Blanca, Danny, and Kokila. Thank you for bringing me back to dance.

To all of the cats who have wandered into my life but especially Tuna, Cashew, and Nico.

To my family, for their lifelong support.

To David and Evan, for getting me into rock climbing.

To John, whose energy is infectious.

To Kyle and Mia, for their bi-coastal friendship and support.

To the orthopedic surgeons, who put me back together.

To Tristan, for bringing me along to explore the waves.

To Amelia, for her effective recruitment efforts.

To Jake, Allison, Emily, and Tom, who always reminded me that you don't have to have a plan.

To Nick, who has been on this PL journey with me from the very start.

To all of my friends from undergrad. I think I have slept on all of your couches and know that you wouldn't hesitate to take me in again.

To every member of UAW 4811: solidarity forever.

To Danielle, who continues to refuse to let me go uncelebrated.

EPIGRAPH

"Designated trail ends here"
I read
and go no further.

TABLE OF CONTENTS

Dissertation Approval Page . . . iii

Dedication . . . iv

Epigraph . . . vi

Table of Contents . . . vii

List of Figures . . . ix

List of Tables . . . xii

Acknowledgements . . . xiii

Vita . . . xiv

Abstract of the Dissertation . . . xv

Introduction . . . 1
- 0.1 Software Fault Isolation . . . 3
- 0.2 Constant Time . . . 5
  - 0.2.1 Speculative Constant Time . . . 6

Chapter 1 Near Zero Cost Transitions . . . 9
- 1.1 Overview . . . 13
  - 1.1.1 The Need for Secure Transitions . . . 13
  - 1.1.2 Heavyweight Transitions . . . 15
  - 1.1.3 Zero-Cost Transitions . . . 16
- 1.2 A Gated Assembly Language . . . 19
  - 1.2.1 Secure Transitions . . . 25
  - 1.2.2 Security of Heavyweight Transitions . . . 31
- 1.3 Zero-Cost Transition Conditions . . . 36
  - 1.3.1 Overlay Monitor . . . 40
  - 1.3.2 Overlay Semantics Enforce Security . . . 43
- 1.4 Instantiating Zero-Cost . . . 44
  - 1.4.1 WebAssembly . . . 45
  - 1.4.2 SegmentZero32 . . . 46
- 1.5 Verifying Compiled WebAssembly . . . 48
  - 1.5.1 The VeriZero Analyzers . . . 49
  - 1.5.2 The Dataflow Abstract Domain . . . 50
  - 1.5.3 Checking the Zero-Cost Conditions . . . 51
  - 1.5.4 Proving WebAssembly Secure . . . 52
- 1.6 Evaluation . . . 56

1.6.1 The Cost of Transitions 60
1.6.2 End-to-End Performance Improvements of Zero-Cost Transitions for WebAssembly 61
1.6.3 Performance Overhead of Purpose-Built Zero-Cost SFI Enforcement 66
1.6.4 Effectiveness of the VeriZero Verifier 69
1.7 Limitations 71
1.8 Related work 72
1.9 Acknowledgments 76

Chapter 2 Robust Constant Time 77
2.1 Security Semantics 83
2.1.1 Non-Speculative Trace Semantics 83
2.1.2 Speculative Semantics 89
2.1.3 Concurrent Observer Semantics 94
2.1.4 Modeling Spectre Attacks 95
2.2 Robust Constant Time 104
2.2.1 Programs and Traces 104
2.2.2 Robust Constant Time 107
2.2.3 Attackers 109
2.3 A Robust Compiler 113
2.3.1 Making Libraries Robust 113
2.3.2 Proving RoboCop Secure 116
2.4 Evaluation 119
2.4.1 Read-Only and Speculative Attackers 121
2.4.2 Concurrent Attackers 127
2.5 Limitations 129
2.6 Related Work 130
2.7 Acknowledgments 133

Appendix A Proofs 134
A.1 Proofs of Security for Heavyweight transitions 134
A.2 Proofs of Security for Zero-Cost WebAssembly 142
A.3 Formal Definitions of RoboCop Compilers 144
A.4 Proofs of Security for RoboCop Compilers 147
A.4.1 Read-only Protections 147
A.4.2 Speculative Protections 153
A.4.3 Concurrent Protections 166

LIST OF FIGURES

Figure 1.1. Syntax of SFIasm. 20

Figure 1.2. Operational semantics for SFIasm. 23

Figure 1.3. Operational semantics for SFIasm: errors. 24

Figure 1.4. Operational semantics for SFIasm: auxiliary definitions. 24

Figure 1.5. Well-bracketed transitions in SFIasm. 25

Figure 1.6. Call stack return address calculation. 27

Figure 1.7. Layout of contexts for heavyweight transitions. 31

Figure 1.8. Heavyweight springboards. 32

Figure 1.9. Heavyweight trampolines. 33

Figure 1.10. Heavyweight callback springboards. 33

Figure 1.11. Heavyweight callback trampoline. 34

Figure 1.12. Extended syntax for overlay semantics of oSFIasm. 37

Figure 1.13. Operational semantics for oSFIasm. 38

Figure 1.14. Operational semantics for oSFIasm, continued. 39

Figure 1.15. Operational semantics for oSFIasm: auxiliary predicates. 40

Figure 1.16. Operational semantics for oSFIasm: auxiliary definitions 41

Figure 1.17. Disassembled and lifted WebAssembly functions. 49

Figure 1.18. Structure of WebAssembly libraries. 52

Figure 1.19. Layout of WebAssembly stack frame. 53

Figure 1.20. Definition of Kripke worlds for WebAssembly logical relation. 54

Figure 1.21. Logical relation for zero-cost WebAssembly. 55

Figure 1.22. Performance of different WebAssembly transitions on rendering of (a) a simple image with one color, (b) a stock image, and (c) a complex image with random pixels, normalized to `WasmZero`. `WasmZero` transitions outperform other transitions. The difference diminishes with width, but narrower images are more common on the web. . . 62

Figure 1.23. Performance of the `WasmLucet` heavyweight transitions included in the Lucet runtime on the image benchmarks. Performance when rendering: 1. a simple image with one color, 2. a stock image, and 3. a complex image with random pixels. The performance is the overhead compared to `WasmZero`. 63

Figure 1.24. Performance of an ideal isolation scheme (no enforcement overhead) with heavy trampolines when rendering images. WebAssembly compilers whose enforcement overhead is lower than this can outperform even an ideal isolation scheme that uses heavy weight transitions. 63

Figure 1.25. Cumulative distribution of image widths on the landing pages of the Alexa top 500 websites. Over 80% of the images have widths under 480 pixels. Narrower images have a higher transition rate, and thus higher relative overheads when using expensive transitions. 65

Figure 1.26. Performance of image rendering with libjpeg sandboxed with `SegmentZero32` and `NaCl32` and `IdealHeavy32`. Times are relative to unsandboxed code. `NaCl32` and `IdealHeavy32` relative overheads are as high as 312% and 208% respectively, while `SegmentZero32` relative overheads do not exceed 24%. 67

Figure 2.1. An excerpt from the reference implementation of Salsa20 in LibSodium. 78

Figure 2.2. Syntax of λ_{spec}. 84

Figure 2.3. Syntax of events in λ_{spec}. 85

Figure 2.4. Non-speculative operational semantics for λ_{spec}: transition reductions. 87

Figure 2.5. Non-speculative operational semantics for λ_{spec}: domain reductions. 88

Figure 2.6. Speculative operational semantics for λ_{spec}. 90

Figure 2.7. Speculative operational semantics for λ_{spec}: fencing. 90

Figure 2.8. Speculative operational semantics for λ_{spec}: auxiliary definitions. . . 91

Figure 2.9. Speculative operational semantics for λ_{spec}: auxiliary definitions, continued. 92

Figure 2.10. Concurrent observer semantics for λ_{spec}. 95

Figure 2.11. Syntax of programs and traces for λ_{spec}. 104

Figure 2.12. Well-formedness of λ_{spec} programs. 105

Figure 2.13. Substitution in λ_{spec}. 106

Figure 2.14. Definition of traces for λ_{spec}. 106

Figure 2.15. Constant time events for λ_{spec}. 108

Figure 2.16. Definition of robust constant time (RCT) attackers. 110

Figure 2.17. Well-formedness of traces for robust constant time (RCT) attackers. 111

Figure 2.18. RoboCop protections. 114

Figure 2.19. Initial well-formedness of states with protected memory. 118

Figure A.1. Semantic interpretation of non-speculative traces. 150

LIST OF TABLES

Table 1.1. Costs of transitions in different isolation models. Zero-cost transitions are shown in **boldface**. `Vanilla` is the performance of an unsandboxed C function call, to serve as a baseline. 60

Table 1.2. Costs of font rendering in different isolation models. Zero-cost transitions are shown in **boldface**. `Vanilla` is the performance of an unsandboxed C function call, to serve as a baseline. 64

Table 1.3. Overheads compared to native code on SPEC CPU® 2006 (nc), for `NaCl32` and `SegmentZero32`. 68

Table 1.4. Overheads compared to native code on font rendering for `NaCl32` and `SegmentZero32`. 68

Table 2.1. Overheads for read-only and speculative protections vs. unprotected baseline. N is the number of algorithms in the dataset, Size is the size of the operation's input in bytes, Q_1 and Q_3 are the first and third quartile overheads, and the median overhead is of the overheads of the mean runtimes. 122

Table 2.2. Read-only and speculative protection overheads for **stream** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline. 123

Table 2.3. Read-only and speculative protection overheads for **aead** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline. 124

Table 2.4. Read-only and speculative protection overheads for **encrypt** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline. 125

Table 2.5. Read-only and speculative protection overheads for **sign** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline. 126

Table 2.6. Overhead of read-only and speculative protections vs. overhead with concurrent protections. 128

ACKNOWLEDGEMENTS

I would like to thank my advisors: Ranjit Jhala, whose editing has made me the writer that I am today. Deian Stefan, who has been an unending source of ideas. I am immensely thankful for the role both have played in these past years. I would not be the same researcher without them.

Thanks also to my undergraduate advisor, Amal Ahmed, who brought me into the world of programming languages and is directly responsible for the use and abuse of logical relations throughout this dissertation.

Thanks to my committee members, Nadia Polikarpova and Samuel Buss, for putting up with my scheduling and asking insightful questions.

Thank you to my co-authors, this work would be nothing without you.

Thanks also to all of the reviewers who provided valuable feedback along the way, my work was made better by it.

Chapter 1, in full, is adapted from material as it appears in "Isolation without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI." Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Proc. ACM Program. Lang. 6, POPL, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in full, is adapted from material as it appears in "Robust Constant-Time Cryptography." Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. Proc. ACM Program. Lang. 9, PLDI, 2025. The dissertation author was the primary investigator and author of this paper.

VITA

2018	Bachelor of Science, Northeastern University
2018–2025	Teaching Assistant, Department of Computer Science University of California San Diego
2018–2025	Research Assistant, Department of Computer Science University of California San Diego
2025	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

"Isolation without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI." Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Proc. ACM Program. Lang. 6, POPL, 2022.

"Robust Constant-Time Cryptography." Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. Proc. ACM Program. Lang. 9, PLDI, 2025.

"Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types." Anish Tondwalkar, Matthew Kolosick, and Ranjit Jhala. Leibniz International Proceedings in Informatics 194, ECOOP, 2021.

ABSTRACT OF THE DISSERTATION

Security at the Boundary: Formally Securing Transitions in Component Isolation

by

Matthew Kolosick

Doctor of Philosophy in Computer Science

University of California San Diego, 2025

Professor Ranjit Jhala, Chair

There has been significant work on defining and guaranteeing security for the components that make up software systems, in particular software fault isolation for untrusted libraries and constant time for cryptographic libraries. But these components exist in the context of a larger application, and there has been comparatively little work on what happens at the boundary between library and application. In this dissertation we examine security in two settings: trusted applications interacting with untrusted third-party libraries and untrusted applications interacting with trusted cryptographic libraries. For each we examine the boundary between application and library, formally defining the security properties required of the transitions between the components. In

the case of untrusted third-party libraries we develop a set of *zero-cost conditions* that capture sufficient structure to ensure a sandboxed library can securely and efficiently context switch to and from the trusted application. In the case of cryptographic libraries we define a notion of *robust constant time*, characterizing security for cryptographic libraries used in a (potentially unsafe) application. We use our formal security properties to identify and develop compilers to automatically provide security for the overall application and *prove* that the compilers do indeed guarantee security.

Introduction

Software systems are commonly composed of components written by different developers: for instance, web browsers use third-party libraries to handle media processing and application developers use cryptographic libraries instead of attempting to write cryptographic algorithms themselves. This leads to a mix of assumptions and security requirements for the varying components within a software system, yet they must all be brought together to make one, hopefully secure, program. Throughout this dissertation we will separate these components into two categories based on their role within a software system and their origin within the social process of software development. The first, *applications*, are the components that make up the bulk of a program and dictate the overall behavior of a program: web browsers will be a running example throughout the dissertation. The second, *libraries*, are typically smaller, special purpose components, included by an application developer to accomplish a specific task. While an application developer might split their work into separate sub-components, we will use the categorization *library* specifically to refer to *third-party* libraries, that is those developed separately (either in terms of developers or development process) from the application in which they are included.

We take the stance that understanding the *semantics* of software systems is key to ensuring their security, and approach this understanding from the perspective of formal semantics. With our categorization of components there are then three semantic aspects we must consider:

1. the security assumptions, requirements, and behavior of application components;
2. the security assumptions, requirements, and behavior of library components; and
3. the behavior at the boundary between application and library.

The first and second are well-studied and will be discussed in Sections 0.1 and 0.2. The third has been relatively neglected and is the focus of this dissertation. We argue that understanding the interactions of the semantics of applications and libraries at their boundary can be used to guarantee efficient, whole system security.

In Chapter 1: Near Zero Cost Transitions we first examine the setting of untrusted libraries interacting with a trusted application. A lightweight approach to security with such untrusted components is software sandboxing also known as software-based fault isolation or software fault isolation (SFI). While there have been significant efforts to optimize and verify SFI enforcement,[1] context switching in SFI systems remains largely unexplored: almost all SFI systems use *heavyweight transitions* that are not only error-prone but incur significant performance overhead from saving, clearing, and restoring registers when context switching. We build an operational semantics capturing the interaction of application and library at this boundary, which we deploy to characterize the boundary security conditions. From there we identify a set of *zero-cost conditions* that characterize when sandboxed code has sufficient structured to guarantee security via lightweight *zero-cost* transitions (that is, simple function calls). We show how to implement a zero-cost SFI system using WebAssembly, develop a static binary verifier to check that produced code satisfies the zero-cost conditions, and develop a proof of security. We then evaluate the costs of heavyweight and zero-cost conditions.

Where Chapter 1 covers a trusted application interacting with untrusted libraries, in Chapter 2: Robust Constant Time we examine the mirror image with trusted cryptographic libraries interacting with an untrusted application. Here, the property

[1]See Section 0.1.

of constant time (CT) serves as the standard notion of security for the cryptographic library, capturing when cryptographic code is protected against side-channels.[2] But cryptographic *library* developers take care to ensure their library does not leak secrets even when there are (inevitably) exploitable vulnerabilities in the *applications* the library is linked against. To do so, they choose some class of application vulnerabilities to defend against and hardcode protections against those vulnerabilities in the library code. A single set of choices is a poor fit for all contexts: a chosen protection could impose unnecessary overheads in contexts where those attacks are impossible, and an ignored protection could render the library insecure in contexts where the attack is feasible. We present an operational semantics that describes the behavior of a cryptographic library executing in the context of a potentially vulnerable application, allowing us to precisely specify what different attackers can observe. From there we use our semantics to define a novel security property, robust constant time (RCT), that defines when a cryptographic library is secure *in the context of* a vulnerable application. This definition is parameterized by an attacker model, allowing us to factor out the classes of attackers that a library may wish to secure against. With this parameterization we develop a compiler that can synthesize bespoke cryptographic libraries with security tailored to the specific application context against which a cryptographic library will be linked, guaranteeing that the library is RCT in that context. We prove the security of the parameterized compiler and evaluate its overhead.

0.1 Software Fault Isolation

Memory safety bugs are the single largest source of critical vulnerabilities in modern software. Recent studies found that roughly 70% of all critical vulnerabilities were caused by memory safety bugs [Chromium Team 2020; M. Miller 2019] and that malicious attackers are exploiting these bugs before they can be patched [Google

[2] See Section 0.2.

Project Zero 2021; Metrick et al. 2020]. As such, they form the core of the security threat from untrusted library components. To address this, toolkits for software fault isolation (SFI), such as Native Client (NaCl) [Yee et al. 2009] and WebAssembly [W3C 2019], promise to reduce the danger of these vulnerabilities by isolating untrusted components to their own *sandboxed* regions of memory, thus limiting the damage that can be caused by memory safety bugs in these components [Tan 2017; Wahbe et al. 1993]. Mozilla, for example, uses WebAssembly to sandbox third-party C libraries in the Firefox browser [Froyd 2020; Narayan, Disselkoen, Garfinkel, et al. 2020]: SFI allows the browser to use libraries like `libgraphite` (font rendering), `libexpat` (XML parsing), `libsoundtouch` (audio processing), and `hunspell` (spell checking) without risking whole-browser compromise due to library vulnerabilities. Others have used SFI to isolate code in OS kernels [Castro et al. 2009; Erlingsson et al. 2006; Herder et al. 2009; Seltzer et al. 1996], databases [Ford 2005; Ford and Cox 2008; Wahbe et al. 1993], browsers [Haas et al. 2017; Lucco et al. 1995; Yee et al. 2009], language runtimes [Niu and Tan 2014; Siefers et al. 2010], and serverless clouds [Gadepalli et al. 2020; McMullen 2020; Varda 2018].

SFI toolkits enforce this sandboxing by 1. allocating a separate memory region for the isolated component, 2. restricting memory accesses to remain within the separate memory region, and 3. restricting control flow to guarantee the execution remains *safely* within the component. Typically this is accomplished by re-compiling the untrusted code with the addition of dynamic runtime protections, with the compilation starting from either source or from existing assembly code.

There is a large body of work on ensuring that this runtime enforcement is *fast* on different architectures, e.g., x86 [Ford and Cox 2008; McCamant and Morrisett 2006; Payer and Gross 2011; Yee et al. 2009], x86-64 [Sehr et al. 2010], SPARC® [Adl-Tabatabai et al. 1996], and ARM® [Sehr et al. 2010; Zhao et al. 2011; Zhou et al. 2014], as otherwise they incur unacceptable overheads on the code executing in the sandbox. At a high-level

restricting memory access is handled by either trapping on out-of-bounds accesses (crashing the library component on unsafe behavior) or rewriting the out-of-bounds accesses to remain in-bounds (thus breaking the intended usage of the library component but maintaining *safe* execution). Control flow restrictions are handled via control flow integrity (CFI) protections: under erroneous or exploited execution these restrict the control flow paths to some subset of all possible paths. It is generally infeasible to restrict exploited execution to exactly the non-erroneous control flow paths [Carlini et al. 2015]; instead CFI protections vary in their *granularity*, that is the size of the set of allowed targets of control flow instructions. For instance, NaCl uses a coarse-grain CFI: control flow instructions are only restricted to jumping to instructions at specific alignments (thus ensuring that control flow remains both within the sandbox and cannot skip memory access restrictions). In contrast, WebAssembly enforces a fine-grain CFI: dynamic control flow instructions are restricted to be to targets that correspond to the type of the control flow instruction.

Because of the critical security nature of SFI enforcement there is also considerable literature that establishes that the checks and restrictions are *correct* [Besson, Blazy, et al. 2019; Besson, Jensen, et al. 2018; Johnson et al. 2021; Kroll et al. 2014; Morrisett, Tan, et al. 2012], as even a single missing check can let an attacker escape the sandbox. This correctness is generally established via either verification of the SFI compiler in theorem provers such as Roq [Morrisett, Tan, et al. 2012; The Coq Development Team 2025] or with a verifier that ensures that the correct checks are inserted into the code output by the SFI compiler [Johnson et al. 2021; Wahbe et al. 1993].

0.2 Constant Time

Every application that works with sensitive or personal user data uses cryptography to ensure the confidentiality or integrity of the data. As cryptographic operations

work directly with information that *must* remain secret (e.g. an attacker that gains access to a user's secret key could undetectably impersonate the user or steal the user's data) it is vital that the software implementing them be free of security vulnerabilities. While cryptographic implementations must contend with the threat of memory safety bugs, the fact of their direct operation over secret values means they must also contend with the possibility of timing side channel attacks. Timing side channels are created when secret data is operated on via operations with data-dependent timing. For instance, the running time of a naïve implementation of modular exponentiation (as used in RSA and other algorithms) varies based on the value of the exponent [P. C. Kocher 1996]. An attacker that can observe multiple inputs can then measure this timing variation to determine the value of the exponent, allowing extraction of a user's secret key.

To be free from such timing side channel attacks, cryptographic code must be constant time (CT). Constant time properties are a subset of noninterference properties, which broadly capture the notion that public outputs do not vary based on private inputs, that is that the outputs that an attacker can observe are independent from the inputs that should be secret [Goguen and Meseguer 1982]. To capture timing side channels, constant time models generally augment the set of public outputs with a trace of events capturing the trace of operations with data-dependent timing. There has been significant work on characterizing when a cryptographic library is constant-time and designing recipes to write constant time code, and this work guides both the design of cryptographic algorithms and their implementation in cryptographic libraries.

0.2.1 Speculative Constant Time

While constant time has been a known security model for cryptographic code, recently a new source of vulnerabilities has arisen due to the speculative nature of execution on modern processors. To take advantage of their wide pipelines and parallel execution model, modern processors execute instructions optimistically, for example,

beginning to execute the target of a branch before the condition has been fully resolved. Unfortunately, this speculatively executed code leaves detectable traces, via timing, changes to caches, or potentially other side channels, thereby allowing attackers to glean information from speculative execution.

Attacks exploiting processors' speculative execution are referred to as Spectre attacks [P. Kocher et al. 2019], with different variants targeting different architectures and Spectre models. As an example, the original Spectre attack (Spectre v1 or Spectre PHT) attacks the pattern history table (also known as the branch predictor) on x86 processors. The branch predictor tracks the expected target of branches, so, if an attacker can train the predictor that a certain branch condition is typically true, they can execute the code in the true branch even if the condition isn't met. The following code demonstrates the classic instantiation of this attack:

```
// array is of length 10
void main() {
  ...
  if (index < 10) {
    array[index];
    ...
  }
  ...
}
```

The attacker controls `index` and sets it to `100`. Having trained the branch predictor with indices under `10`, the processor predicts that the branch condition will be true and speculatively runs `array[index]`, thus reading out of bounds before eventually rolling back, allowing an attacker to perform an out-of-bounds memory read.

Similar to constant time, to be free from speculative execution based attacks, cryptographic code must be speculatively constant time (SCT). SCT properties are

essentially CT properties in the context of speculative execution. As such SCT properties fundamentally depend on *how* speculative execution works and is modeled. There is significant ongoing work into modeling speculative execution (we refer interested readers to Cauligi, Disselkoen, Moghimi, et al. [2022]) and investigating the speculative behavior and attack surfaces of modern processors.

Chapter 1

Near Zero Cost Transitions

As discussed in Section 0.1, software fault isolation (SFI) toolkits (software sandboxing) are used to provide security in the setting of untrusted libraries operating inside trusted applications by transforming untrusted library code to ensure that memory safety bugs cannot leak outside of the sandboxed component. This handles the security requirements of the untrusted library, however, the security and overhead of software sandboxing also crucially depends on the correctness and cost of context switching: the *trampolines* and *springboards* used to transition into and out of sandboxes. Almost all SFI systems, from Wahbe et al. [1993]'s original SFI implementation to recent WebAssembly SFI toolkits [Bytecode Alliance 2020; McMullen 2020], use *heavyweight transitions* for context switching. These transitions 1. switch protection domains by tying into the underlying memory isolation mechanism (e.g., by setting segment registers [Yee et al. 2009], memory protection keys [Hedayati et al. 2019; Vahldiek-Oberwagner et al. 2019], or sandbox context registers [Bytecode Alliance 2020; McMullen 2020]), and 2. save, scrub, and restore machine state (e.g. the stack pointer, program counter, and callee-save registers) across the boundary. This code is complicated and hard to get right, as it has to account for the particular quirks of different architectures and operating system platforms [Alder et al. 2020]. Consequently, bugs in transition code have led to vulnerabilities in both NaCl and WebAssembly: from sandbox

breakouts [Chromium Team 2019, 2011], to information leaks [Chromium Team 2012, 2010], and application state corruption [Rydgård 2020]. Furthermore, in applications with high application-sandbox context switching rates, the cost of transitions dominates the overall sandboxing overhead. For example, heavyweight transitions prohibitively slowed down font rendering in Firefox, preventing Mozilla from shipping a sandboxed `libgraphite` [Narayan, Disselkoen, Garfinkel, et al. 2020].

In this chapter, we develop the principles and pragmatics needed to implement SFI systems with near-zero-cost transitions, thereby realizing the three-decade-old vision of reducing the cost of SFI context switches to (almost) that of a function call. We do this via five contributions:

1. **A formal model of secure transitions (§ 1.2):** Simply eliminating heavyweight transitions is unsafe, potentially allowing an attacker to escape the SFI sandbox. To understand this threat, our first contribution is a formal, declarative, and high-level model that elucidates the role of transitions in making SFI secure. Intuitively, our model shows how secure transitions protect the integrity and confidentiality of machine state across the domain transition by providing *well-bracketed* control flow, i.e., ensuring that returns actually return to their call sites.

2. **Zero-cost conditions for isolation (§ 1.3):** Heavyweight transitions provide security by wrapping cross-domain calls and returns to ensure that sandboxed code cannot, for example, read secret registers or tamper with the stack pointer. While this wrapping is necessary when sandboxing arbitrary code, our insight is these wrappers can be made redundant when the code enjoys additional structure, not dissimilar to the additional structure typically imposed by most SFI systems to ensure memory isolation. For example, NaCl uses *coarse-grained* control flow integrity (CFI) to restrict the sandbox's control flow to its own code region [Haas et al. 2017; Tan 2017; Yee et al. 2009].

We concretize this insight via our second contribution, a precise definition of *zero-cost conditions* that guarantee that sandboxed code can safely use zero-cost transitions. In particular, we show that transitions can be eliminated when sandboxed code follows a *type-directed* CFI discipline, has well-bracketed control flow, enforces local state (stack and register) encapsulation, and ensures registers and stack slots are initialized before use. Our notion of zero-cost conditions is inspired, in part, by techniques that use type- and memory-safe languages to isolate code via language-level enforcement of well-bracketed control flow and local state encapsulation [Grimmer et al. 2015; Hunt and Larus 2007; Maffeis et al. 2010; Mettler et al. 2010; M. S. Miller et al. 2008; Morrisett, Crary, Glew, Grossman, et al. 1999]. However, instead of requiring developers to rewrite millions of lines of code in high-level languages [Tan 2017], our zero-cost conditions distill the semantic guarantees provided by high-level languages to allow retrofitting zero-cost transitions in the SFI setting.

3. **Instantiating the zero-cost model (§ 1.4):** We demonstrate the retrofitting of zero-cost transitions via our third contribution, an instantiation of our zero-cost model to two SFI systems: WebAssembly and `SegmentZero32`. Previous work has shown how WebAssembly can provide SFI by compiling untrusted C/++libraries to native code using WebAssembly as an intermediate representation (IR) [Bosamiya et al. 2020; Narayan, Disselkoen, Garfinkel, et al. 2020; Narayan, Garfinkel, et al. 2019; Zakai 2020]. We show that WebAssembly satisfies our zero-cost conditions, and replace the heavyweight transitions used by the industrial Lucet WebAssembly SFI toolkit with zero-cost transitions. WebAssembly imposes more structure than required by our zero-cost conditions (and WebAssembly compilers are still relatively new and slow [Jangda et al. 2019]), so, in order to compare the overhead of our zero-cost model to the still fastest SFI implementation (NaCl [Yee et al.

2009]) we design a new prototype SFI system, `SegmentZero32`, that: (a) enforces our zero-cost conditions through LLVM-level transformations, and (b) enforces memory isolation in hardware, using 32-bit x86 segmentation.[1]

4. **Verifying security at the binary level (§ 1.5):** Our fourth contribution is a *static verifier*, VeriZero, that checks whether a potentially malicious binary produced by the Lucet toolkit satisfies our zero-cost conditions. This removes the need to trust the Lucet compiler when, for example, compiling third-party Firefox libraries [Narayan, Disselkoen, Garfinkel, et al. 2020] or untrusted tenant code running on Fastly's serverless cloud [McMullen 2020]. To prove the soundness of VeriZero, we develop a logical relation that captures when a compiled WebAssembly function is well-behaved with respect to our zero-cost conditions and use it to prove that the checks of VeriZero guarantee that the zero-cost conditions are met. We implement VeriZero by extending VeriWasm [Johnson et al. 2021] and show that in just a few seconds, it can (a) verify sandboxed libraries that ship (or are in the process of being shipped) with Firefox, WebAssembly-compiled SPEC CPU® 2006 benchmarks, and 100,000 programs randomly generated by Csmith [X. Yang et al. 2011], and (b) catch previous NaCl and WebAssembly vulnerabilities (§ 1.6.4). VeriZero has been integrated into the Lucet industrial WebAssembly compiler [Johnson 2021].

5. **Implementation and evaluation (§ 1.6):** Our last contribution is an implementation of our zero-cost sandboxing toolkits, and an evaluation of how they improve the performance of a transition micro-benchmark and two macro-benchmarks: image decoding (`libjpeg`) and font rendering (`libgraphite`) in Firefox. First, we demonstrate the potential performance of a purpose-built zero-cost SFI system,

[1]While the prevalence of 32-bit x86 systems is declining, it nevertheless still constitutes over 20% of the Firefox web browser's user base (over forty million users) [Mozilla 2021]; `SegmentZero32` would allow for high performance library sandboxing on these machines.

by evaluating SegmentZero32 on SPEC CPU® 2006 and our macro-benchmarks. We find that SegmentZero32 imposes at most 25% overhead on SPEC CPU® 2006 (nc), and at most 24% on image decoding and 22.5% on font rendering. These overheads are lower than the state-of-the-art NaCl SFI system. On the macro-benchmarks, SegmentZero32 even outperforms an idealized SFI system that enforces memory isolation for free but requires heavyweight transitions. Second, we find that zero-cost transitions speed up WebAssembly-sandboxed image decoding by (up to) 29.7% and font rendering by 10%. The speedup resulting from our zero-cost transitions allowed Mozilla to ship the WebAssembly-sandboxed libgraphite library in production.

1.1 Overview

In this section we describe the role of transitions in making SFI secure, give an overview of existing heavyweight transitions, and introduce our zero-cost model, which makes it possible for SFI systems to replace heavyweight transitions with simple function calls.

1.1.1 The Need for Secure Transitions

As an example, consider sandboxing an untrusted font rendering library (e.g., libgraphite) as used in a browser like Firefox:

```
1 void onPageLoad(int* text) {
2   ...
3   int* screen = ...; // stored in r12
4   int* temp_buf = ...;
5   gr_get_pixel_buffer(text, temp_buf);
6   memcpy(screen, temp_buf, 100);
7   ...
8 }
```

This code calls the `libgraphite` `gr_get_pixel_buffer` function to render text into a temporary buffer and then copies the temporary buffer to the variable `screen` to be rendered.

Using SFI to sandbox this library ensures that the browser's memory is isolated from `libgraphite`: memory isolation ensures that `gr_get_pixel_buffer` cannot access the memory of `onPageLoad` or any other parts of the browser stack and heap. Unfortunately, memory isolation alone is not enough: if transitions are simply function calls, attackers can violate the calling convention at the application-library boundary (e.g., the `gr_get_pixel_buffer` call and its return) to break isolation. Below, we describe the different ways a compromised `libgraphite` can do this.

- **Clobbering callee-save registers:** Suppose the `screen` variable in the above `onPageLoad` snippet is compiled down to the register `r12`. In the System V calling convention `r12` is a *callee-saved* register [Lu et al. 2018], so if `gr_get_pixel_buffer` clobbers `r12`, then it is also supposed to restore it to its original value before returning to `onPageLoad`. A compromised `libgraphite` doesn't have to do this; instead, the attacker can poison the register:

```
1 mov r12, ☣
2 ret
```

  Since `r12` (`screen`) in our hypothetical example is then used on Line 6 to `memcpy` the `temp_buf` from the sandbox memory, this gives the attacker a write gadget that they can use to hijack Firefox's control flow. To prevent such attacks, we need *callee-save register integrity*, i.e., we must ensure that sandboxed code restores callee-save registers upon returning to the application.

- **Leaking scratch registers:** Dually, *scratch registers* can potentially leak sensitive information into the sandbox. Suppose that Firefox keeps a secret (e.g., an encryption key) in a scratch register. Memory isolation alone would not prevent

an attacker-controlled `libgraphite` from using uninitialized registers, thereby reading this secret. To prevent such leaks, we need *scratch register confidentiality*.

- **Reading and corrupting stack frames:** Finally, if the application and sandboxed library share a stack, the attacker could potentially read and corrupt data (and pointers) stored on the stack. To prevent such attacks, we need *stack frame encapsulation*, i.e., we need to ensure that sandboxed code cannot access application stack frames.

1.1.2 Heavyweight Transitions

SFI toolchains, from NaCl [Yee et al. 2009] to WebAssembly native compilers like Lucet [McMullen 2020] and WAMR [Bytecode Alliance 2020], use *heavyweight transitions* to wrap calls and returns and address the aforementioned attacks. These heavyweight transitions are secure transitions. They provide:

1. **Callee-save register integrity:** The *springboard*, the transition code which wraps calls, saves callee-save registers to a separate stack stored in protected application memory. When returning from the library to the application, the *trampoline*, the code which wraps returns, restores the registers.

2. **Scratch register confidentiality:** Since any scratch register may contain secrets, the springboard clears *all* scratch registers before transitioning into the sandbox.

3. **Stack frame encapsulation:** Most (but not all) SFI systems provision separate stacks for trusted and sandboxed code and ensure that the trusted stack is not accessible from the sandbox. The springboard and trampoline account for this in three ways. First, they track the separate stack pointers at each transition in order to switch stacks. Second, the springboard copies arguments passed on the stack to the sandbox stack, since sandboxed code cannot access arguments stored on

the application stack. Finally, the trampoline tracks the actual return address on transition by keeping it in the protected memory, so that the sandboxed library cannot tamper with it.

Heavyweight springboards and trampolines guarantee secure transitions but have two significant drawbacks. First, they impose an overhead on SFI: calls into the sandboxed library become significantly more expensive than simple application function calls (see Section 1.6). Heavyweight transitions conservatively save and clear more state than might be necessary, essentially reimplementing aspects of an OS process switch and duplicating work done by well-behaved libraries. Second, springboards and trampolines must be customized to different platforms, i.e., different processors and calling conventions, and, in extreme cases such as in Vahldiek-Oberwagner et al. [2019], even different applications. Implementation mistakes can (and have [Bartel and Doe 2018; Chromium Team 2019, 2012, 2011, 2010]) resulted in sandbox escape attacks.

1.1.3 Zero-Cost Transitions

Heavyweight transitions are conservative because they make few assumptions about the structure (or possible behavior) of the code running in the sandbox. SFI systems like NaCl and WebAssembly *do*, however, impose structure on sandboxed code to enforce memory isolation. In this section we show that by imposing structure on sandboxed code we can make transitions less conservative. Specifically, we describe a set of *zero-cost conditions* that impose *just enough* internal structure on sandboxed code to ensure that it will behave like a high-level, compositional language while maintaining SFI's high performance. SFI systems that meet these conditions can safely elide almost all the extra work done by heavyweight springboards and trampolines, thus moving toward the ideal of SFI transitions as simple, fast, and portable function calls.

We assume that the sandboxed library code is split into functions and that each function has an expected number of arguments. We *formalize* the internal structure

required of library code via a *safety monitor* that checks the zero-cost conditions, i.e., the local requirements necessary to ensure that calls-into and returns-from the untrusted library functions are "well-behaved" and, hence, that they satisfy the secure transition requirements.

1. **Callee-save register restoration:** First, our monitor enforces function-call level adherence to callee-save register conventions: our monitor tracks callee-save state and checks that it has been correctly restored upon returning. Importantly, satisfying the monitor means that application calls to a well-behaved library function do not require a transition which separately saves and restores callee-save registers, since the function is known to obey the standard calling convention.

2. **Well-bracketed control-flow:** Second, our monitor requires that the library code adheres to well-bracketed return edges. Abstractly, calls and returns should be well-bracketed: when `f` calls `g` and then `g` calls `h`, `h` ought to return to `g` and then `g` ought to return to `f`. However, untrusted functions may subvert the control stack to implement arbitrary control flow between functions. This unrestricted control flow is at odds with compositional reasoning, preventing *local* verification of functions. Further, subverting well-bracketing could enable an attacker to cause `h` to return directly to `f`. Then, even if `h` and `f` both restore their callee-save registers, those of `g` would be left unrestored. Accordingly, we require two properties of the library to ensure that calls and returns are well-bracketed. First, each jump must stay within the same function. This limits inter-function control flow to function calls and returns. Second, the (specification) monitor maintains a "logical" call stack, which is used to ensure that returns go only to the preceding caller.

3. **Type-directed forward-edge CFI:** Our monitor also requires that library code obeys type-directed forward-edge CFI. That is, for every call instruction encountered during execution, the jump target address is the start of a library function and the

arguments passed match those expected by the called function. This ensures that each function starts from a (statically) known stack shape, preventing a class of attacks where a benign function can be tricked into overwriting other stack frames or hijacking control flow because it is passed too few (or too many) arguments. If this were not the case, a locally well-behaved function that was passed too few arguments could write to a saved register or the saved return address, expecting that stack slot to be the location of an argument.

4. **Local state encapsulation:** Our monitor establishes *local state encapsulation* by checking that all stack reads and writes are within the current stack frame. This check allows us to *locally*, i.e., by checking each function in isolation, ensure that a library function correctly saves and restores callee-save registers upon entry and exit. To see why local state encapsulation is needed, consider the following idealized assembly function `library_func`:

```
1 library_func:          library_helper:
2   push r12                store sp - 1 := ☣
3   mov r12 ← 1            ret
4   load r1 ← sp - 1
5   add r1 ← r12
6   call library_helper
7   pop r12
8   ret
```

If `library_helper` is called it will overwrite the stack slot where `library_func` saved `r12`, and `library_func` will then "restore" `r12` to the attacker's desired value. Our monitor prohibits such cross-function tampering, thus ensuring that all subsequent reasoning about callee-save integrity can be carried out locally in each function.

5. **Confidentiality:** Finally, our monitor uses dynamic information flow control (IFC)

tracking to define the confidentiality of scratch registers. The monitor tracks how (secret application) values stored in scratch registers flow through the sandboxed code, and checks that the library code does not leak this information. Concretely, our implementations enforce this by ensuring that, within each function's localized control flow, all register and local stack variables are initialized before use.

The individual properties making up our zero-cost conditions are well-known to be beneficial to software security, and their enforcement in low-level code has been extensively studied (see Section 1.8): our insight (made manifest in the monitor soundness proofs of Section 1.3.2) is that in conjunction these conditions are *sufficient* to eliminate heavyweight transitions in SFI systems, which can currently be a source of significant overhead when sandboxing arbitrary code. Indeed, in Section 1.4.1 we show that the WebAssembly type system is strict enough to ensure that a WebAssembly compiler generates native code that already meets these conditions. To increase the trustworthiness of this zero-cost compatible WebAssembly, in Section 1.5 we describe a verifier that statically checks that compiled WebAssembly code meets the zero-cost conditions. In Section 1.5.4 we describe our proof of soundness for the verifier, proving that the verifier's checks ensure monitor safety and therefore zero-cost security. Further, in Section 1.4.2 we demonstrate how the zero-cost conditions can be used to design a new SFI scheme by combining hardware-backed memory isolation with existing LLVM compiler passes.

1.2 A Gated Assembly Language

We formalize zero-cost transitions via an assembly language, SFIasm, that captures key notions of an application interacting with a sandboxed library, focusing on capturing properties of the transitions between the application and sandboxed library. Figure 1.1 defines the syntax of SFIasm: a RISC-style language with natural numbers

$$
\begin{array}{rrcrcl}
\text{numbers} & & & n, \ell & \in & \mathbb{N} \\
\text{privileges} & \mathit{Priv} & \ni & p & ::= & \texttt{app} \mid \texttt{lib} \\
\text{values} & \mathit{Val} & \ni & v & ::= & n \\
\text{registers} & \mathit{Reg} & \ni & r & ::= & \texttt{r}_n \mid sp \mid pc \\
\text{memory regions} & \mathit{Region} & \ni & k & \in & \mathbb{N} \rightharpoonup \mathbb{N} \\
\text{expressions} & \mathit{Expr} & \ni & e & ::= & r \mid v \mid e \oplus e \\
\text{commands} & \mathit{Command} & \ni & c & ::= & r \leftarrow \texttt{pop}_p \\
 & & & & \mid & \texttt{push}_p\ e \\
 & & & & \mid & r \leftarrow \texttt{load}_k\ e \\
 & & & & \mid & \texttt{store}_k\ e := e \\
 & & & & \mid & r \leftarrow \texttt{mov}\ e \\
 & & & & \mid & \texttt{call}_k\ e \\
 & & & & \mid & \texttt{ret}_k \\
 & & & & \mid & \texttt{jmp}_k\ e \\
 & & & & \mid & \texttt{gatecall}_n\ e \\
 & & & & \mid & \texttt{gateret} \\
\text{code} & \mathit{Code} & \ni & C & \in & \mathbb{N} \rightharpoonup \mathit{Priv} \times \mathit{Command} \\
\text{register maps} & \mathit{RegVals} & \ni & R & \in & \mathit{Reg} \to \mathit{Val} \\
\text{data} & \mathit{Memory} & \ni & M & \in & \mathbb{N} \to \mathit{Val} \\
\text{states} & \mathit{State} & \ni & \Psi & ::= & \texttt{error} \\
 & & & & \mid & \left\{ \begin{array}{lcl} pc & : & \mathbb{N} \\ sp & : & \mathbb{N} \\ R & : & \mathit{RegVals} \\ M & : & \mathit{Memory} \\ C & : & \mathit{Code} \end{array} \right\}
\end{array}
$$

Figure 1.1. Syntax of SFIasm.

($\mathbb{N}$) as the sole data type. Code (C) and data (M) memory are separated, and, to capture the separation of application code from sandboxed library code, C is an (immutable) partial map from $\mathbb{N}$ to pairs of a privilege (p) (`app` or `lib`) and a command (c), where `app` and `lib` are our *security domains*. We order `app` and `lib` via the "less trusted than" relation ($p \sqsubseteq p'$) where $\texttt{lib} \sqsubseteq \texttt{app}$ and $\texttt{app} \not\sqsubseteq \texttt{lib}$.

Memory is a (total) map from $\mathbb{N}$ to values (v). We assume that the memory is subdivided into disjoint regions (M_p) so that the application and library have separate memory. Each of these regions is further divided into a disjoint heap H_p and stack S_p. We write Ψ to denote the states or machine configurations, which comprise code, memory, and a fixed, finite set of registers mapping register names (r_n) to values, with a distinguished stack pointer (sp) and program counter (pc) register. We write $\Psi(\!|c|\!)_p$ for $\Psi.C(\Psi.pc) = (p, c)$, that is that the current instruction is c in security domain p. We write $\Psi_0 \in \mathit{Program}$ to mean that Ψ_0 is a valid initial program state. The definition of validity varies between different SFI techniques (e.g., heavyweight transitions make assumptions about the initial state of the separate stack).

We capture the transitions between the application and the library by defining a pair of instructions $\texttt{gatecall}_n\ e$ and `gateret`, that serve as the *only* way to switch between the two security domains (that is, `call` and `ret` check that the target is in the same security domain). The first, $\texttt{gatecall}_n\ e$, represents a call from the application into the sandbox or a callback from the sandbox to the application with the n annotation representing the number of arguments to be passed. The second, `gateret`, represents the corresponding return from sandbox to application or vice-versa. We leave the reduction rule for both *implementation specific* in order to capture the details of a given SFI system's trampolines and springboards.

SFIasm provides abstract mechanisms for enforcing SFI memory isolation by equipping the standard `load`, `store`, `push`, and `pop` with (optional) statically annotated checks. To capture different styles of enforcement we model these checks as partial

functions that map a pointer to its new value or are undefined when a particular address is invalid. This lets us, for instance, capture NaCl's coarse grained, dynamically enforced isolation (sandboxed code may read and write anywhere in the sandbox memory) by requiring that all loads and stores are annotated with the check $k(n)|_{n \in M_{\texttt{lib}}} = n$. This captures that NaCl's memory isolation does not remap addresses but traps when an address is outside the sandbox memory region ($M_{\texttt{lib}}$).[2] The rule for `load` below demonstrates the use of these region annotations in the semantics.

SFIasm also provides abstract control-flow integrity enforcement via annotations on `jmp`, `call`, and `ret`. These are also enforced dynamically. However, we require that the standard control flow operations remain within their own security domain so that `gatecall` and `gateret` remain the only way to switch security domains.

We capture the dynamic behavior via a deterministic small-step operational semantics ($\Psi \to \Psi'$) shown in Figures 1.2, 1.3, and 1.4. The rules are standard for an assembly language: to explain the notation we describe the reduction for a `load` instruction. The rule RED-LOAD handles a "correct" `load` instruction. $\mathcal{V}_{\Psi}(e)$ evaluates the expression to a value n based on the register file. If the memory isolation check function $k(n)$ is defined and maps n to a value n' then RED-LOAD applies and a value is loaded from memory into the appropriate register. Ψ^{++} increments pc, checking that it remains within the same security domain and returning an error otherwise. If the function $k(n)$ is undefined (n is not within bounds), the rule RED-LOAD-ERROR applies instead and the program will step to a distinguished, terminal state `error`. $\Psi(\!|c|\!)$ is simply shorthand for $\Psi(\!|c|\!)_p$ when we do not care about the security domain. Lastly, we do not include a specific halt command, instead halting when pc is not in the domain of C.

[2]NaCl implements memory protection differently on different platforms. The 32-bit implementation traps whereas the 64-bit implementation masks addresses. Without loss of generality we use the former in our model.

$$\boxed{\Psi(\!|c|\!) \to \Psi'}$$

$$\textsc{red-pop}\quad \frac{\Psi.sp \in S_{p_s} \qquad p_s \sqsubseteq p \qquad v = \Psi.M(\Psi.sp) \qquad R' = R[r \mapsto v]}{\Psi(\!|r \leftarrow \texttt{pop}_p|\!) \to \Psi^{++}[sp := \Psi.sp - 1, R := R']}$$

$$\textsc{red-push}\quad \frac{v = \mathcal{V}_\Psi(e) \qquad sp' = \Psi.sp + 1 \qquad M' = \Psi.M[sp' \mapsto v] \qquad sp' \in S_{p_s} \qquad p_s \sqsubseteq p}{\Psi(\!|\texttt{push}_p\ e|\!) \to \Psi^{++}[sp := sp', M := M']}$$

$$\textsc{red-load}\quad \frac{n = \mathcal{V}_\Psi(e) \qquad n' = k(n) \qquad v = \Psi.M(n') \qquad R' = \Psi.R[r \mapsto v]}{\Psi(\!|r \leftarrow \texttt{load}_k\ e|\!) \to \Psi^{++}[R := R']}$$

$$\textsc{red-store}\quad \frac{n = \mathcal{V}_\Psi(e) \qquad v = \mathcal{V}_\Psi(e') \qquad n' = k(n) \qquad M' = \Psi.M[n' \mapsto v]}{\Psi(\!|\texttt{store}_k\ e := e'|\!) \to \Psi^{++}[M := M']}$$

$$\textsc{red-jmp}\quad \frac{n = \mathcal{V}_\Psi(e) \qquad n' = k(n)}{\Psi(\!|\texttt{jmp}_k\ e|\!) \to \Psi[pc := n']}$$

$$\textsc{red-mov}\quad \frac{v = \mathcal{V}_\Psi(e) \qquad R' = \Psi.R[r \mapsto v]}{\Psi(\!|r \leftarrow \texttt{mov}\ e|\!) \to \Psi^{++}[R := R']}$$

$$\textsc{red-mov-sp}\quad \frac{v = \mathcal{V}_\Psi(e)}{\Psi(\!|sp \leftarrow \texttt{mov}\ e|\!) \to \Psi^{++}[sp := v]}$$

$$\textsc{red-call}\quad \frac{n = \mathcal{V}_\Psi(e) \qquad n' = k(n) \qquad sp' = \Psi.sp + 1 \qquad M' = \Psi.M[sp' \mapsto \Psi.pc + 1] \qquad sp' \in S_{p_s}}{\Psi(\!|\texttt{call}_k\ e|\!) \to \Psi[pc := n', sp := sp', M := M']}$$

$$\textsc{red-ret}\quad \frac{n = \Psi.M(\Psi.sp) \qquad n' = k(n) \qquad \Psi.sp \in S_{p_s}}{\Psi(\!|\texttt{ret}_k|\!) \to \Psi[pc := n', sp := \Psi.sp - 1]}$$

Figure 1.2. Operational semantics for SFIasm.

$$\boxed{\Psi(\!|c|\!) \rightarrow \texttt{error}}$$

$$\frac{\Psi.sp \in S_{p_s} \qquad p_s \not\sqsubseteq p}{\Psi(\!|r \leftarrow \texttt{pop}_p|\!) \rightarrow \texttt{error}}\ \textsc{red-pop-error}$$

$$\frac{\Psi.sp + 1 \in S_{p_s} \qquad p_s \not\sqsubseteq p}{\Psi(\!|\texttt{push}_p\ e|\!) \rightarrow \texttt{error}}\ \textsc{red-push-error}$$

$$\frac{n = \mathcal{V}_\Psi(e) \qquad k(n)\ \text{undefined}}{\Psi(\!|r \leftarrow \texttt{load}_k\ e|\!) \rightarrow \texttt{error}}\ \textsc{red-load-error}$$

$$\frac{n = \mathcal{V}_\Psi(e) \qquad k(n)\ \text{undefined}}{\Psi(\!|\texttt{store}_k\ e := e'|\!) \rightarrow \texttt{error}}\ \textsc{red-store-error}$$

$$\frac{\Psi.sp + 1 \notin S_{p_s}}{\Psi(\!|\texttt{call}_k\ e|\!) \rightarrow \texttt{error}}\ \textsc{red-call-sp-error}$$

$$\frac{n = \mathcal{V}_\Psi(e) \qquad k(n)\ \text{undefined}}{\Psi(\!|\texttt{call}_k\ e|\!) \rightarrow \texttt{error}}\ \textsc{red-call-target-error}$$

$$\frac{\Psi.sp \notin S_{p_s}}{\Psi(\!|\texttt{ret}_k|\!) \rightarrow \texttt{error}}\ \textsc{red-ret-sp-error}$$

$$\frac{n = \Psi.M(\Psi.sp) \qquad k(n)\ \text{undefined}}{\Psi(\!|\texttt{ret}_k|\!) \rightarrow \texttt{error}}\ \textsc{red-ret-target-error}$$

$$\frac{n = \mathcal{V}_\Psi(e) \qquad k(n)\ \text{undefined}}{\Psi(\!|\texttt{jmp}_k\ e|\!) \rightarrow \texttt{error}}\ \textsc{red-jmp-error}$$

Figure 1.3. Operational semantics for SFIasm: errors.

$$\begin{array}{rcl}
\mathcal{V}_\Psi(v) & \triangleq & v \\
\mathcal{V}_\Psi(r) & \triangleq & \Psi.R(r) \\
\mathcal{V}_\Psi(sp) & \triangleq & \Psi.sp \\
\mathcal{V}_\Psi(pc) & \triangleq & \Psi.pc \\
\mathcal{V}_\Psi(e \oplus e') & \triangleq & \mathcal{V}_\Psi(e) \oplus \mathcal{V}_\Psi(e')
\end{array}$$

$$\Psi^{++} \triangleq \begin{cases} \Psi[pc := \Psi.pc + 1] & \text{when } \pi_1(\Psi.C(\Psi.pc)) = \pi_1(\Psi.C(\Psi.pc + 1)) \\ \texttt{error} & \text{otherwise} \end{cases}$$

Figure 1.4. Operational semantics for SFIasm: auxiliary definitions.

$$\frac{\Psi_1 \rightarrow \Psi_2 \qquad \Psi_1(\!|c_1|\!)_{p_1} \qquad \Psi_2(\!|c_2|\!)_{p_2} \qquad p_1 = p_2 = p}{\Psi_1 \xrightarrow{p} \Psi_2} \qquad \frac{\Psi \xrightarrow{p} \Psi'}{\Psi \xrightarrow{\Box} \Psi'} \qquad \frac{\Psi \xrightarrow{\text{wb}} \Psi'}{\Psi \xrightarrow{\Box} \Psi'} \qquad \frac{\Psi \rightarrow \Psi_1 \xrightarrow{\Box}{}^{*} \Psi_2 \rightarrow \Psi' \qquad \Psi(\!|\texttt{gatecall}_n\ i|\!) \qquad \Psi_2(\!|\texttt{gateret}|\!)}{\Psi \xrightarrow{\text{wb}} \Psi'}$$

Figure 1.5. Well-bracketed transitions in SFIasm.

1.2.1 Secure Transitions

Next, we use SFIasm to declaratively specify high-level properties that capture the intended security goals of transition systems. This lets us use SFIasm both as a setting to study zero-cost transitions and to explore the correctness of implementations of springboards and trampolines. As a demonstrative example we prove that NaCl-style heavyweight transitions satisfy the high-level properties in Section 1.2.2.

SFI systems may allow arbitrary *nesting* of calls into and callbacks out of the sandbox. Thus, it is insufficient to define that callee-save registers have been properly restored by simply equating register state upon entry to the sandbox and the following exit. Instead we make the notion of an entry and its *corresponding* exit precise, by using SFIasm's `gatecall` and `gateret` to define a notion of *well-bracketed gated calls* that serve as the backbone of transition integrity properties. A well-bracketed gated call, which we write $\Psi \xrightarrow{\text{wb}} \Psi'$ (Figure 1.5), captures the idea that Ψ is a gated call from one security domain to another, followed by running in the new security domain, and then Ψ' is the result of a gated return that balances the gated call from Ψ. This can include potentially recursive but properly bracketed gated calls. Well-bracketed gated calls let us relate the state before a gated call with the state after the *corresponding* gated return, capturing when the library has fully returned to the application.

Integrity

Relations between the states before calling into the sandbox and then after the corresponding return capture SFI transition system *integrity* properties. We identify two key integrity properties that SFI transitions must maintain:

1. *Callee-save register integrity:* requires that callee-save registers are restored after returning from a gated call into the library. This ensures that an attacker cannot unexpectedly modify the private state of an application function.

2. *Return address integrity:* requires that the sandbox (a) returns to the instruction after the `gatecall`, (b) does not tamper with the stack pointer, and (c) does not modify the call stack itself.

Together these ensure that an attacker cannot tamper with the application control flow.

These integrity properties are crucial to ensure that the sandboxed library cannot break application invariants. To capture them formally, we first define an abstract notion of integrity across a well-bracketed gated call. This not only allows us to cleanly define the above properties, but also provides a general framework that can capture integrity properties for different architectures. Specifically, we define an integrity property by a predicate $\mathcal{I} : \mathit{Trace} \times \mathit{State} \times \mathit{State} \rightarrow \mathbb{P}$ that captures when integrity is preserved across a call ($\mathbb{P}$ is the type of propositions). The first argument is a trace, a sequence of steps that our program has taken before making the gated call. The next two arguments are the states before and after the well-bracketed gated call. $\mathcal{I}$ defines when these two states are properly related. This leads to the following definition of $\mathcal{I}$-Integrity:

Definition 1 ($\mathcal{I}$-Integrity)**.** *Let $\mathcal{I} : \mathit{Trace} \times \mathit{State} \times \mathit{State} \rightarrow \mathbb{P}$. We say that an SFI transition system has $\mathcal{I}$-integrity if, for all initial configurations $\Psi_0 \in \mathit{Program}$, trace prefixes $\pi = \Psi_0 \rightarrow^* \Psi_1$, that run to an application instruction $\Psi_1(\!|_|\!)_{\mathsf{app}}$ that then makes a well-bracketed call $\Psi_1 \xrightarrow{wb} \Psi_2$, we have that $\mathcal{I}(\pi, \Psi_1, \Psi_2)$.*

$$\boxed{\text{return-address}_p : \mathit{Trace} \to \wp(\mathbb{N})}$$

$$\text{return-address}_p(\Psi_0 \to^* \Psi(\!|\texttt{call}_k\ e|\!)_p \to \Psi')$$
$$\triangleq \text{return-address}_p(\Psi_0 \to^* \Psi) \cup \{\Psi.sp + 1\}$$

$$\text{return-address}_p(\Psi_0 \to^* \Psi(\!|\texttt{ret}_k|\!)_p \to \Psi')$$
$$\triangleq \text{return-address}_p(\Psi_0 \to^* \Psi) - \{\Psi.sp\}$$

$$\text{return-address}_p(\Psi_0 \to^* \Psi(\!|\texttt{gatecall}_n\ e|\!)_p \to \Psi')$$
$$\triangleq \text{return-address}_p(\Psi_0 \to^* \Psi) \cup \{\Psi.sp + 1\}$$

$$\text{return-address}_p(\Psi_0 \to^* \Psi(\!|\texttt{gateret}|\!)_p \to \Psi')$$
$$\triangleq \text{return-address}_p(\Psi_0 \to^* \Psi) - \{\Psi.sp\}$$

$$\text{return-address}_p(\Psi_0 \to^* \Psi(\!|c|\!) \to \Psi')$$
$$\triangleq \text{return-address}_p(\Psi_0 \to^* \Psi)$$

$$\text{return-address}_p(\Psi_0 \to^0 \Psi_0)$$
$$\triangleq \emptyset$$

Figure 1.6. Call stack return address calculation.

We instantiate this to define our two integrity properties:

1. **Callee-save register integrity:** We define callee-save register integrity as an $\mathcal{I}$-integrity property that requires the callee-save registers' values to be equal in both states:

   Definition 2 (Callee-Save Register Integrity)**.** *Let* $\mathbb{CSR}$ *be the callee-save registers and define* $\mathcal{CSR}(_, \Psi_1, \Psi_2) \triangleq \Psi_2.R(\mathbb{CSR}) = \Psi_1.R(\mathbb{CSR})$*. If an SFI transition system has* $\mathcal{CSR}$*-integrity then we say it has callee-save register integrity.*

2. **Return address integrity:** We specify that the library returns to the expected instruction as a relation between Ψ_1 and Ψ_2, namely that $\Psi_2.pc = \Psi_1.pc + 1$.

Restoration of the stack pointer is similarly specified as $\Psi_2.sp = \Psi_1.sp$. Specifying call stack integrity is more involved as Ψ_1 lacks information on where return addresses are saved: they look like any other stack data. Instead, return addresses are defined by the history of calls and returns leading up to Ψ_1, which we capture with the trace argument π. We thus define a function return-address(π) (Figure 1.6) that computes the locations of return addresses from a trace. The third clause of return address integrity is then that these locations' values are preserved from Ψ_1 to Ψ_2, yielding:

Definition 3 (Return Address Integrity)**.**

$$\begin{aligned}\mathcal{RA}(\pi, \Psi_1, \Psi_2) \triangleq\ & \Psi_2.pc = \Psi_1.pc + 1 \land \Psi_2.sp = \Psi_1.sp \\ & \land \Psi_2.M(\text{return-address}(\pi)) = \Psi_1.M(\text{return-address}(\pi))\end{aligned}$$

If an SFI transition system has $\mathcal{RA}$-integrity then we say the system has return address integrity.

Confidentiality

SFI systems must ensure that secrets cannot be leaked to the untrusted library, i.e., they must provide *confidentiality*. We specify confidentiality as noninterference, which informally states that "changing secret inputs should not affect public outputs." In the context of library sandboxing, application data is secret whereas library data is non-secret (public).[3] To capture this formally, we pair programs with a confidentiality policy, $\mathbb{C} \in State \rightharpoonup (\mathbb{N} + Reg \rightarrow Priv)$, that labels all memory and registers as `app` or `lib` at each gated call into the library.

To prove noninterference, that is, that changing secret data does not affect public (or non-secret) outputs, we need to define public outputs. We over-approximate public

[3]This could also be extended to a setting with mutually distrusting components.

outputs as the set of values *exposed* to the application. This includes all arguments to a `gatecall` callback, the return value when returning to the application via `gateret`, and all values stored in the sandboxed library's heap ($H_{\texttt{lib}}$) (which may be referenced by other returned values). Alas, this is not enough: in a callback, the application may choose to declassify secret data. For instance, a sandboxed image decoding library might, after parsing the file header, make a callback requesting the data to decode the rest of the image. This application callback will then transfer that data (which was previously confidential) to the sandbox, declassifying it in the transfer.

To account for such intentional declassifications, we follow Matos and Boudol [2005] and define confidentiality as *disjoint noninterference*. We define $\Psi =_{\mathbb{C}} \Psi'$ to mean that Ψ and Ψ' agree on all values labeled `lib` by the confidentiality policy, capturing varying secret inputs:

Definition 4. *We say* $\Psi =_{\mathbb{C}} \Psi'$ *when*

1. $\Psi.pc = \Psi'.pc$
2. $\Psi.sp = \Psi'.sp$
3. $\Psi(\!|\texttt{gatecall}_n\ e|\!)_{\texttt{app}}$ *and* $\Psi'(\!|\texttt{gatecall}_n\ e|\!)_{\texttt{app}}$
4. $\Psi.R|_{\{r|\mathbb{C}(\Psi)(r)=\texttt{lib}\}} = \Psi'.R|_{\{r|\mathbb{C}(\Psi')(r)=\texttt{lib}\}}$
5. $\Psi.M|_{\{n|\mathbb{C}(\Psi)(n)=\texttt{lib}\}} = \Psi'.M|_{\{n|\mathbb{C}(\Psi')(n)=\texttt{lib}\}}$

We further define $\Psi =_{\texttt{call}\ m} \Psi'$ when Ψ and Ψ' agree on all sandboxed heap values, the program counter, and the m arguments passed to a callback and $\Psi =_{\texttt{ret}} \Psi'$ when Ψ and Ψ' agree on all sandboxed heap values, the program counter, and the value in the calling convention's return register (written r_{ret}). The formal definitions are shown below:

Definition 5. *We say* $\Psi =_{\texttt{call}\ m} \Psi'$ *if*

1. $\Psi.M(H_{\texttt{lib}}) = \Psi'.M(H_{\texttt{lib}})$

2. $\Psi.pc = \Psi'.pc$

3. $\Psi.sp = \Psi'.sp$

4. *For all* $i \in [1, m]$, *there exists some* m' *such that* $m' = \Psi.M(\Psi.sp - i) = \Psi'.M(\Psi.sp - i)$.

Definition 6. *We say* $\Psi =_{\texttt{ret}} \Psi'$ *if*

1. $\Psi.M(H_{\texttt{lib}}) = \Psi'.M(H_{\texttt{lib}})$

2. $\Psi.pc = \Psi'.pc$

3. *There exists some* n *such that* $n = \Psi.R(r_{ret}) = \Psi'.R(r_{ret})$.

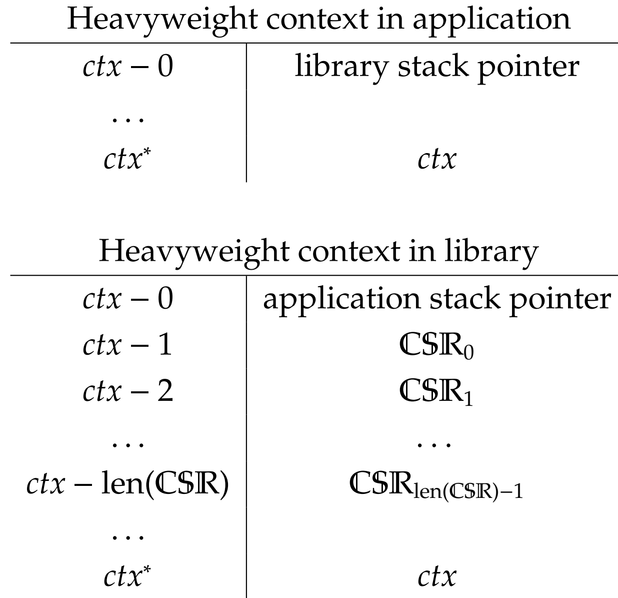
This lets us formally define noninterference as follows:

Definition 7 (Disjoint Noninterference)**.** *We say that an SFI transition system has the disjoint noninterference property if, for all initial configurations and their confidentiality policy* $(\Psi_0, \mathbb{C}) \in Program$, *traces* $\Psi_0 \rightarrow^* \Psi_1 \rightarrow \Psi_2 \xrightarrow{\texttt{lib}}{}^* \Psi_3 \rightarrow \Psi_4$, *where* Ψ_1 *is a gated call into the library* ($\Psi_1 (\!| \texttt{gatecall}_n\ e |\!)_{\texttt{app}}$), *and* $\Psi_3 \rightarrow \Psi_4$ *leaves the library and reenters the application* ($\Psi_4 (\!| _ |\!)_{\texttt{app}}$), *and, for all* Ψ'_1 *such that* $\Psi_1 =_{\mathbb{C}} \Psi'_1$, *we have that* $\Psi'_1 \rightarrow \Psi'_2 \xrightarrow{\texttt{lib}}{}^* \Psi'_3 \rightarrow \Psi'_4$, $\Psi'_4 (\!| _ |\!)_{\texttt{app}}$, $\Psi_4.pc = \Psi'_4.pc$, *and either*

1. Ψ_3 *is a gated call to the application* ($\Psi_3 (\!| \texttt{gatecall}_m\ e |\!)$ *and* $\Psi'_3 (\!| \texttt{gatecall}_m\ e |\!)$) *and* $\Psi_4 =_{\texttt{call}\ m} \Psi'_4$ *or*

2. Ψ_3 *is a gated return to the application* ($\Psi_3 (\!| \texttt{gateret} |\!)$ *and* $\Psi'_3 (\!| \texttt{gateret} |\!)$) *and* $\Psi_4 =_{\texttt{ret}} \Psi'_4$.

This definition captures that, for any sequence of executing within the library then returning control to the application, varying confidential inputs does not influence the public outputs and the library returns control to the application in the same number of steps. Thus, an SFI system that satisfies disjoint noninterference is guaranteed to not leak data while running within the sandbox.



Heavyweight context in application	
$ctx - 0$	library stack pointer
...	
ctx^*	ctx

Heavyweight context in library	
$ctx - 0$	application stack pointer
$ctx - 1$	$\mathbb{C}\$\mathbb{R}_0$
$ctx - 2$	$\mathbb{C}\$\mathbb{R}_1$
...	...
$ctx - \text{len}(\mathbb{C}\$\mathbb{R})$	$\mathbb{C}\$\mathbb{R}_{\text{len}(\mathbb{C}\$\mathbb{R})-1}$
...	
ctx^*	ctx

Figure 1.7. Layout of contexts for heavyweight transitions.

1.2.2 Security of Heavyweight Transitions

After formally defining the zero-cost conditions in Section 1.3 we will discuss our proof that zero-cost WebAssembly meets the above secure transition properties in Section 1.5.4. But first we will exercise the framework by defining NaCl style heavyweight transitions and the associated proofs of security.

We begin by defining the heavyweight springboards and trampolines. These rely on a stack-structured transition context described in Figure 1.7. There is first a fixed memory location ctx^* that stores a pointer (ctx) to the top of the transition context stack. Because NaCl style SFI systems cannot guarantee there is no stack smashing within the sandboxed library they allocate a library control stack within the sandbox memory. As such, when operating within the application the transition context stores the library stack pointer. Similarly, when operating within the library it is necessary to save the application stack pointer. Beyond this, the heavyweight transitions use the context to store the callee-save registers for restoration when returning to the application.

heavyweight-springboard$(n, e) \triangleq$

	$r_0 \leftarrow \texttt{load}\ ctx^*$
	// r_1 holds the library stack pointer
	$r_1 \leftarrow \texttt{load}\ r_0$
	// save callee-save registers
$j \in (\text{len}(\mathbb{CSR}), 0]$	$\texttt{store}\ r_0 := \mathbb{CSR}_j; r_0 \leftarrow \texttt{mov}\ r_0 + 1$
	// set r_1 to the new top of the library stack
	$r_1 \leftarrow \texttt{mov}\ r_1 + n$
	// move the stack pointer to the first argument
	$sp \leftarrow \texttt{mov}\ sp - 1$
	// copy arguments to library stack
$j \in [0, n)$	$r_2 \leftarrow \texttt{pop}; \texttt{store}_{M_{\texttt{lib}}}\ r_1 := r_2; r_1 \leftarrow \texttt{mov}\ r_1 - 1$
	// save stack pointer
	$r_2 \leftarrow \texttt{mov}\ sp + (n + 1); \texttt{store}\ r_0 := r_2$
	// set new stack pointer
	$sp \leftarrow \texttt{mov}\ r_1 + n$
	// update ctx
	$\texttt{store}\ ctx^* := r_0$
	// clear registers
$r \in \mathbb{R}$	$r \leftarrow \texttt{mov}\ 0$
	$\texttt{jmp}\ e$

Figure 1.8. Heavyweight springboards.

heavyweight-trampoline $\triangleq$

	$r_0 \leftarrow$ `load` ctx^*
	// restore callee-save registers
$j \in [0, \text{len}(\mathbb{CSR}))$	$r_0 \leftarrow$ `mov` $r_0 - 1$; $\mathbb{CSR}_j \leftarrow$ `load` r_0
	$r_0 \leftarrow$ `load` ctx^*
	// r_1 holds the application stack pointer
	$r_1 \leftarrow$ `load` r_0
	// save library stack pointer
	$r_0 \leftarrow$ `mov` $r_0 - \text{len}(\mathbb{CSR})$; `store` $r_0 := sp$
	// update ctx
	`store` $ctx^* := r_0$
	// switch to application stack
	$sp \leftarrow$ `mov` r_1
	`ret`

Figure 1.9. Heavyweight trampolines.

heavyweight-cb-springboard$(n, e) \triangleq$

	$r_0 \leftarrow$ `load` ctx^*
	// r_1 holds the application stack pointer
	$r_1 \leftarrow$ `load` r_0
	// save stack pointer
	$r_0 \leftarrow$ `mov` $r_0 + 1$; `store` $r_0 := sp$
	// update ctx
	`store` $ctx^* := r_0$
	// move the stack pointer to the first argument
	$sp \leftarrow$ `mov` $sp - 1$
	// set r_1 to the new top of the library stack
	$r_1 \leftarrow$ `mov` $r_1 + n$
	// copy arguments to application stack
$j \in [0, n)$	$r_2 \leftarrow \texttt{pop}_{M_{\texttt{lib}}}$; `store` $r_1 := r_2$; $r_1 \leftarrow$ `mov` $r_1 - 1$
	// set new stack pointer
	$sp \leftarrow$ `mov` $r_1 + n$
	$\texttt{jmp}_I\ e$

Figure 1.10. Heavyweight callback springboards.

$$
\begin{array}{r|l}
\multicolumn{2}{l}{\text{heavyweight-cb-trampoline} \triangleq} \\
\hline
 & r_0 \leftarrow \texttt{load}\ ctx^* \\
 & \text{// } r_1 \text{ holds the library stack pointer} \\
 & r_1 \leftarrow \texttt{load}\ r_0 \\
 & \text{// update } ctx \\
 & r_0 \leftarrow \texttt{mov}\ r_0 - 1; \texttt{store}\ ctx^* := r_0 \\
 & \text{// switch to library stack} \\
 & sp \leftarrow \texttt{mov}\ r_1 \\
 & \text{// clear registers} \\
r \in \mathbb{R} & \overline{r \leftarrow \texttt{mov}\ 0} \\
 & \texttt{ret}_{C_{\texttt{lib}}}
\end{array}
$$

Figure 1.11. Heavyweight callback trampoline.

To instantiate `gatecall` and `gateret` for heavyweight transitions we mirror the real-world implementations of NaCl and other existing SFI systems: defining assembly blocks that implement the entry and exit behavior. We show the implementations for normal calls and callbacks in Figures 1.8, 1.9, 1.10, and 1.11 (note that the springboards, which transition into the library, are parameterized by the number of arguments to be passed). The implementations are unsurprising given a description of heavyweight transitions, but we would like to draw attention to the tricky nature of ensuring correct use of the heavyweight context and properly ordering stack switching as a reminder of the difficulty of correctly implementing heavyweight transitions. `gatecall` and `gateret` are then implemented as running the associated assembly code.

NaCl Programs

Before we prove the security of heavyweight transitions we must instantiate an SFI system. Following our running example we choose to mode NaCl. To do so we define a NaCl program Ψ is as meeting the following conditions (these model the properties checked by the NaCl verifier):

1. All memory operations in the sandboxed library are guarded:

$$\begin{aligned} \forall n. \Psi.C(n) = (\texttt{lib}, r \leftarrow \texttt{pop}_p) &\implies p = \texttt{lib} \\ \Psi.C(n) = (\texttt{lib}, \texttt{push}_p\ e) &\implies p = \texttt{lib} \\ \Psi.C(n) = (\texttt{lib}, r \leftarrow \texttt{load}_k\ e) &\implies \mathrm{cod}(k) \subseteq M_{\texttt{lib}} \\ \Psi.C(n) = (\texttt{lib}, \texttt{store}_k\ e := e') &\implies \mathrm{cod}(k) \subseteq M_{\texttt{lib}}. \end{aligned}$$

2. The application does not write app data to the sandbox memory.

3. Gated calls are the only way to move between application and library code:

$$\begin{aligned} \forall n. \Psi.C(n) = (p, \texttt{call}_k\ e) &\implies \mathrm{cod}(k) \subseteq C_p \\ \Psi.C(n) = (p, \texttt{ret}_k) &\implies \mathrm{cod}(k) \subseteq C_p \\ \Psi.C(n) = (p, \texttt{jmp}_k\ e) &\implies \mathrm{cod}(k) \subseteq C_p \end{aligned}$$

4. The program starts in the application: $\Psi.pc = 0$ and $\Psi.C(0) = (\texttt{app}, _)$.

5. ctx^* and ctx start initialized to the library stack:

$$\begin{aligned} &ctx^* \in H_{\text{app}} \\ &ctx = \Psi.M(ctx^*) \in H_{\text{app}} \\ &\Psi.M(ctx) = S_{\texttt{lib}}[0] - 1. \end{aligned}$$

6. When calling into the library NaCl considers all registers except the stack pointer and program counter confidential. It further labels sandbox memory and the n' arguments in the application stack as public with the remainder of application

memory labeled as confidential. This is captured by the following confidentiality policy:

$$\mathbb{C}(\Psi)(r) = \begin{cases} \texttt{lib} & \text{when } r = sp \\ \texttt{lib} & \text{when } r = pc \\ \texttt{app} & \text{otherwise} \end{cases}$$

$$\mathbb{C}(\Psi)(n) = \begin{cases} \texttt{lib} & \text{when } n \in M_{\texttt{lib}} \\ \texttt{lib} & \text{when } n \in (\Psi.sp - n', \Psi.sp] \text{ where } \Psi(\!|\texttt{gatecall}_{n'}\ e|\!)_{\texttt{app}} \\ \texttt{app} & \text{when } n \in M_{\texttt{app}} \wedge n \notin (\Psi.sp - n', \Psi.sp] \\ & \quad \text{where } \Psi(\!|\texttt{gatecall}_{n'}\ e|\!)_{\texttt{app}} \end{cases}$$

We then show that the combined NaCl SFI instantiation and the heavyweight transitions satisfy our integrity and confidentiality properties. The proofs can be found in Appendix A.1.

1.3 Zero-Cost Transition Conditions

Having laid out the security properties required of an SFI transition system, we turn to formally defining the set of zero-cost conditions on sandboxed code that capture when we may securely elide springboards and trampolines. To this end we define our zero-cost conditions as a safety monitor via a language oSFIasm overlaid on top of SFIasm. oSFIasm extends SFIasm with additional structure and dynamic type checks that ensure the invariants needed for zero-cost transitions are maintained upon returning from library functions, providing both an inductive structure for proofs of security for zero-cost implementations and providing a top-level guarantee that our

$$
\begin{array}{rcrcl}
\mathit{Val} & \ni & v & ::= & \langle n, p \rangle \\
\mathit{Frame} & \ni & \mathit{SF} & ::= & \{\ \mathit{base} : \mathbb{N} \\
& & & & \quad \mathit{ret\text{-}addr\text{-}loc} : \mathbb{N} \\
& & & & \quad \mathit{csr\text{-}vals} : \wp(\mathit{Reg} \times \mathbb{N})\ \} \\
\mathit{Function} & \ni & F & ::= & \{\ \mathit{instrs} : \mathbb{N} \rightharpoonup \mathit{Command} \\
& & & & \quad \mathit{entry} : \mathbb{N} \\
& & & & \quad \mathit{type} : \mathbb{N}\ \} \\
\mathit{oState} & \ni & \Phi & ::= & \texttt{oerror} \\
& & & | & \{\ \Psi : \mathit{State} \\
& & & & \quad \mathit{funcs} : \mathbb{N} \rightharpoonup \mathit{Function} \\
& & & & \quad \mathit{stack} : [\mathit{Frame}]\ \}
\end{array}
$$

Figure 1.12. Extended syntax for overlay semantics of oSFIasm.

integrity and confidentiality properties are maintained. In Section 1.3.2 we outline the proofs of overlay soundness, showing that oSFIasm captures when a system is zero-cost secure.

Figure 1.12 shows the extended syntax of oSFIasm. Values (v) are extended with a security label p. Overlay states, written Φ, wrap the state of SFIasm, extending it with two extra pieces of data. First, oSFIasm requires the sandboxed code be organized into functions ($\Phi.\mathit{funcs}$). $\Phi.\mathit{funcs}$ maps each command in the sandboxed library to its parent function. Functions (F) also store the code indices of their commands as the field $F.\mathit{instrs}$, store the entry point ($F.\mathit{entry}$), and track the number of arguments the function expects ($F.\mathit{type}$). This partitioning of sandboxed code into functions is static. Second, the overlay state dynamically tracks a list of overlay stack frames ($\Phi.\mathit{stack}$). These stack frames (SF) are solely logical and inaccessible to instructions. They instead serve as bookkeeping to implement the dynamic type checks of oSFIasm by tracking the base address of each stack frame ($\mathit{SF}.\mathit{base}$), the stack location of the return address ($\mathit{SF}.\mathit{ret\text{-}addr}$), and the values of the callee save registers upon entry to the function ($\mathit{SF}.\mathit{csr\text{-}vals}$). We are concerned with the behavior of the untrusted library, so the logical stack does not finely

$$\boxed{\Psi(\!|c|\!) \rightsquigarrow \Psi'}$$

$$\frac{\langle n, p_e \rangle = \mathcal{V}_{\Phi}(e) \qquad n' = k(n) \qquad sp' = \Phi.sp + 1 \qquad M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \qquad \text{typechecks}(\Phi, n', sp') \qquad SF = \text{new-frame}(\Phi, n', sp') \qquad p_e \sqsubseteq \texttt{lib}}{\Phi(\!|\texttt{call}_k\ e|\!)_{\texttt{lib}} \rightsquigarrow \Phi[stack := [SF] +\!\!+ \Phi.stack, pc := n', sp := sp', M := M']}\ \textsc{oCall}$$

$$\frac{\text{is-ret-addr-loc}(\Phi, \Phi.sp) \qquad \langle n \rangle = \Phi.M(\Phi.sp) \qquad n' = k(n) \qquad \text{csr-restored}(\Phi) \qquad \Phi' = \text{pop-frame}(\Phi)}{\Phi(\!|\texttt{ret}_k|\!)_{\texttt{lib}} \rightsquigarrow \Phi'[pc := n', sp := \Phi.sp - 1]}\ \textsc{oRet}$$

$$\frac{\langle n' \rangle = \mathcal{V}_{\Phi}(e) \qquad sp' = \Phi.sp + 1 \qquad M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \qquad n' \in I \qquad \text{typechecks}(\Phi, n', sp') \qquad \text{args-secure}(\Phi, sp', n) \qquad SF = \text{new-frame}(\Phi, n', sp')}{\Phi(\!|\texttt{gatecall}_n\ e|\!)_{\texttt{lib}} \rightsquigarrow \Phi[stack := [SF] +\!\!+ \Phi.stack, pc := n', sp := sp', M := M']}\ \textsc{oGatecall-lib}$$

$$\frac{\Phi' = \text{classify}_{\mathbb{C}}(\Phi) \qquad \langle n' \rangle = \mathcal{V}_{\Phi'}(e) \qquad sp' = \Phi'.sp + 1 \qquad M' = \Phi'.M[sp' \mapsto \Phi'.pc + 1] \qquad SF = \text{new-frame}(\Phi', n', sp')}{\Phi(\!|\texttt{gatecall}_n\ e|\!)_{\text{app}} \rightsquigarrow \Phi'[stack := [SF] +\!\!+ \Phi'.stack, pc := n', sp := sp', M := M']}\ \textsc{oGatecall-app}$$

$$\frac{\Phi(\!|\texttt{ret}|\!) \rightsquigarrow \Phi' \qquad p' \sqsubseteq p \qquad \langle n, p' \rangle = \Phi.R(r_{ret})}{\Phi(\!|\texttt{gateret}|\!)_p \rightsquigarrow \Phi'}\ \textsc{oGateret}$$

$$\frac{v = \mathcal{V}_{\Phi}(e) \qquad sp' = \Phi.sp + 1 \qquad sp' \in S_p \qquad M' = \Phi.M[sp' \mapsto v] \qquad \text{writeable}(\Phi, sp')}{\Phi(\!|\text{push}_p\ e|\!) \rightsquigarrow \Phi^{++}[sp := sp', M := M']}\ \textsc{oPush}$$

Figure 1.13. Operational semantics for oSFIasm.

track application stack frames, but keeps a single large "stack frame" for all nested application stack frames.

When code fails the overlay's dynamic checks it will result in the state `oerror`. Our definition of monitor safety, which will ensure that zero-cost transitions are secure, is then simply that a program does not step to an `oerror`.

$$\boxed{\Psi(\!|c|\!) \rightsquigarrow \Psi'}$$

oStore
$$\frac{\begin{array}{c}\langle n, p_e\rangle = \mathcal{V}_\Phi(e) \qquad v = \langle _, p_{e'}\rangle = \mathcal{V}_\Phi(e') \\ M' = \Phi.M[n' \mapsto v] \qquad \text{writeable}(\Phi, n') \\ n' = k(n) \qquad p_e \sqsubseteq p \qquad p_{e'} \not\sqsubseteq p \implies n' \notin H_{\texttt{lib}}\end{array}}{\Phi(\!|\texttt{store}_k\ e := e'|\!)_p \rightsquigarrow \Phi^{++}[M := M']}$$

oLoad
$$\frac{\begin{array}{c}\langle n, p_e\rangle = \mathcal{V}_\Phi(e) \qquad n' = k(n) \qquad p_e \sqsubseteq p \\ v = \langle _, p_{n'}\rangle = \Phi.M(n') \qquad R' = \Phi.R[r \mapsto v]\end{array}}{\Phi(\!|r \leftarrow \texttt{load}_k\ e|\!)_p \rightsquigarrow \Phi^{++}[R := R']}$$

oMov
$$\frac{\langle v, p_e\rangle = \mathcal{V}_\Phi(e) \qquad p_e \sqsubseteq p}{\Phi(\!|sp \leftarrow \texttt{mov}\ e|\!)_p \rightsquigarrow \Phi^{++}[sp := v]}$$

oJmp
$$\frac{\begin{array}{c}\langle n, p_e\rangle = \mathcal{V}_\Phi(e) \qquad n' = k(n) \\ p_e \sqsubseteq p \qquad \text{in-same-func}(\Phi, \Phi.pc, n')\end{array}}{\Phi(\!|\texttt{jmp}_k\ e|\!)_p \rightsquigarrow \Phi[pc := n']}$$

oMovlabel
$$\frac{\begin{array}{c}p_r \not\sqsubseteq p' \implies p_r \sqsubseteq p \\ \langle n, p_r\rangle = \Phi.R(r) \qquad R' = \Phi.R[r := \langle n, p'\rangle]\end{array}}{\Phi(\!|r \leftarrow \texttt{movlabel}_{p'}|\!)_p \rightsquigarrow \Phi^{++}[R := R']}$$

oStorelabel
$$\frac{\begin{array}{c}\langle n, p_e\rangle = \mathcal{V}_\Phi(e) \qquad \langle m, p_m\rangle = \Phi.M(n) \qquad p_e \sqsubseteq p \\ M' = \Phi.M[n := \langle m, p'\rangle] \qquad p_m \not\sqsubseteq p' \implies p_m \sqsubseteq p\end{array}}{\Phi(\!|\texttt{storelabel}_{p'}\ e|\!)_p \rightsquigarrow \Phi^{++}[M := M']}$$

oSame
$$\frac{\begin{array}{c}\Phi.\Psi(\!|c|\!) \rightarrow \Psi' \\ \text{in-same-func}(\Phi, \Phi.\Psi.pc, \Psi'.pc)\end{array}}{\Phi(\!|c|\!) \rightsquigarrow \Phi[\Psi := \Psi']}$$

Figure 1.14. Operational semantics for oSFIasm, continued.

$$\frac{[SF] +\!\!+ _ = \Phi.stack \quad n \in S_p \implies n \geq SF.base \land n \neq SF.\mathit{ret\text{-}addr\text{-}loc}}{\text{writeable}(\Phi, n)}$$

$$\frac{F = \Phi.funcs(target) \quad F.entry = target \quad sp \in S_p \quad [SF] +\!\!+ _ = \Phi.stack \quad sp \geq SF.\mathit{ret\text{-}addr\text{-}loc} + F.type}{\text{typechecks}(\Phi, target, sp)}$$

$$\frac{[SF] +\!\!+ _ = \Phi.stack \quad \mathit{ret\text{-}addr\text{-}loc} = SF.\mathit{ret\text{-}addr\text{-}loc}}{\text{is-ret-addr-loc}(\Phi, \mathit{ret\text{-}addr\text{-}loc})}$$

$$\frac{[SF] +\!\!+ _ = \Phi.stack \quad \forall (r, n) \in SF.\mathit{csr\text{-}vals}.\ \Phi.R(r) = n}{\text{csr-restored}(\Phi)}$$

$$\frac{F \in \text{cod}(\Phi.funcs) \quad n, n' \in \text{dom}(F.instrs)}{\text{in-same-func}(\Phi, n, n')}$$

$$\frac{\forall F \in \text{cod}(\Phi.funcs).\ n \notin \text{dom}(F.instrs)}{\text{in-same-func}(\Phi, n, n')}$$

$$\frac{\forall i \in [1, n].\ \Phi.M(sp - i) = \langle _, \texttt{lib} \rangle}{\text{args-secure}(\Phi, sp, n)}$$

Figure 1.15. Operational semantics for oSFIasm: auxiliary predicates.

1.3.1 Overlay Monitor

oSFIasm enforces our zero-cost conditions by extending the operational semantics of SFIasm with additional checks in the overlay's small step operational semantics, written $\Phi \rightsquigarrow \Phi'$. Each of these steps is a refinement of the underlying SFIasm step, that is $\Phi.\Psi \to \Phi'.\Psi$ whenever Φ' is not `oerror`. Figures 1.13 and 1.14 (with auxiliary definitions shown in Figures 1.15 and 1.16) show the checks, which we describe below. We elide the rules for stepping to `oerror`: they are identical to the displayed rules except with one of the checks negated.

In the overlay, the reduction rule for library `call` instructions (oCall) checks type-safe execution with typechecks, a predicate over the state (Φ), call target (*target*), and stack pointer (*sp*) that checks that 1. the address we are jumping to is the entry instruction of one of the functions, 2. the stack pointer remains within the stack ($sp \in S_p$), and 3. the number of arguments expected by the callee have been pushed to the stack. On top of this, `call` also creates a new logical stack frame recording the base of the

$$\text{new-frame}(\Phi, \mathit{target}, \mathit{ret\text{-}loc}) \triangleq \{ \; \mathit{base} = \mathit{ret\text{-}loc} - \Phi.\mathit{funcs}(\mathit{target}).\mathit{type} \;\; \mathit{ret\text{-}loc} = \mathit{ret\text{-}loc} \;\; \mathit{csr\text{-}vals} = \{(r, \Phi.R(r))\}_{r \in \mathbb{CSR}} \; \}$$

$$\text{pop-frame}(\Phi) \triangleq \Phi[\mathit{stack} := S] \quad \text{where } [\mathit{SF}] +\!\!+ S = \Phi.\mathit{stack}$$

$$\Phi^{++} \triangleq \begin{cases} \Phi[\Psi := \Psi^{++}] & \text{in-same-func}(\Phi, \Phi.\mathit{pc}, \Phi.\mathit{pc} + 1) \\ \texttt{oerror} & \text{otherwise} \end{cases}$$

$$\text{classify}_{\mathbb{C}}(\Phi) \triangleq \Phi[M := M', R := R'] \quad \text{where } M'(n) = \langle \langle \Phi.M(n) \rangle, \mathbb{C}(\Phi.\Psi)(n) \rangle, \;\; R'(r) = \langle \langle \Phi.R(r) \rangle, \mathbb{C}(\Phi.\Psi)(r) \rangle$$

$$\begin{aligned}
\mathcal{V}_\Psi(v) &\triangleq v \\
\mathcal{V}_\Psi(r) &\triangleq \Psi.R(r) \\
\mathcal{V}_\Psi(\mathit{sp}) &\triangleq \langle \Psi.\mathit{sp}, \texttt{lib} \rangle \\
\mathcal{V}_\Psi(\mathit{pc}) &\triangleq \langle \Psi.\mathit{pc}, \texttt{lib} \rangle \\
\mathcal{V}_\Psi(e \oplus e') &\triangleq \langle v \oplus v', p \sqcup p' \rangle \\
&\quad \text{where } \langle v, p \rangle = \mathcal{V}_\Psi(e) \\
&\quad \phantom{\text{where }} \langle v', p' \rangle = \mathcal{V}_\Psi(e')
\end{aligned}$$

$$\langle n \rangle \triangleq \langle n, _ \rangle$$

$$\begin{aligned}
\texttt{app} \sqcup p &\triangleq \texttt{app} \\
p \sqcup \texttt{app} &\triangleq \texttt{app} \\
\texttt{lib} \sqcup \texttt{lib} &\triangleq \texttt{lib}
\end{aligned}$$

Figure 1.16. Operational semantics for oSFIasm: auxiliary definitions

new frame, location of the return address, and the current callee-save register values, pushing the new frame onto the overlay stack. To ensure IFC, we require that i has the label `lib` to ensure that control flow is not influenced by confidential values; a similar check is done when jumping within library code, obviating the need for a program counter label. Further, because the overlay captures zero-cost transitions, `gatecall` behaves in the exact same way except for an additional IFC check that the arguments are not influenced by confidential values.

Our zero-cost conditions rely on preventing invariants internal to a function from being interfered with by other functions. A key protection enabling this is illustrated by the reduction for `jmp` (oJmp), which enforces that the only inter-function control flow is via `call` and `ret`: the in-same-func predicate checks that the current (n) and target (n') instructions are within the same overlay function. The same check is added to the program counter increment operation, Φ^{++}. These checks ensure that the logical call stack corresponds to the actual control flow of the program, enabling the overlay stack's use in maintaining invariants at the level of function calls.

The reduction rule for `store` (oStore) demonstrates the other key protection enabling function local reasoning, with the check that the address (n) is writeable given the current state of the overlay stack. The predicate writeable guarantees that, if the operation is writing to the stack, then that write must be within the current frame and cannot be the location of the stored return address. This allows reasoning to be localized to each function: they do not need to worry about their callees tampering with their local variables. Protecting the stored return address is crucial for ensuring well-bracketing, which guarantees that each function returns to its caller.

To guarantee IFC, oStore first requires that the pointer have the label `lib`, ensuring that the location we write to is not based on confidential data. Second, the check $p_{i'} = \texttt{app} \implies n' \notin H_{\texttt{lib}}$ enforces that confidential values cannot be written to the library heap. Similar checks, based on standard IFC techniques, are implemented for all

other instructions.

With control flow checks and memory write checks in place, we guarantee that, when we reach a `ret` instruction, the logical call frame will correspond to the "actual" call frame. `ret` is then responsible for guaranteeing well-bracketing and ensuring callee-save registers are restored. This is handled by two extra conditions on `ret` instructions: is-ret-addr and csr-restored. csr-restored checks that callee-save registers have been properly restored by comparing against the values that were saved in the logical stack frame by `call`. is-ret-addr checks that the value pointed to by the stack pointer (*ret-addr*) corresponds to the location of the return address saved in the logical stack frame. Memory writes were checked to enforce that the return address cannot be overwritten, so this guarantees the function will return to the expected program location.

1.3.2 Overlay Semantics Enforce Security

The goal of the overlay semantics and our zero-cost conditions is to capture the essential behavior necessary to ensure that individual, well-behaved library functions can be composed together into a sandboxed library call that enforces SFI integrity and confidentiality properties. Thus, library code that is well-behaved under the dynamic overlay type system will behave equivalently to library code with springboard and trampoline wrappers, and therefore well-behaved library code can safely elide those wrappers and their overhead. We prove that the overlay semantics is sound with respect to each of our security properties:

Theorem 1 (Overlay Integrity Soundness). *If* $\Phi_0 \in \mathit{Program}$, $\Phi_0 \rightsquigarrow^n \Phi_1$, $\Phi_1(\!|_|\!)_{\mathsf{app}}$, *and* $\Phi_1 \rightsquigarrow^* \Phi_2$ *such that* $\Phi_1.\Psi \xrightarrow{wb} \Phi_2.\Psi$ *with* $\pi = \Phi_0.\Psi \rightarrow^n \Phi_1.\Psi$, *then*

1. $\mathcal{CSR}(\pi, \Phi_1.\Psi, \Phi_2.\Psi)$ *and*

2. $\mathcal{RA}(\pi, \Phi_1.\Psi, \Phi_2.\Psi)$.

Proof. By induction over the definition of a well-bracketed step and nested induction over the logical call stack. The last step follows by the fact that $\Phi_2 \neq \texttt{oerror}$, and therefore the restoration checks in the overlay monitor passed. □

Theorem 2 (Overlay Confidentiality Soundness)**.** *If* $\Phi_0 \in Program$, $\Phi_1(\!|_|\!)_{\texttt{lib}}$, $\Phi_3(\!|_|\!)_{\texttt{app}}$, $\Phi_0.\Psi \to^* \Phi_1.\Psi \xrightarrow{\texttt{lib}}{}^{n} \Phi_2.\Psi \to \Phi_3.\Psi$, $\Phi_1 \rightsquigarrow^{n+1} \Phi_3$, *and* $\Phi_1 =_{\texttt{lib}} \Phi'_1$, *then* $\Phi'_1.\Psi \xrightarrow{\texttt{lib}}{}^{n} \Phi'_2.\Psi \to \Phi'_3.\Psi$, $\Phi'_1 \rightsquigarrow^{n+1} \Phi'_3$ $\Phi'_3(\!|_|\!)_{\texttt{app}}$, $\Phi_3.pc = \Phi'_3.pc$, *and*

1. $\Phi_2(\!|\texttt{gatecall}_{n'}\ e|\!)$, $\Phi'_2(\!|\texttt{gatecall}_{n'}\ e|\!)$, *and* $\Phi_3 =_{\texttt{call}\ n'} \Phi'_3$ *or*
2. $\Phi_2(\!|\texttt{gateret}|\!)$, $\Phi'_2(\!|\texttt{gateret}|\!)$, *and* $\Phi_3 =_{\texttt{ret}} \Phi'_3$.

Proof. Proof is standard for an IFC enforcement system. □

1.4 Instantiating Zero-Cost

We describe two isolation systems that securely support zero-cost transitions: that is, they meet the overlay monitor zero-cost conditions. The first, described in Section 1.4.1, is an SFI system using WebAssembly as an IR before compiling to native code using the Lucet toolchain [McMullen 2020]. Here we rely on the language-level invariants of WebAssembly to satisfy our zero-cost requirements. To ensure that these invariants are maintained, in Section 1.5 we describe a verifier, VeriZero, that checks that compiled binaries meet the zero-cost conditions. In Section 1.5.4 we outline our proof that the verifier guarantees that compiled WebAssembly can safely elide springboards and trampolines.

The second system, `SegmentZero32`, is our novel SFI system combining the x86 segmented memory model for memory isolation with several security-hardening LLVM compiler passes to enforce our zero-cost conditions. While WebAssembly meets the zero-cost conditions, it imposes additional restrictions that lead to unrelated slowdowns. `SegmentZero32` thus serves as a platform for evaluating the potential cost of enforcing

the zero-cost conditions directly as well as a proof-of-concept SFI implementation designed using the zero-cost framework.

1.4.1 WebAssembly

WebAssembly is a low-level bytecode with a sound, static type system. WebAssembly's abstract state includes global variables and heap memory, which are zero-initialized at start-up. All heap accesses are explicitly bounds checked, meaning that compiled WebAssembly programs inherently implement heap isolation. Beyond this, WebAssembly programs enjoy several language-level properties, which ensure compiled binaries satisfying the zero-cost conditions. We describe these below.

- **Control flow:** There are no arbitrary jump instructions in WebAssembly, only structured intra-function control flow. Functions may only be entered through a call instruction, and may only be exited by executing a return instruction. Functions also have an associated type; direct calls are type-checked at compile time while indirect calls are subject to a runtime type check. This ensures that compiled WebAssembly meets our type-directed forward-edge CFI condition.

- **Protecting the stack:** A WebAssembly function's type precisely describes the space required to allocate the function's stack frame (including spilled registers). All accesses to local variables and arguments are performed through statically known offsets from the current stack base. It is therefore impossible for a WebAssembly operation to access other stack frames or alter the saved return address. This ensures that compiled WebAssembly meets our local state encapsulation condition, and, in combination with type-checking function calls, guarantees that WebAssembly's control-flow is well-bracketed. We therefore know that compiled WebAssembly functions will always execute the register-saving preamble and, upon termination, will execute the register-restoring epilogue. Further, the func-

tion body will not alter the values of any registers saved to the stack, thereby ensuring restoration of callee-save registers.

- **Confidentiality:** WebAssembly code may store values into function-local variables or a function-local "value stack" similar to that of the Java Virtual Machine [Oracle 2019]. The WebAssembly spec requires that compilers initialize function-local variables either with a function argument or with a default value. Further, accesses to the WebAssembly value stack are governed by a coarse-grained data-flow type system, with explicit annotations at control flow joins. These are used to check at compile-time that an instruction cannot pop a value from the stack unless a corresponding value was pushed earlier in the same function. This guarantees that local variable and value stack accesses can be compiled to register accesses or accesses to a statically-known offset in the stack frame.

When executing a compiled WebAssembly function without heavyweight transitions, confidential values from prior computations may linger in these spilled registers or parts of the stack. However, the above checks ensure that these locations will only be read if they have been previously overwritten during execution of the same function by a low-confidentiality WebAssembly library value.

1.4.2 SegmentZero32

To demonstrate that zero-cost conditions can be applied outside of highly structured languages such as WebAssembly, we demonstrate their enforcement in our novel SFI system for C code called `SegmentZero32`. As we mention in Section 1.1.3, our zero-cost conditions amalgamate a number of individual conditions which separately have well-studied enforcement mechanisms, and so we are able to compose a series of off-the-shelf Clang/LLVM security-hardening passes to form the core of `SegmentZero32`. The memory bounds checks are performed using the x86 segmented memory model [Intel

Corporation 2023] (Similar to NaCl [Yee et al. 2009], however we use an additional segment to separate the sandboxed heap and stack).

Since `SegmentZero32` directly enforces the structure required for zero-cost transitions on C code (rather than relying on WebAssembly as an IR), it allows us to investigate the intrinsic cost of enforcing zero-cost (see Section 1.6.3), without suffering from irrelevant WebAssembly overheads. We additionally compare `SegmentZero32` against NaCl's 32-bit SFI scheme for the x86 architecture, which we believe is the fastest production-quality SFI toolchain currently available. Below we discuss specific details of `SegmentZero32` zero-cost condition enforcement.

- **Protecting the stack:** We apply the SafeStack [Kuznetsov et al. 2014; The LLVM Foundation 2021b] compiler pass to further split the sandboxed stack into a safe and unsafe stack. The safe stack contains only data that the compiler can statically verify is always accessed safely, e.g., return addresses, spilled registers, and allocations that are only accessed locally using verifiably safe offsets within the function that allocates them.[4] All other stack values are moved to the heap segment. This ensures that pointer manipulation of unsafe stack references cannot be used to corrupt the return address and saved context of the current call. We write a small LLVM pass to add additional support for tracking whether an access must be made through the heap segment or the stack segment, ensuring correct code generation.

  These transformations ensure that malicious code cannot programmatically access anything stored in the stack segment, except through offsets statically determined to be safe by the SafeStack pass. This protects the stored callee-save registers and return address, guaranteeing the restoration of callee-save registers and well-bracketing *iff forward control flow is enforced*.

[4] We also use LLVM's stack-heap clash detection (-fstack-clash-protection) to prevent the stack growing into the heap.

- **Control flow:** Fortunately, enforcing forward-edge CFI has been widely studied [Burow, Carr, et al. 2017]. We use a CFI pass as implemented in Clang/LLVM [The LLVM Foundation 2021a; Tice et al. 2014] including flags to dynamically protect indirect function calls, ensuring forward control flow integrity. Further, `SegmentZero32` conservatively bans non-local control flow (e.g. `setjmp/longjmp`) in the C source code. A more permissive approach is possible, but we leave this for future work.

- **Confidentiality:** To guarantee confidentiality we implement a small change in Clang to zero initialize all stack variables.[5] This ensures that scratch registers cannot leak secrets as all sandbox values are semantically written before use. In practice, many of these writes are statically known to be dead and therefore optimized away.

1.5 Verifying Compiled WebAssembly

Instead of trusting the WebAssembly compiler, we build a *zero-cost verifier*, VeriZero, to check that the native, compiled output meets the zero-cost conditions and is thus safe to run without springboards and trampolines. VeriZero is a static x86 assembly analyzer that takes as input potentially untrusted native programs and verifies a series of local properties via abstract interpretation. Together these local properties guarantee that the monitor checks defined in oSFIasm are met; we discuss the proof of soundness in Section 1.5.4.

VeriZero extends the VeriWasm SFI verifier [Johnson et al. 2021]. Both operate over WebAssembly modules compiled by the Lucet WebAssembly compiler [McMullen 2020], first disassembling the native x86 code before computing a control-flow graph (control flow graph (CFG)) for each function in the binary. The disassembled code is

[5] We can't use Clang's existing pass for variable initialization [Bastien 2018] as it zero initializes data on the unsafe stack leading to poor performance.

```
1  bad_func: [] → rax                                good_func: [rdi] → rax
2    push r12                                          mov rax ← rdi
3    ; TRACK: stack[0] = initial r12 value             ret
4    mov r12 ← 1
5    ; TRACK: r12 initialized
6    mov r11 ← r13 + r12
7    ; TRACK: r11 uninitialized
8    mov rdi ← 2
9    ; TRACK: rdi initialized
10   ; ASSERT: good_func arguments initialized
11   call good_func
12   ; TRACK: good_func return value initialized
13   pop r12
14   ; TRACK: r12 = initial r12 value
15   ; ASSERT: callee-save registers restored
16   gateret
17
```

Figure 1.17. Disassembled and lifted WebAssembly functions.

then lifted to a subset of SFIasm, which serves as the first abstract domain in our analysis. Unfortunately, the properties checked by VeriWasm, while sufficient to guarantee SFI security, are insufficient to guarantee zero-cost security. Below we will describe how VeriZero extends VeriWasm to guarantee the stronger zero-cost conditions are met.

1.5.1 The VeriZero Analyzers

VeriZero adds two new analyses to VeriWasm. The first extends VeriWasm's CFI analysis, which only captures coarse grained control-flow (i.e., that all calls target valid sandboxed functions), to also extract type information. Extracting type information from the binary code is possible without any complex type inference because Lucet leaves the type signatures in the compiled output (though we do not need to trust Lucet to get these type signatures correct since VeriZero would catch any deviations at the binary level). For direct calls, VeriZero simply extracts the WebAssembly type stored in the binary. For indirect calls we extend the VeriWasm indirect call analysis to track the type of each indirect call table entry, enabling us to resolve each indirect call to a

statically known type. These types correspond to the input registers and stack slots, and the output registers (if any) used by a function. For example, in Figure 1.17 `bad_func` takes no input and outputs to `rax` and `good_func` takes `rdi` as input and outputs to `rax`.

The second analysis tracks dataflow in local variables, i.e., in registers and stack slots. Continuing with `bad_func` as our example this analysis captures that: in Line 2 stack slot 0 now holds the initial value of `r12`, in Line 4 `r12` holds an initialized (and therefore public) value, in Line 6 `r13` has not been initialized and therefore potentially contains confidential data so `r11` may also contain confidential data, etc. This analysis is used to check confidentiality, callee-save register restoration, local state encapsulation, and is combined with the previous analysis to check type-directed CFI.

1.5.2 The Dataflow Abstract Domain

To track local variable dataflow, VeriZero uses an abstract domain with three elements: `Uninitialized` which represents an uninitialized, potentially confidential value; `Initialized` which represents an initialized, public value; and `UninitializedCallee`(r) which represents a potentially confidential value which corresponds to the original value of the callee-save register r. The domain forms a meet-semilattice with `Uninitialized` the least element and all other elements incomparable.

From here, analysis is straightforward, with a function's argument registers and stack slots initialized to `Initialized`, each callee-save register r initialized to `UninitializedCallee`(r), and everything else `Uninitialized`. Instructions are interpreted as expected, e.g., `mov` simply copies the abstract value of its source into the target, operations return the meet of their operands, and all constants and reads from the heap are treated as initialized. Across calls we assume that callee-save register conventions are followed (as we will be checking this), preserving the value of all callee-save registers and clearing all other registers' values. We extract the type information from the

extended CFI analysis to determine the return register that is initialized after a function call.

1.5.3 Checking the Zero-Cost Conditions

The above two analyses, along with additional information from VeriWasm's existing analyses enable us to check the zero-cost conditions.

1. **Callee-save register restoration:** The `UninitializedCallee`(r) value enables straightforward checking that callee-save registers have been restored by checking that, at each `ret` instruction, each callee-save register r has the abstract value `UninitializedCallee`(r).

2. **Well-bracketed control-flow:** VeriWasm already implements a stack checker that guarantees that all writes to the stack are to local variables, ensuring that the saved return address on the stack cannot be tampered with. Further, it checks that the stack pointer is restored to its original location at the end of every function, ensuring the saved return address is used.

3. **Type-directed forward-edge CFI:** The dataflow analysis gives us the registers that are initialized when we reach a `call` instruction, enabling us to check that the input arguments of the target have been initialized. For example, when we reach Line 11 we know that `rdi` has the value `Initialized`. The type-based CFI analysis tells us that `good_func` expects `rdi` as an input, so this call is marked as safe.

4. **Local state encapsulation:** To ensure SFI security, VeriWasm checks that no writes are below the current stack frame, ensuring that verified WebAssembly functions cannot tamper with other frames.

5. **Confidentiality:** We check confidentiality using the information obtained in

$$
\begin{array}{rrcll}
\textit{Library} & L & ::= & \{ & \textit{interface} : \mathbb{N} \rightharpoonup \mathbb{N} \\
 & & & & \textit{library} : \mathbb{N} \rightharpoonup \textit{WasmFunction} \\
 & & & & \textit{code} : \textit{Code} \quad \} \\
\textit{WasmFunction} & \textit{WF} & ::= & \{ & |\textit{args}| : \mathbb{N} \\
 & & & & |\textit{locals}| : \mathbb{N} \\
 & & & & \textit{entry} : \textit{Block} \\
 & & & & \textit{exits} : \wp(\textit{Block}) \\
 & & & & \textit{blocks} : \wp(\textit{Block}) \quad \} \\
\textit{Block} & B & ::= & \{ & \textit{start} : \mathbb{N} \\
 & & & & \textit{end} : \mathbb{N} \\
 & & & & |\textit{slots}| : \mathbb{N} \\
 & & & & \textit{inputs} : \wp(\mathbb{N} + \textit{Reg}) \\
 & & & & \textit{indirects} : \wp(\textit{IndirectBlock}) \quad \} \\
\textit{IndirectBlock} & \textit{IB} & ::= & \{ & \textit{start} : \mathbb{N} \\
 & & & & \textit{end} : \mathbb{N} \\
 & & & & \textit{parent} : \textit{Block} \quad \}
\end{array}
$$

Figure 1.18. Structure of WebAssembly libraries.

our dataflow analysis, where the value `Initialized` ensures that a value is initialized with a public, non-confidential value. This enables us to check each of the confidentiality checks encoded in oSFIasm are met: for instance the type-safe forward-edge CFI check described above already ensures each argument is initialized. In Figure 1.17, the confidentiality checker will flag Line 6 as unsafe because `r13` still has the value `UninitializedCallee(r13)`, which potentially contains confidential information leaked from the application.

1.5.4 Proving WebAssembly Secure

We prove that compiled and verified WebAssembly libraries can safely elide springboards and trampolines while maintaining integrity and confidentiality, by showing that the verified code would not violate the safety monitor. Formally, this amounts to showing that WebAssembly code verified by VeriZero never reaches an

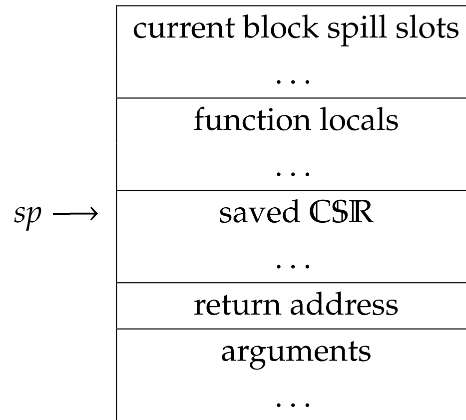


Figure 1.19. Layout of WebAssembly stack frame.

oerror state. This then allows us to apply Theorem 1 and Theorem 2 to derive that verified WebAssembly code is secure.

We first describe the structure of WebAssembly libraries, with syntax and stack structure shown in Figures 1.18 and 1.19 respectively. We split libraries into a collection of functions (*WF*) with statically defined arguments, local variables, and blocks (with entry and exit blocks separately identified). Blocks are similarly structured with start and end locations identified as well as identifying the inputs and slots used. Blocks my make an indirect jump at their end in which case they carry a set of indirect blocks (*IB*) that they may jump to. The structure of the control stack is standard, we show it in Figure 1.19 as the details are key in our proof.

It is relatively straightforward (with one exception) to prove that the abstract interpretation as described guarantees the necessary safety conditions. The crucial exception in the soundness proof is when a function calls to other WebAssembly functions. We must inductively assume that the called function is safe, i.e., doesn't change any variables in our stack frame, restores callee-save registers, etc. Unfortunately, a naive attempt does not lead to an inductively well-founded argument. Instead, we use the overlay monitor's notion of a well-behaved function to define a step-indexed logical relation (detailed in Figures 1.21 and 1.20) that captures a semantic notion of

$$
\begin{array}{rcl}
\text{World} & \triangleq & \mathbb{N} \times (\mathbb{N} \rightharpoonup \mathbb{N}) \times (\mathbb{N} \rightharpoonup \mathbb{N}) \\
\\
\triangleright & : & \text{World} \to \text{World} \\
\triangleright(0, \overline{f_i}, \overline{f_l}) & \triangleq & (0, \overline{f_i}, \overline{f_l}) \\
\triangleright(n, \overline{f_i}, \overline{f_l}) & \triangleq & (n-1, \overline{f_i}, \overline{f_l}) \\
\\
(C, \overline{F_2}, \overline{F_3}) :_W & \triangleq & \text{for } i \in \{1, 2\} : \\
& & \overline{F_i.entry} = \text{dom}(\pi_i(W)) \\
& & \forall F_i \in \overline{F_i}.\ F_i.type = \pi_i(W)(F_i.entry) \\
& & \forall F_i \in \overline{F_i}.\ (\triangleright W, F_i, C|_{F_i.instrs}) \in \mathcal{F}
\end{array}
$$

Figure 1.20. Definition of Kripke worlds for WebAssembly logical relation.

well-behaved functions (as a relation $\mathcal{F}$), and then lifts this to a relation over an entire WebAssembly library (as a relation $\mathcal{L}$). This gives a basis for an inductively well-founded argument where we can prove that, locally, the abstract interpretation gives that each WebAssembly function is semantically well-behaved (is in $\mathcal{F}$) and then use this to prove the standard fundamental theorem of a logical relation for a whole WebAssembly library:

Theorem 3 (Fundamental Theorem for WebAssembly Libraries)**.** *For any number of steps $n \in \mathbb{N}$ and compiled WebAssembly library L, $(n, L) \in \mathcal{L}$.*

This theorem states that every function in a compiled WebAssembly library, when making calls to other WebAssembly functions or application callbacks, is well-behaved with respect to the zero-cost conditions. The number of steps is a technical detail related to step-indexing. Transition security then follows by adequacy of the logical relation, Theorem 1, and Theorem 2:

Theorem 4 (Adequacy of WebAssembly Logical Relation)**.** *For any number of steps $n \in \mathbb{N}$, library L such that $(n, L) \in \mathcal{L}$, program $\Phi_0 \in Program$ using L, and $n' \leq n$, if $\Phi_0 \leadsto^{n'} \Phi'$ then $\Phi' \neq \texttt{oerror}$.*

$$
\mathcal{F} \triangleq \left\{ (W, F, c) \;\middle|\; \begin{array}{l}
\forall \rho \in [\mathit{Frame}], \mathit{ret\text{-}addr}, A \in [\mathbb{N}], \mathit{sp}, R, M, C, \overline{F_i}, \overline{F_l}. \\
\quad \text{let } \rho' = \\
\qquad \rho +\!\!+ \{\mathit{base} := \mathit{sp} - |A|, \mathit{ret\text{-}addr\text{-}loc} := \mathit{sp}, \mathit{csr\text{-}vals} := R(\mathbb{CSR})\} \\
\quad |A| = F.\mathit{type} \\
\quad \mathit{sp} > \mathrm{top}(\rho).\mathit{ret\text{-}addr\text{-}loc} + |A| \\
\quad [\mathit{sp} \mapsto \mathit{ret\text{-}addr}, (\mathit{sp} - |A| + i \mapsto A_i)_{i \in [0,|A|)}] \leq M \\
\quad (C, \overline{F_i}, \overline{F_l}) :_W \\
\quad c \leq C \\
\quad \mathrm{dom}(c) = F.\mathit{instrs} \\
\quad \forall \ell \in \mathrm{dom}(M) \cap M_{\mathtt{lib}}.\ \langle n, \mathtt{lib} \rangle = M(\ell) \\
\quad \text{let } \Psi = \{\mathit{pc} := F.\mathit{entry}, \mathit{sp} := \mathit{sp}, R := R, M := M, C := C\} \\
\quad \text{let } \Phi = \{\Psi := \Psi, \mathit{stack} := \rho', \mathit{funcs} := [F'.\mathit{entry} \mapsto F']_{F' \in \overline{F_i} \uplus \overline{F_l}}\} \\
\Longrightarrow \\
\quad \forall n' \leq W.n.\ \Phi \overset{\rho'}{\rightsquigarrow}{}^{n'}\ \Phi' \Longrightarrow \Phi' \neq \mathtt{oerror} \\
\quad \text{or } \exists n' \leq W.n.\ \Phi \overset{\rho'}{\rightsquigarrow}{}^{n'-1}\ \Phi' \rightsquigarrow \Phi'' \\
\qquad \text{where } (\Phi'(\!|\mathtt{ret}|\!) \vee \\
\qquad\qquad \Phi'(\!|\mathtt{gateret}|\!)) \wedge \Phi'.\mathit{stack} = \rho' \wedge \Phi'' \neq \mathtt{oerror}
\end{array} \right\}
$$

$$
\mathcal{L} \triangleq \left\{ (n, L) \;\middle|\; \begin{array}{l}
\forall i \in \mathrm{dom}(L.\mathit{library}). \\
\quad \text{let } \mathit{WF} = L.\mathit{library}(i) \\
\quad \text{let } W = (n, L.\mathit{interface}, \lambda i \to L.\mathit{library}(i).|\mathit{args}|) \\
\quad \text{let } \mathit{instrs} = \biguplus_{B \in \mathit{WF.blocks}} [B.\mathit{start}, B.\mathit{end}] \\
\quad \text{let } F = \{\mathit{instrs} := \mathit{instrs}, \mathit{entry} := \mathit{WF.entry.start}, \mathit{type} := \mathit{WF}.|\mathit{args}|\} \\
\quad (W, F, L.\mathit{code}|_{\mathit{instrs}}) \in \mathcal{F}
\end{array} \right\}
$$

Figure 1.21. Logical relation for zero-cost WebAssembly.

Details of the proofs are in Appendix A.2.

1.6 Evaluation

We evaluate our zero-cost model by asking four questions:

Q1: What is the cost of a context switch? (§ 1.6.1)

Q2: What is end-to-end performance gain of WebAssembly-based SFI due to zero-cost transitions? (§ 1.6.2)

Q3: What is the performance overhead of purpose-built zero-cost SFI enforcement? (§ 1.6.3)

Q4: Is the VeriZero verifier effective? (§ 1.6.4)

Since our zero-cost condition enforcement does incur some runtime overhead, **Q2** and **Q3** are heavily workload-dependent. The benefit a workload receives from the zero-cost approach will be in direct proportion to the frequency with which it performs domain transitions.

Systems

To investigate the first three questions, we consider two groups of SFI systems. The first group compares a number of different transition models for WebAssembly-based SFI for 64-bit binaries, built on top of the Lucet compiler [McMullen 2020]. All of these will have identical runtime overhead, meaning that the only variance between them will be due to transition overhead. The `WasmLucet` build uses the original heavyweight springboards and trampolines shipped with the Lucet runtime written in Rust. `WasmHeavy` adopts techniques from NaCl and uses optimized assembly to save and restore application context during transitions. `WasmZero` implements our zero-cost transition system, meaning transitions are simple function calls. To

understand the overhead of register saving/restoring and stack switching, we also evaluate a `WasmReg` build which saves/restores registers like `WasmHeavy`, but shares the library and application stack like `WasmZero`.

The second group compares optimized SFI techniques for 32-bit binaries. WebAssembly based SFI imposes overheads far beyond what is strictly necessary to enforce our zero-cost conditions, both because of the immaturity of the Lucet compiler in comparison to more established compilers such as Clang, and because WebAssembly inherently enforces additional restrictions on compiled code (e.g., structured intra-function control flow). We design `SegmentZero32` (see Section 1.4.2) to enforce only our zero-cost-conditions and nothing more, aiming to benchmark it against the NaCl 32-bit isolation scheme (`NaCl32`) [Yee et al. 2009], arguably the fastest production SFI system available, which requires heavyweight transitions. Both systems make use of memory segmentation, a 32-bit x86-only feature for fast memory isolation. Unfortunately, we cannot make a uniform comparison between `NaCl32`, `SegmentZero32`, and `WasmZero` since Lucet only supports a 64-bit target.

Each group additionally uses unsandboxed, insecure native execution (`Vanilla`) as a baseline. To represent the best possible performance of schemes relying on heavyweight transitions, we also benchmark `IdealHeavy32` and `IdealHeavy64`, ideal hardware isolation schemes, which incur no runtime overhead but require heavyweight transitions. To simulate the performance of these ideal schemes, we simply measure the performance of native code with heavyweight trampolines.

We integrate all of the above SFI schemes into Firefox using the RLBox framework [Narayan, Disselkoen, Garfinkel, et al. 2020]. Since RLBox already provides plugins for the `WasmLucet` and `NaCl32` builds, we only implement the plugins for the remaining system builds.

Benchmarks

We use a micro-benchmark to evaluate the cost of a single transition for our different transition models, using unsandboxed native calls as a baseline (**Q1**). We answer questions **Q2**–**Q3** by measuring the end-to-end performance of font and image rendering in Firefox, using a sandboxed `libgraphite` and `libjpeg`, respectively. We use these libraries because they have many cross-sandbox transitions, which Narayan, Disselkoen, Garfinkel, et al. [2020] previously observed to affect the overall browser performance. To evaluate the performance of `libgraphite`, we use Kew's benchmark[6], which reflows the text on a page ten times, adjusting the size of the fonts each time to negate the effects of font caches. When calling `libgraphite`, Firefox makes a number of calls into the sandbox roughly proportional to the number of glyphs on the page. We run this benchmark 100 times and report the median execution time below (all values have standard deviations within 1%).

To evaluate the performance of `libjpeg`, we measure the overhead of rendering images of varying complexity and size. Since the work done by the sandboxed `libjpeg` per call is proportional to the width of the image (Firefox executes the library in *streaming mode*, one row at a time) we consider images of different widths, keeping the image height fixed. This allows us to understand the benefits and limitations of zero-cost transitions, since the proportion of execution time spent context-switching decreases as the image width increases. We do this for three images, of varying complexity: a simple image consisting of a single color (`SimpleImage`), a stock image from the Image Compression benchmark suite[7] (`StockImage`), and an image of random pixels (`RandomImage`). We render each image 500 times and report the median time (standard deviations are all under 1%).

Finally, we use SPEC CPU® 2006 to partly evaluate the sandboxing overhead

[6]Available at https://jfkthame.github.io/test/udhr_urd.html
[7]Online: https://imagecompression.info/test_images/. Visited Dec 9, 2020.

of our purpose-built `SegmentZero32` SFI system (**Q3**), and to measure VeriZero's verification speed (**Q4**).

Machine and Software Setup

We run all but the verification benchmarks on an Intel® Core™ i7-6700K machine with four 4GHz cores, 64GB RAM, running Ubuntu 20.04.1 LTS (kernel version 5.4.0-58). We run benchmarks with a shielded isolated cpuset [Tsariounov 2021] consisting of one core with hyperthreading disabled and the clock frequency pinned to 2.2GHz. We generate WebAssembly sandboxed code in two steps: First, we compile C/++to WebAssembly using Clang-11, and then compile WebAssembly to native code using the fork of the Lucet used by RLBox (snapshot from Dec 9, 2020). We generate NaCl sandboxed code using a modified version of Clang-4. We compile all other C/++source code, including `SegmentZero32` sandboxed code and benchmarks using Clang-11. We implement our Firefox benchmarks on top of Firefox Nightly (from August 22, 2020).

Summary of Results

We find that the performance of WebAssembly-based isolation can be significantly improved by adopting zero-cost transitions, but that Lucet-compiled WebAssembly's runtime overhead means that it does not outperform more optimized isolation schemes in end-to-end tests. The low performance overhead of `SegmentZero32` demonstrates that these runtime overheads are not inherent to the zero-cost approach, and that an optimized zero-cost SFI system can significantly outperform more traditional schemes, especially for workloads with a large number of transitions. Finally, we find that we can efficiently check zero-cost conditions at the binary level, for Lucet compiled code, with no false positives.

Table 1.1. Costs of transitions in different isolation models. Zero-cost transitions are shown in **boldface**. `Vanilla` is the performance of an unsandboxed C function call, to serve as a baseline.

Build	Direct call	Indirect call	Callback	Syscall
`Vanilla` (in C)	1ns	56ns	56ns	24ns
`WasmLucet`	—	1137ns	—	—
`WasmHeavy`	120ns	209ns	172ns	192ns
`WasmReg`	120ns	210ns	172ns	192ns
`WasmZero`	**7ns**	**66ns**	**67ns**	**60ns**
`Vanilla` (in C, 32-bit)	1ns	74ns	74ns	37ns
`NaCl32`	—	714ns	373ns	356ns
`SegmentZero32`	**24ns**	**108ns**	**80ns**	**88ns**

1.6.1 The Cost of Transitions

We measure the cost of different cross-domain transitions, direct and indirect calls into the sandbox, callbacks from the sandbox, and syscall invocations from the sandbox, for the different system builds described above. To expose overheads fully we choose extremely fast payloads: either a function that just adds two numbers or the `gettimeofday` syscall, which relies on Linux's vDSO to avoid CPU ring changes. The results are shown in Table 1.1. All numbers are averages of one million repetitions, and repeated runs have negligible standard deviation.[8]

We make several observations. First, among WebAssembly-based SFI schemes, zero-cost transitions (`WasmZero`) are significantly faster than even optimized heavy-weight transitions (`WasmHeavy`). Lucet's existing indirect calls (`WasmLucet`) which are written in Rust are significantly slower than both. Second, the cost of stack switching (the difference of `WasmHeavy` and `WasmReg`) is surprisingly negligible. Third, the performance of `Vanilla` and `WasmZero` should be identical but is not. This is *not*

[8]Lucet and NaCl don't support direct sandbox calls; Lucet further does not support custom callbacks or syscall invocations.

because our transitions have a hidden cost. Rather, it's because we are comparing code produced by two different compilers: `Vanilla` is native code produced by Clang, while `WasmZero` is code produced by Lucet, and Lucet's code generation is not yet highly optimized [Hansen 2019]. For example, in the benchmark that adds two numbers, Clang eliminates the function prologue and epilogue that save and restore the frame pointer, while Lucet does not. We observe similar trends for hardware-based isolation. For example, we find that `SegmentZero32` transitions are much faster than `IdealHeavy32` and `NaCl32` transitions and only 23ns slower than `Vanilla` for direct calls. Finally, we observe that `SegmentZero32` is slower than `WasmZero`: hardware isolation schemes like `SegmentZero32` and `NaCl32` execute instructions to enable or disable the hardware based memory isolation in their transitions.

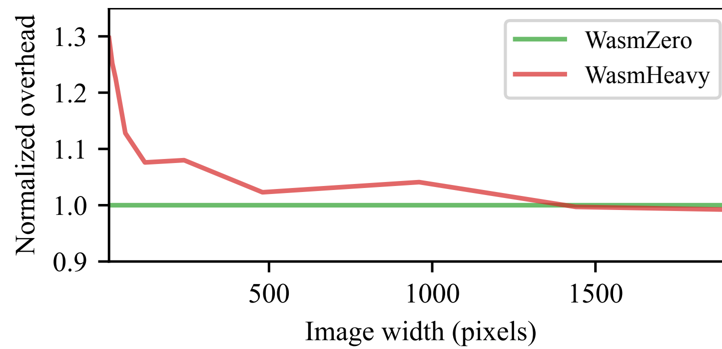
1.6.2 End-to-End Performance Improvements of Zero-Cost Transitions for WebAssembly

We evaluate the end-to-end performance impact of the different transition models on WebAssembly-sandboxed font and image rendering as used in Firefox (see Section 1.6).

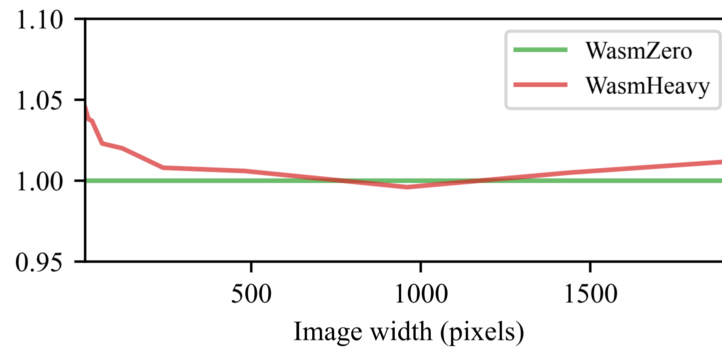
Font Rendering

We report the performance of `libgraphite` isolated with WebAssembly-based schemes on Kew's in Table 1.2. As expected, WebAssembly with zero-cost transitions (`WasmZero`) outperforms the other WebAssembly-based SFI transition models. Compared to `WasmZero`, Lucet's existing transitions slow down rendering by over 4×.[9] But, even the optimized heavyweight transitions (`WasmHeavy`) impose a 10% performance tax. This overhead is due to register saving/restoring; stack switching only accounts for 0.8% overhead.

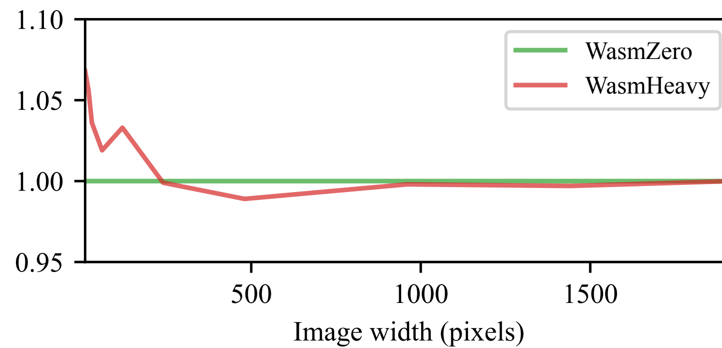
[9] This overhead is smaller than the 8× overhead reported by Narayan, Disselkoen, Garfinkel, et al. [2020]. We attribute this difference to the different compilers: we use a more recent, and faster, version of Lucet.



(a) `SimpleImage`



(b) `StockImage`



(c) `RandomImage`

Figure 1.22. Performance of different WebAssembly transitions on rendering of (a) a simple image with one color, (b) a stock image, and (c) a complex image with random pixels, normalized to `WasmZero`. `WasmZero` transitions outperform other transitions. The difference diminishes with width, but narrower images are more common on the web.

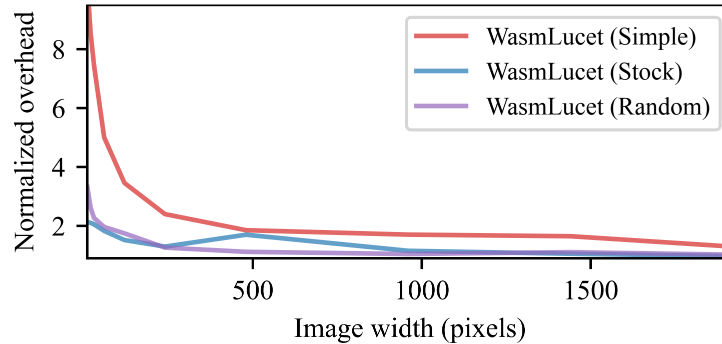


Figure 1.23. Performance of the `WasmLucet` heavyweight transitions included in the Lucet runtime on the image benchmarks. Performance when rendering: 1. a simple image with one color, 2. a stock image, and 3. a complex image with random pixels. The performance is the overhead compared to `WasmZero`.

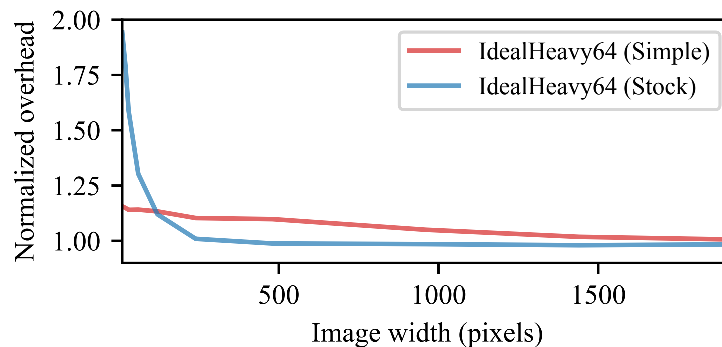


Figure 1.24. Performance of an ideal isolation scheme (no enforcement overhead) with heavy trampolines when rendering images. WebAssembly compilers whose enforcement overhead is lower than this can outperform even an ideal isolation scheme that uses heavy weight transitions.

Table 1.2. Costs of font rendering in different isolation models. Zero-cost transitions are shown in **boldface**. `Vanilla` is the performance of an unsandboxed C function call, to serve as a baseline.

	Font render
`WasmLucet`	8173ms
`WasmHeavy`	2246ms
`WasmReg`	2230ms
`WasmZero`	2032ms
`IdealHeavy64`	1563ms
`Vanilla`	1116ms

While these results show that existing WebAssembly-based isolation schemes can benefit from switching to zero-cost transitions (and indeed the speed-up due to zero-cost transitions allowed Mozilla to ship the WebAssembly-sandboxed `libgraphite`) they also show that Lucet-compiled WebAssembly is slow (~80% slower than Vanilla). This, unfortunately, means that the transition cost savings alone are not enough to beat `IdealHeavy64`, even for a workload with many transitions. To compete with this ideal SFI scheme with heavyweight transitions, we would need to reduce the runtime overhead to ~40%. Jangda et al. [2019] report the average runtime overhead of Mozilla SpiderMonkey JIT-compiled WebAssembly compared to native as ~45% in a different set of benchmarks, while noting many correctable inefficiencies in the generated assembly code, suggesting that there is a lot of room for Lucet to be further optimized.

Image Rendering

Figure 1.22 reports the overhead of WebAssembly-based sandboxing on image rendering, normalized to `WasmZero` to highlight the relative overheads of different transitions as compared to our zero-cost transitions. We report results of `WasmLucet` separately, in Figure 1.23 because the rendering times are up to 9.2× longer than the other builds. Here, we instead focus on evaluating the overheads of optimized heavy

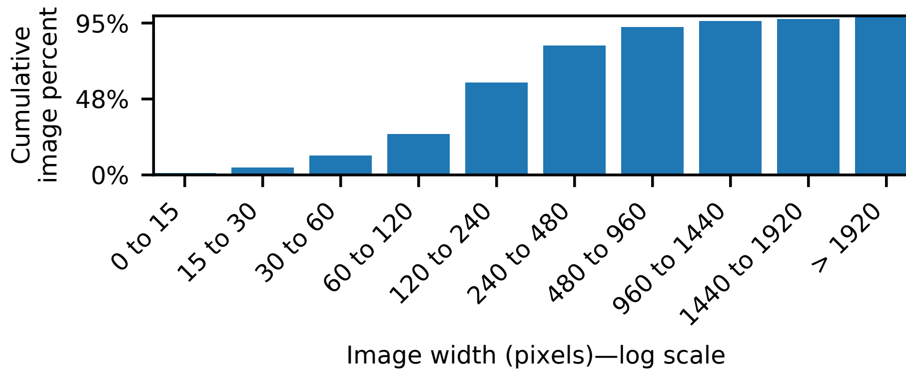


Figure 1.25. Cumulative distribution of image widths on the landing pages of the Alexa top 500 websites. Over 80% of the images have widths under 480 pixels. Narrower images have a higher transition rate, and thus higher relative overheads when using expensive transitions.

transitions.

As expected, `WasmZero` significantly outperforms other transitions when images are narrower and simpler. On `SimpleImage`, `WasmHeavy` and `WasmLucet` can take as much as 29.7% and 9.2× longer to render the image as with `WasmZero` transitions. However, this performance gap diminishes as image width increases (and the relative cost of context switching decreases). For `StockImage` and `RandomImage`, the `WasmHeavy` trends are similar, but the rendering time differences start at about 4.5%. Lucet's existing transitions (`WasmLucet`) are still significantly slower than zero-cost transitions (`WasmZero`) even on wide images.

Though the differences between the transitions are smaller as the image width increases, many images on the Web are narrow. Figure 1.25 shows the distribution of images on the landing pages of the Alexa top 500 websites. Of the 10.6K images, 8.6K (over 80%) have widths between 1 and 480 pixels, a range in which zero-cost transitions noticeably outperform the other kinds of transitions.

Like font rendering, we measure the target runtime overhead Lucet should achieve to beat `IdealHeavy64` end-to-end for rendering images. We report our results in Figure 1.24. For the small simple image, we observe this to be 94%: this is

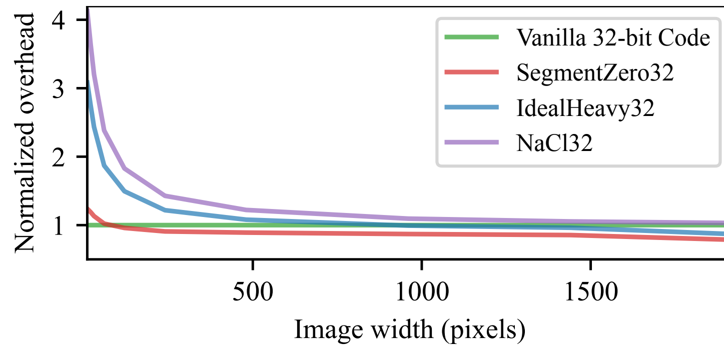
approximately the overhead of Lucet that we see already today. For the small stock image, we observe this to be 15%: this is much smaller than the overhead of Lucet today, but lower overheads have been demonstrated on some benchmarks by the prototype WebAssembly compiler of Gadepalli et al. [2020].

1.6.3 Performance Overhead of Purpose-Built Zero-Cost SFI Enforcement

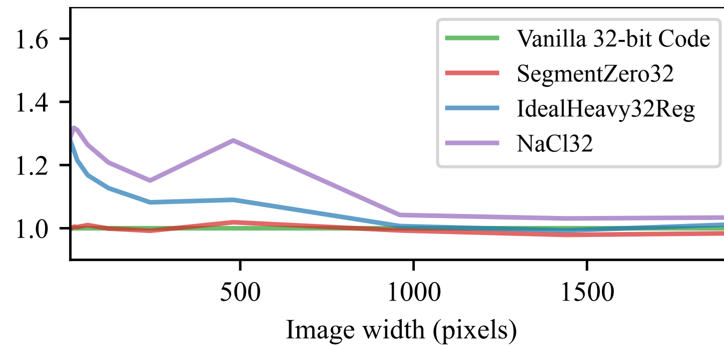
In this section, we measure the performance overhead of `SegmentZero32` with zero-cost transitions. We compare `SegmentZero32` with NaCl (`NaCl32`) and `IdealHeavy32` — a hypothetical SFI scheme with no isolation enforcement overhead, both of which rely on heavyweight transitions. We measure the overhead of these systems on the standard SPEC CPU® 2006 benchmark suite, and the `libgraphite` and `libjpeg` font and image rendering benchmarks. Since both `SegmentZero32` and `NaCl32` use segmentation which is supported only in 32-bit mode, we implement these three isolation builds in 32-bit mode and compare it to native 32-bit unsandboxed code.

SPEC

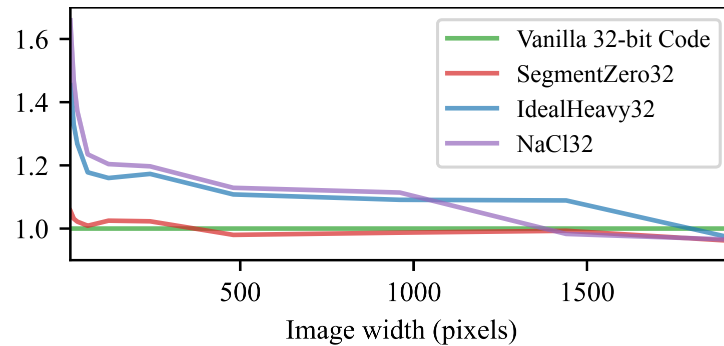
We report the impact of sandboxing on SPEC CPU® 2006 in Table 1.3. Several benchmarks are not compatible with `NaCl32`; augmenting `NaCl32` runtime and libraries to fix such compatibility issues (e.g., as done in Yee et al. [2009] for SPEC2000) is beyond the scope of this work. The `gcc` benchmark, on the other hand, is not compatible with `SegmentZero32`: `gcc` fails (at runtime) because the CFI used by `SegmentZero32`, Clang's CFI, incorrectly computes a target CFI label. Clang's CFI implementation is more precise than necessary for our zero-cost conditions; as with `NaCl32`, we leave the implementation of a coarse-grain and more permissive CFI to future work. On the overlapping benchmarks, `SegmentZero32`'s overhead is comparable to `NaCl32`'s.



(a) `SimpleImage`



(b) `StockImage`



(c) `RandomImage`

Figure 1.26. Performance of image rendering with libjpeg sandboxed with `SegmentZero32` and `NaCl32` and `IdealHeavy32`. Times are relative to unsandboxed code. `NaCl32` and `IdealHeavy32` relative overheads are as high as 312% and 208% respectively, while `SegmentZero32` relative overheads do not exceed 24%.

Table 1.3. Overheads compared to native code on SPEC CPU® 2006 (nc), for `NaCl32` and `SegmentZero32`.

	`NaCl32`	**`SegmentZero32`**
`400.perlbench`	—	1.20×
`401.bzip2`	—	1.08×
`403.gcc`	1.10×	—
`429.mcf`	—	1.04×
`445.gobmk`	1.27×	1.25×
`456.hmmer`	0.97×	0.82×
`458.sjeng`	1.20×	1.16×
`462.libquantum`	1.06×	1.02×
`464.h264ref`	1.34×	1.01×
`471.omnetpp`	1.06×	1.01×
`473.astar`	1.31×	1.10×
`483.xalancbmk`	—	1.05×

Table 1.4. Overheads compared to native code on font rendering for `NaCl32` and `SegmentZero32`.

	`Vanilla` (32-bit)	`IdealHeavy32`	`NaCl32`	**`SegmentZero32`**
Font render	1441ms	2399ms	2769ms	1765ms

Font rendering

The impact of these isolation schemes on font rendering is shown in Table 1.4. We observe that `NaCl32` and `IdealHeavy32` impose an overhead of 92% and 66% respectively. In contrast, `SegmentZero32` has a smaller overhead (22.5%) as it does not have to save and restore registers or switch stacks. We attribute the overhead of `SegmentZero32` (over Vanilla) to three factors: (1) changing segments to enable/disable isolation during function calls, (2) using indirect function calls for cross-domain calls (a choice that simplifies engineering but is not fundamental), and (3) the structure imposed by our zero-cost condition enforcement.

Image rendering

We report the impact of sandboxing on image rendering in Figure 1.26. For narrow images (10 pixel width), `SegmentZero32` overheads relative to the native unsandboxed code are 24%, 1%, and 6.5% for `SimpleImage`, `StockImage` and `RandomImage`, respectively. These overheads are lower than the corresponding overheads for `NaCl32` (312%, 29%, and 66%) as well as `IdealHeavy32` (208%, 28% and 45%). As in the WebAssembly measurements, these overheads shrink as image width increases and the complexity of the image increases (e.g., the overheads for images wider than 480 pixels are negligible).

1.6.4 Effectiveness of the VeriZero Verifier

We evaluate VeriZero's effectiveness by using it to 1. verify 13 binaries (five third–party libraries shipping (or about to ship) with Firefox compiled across 3 binaries, and 10 binaries from the SPEC CPU® 2006 benchmarks) and 2. find nine manually introduced bugs, inspired by real calling convention bugs in previous SFI toolchains [Chromium Team 2012, 2010; Rydgård 2020]. We measure VeriZero's performance verifying the aforementioned 13 binaries. Finally, we stress test VeriZero by running it on random

binaries generated by Csmith [X. Yang et al. 2011].

Experimental setup

We run all VeriZero experiments on a 2.1GHz Intel® Xeon® Platinum 8160 machine with 96 cores and 1 TB of RAM running Arch Linux 5.11.12. All experiments run on a single core and use no more than 2GB of RAM. We compile the SPEC CPU® 2006 binaries using the Lucet toolkit described in Section 1.6.2. We verify three of the Firefox libraries from Firefox Nightly; we compile the other two from the patches that are in the process of being integrated into Firefox.

Effectiveness and Performance Results

VeriZero successfully verifies the 13 Firefox and SPEC CPU® 2006 binaries. These binaries vary in size from 150 functions (the `lbm` benchmark from SPEC CPU® 2006) to 4094 functions (the binary consisting of the Firefox Nightly libraries `libogg`, `libgraphite`, and `hunspell`). It took VeriZero between 1.77 seconds and 19.28 seconds to verify these binaries, with an average of 9.2 seconds and median of 5.93 seconds. VeriZero's performance is on par with the original VeriWasm's performance: on the 10 SPEC CPU® 2006 benchmarks evaluated in the VeriWasm paper [Johnson et al. 2021] VeriZero is slightly (15%) faster, despite checking zero-cost conditions in addition to all of VeriWasm's original checks. This is due to various engineering improvements that were made to VeriWasm in the course of developing VeriZero.

VeriZero also successfully found bugs injected into nine binaries. These bugs tested all the zero-cost properties that VeriZero was designed to check: when possible they were based on real bugs (like those in Cranelift [Rydgård 2020]). VeriZero successfully detected all nine of these bugs, giving us confidence that VeriZero is capable of finding violations of the zero-cost conditions.

Fuzzing Results

We fuzzed VeriZero to both search for potential bugs in Lucet, as well as to ensure VeriZero does not incorrectly declare safe programs unsafe. The fuzzing pipeline works in four stages: first, we use Csmith [X. Yang et al. 2011] to generate random C and ++programs, next we use Clang to compile the generated C/++program to WebAssembly, followed by compiling the Wasm file to native code using Lucet, and finally we verify the generated binary with VeriZero. As these programs were compiled by Lucet, we expect them to adhere to the zero-cost conditions, and any binaries flagged by VeriZero are either bugs in Lucet or are spurious errors in VeriZero.

While we did not find bugs in Lucet, fuzzing did find cases where VeriZero triggered spurious errors. After fixing these errors, we verified 100,000 randomly generated programs with no false positives.

1.7 Limitations

Our WebAssembly SFI scheme is capable of sandboxing any C/++-like language (with arbitrary intra-function control flow, arbitrary type casting, arbitrary pointer aliasing, arithmetic etc.) that can compile to WebAssembly, so long as it does not use features which WebAssembly must emulate through JavaScript[10] (most prominently ++-style exceptions, `setjmp/longjmp`, and multithreading). These limitations are not inherent to our zero-cost conditions, and WebAssembly is in the process of being extended with support for all of the above features [Watt, Rossberg, et al. 2019; WebAssembly Community Group 2021].

Our `SegmentZero32` scheme is built as a proof-of-concept, using mostly existing LLVM passes to sandbox C programs compiled to 32-bit x86, as an approach to understanding the overhead of zero-cost conditions on native code. As such, our

[10] See https://emscripten.org/

SegmentZero32 implementation does not support, for instance, setjmp/longjmp or multithreading (similar to WebAssembly). It also does not support user-defined variadic function arguments or position independent code. However, these limitations are not fundamental. For example, variadic function arguments could be supported by extending the SafeStack pass to move the variadic argument buffer into the unsafe stack, and position independent code could be supported through minor generalizations of existing compiler primitives.

Both WebAssembly and SegmentZero32 rely on a type-directed forward-edge CFI which requires us to statically infer a limited amount of information about arguments to functions.[11] This information includes the number of arguments, their width, and the calling convention. In practice, this information is readily available as part of compilation and does not require any complex control flow inference (unlike more precise fine grain CFI schemes), so this is only a limitation when analyzing certain binary programs. Like most SFI schemes, both WebAssembly and our SegmentZero32 do not currently support just in time (JIT) code compilation within the isolated component; adding this would require engineering work, but can be done following the approaches of [Ansel et al. 2011; Vahldiek-Oberwagner et al. 2019]. Finally, side channels are out of scope for this work.

1.8 Related work

A considerable amount of research has gone into efficient implementations of memory isolation and CFI techniques to provide SFI across many platforms [Adl-Tabatabai et al. 1996; Bittau et al. 2008; Castro et al. 2009; Chen et al. 2016; Ford and Cox 2008; Goonasekera et al. 2015; Herder et al. 2009; Litton et al. 2016; Lucco et al. 1995; McCamant and Morrisett 2006; McMullen 2020; Niu and Tan 2014; Payer and Gross

[11]We do not need to infer any information about the heap or unsafe stack. Variadic functions, for example, can pass a dynamic number of arguments on the unsafe stack.

2011; Sehr et al. 2010; Seltzer et al. 1996; Siefers et al. 2010; Tan 2017]. However, these systems either implement or require the user to implement heavyweight springboards and trampolines to guarantee security.

SFI systems

Wahbe et al. [1993] suggest two ways to optimize transitions: 1. partitioning the registers used by the application and the sandboxed component and 2. performing link time optimizations (LTO) that conservatively eliminates register saves that are never used in the entire sandboxed component (not just the callee). Register partitioning would cause slowdowns due to increased spilling. Native Client [Yee et al. 2009] optimized transitions by clearing and saving contexts using machine specific mechanisms to, e.g., clear floating point state and SIMD registers in bulk. However, we show (see Section 1.6) that, even with those optimizations, the software model imposes significant transition overheads. While CPU makers continue to add optimized context switching instructions, such instructions do not yet eliminate all overhead.

Our work is inspired by Besson, Jensen, et al. [2018], who define a defensive semantics for SFI that captures a notion of sandboxing via simple function calls with a stack shared between the sandbox and host application. Our work builds on this work by addressing two shortcomings: First, their definition does not account for confidentiality of application data, and implementations based on their system would thus need heavyweight transitions to prevent such attacks. Second, their defensive semantics makes fundamental use of guard zones, which limits the flexibility of the framework. Our definition of zero-cost transitions has no such limitations and fully realizes their goal of defining flexible, secure SFI with zero-cost transitions between application and sandbox.

Zeng et al. [2011] combine an SFI scheme with a rich CFI scheme enforcing structure on executing code. While a similar approach, their goal is to safely perform

optimizations to elide SFI and CFI bounds checks, and they do not impose sufficient structure to enforce well-bracketing, a necessary property for zero-cost transitions. XFI [Erlingsson et al. 2006] also combines an SFI scheme with a rich CFI scheme and adopts a safe stack model. While meeting many of the zero-cost conditions, it does not prevent reading uninitialized scratch registers and therefore cannot ensure confidentiality without heavyweight springboards that clear scratch registers. They also do not specify the CFG granularity, so it is not clear if is strong enough to satisfy the zero-cost type-safe CFI requirement.

WebAssembly based isolation

WasmBoxC [Zakai 2020] sandboxes C code by compiling to WebAssembly followed by (de)compiling back to C, ensuring that the sandboxed code will inherit isolation properties from WebAssembly. The sandboxed library code can be safely linked with C applications, enabling a form of zero-cost transition. The zero-cost WebAssembly SFI system described in this chapter was designed and released prior to and independently of WasmBoxC. Moreover, we believe that the theory developed in this chapter provides a foundation for analyzing and proving the security of WasmBoxC though such analysis would need to account for possible undefined behavior introduced in compiling to C.

Sledge [Gadepalli et al. 2020] describes a WebAssembly runtime for edge computing, that relies on WebAssembly properties to enable efficient isolation of serverless components. However, Sledge focuses on function scheduling including preempting running WebAssembly programs, so its needs for context saving differ from library sandboxing as contexts must be saved even in the middle of function calls.

SFI verification

Previous work on SFI (e.g., [Erlingsson et al. 2006; Johnson et al. 2021; McCamant and Morrisett 2006; Yee et al. 2009]) uses a *verifier* or a theorem prover [Kroll et al. 2014; Zhao et al. 2011] to validate the relevant SFI properties of compiled sandbox code. However, unlike VeriZero, none of these verifiers establish sufficient properties for zero-cost transitions.

Hardware based isolation

Hardware features such as memory protection keys [Hedayati et al. 2019; Vahldiek-Oberwagner et al. 2019], extended page tables [Qiang et al. 2017], virtualization instructions [Belay et al. 2012; Qiang et al. 2017], or even dedicated hardware designs [Schrammel, Weiser, Steinegger, et al. 2020] can be used to speed up memory isolation. These works focus on the efficiency of memory isolation as well as switching between protected memory domains; however these approaches also use a single memory region that contain both the stack and heap making them incompatible with zero-cost conditions, i.e. they require heavyweight transitions. `IdealHeavy32` and `IdealHeavy64` in Section 1.6 studies an idealized version of such a scheme.

Capabilities

Karger [1989] and Skorstengaard et al. [2019] look at protecting interacting components on systems that provide hardware-enforced capabilities. Karger [1989] specifically looks at how register saving and restoration can be optimized based on different levels of trust between components, however their analysis does not offer formal security guarantees. Skorstengaard et al. [2019] investigate a calling-convention based on capabilities (à la CHERI [Watson et al. 2015]) that allow safe sharing of a stack between distrusting components. Their definition of well-bracketed control flow and local state encapsulation via an overlay inspired our work, and our logical relation is also

based on their work. However, their technique does not yet ensure an equivalent notion to our confidentiality property, and further is tied to machine support for hardware capabilities.

Type safety for isolation

There has also been work on using strongly-typed languages to provide similar security benefits. SingularityOS [Aiken et al. 2006; Fähndrich et al. 2006; Hunt and Larus 2007], explored using Sing# to build an OS with cheap transitions between mutually untrusting processes. Unlike the work on SFI techniques that zero-cost transitions extend, tools like SingularityOS require engineering effort to rewrite unsafe components in new safe languages.

At a lower level, Typed Assembly Language (TAL) [Morrisett, Crary, Glew, Grossman, et al. 1999; Morrisett, Crary, Glew, and Walker 2002; Morrisett, Walker, et al. 1999] is a type-safe compilation target for high-level type-safe languages. Its type system enables proofs that assembly programs follow calling conventions, and enables an elegant definition of stack safety through polymorphism. Unfortunately, SFI is designed with unsafe code in mind, so cannot generally be compiled to meet TAL's static checks. To handle this our zero-cost and security conditions instead capture the *behavior* that TAL's type system is designed to ensure.

1.9 Acknowledgments

This chapter, in full, is adapted from material as it appears in "Isolation without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI." Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Proc. ACM Program. Lang. 6, POPL, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Robust Constant Time

“Don’t roll your own crypto” is a well known adage directed at application developers when they are considering using cryptography in their code. Instead developers are exhorted to use cryptographic libraries written by experts who have hopefully learned the hard-won lessons of decades of cryptographic and software security research and practice. For such cryptographic library developers, as well as algorithm designers, one of those hard-won lessons is that cryptographic code must be constant time (CT) [D. Bernstein 2005]. More recently (due to microarchitectural attacks like Spectre [P. Kocher et al. 2019]) this lesson has been expanded to the requirement that, under certain circumstances, it is also crucial for cryptographic code to be speculative constant time (SCT) [Cauligi, Disselkoen, Gleissenthall, et al. 2020]. Together, as discussed in Section 0.2 these ensure that the code in the library does not leak secrets (such as cryptographic keys) through either timing channels or speculative execution, respectively.

Sadly, this is not enough. While a constant time cryptographic library will not *itself* leak secrets, it is but one component executing within the context of a larger *application*. Security vulnerabilities in application code could themselves lead to inadvertent disclosure of secrets, no matter how careful library authors were to avoid vulnerabilities. Library authors are keenly aware of this problem and routinely

```c
int stream(u8 *c, u64 clen, u8 *n, u8 *k) {
  ... u8 kcopy[32]; ...
  for (i = 0; i < 32; i++) {
    kcopy[i] = k[i];
  }
  ...
  while (clen >= 64) {
    crypto_core_salsa20(c, in, kcopy, NULL);
    ...
  }
  ...
  sodium_memzero(kcopy, sizeof kcopy);
  return 0;
}
```

Figure 2.1. An excerpt from the reference implementation of Salsa20 in LibSodium.

add protections to harden their code against application vulnerabilities. For example, consider the excerpt of the implementation of the `Salsa20` stream cipher [D. J. Bernstein 2008] from `LibSodium` [Denis 2024] shown in Figure 2.1. We might hope that the (elided) body being constant time suffices to ensure that the secrets in `kcopy` are not leaked. However the *classic* constant time guarantee only ensures that `stream` does not leak the secrets: it makes no guarantees about what happens if there is a memory safety vulnerability in the application linked against the library. Such a vulnerability can lead to a *read-gadget* that may be used to exfiltrate the secrets in `kcopy`! To defend against such gadgets, `LibSodium`'s developers take care to zero the intermediate memory used in `stream` (Line 12) to ensure those secrets cannot be leaked through memory vulnerabilities that reside in the application. Unfortunately, optimizations like dead-store elimination can remove secret scrubbing and nullify the efforts of `LibSodium`'s developers [Z. Yang et al. 2017].

Read gadgets are but one of several possible attacks that library developers must defend against within their host applications. We capture these threats (attacker capabilities) as *assumptions* about application behavior. We specifically adopt the threat models assumed by real world cryptographic libraries: applications may be vulnerable to (speculative) read gadgets, but those vulnerable to control flow hijacking (e.g., because

of write gadgets) are out of scope as most cryptographic libraries assume the application is not *completely* under attacker control.[1]

1. **Attacks via memory unsafety:** The first class of assumptions, illustrated by the code in Figure 2.1, is that owing to memory unsafety: there exist *memory read gadgets* within the application. Zeroing buffers (like `kcopy`) is one key defense against such gadgets, but does nothing to prevent the attacker from reading the original version of the key `k` which remains unzeroed. To protect `k` from read gadgets, libraries like `LibSodium` offer memory protection APIs which must be manually inserted and toggled on and off by application developers, and hence, are prone to incorrect usage.

2. **Attacks via speculation:** If the application is written in a memory safe language like Rust, then the library developer need not fret about read gadgets, and can avoid the overhead of zeroing out secrets. However, the library developer must still contend with the spectre of hardware speculation and its attendant vulnerabilities [Göktas et al. 2020; P. Kocher et al. 2019; Koruyeh et al. 2018; Maisuradze and Rossow 2018; Shivakumar et al. 2023; Wikner and Razavi 2022]. There is work on extending constant time protections to Spectre vulnerabilities, but it focuses on protecting cryptographic code *itself* from speculatively leaking secrets. Owing to the overheads imposed by such protections, it is currently unreasonable to apply the same protections to the entirety of application code. As such, library developers may have to contend with attacks based on Spectre vulnerabilities *in the application* [Mambretti et al. 2021].

   Indeed, in the case of `stream`, speculation leads to a potential security issue with the clearing of `kcopy`: The `LibSodium` developers forgo an appropriate

[1] Applications are also starting to employ software and hardware CFI techniques such as Clang's CFI [The LLVM Foundation 2021a] and Intel's CET [Intel Corporation 2018a].

memory fence in `sodium_memzero`, leading to the possibility of `kcopy` being read by speculatively executing application code *before* it is zeroed [Urban 2019].[2] The fence was eschewed due to its performance overhead: *all* clients of `LibSodium` would suffer the performance degradation for Spectre protections. By manually adding or changing protections, `LibSodium`'s developers implicitly restrict their defenses to certain classes of attackers: i.e. they assume that the only application vulnerabilities are *non-speculative* read gadgets.

3. **Attacks via concurrency:** The last category of application assumptions that we consider is whether the application is *concurrent*. In a concurrent context, work like Spectre-Declassified [Shivakumar et al. 2023] has shown that an attacker can recover secret information if they can observe intermediate results, thus enhancing the reach of (speculative) read gadgets. In fact, the possibility of such observations are cited by the `LibSodium` developers as a reason to forgo the memory fence in `sodium_memzero` [Urban 2019].

For each attack, the library developer must either manually add relevant defenses to their code, or make an explicit choice *not* to do so (e.g. due to prohibitive overheads). Consequently, library code "bakes in" a subset of possible protections against application vulnerabilities: these may be unnecessary or insufficient depending upon the application context. For example, `LibSodium`'s zeroization protection against read gadgets is unnecessary when linked against a memory safe Rust application. On the other hand, `LibSodium`'s protections are insufficient against Spectre attacks [Urban 2019]. The library's authors chose to elide an appropriate fence needed to protect against speculative read gadgets as *all* the clients of the library would have to suffer the corresponding performance degradation, not just the ones where Spectre was a legitimate concern.

[2] Well-documented issues with zeroing functions in high-level code make the zeroing best effort regardless of speculation [Percival 2014].

In a nutshell, cryptographic library developers are currently in a difficult position: they must make one-size-fits-all security-performance trade offs by manually inserting fragile protections orthogonal to the cryptographic algorithms and protocols they are implementing. Luckily, secure compilers offer a promising solution to escape this dilemma [Patrignani, Ahmed, et al. 2019]. Library users are in a better position to make security-performance trade offs and can provide a compiler with security policies to be automatically enforced through inserted protections. To realize this vision, we introduce RoboCop, a new methodology and toolchain for building secure *and* efficient applications from cryptographic libraries. We develop RoboCop via four concrete contributions:

1. **Abstraction: libraries and attackers (§ 2.1, § 2.2.3):** Our first contribution is a formal operational semantics describing the behavior of a library executing within a potentially vulnerable application. We further define a semantics capturing a high-level, abstract model of speculative execution, based on the notion of a speculation oracle that "guesses" the result of evaluating an expression and then later rolling back or committing if the guesses were correct. These semantics provide a unified setting for precisely stating different attackers' observations, guiding the design of our protections.

2. **Specification: robust constant time (§ 2.2.2):** Our second contribution is a novel security property, robust constant time (RCT), using our model to precisely define security for a cryptographic library running *in the context of* a potentially vulnerable application. Crucially, our definition is parameterized by an attacker model, capturing the set of attackers that a library is supposed to be secure against. We further define a speculative version, robust speculative constant time (RSCT), capturing constant time in the presence of speculation.

3. **Implementation: the RoboCop compiler (§ 2.3):** By factoring out assumptions

about the context, our notion of RCT enables our third contribution: a compiler that takes a cryptographic library and synthesizes a bespoke binary tailored to the application against which the library will be linked. To do so, we show how to map each kind of attacker to a concrete code transform that provably, with respect to our definition of RCT, protects against that attacker.[3] Thus, our RoboCop compiler lets library developers focus on implementing constant-time cryptographic algorithms, without having to worry about baking in potentially inefficient or insecure protections against application vulnerabilities. Instead, protections can be automatically inserted based on the application context, thereby ensuring the same library code can be securely *and* efficiently reused in all contexts.

4. **Evaluation: SUPERCOP (§ 2.4):** Finally, our fourth contribution is an empirical evaluation that shows that our RoboCop compiler can automatically generate protections for a wide range of cryptographic library code defending against a variety of attacks. Here we modify the `SUPERCOP` [VAMPIRE 2022] cryptographic benchmarking suite. We instrument `SUPERCOP` to generate and measure the overhead of protecting against three classes of attacks: read gadgets (due to memory safety vulnerabilities in the application), speculative read gadgets (due to speculative execution in the application), and concurrent observations (due to threads in the application). In our test suite of 507 different implementations of cryptographic operations, we show that our RoboCop compiler can automatically generate code that is secure against application vulnerabilities with the majority of overheads under 2% when protecting against read gadgets and under 4% when protecting against speculative read gadgets, thereby demonstrating that RCT reconciles the tension between security and efficiency when reusing cryptographic

[3] Our compiler assumes that the library is already (speculatively) constant-time. In § 2.2.2 we discuss how RoboCop's robust protections are orthogonal to existing (speculative) constant time protections, thus allowing library developers to use existing automated tools or manual technique to meet this assumption.

libraries in different application contexts.

2.1 Security Semantics

To formally ground robust constant time, the attacker models, and our compiler we develop a high-level, stateful calculus, λ_{spec}, whose syntax is shown in Figure 2.2. Syntactically, λ_{spec} is relatively standard, following λ_{rust}, CompCert, and others [Jung et al. 2017; Leroy 2009] in employing a block-based memory model: memory is structured as "disjoint", fixed width blocks addressed via a block label and offset ($z_b[z_o]$). New blocks are allocated with $\texttt{new}_p\ e$ with e the size of the block and the protection label p determines whether the allocation is in a protected or unprotected memory "page". The size, set of values, and memory page are tracked in the second component of the non-speculative states (S). Correspondingly there is a protection operation, $\texttt{protect}_p$, which models hardware memory protection. $\texttt{protect}_p$ sets the memory access policy in the first component of the state: `public` only allows access to the public memory page whereas `protected` allows access to all memory. Dereferences are written $!\,e$ and assignments are written $e_{ptr} := e_{val}$. Functions are also stored (immutably) in memory.

We equip λ_{spec} with these semantics: a non-speculative semantics capturing a trace of events that we use to define our attacker models (§ 2.1.1) a novel, high-level speculative semantics (§ 2.1.2), and (speculative) concurrent semantics capturing a passive observer (§ 2.1.3). Throughout we use $\bullet$ for an empty list, $+\!\!+$ for list concatenation, and an overline (e.g. $\overline{e}$) for a list of elements.

2.1.1 Non-Speculative Trace Semantics

We first describe the structure of the traces the non-speculative semantics is designed to capture. We are interested in capturing three aspects of the execution: 1. *who* (which party, `app` or `lib`) is executing code at a given time as well as the transfer of control between the parties, 2. *what memory* each party accesses, and 3. the internal

$$
\begin{array}{rccl}
\text{numbers} & z & \in & \mathbb{Z} \\
\text{labels} & \ell & ::= & \texttt{app} \mid \texttt{lib} \\
\text{protection} & p & ::= & \texttt{public} \mid \texttt{protected} \\
\text{values} & v & ::= & z \mid z[z] \\
\text{expressions} & e & ::= & v \\
 & & \mid & x \\
 & & \mid & x\{v\} \\
 & & \mid & \texttt{op}(\overline{e}) \\
 & & \mid & e[e] \\
 & & \mid & \texttt{!}\,e \\
 & & \mid & \texttt{new}_p\ e \\
 & & \mid & e := e \\
 & & \mid & \texttt{protect}_p \\
 & & \mid & e; e \\
 & & \mid & \texttt{get-block}\ e \\
 & & \mid & \texttt{get-offset}\ e \\
 & & \mid & e(\overline{e}) \\
 & & \mid & \texttt{fence} \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{states} & S & \in & p \times (\mathbb{Z} \rightharpoonup m) \\
\text{memory slots} & m & ::= & \lambda_\ell \overline{x}.e \\
 & & \mid & \{\ \ size : \mathbb{Z} \\
 & & & \quad\ \ p : p \\
 & & & \quad\ \ v : [size] \rightharpoonup v \quad \} \\
\text{speculation frame} & \Xi & ::= & (S, \widehat{\overline{K} :: e}, \overline{\delta}) \\
 & & \mid & (S, \overline{K} :: e, \overline{\delta}, \mu) \\
\text{speculative states} & \Phi & ::= & \{\ \ S : S \\
 & & & \quad\ \ a : A \\
 & & & \quad\ \ \Xi : \overline{\Xi} \quad \} \\
\text{speculation directives} & d & ::= & \mathsf{nonspec} \mid \mathsf{spec}\ v \mid \mathsf{fence}
\end{array}
$$

Figure 2.2. Syntax of λ_{spec}.

$$
\begin{array}{rcll}
\text{events} & \epsilon & ::= & \delta^{\ell} \mid \tau^{\ell \to \ell} \\
\text{transition events} & \tau & ::= & \mathsf{call}\ z_f \\
& & \mid & \mathsf{ret}\ v \\
& & \mid & \mathsf{begin} \\
& & \mid & \mathsf{end}\ v \\
\text{domain events} & \delta & ::= & \mu \mid \mathsf{call}\ z \\
& & \mid & \mathsf{ret}\ v \\
& & \mid & \mathsf{branch}\ v \\
& & \mid & \mathsf{fence} \\
& & \mid & 0 \\
\text{memory label} & b & ::= & \mathsf{ib} \mid \mathsf{oob} \\
\text{memory events} & \mu & ::= & \mathsf{new}_p\ z@z \\
& & \mid & \mathsf{read}_b\ v \hookleftarrow z[z] \\
& & \mid & \mathsf{write}_b\ v \mapsto z[z] \\
& & \mid & \mathsf{protect}_p \\
\text{observable events} & c & ::= & 0 \\
& & \mid & \mathsf{branch}\ v \\
& & \mid & \mathsf{read} \hookleftarrow z[z] \\
& & \mid & \mathsf{write} \mapsto z[z] \\
& & \mid & \mathsf{new}_p\ z@z \\
& & \mid & \mathsf{call}\ z \\
& & \mid & \mathsf{end}\ v
\end{array}
$$

Figure 2.3. Syntax of events in λ_{spec}.

branching which will be used to define the various constant time properties. To this end each reduction is labeled with an event, ϵ, whose syntax is shown in Figure 2.3. Labels are also attached to every function ($\lambda_\ell \overline{x}.e$) to distinguish application and library code. An event is either a labeled *domain event*, δ^ℓ, which captures an event executed by the party ℓ or a *transition event*, $\tau^{\ell \to \ell}$, which captures the transfer of control between the two parties.

The main transition events are $\textsf{call}\ z_f$ and $\textsf{ret}\ v$ which represent a call to the function at address z_f and returning from a function with return value v. Beyond the call and return events, there are $\textsf{begin}$ and $\textsf{end}\ v$ events that capture the (implicit) beginning and end of a trace. Domain events are either a *memory event* (μ), a $\textsf{call}\ z_f$ and its corresponding $\textsf{ret}\ v$ events (capturing a function call that stays within a single domain), a $\textsf{branch}\ v$ event (capturing branching on the value v), or the empty event 0. Memory events are one of an allocation, a protection operation, a read, or a write and track the data associated with each operation (e.g. the value read/written, the location it was read/written from/to, and whether the memory access was in bounds or out of bounds).

Transition operational semantics

Our non-speculative semantics are split between two labeled reduction judgments: top-level transition reductions (Figure 2.4) and domain reductions (Figure 2.5). Transition reductions are of the form $\langle S \mid \overline{K^\ell} :: e^\ell \rangle \overset{\epsilon}{\Rightarrow} \langle S \mid \overline{K^\ell} :: e^\ell \rangle$ where K are the standard call-by-value evaluation contexts. The state S tracks the memory and the current access level. To explain the control stack $\overline{K^\ell} :: e^\ell$ we examine the rule RED-CALL. The top of our stack (e^ℓ) is the mid-reduction body of the currently executing function (with label ℓ). For RED-CALL we are reducing a function call in the evaluation context K. We push this continuation (where the called function will return to) onto the stack of labeled continuations ($\overline{K'^{\ell'}}$) and then use the substituted body as the new execution frame. If

$$\boxed{\langle S \mid \overline{K^\ell} :: e^\ell \rangle \overset{\epsilon}{\Rightarrow} \langle S \mid \overline{K^\ell} :: e^\ell \rangle}$$

$$\textsc{red-call}\quad \frac{S(z) = \lambda_{\ell_f}\overline{x}.e \qquad \epsilon = \begin{cases} (\mathsf{call}\ z)^\ell & \text{when } \ell_f = \ell \\ (\mathsf{call}\ z)^{\ell \to \ell_f} & \text{otherwise} \end{cases}}{\langle S \mid \overline{K'^{\ell'}} :: K[z(\overline{v})]^\ell \rangle \overset{\epsilon}{\Rightarrow} \langle S \mid \overline{K'^{\ell'}} :: K^\ell :: e\overline{[v/x]}^{\ell_f} \rangle}$$

$$\textsc{red-ret}\quad \frac{\epsilon = \begin{cases} (\mathsf{ret}\ v)^\ell & \text{when } \ell_K = \ell \\ (\mathsf{ret}\ v)^{\ell \to \ell_K} & \text{otherwise} \end{cases}}{\langle S \mid \overline{K'^{\ell'}} :: K^{\ell_K} :: v^\ell \rangle \overset{\epsilon}{\Rightarrow} \langle S \mid \overline{K'^{\ell'}} :: K[v]^{\ell_K} \rangle} \qquad \textsc{red-}\beta\quad \frac{\langle S \mid e \rangle \overset{\delta}{\to} \langle S' \mid e' \rangle}{\langle S \mid \overline{K'^{\ell'}} :: K[e]^\ell \rangle \overset{\delta^\ell}{\Longrightarrow} \langle S' \mid \overline{K'^{\ell'}} :: K[e']^\ell \rangle}$$

Figure 2.4. Non-speculative operational semantics for λ_{spec}: transition reductions.

the current label, ℓ, and the label of the function we are calling, ℓ_f, differ then we emit a transition event with label $\ell \to \ell_f$ otherwise we omit a domain event for the call.

Dually, rule RED-RET handles returning from a function. This is a transfer of control flow from ℓ, the label of the currently executing function, back to ℓ_K, the function caller. The return value is plugged into the top continuation on the stack and a corresponding return event is generated (we ignore same domain returns). The last "transition" reduction rule, RED-β, dispatches to the domain reduction relation ($\langle S \mid e \rangle \overset{\delta}{\to} \langle S \mid e \rangle$), and labels the domain event δ with the current label.

Domain operational semantics

Most of the domain rules are standard and produce an empty trace event. Conditionals are also standard, but produce a branch event based on the condition. The rule β-NEW takes a protection domain p in which to allocate a new block of size z, checking that our current access level allows writing to the domain p using the "can-access" judgment $S.p \sqsubseteq p$.

The most notable of the reduction rules are those related to dereferencing and

$$\boxed{\langle S \mid e\rangle \xrightarrow{\delta} \langle S \mid e\rangle}$$

β-SUBST
$$\frac{}{\langle S \mid x\{v\}\rangle \xrightarrow{0} \langle S \mid v\rangle}$$

β-OP
$$\frac{v' = \delta(op)(\overline{v})}{\langle S \mid \mathsf{op}(\overline{v})\rangle \xrightarrow{0} \langle S \mid v'\rangle}$$

β-DEREF
$$\frac{\mathsf{accessible}(S, z_b) \qquad z_o \in [S(z_b).size) \qquad v = S(z_b).v(z_o)}{\langle S \mid !\,(z_b[z_o])\rangle \xrightarrow{\mathsf{read_{ib}}\ v \hookleftarrow z_b[z_o]} \langle S \mid v\rangle}$$

β-DEREF-OOB
$$\frac{z_b \notin \mathrm{dom}(S) \lor z_o \notin [S(z_b).size] \qquad z_b' \in \mathrm{dom}(S) \qquad z_o' \in [S(z_b').size) \qquad v = S(z_b').v(z_o') \qquad \mathsf{accessible}(S, z_b')}{\langle S \mid !\,(z_b[z_o])\rangle \xrightarrow{\mathsf{read_{oob}}\ v \hookleftarrow z_b'[z_o']} \langle S \mid v\rangle}$$

β-SEQ
$$\frac{}{\langle S \mid v; e\rangle \xrightarrow{0} \langle S \mid e\rangle}$$

β-WRITE
$$\frac{\mathsf{accessible}(S, z_b) \qquad z_o \in [S(z_b).size] \qquad S' = S(z_b).v[z_o := v]}{\langle S \mid z_b[z_o] := v\rangle \xrightarrow{\mathsf{write_{ib}}\ v \mapsto z_b[z_o]} \langle S' \mid 0\rangle}$$

β-FENCE
$$\frac{}{\langle S \mid \mathtt{fence}\rangle \xrightarrow{\mathsf{fence}} \langle S \mid 0\rangle}$$

β-WRITE-OOB
$$\frac{z_b \notin \mathrm{dom}(S) \lor z_o \notin [S(z_b).size] \qquad z_b' \in \mathrm{dom}(S) \qquad z_o' \in [S(z_b').size) \qquad S' = S(z_b').v[z_o' := v] \qquad \mathsf{accessible}(S, z_b')}{\langle S \mid z_b[z_o] := v\rangle \xrightarrow{\mathsf{write_{oob}}\ v \mapsto z_b'[z_o']} \langle S' \mid 0\rangle}$$

β-NEW
$$\frac{z > 0 \qquad z_b = \mathsf{fresh}(S) \qquad S.p \sqsubseteq p \qquad S' = S[z_b := \{size = z, v = \bot, p = p\}]}{\langle S \mid \mathtt{new}_p\ z\rangle \xrightarrow{\mathsf{new}_p\ z @ z_b} \langle S' \mid z_b[0]\rangle}$$

β-GET-BLOCK
$$\frac{}{\langle S \mid \mathtt{get\text{-}block}\ (z_b[z_o])\rangle \xrightarrow{0} \langle S \mid z_b\rangle}$$

β-GET-OFFSET
$$\frac{}{\langle S \mid \mathtt{get\text{-}offset}\ (z_b[z_o])\rangle \xrightarrow{0} \langle S \mid z_o\rangle}$$

β-IF-FALSE
$$\frac{}{\langle S \mid \mathtt{if}\ 0\ \mathtt{then}\ e\ \mathtt{else}\ e'\rangle \xrightarrow{\mathsf{branch}\ 0} \langle S \mid e'\rangle}$$

β-IF-TRUE
$$\frac{v \neq 0}{\langle S \mid \mathtt{if}\ v\ \mathtt{then}\ e\ \mathtt{else}\ e'\rangle \xrightarrow{\mathsf{branch}\ v} \langle S \mid e\rangle}$$

β-PROTECT
$$\frac{}{\langle S \mid \mathtt{protect}_p\rangle \xrightarrow{\mathsf{protect}_p} \langle S[p := p] \mid 0\rangle}$$

Figure 2.5. Non-speculative operational semantics for λ_{spec}: domain reductions.

writing through pointers. The rules mirror each other so we will focus on dereferencing as it is simpler. In the rule β-DEREF we are dereferencing the pointer $z_b[z_o]$ with block label z_b and offset into that block z_o. To actually obtain the value at that location the following must hold: 1. the block must be accessible ($\mathsf{accessible}(S, z_b)$), 2. the offset must be within the allocated size of the block ($z_o \in [S(z_b).size]$), and 3. a value must have been written to the block at the offset z_o ($v = S(z_b).v(z_o)$). When these conditions are met the value is read and a corresponding *in bounds* read trace event is generated. On the other hand, if either of the latter two conditions are not met the dereference is considered out-of-bounds and the rule β-DEREF-OOB applies instead. Here, we model the out-of-bounds read as a nondeterministic read from an arbitrary, valid, and accessible location $z'_o[z'_b]$.

2.1.2 Speculative Semantics

To model Spectre attacks and speculative execution broadly we define a second operational semantics for λ_{spec}. Instead of directly modeling a specific version of Spectre attack we seek to capture a high-level essence of speculation, namely that speculation is the combination of guessing how an expression might evaluate and then either rolling back or committing if the guess was correct. That is, different types of speculation can be captured as the following sequence: First, instead of evaluating an expression, make up the value you think it would evaluate to and continue running using that value instead. While running "under speculation" check if any operation invalidates the speculative guess. Then, once you hit a "fence", rollback to the speculation point if the guess was invalid, or commit the effects of the skipped expression and continue evaluating. In Section 2.1.4 we step through how λ_{spec} models Spectre-PHT, -BTB, -STL, and (with choices of encoding)-RSB [Horn 2018; Intel Corporation 2017a,b, 2018b; P. Kocher et al. 2019; Maisuradze and Rossow 2018].

We capture these notions in Figures 2.6, 2.7, 2.8, and 2.9. Figure 2.6 defines the top-level reduction relation $\langle \Phi \mid e \rangle \overset{\bar{\delta}}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi \mid e \rangle$. Speculative states, Φ, extend the

$$\boxed{\langle \Phi \mid \overline{K} :: e\rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi \mid \overline{K} :: e\rangle}$$

SPEC-NONSPEC

$$\frac{(a', \mathsf{nonspec}) = \mathit{spec}(\Phi.a, e) \qquad \langle \Phi \mid \overline{K} :: e\rangle \overset{\delta}{\hookrightarrow} \langle \Phi' \mid \overline{K'} :: e'\rangle}{\langle \Phi \mid \overline{K} :: e\rangle \overset{\delta}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi'[a := a'] \mid \overline{K'} :: e'\rangle}$$

SPEC-TRY-COMMIT

$$\frac{(a', \mathsf{fence}) = \mathit{spec}(\Phi.a, e) \qquad \mathsf{fence}\ \langle \Phi \mid \overline{K} :: e\rangle\ \mathsf{to}\ \langle \Phi' \mid \overline{K'} :: e'\rangle_{\overline{\delta}}}{\langle \Phi \mid \overline{K} :: e\rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi'[a := a'] \mid \overline{K'} :: e'\rangle}$$

SPEC-SPEC

$$\frac{(a', \mathsf{spec}\ v) = \mathit{spec}(\Phi.a, K[e]) \qquad \langle \Phi \mid \bullet :: e\rangle \overset{\overline{\delta}}{\hookrightarrow}{}^{*} \langle \Phi' \mid \bullet :: v'\rangle \qquad \overline{\Xi'} = \mathsf{makeFrame}_{v=v'}(\Phi.S, \overline{K'} :: K[e], \overline{\delta}) :: \Phi.\Xi \qquad e \neq \mathtt{fence}}{\langle \Phi \mid \overline{K'} :: K[e]\rangle \overset{0}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi[\Xi := \overline{\Xi'}, a := a'] \mid \overline{K'} :: K[v]\rangle}$$

$$\boxed{\langle \Phi \mid \overline{K} :: e\rangle \overset{\delta}{\hookrightarrow} \langle \Phi \mid \overline{K} :: e\rangle}$$

SPEC-β

$$\frac{\langle \Phi.S \mid \overline{K^{\ell}} :: e^{\ell}\rangle \overset{\epsilon}{\Rightarrow} \langle S' \mid \overline{K'^{\ell'}} :: e'^{\ell'}\rangle \qquad \neg\mathsf{stalled}(\Phi.\Xi, \Phi.S, \mathsf{unlabel}(\epsilon)) \qquad \overline{\Xi} = \mathsf{addEvent}(\Phi.\Xi, \mathsf{unlabel}(\epsilon))}{\langle \Phi \mid \overline{K} :: e\rangle \xhookrightarrow{\mathsf{unlabel}(\epsilon)} \langle \Phi[S := S', \Xi := \overline{\Xi}] \mid \overline{K'} :: e'\rangle}$$

Figure 2.6. Speculative operational semantics for λ_{spec}.

$$\boxed{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e\rangle\ \mathsf{to}\ \langle \Phi \mid \overline{K} :: e\rangle_{\overline{\delta}}}$$

FENCE-NO-SPEC

$$\frac{\Phi.\Xi = \bullet}{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e\rangle\ \mathsf{to}\ \langle \Phi \mid \overline{K} :: e\rangle_{\bullet}}$$

FENCE-ROLLBACK

$$\frac{\Phi.\Xi = (S, \widehat{\overline{K'} :: e'}, \overline{\delta}) :: \overline{\Xi}}{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e\rangle\ \mathsf{to}\ \langle \Phi[S := S, \Xi := \overline{\Xi}] \mid \overline{K'} :: e'\rangle_{\bullet}}$$

FENCE-COMMIT

$$\frac{\Phi.\Xi = (S, \overline{K'} :: e', \overline{\delta}, \overline{\mu}) :: \overline{\Xi} \qquad S' = \mathsf{commit}(S, \overline{\delta} +\!\!+ \overline{\mu}) \qquad \overline{\Xi'} = \mathsf{addEvents}(\overline{\Xi}, \overline{\delta} +\!\!+ \overline{\mu})}{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e\rangle\ \mathsf{to}\ \langle \Phi[S := S', \Xi := \overline{\Xi'}] \mid \overline{K} :: e\rangle_{\overline{\delta}}}$$

Figure 2.7. Speculative operational semantics for λ_{spec}: fencing.

$$\boxed{\mathsf{stalled}(\overline{\Xi}, S, \delta) : \overline{\Xi} \times S \times \delta \to \mathbb{2}}$$

$$\begin{aligned}
\mathsf{stalled}(\bullet, S, \mathsf{fence}) &\triangleq \top \\
\mathsf{stalled}(\Xi :: \overline{\Xi}, S, \mathsf{read}_b\ v \leftarrow z_b[z_o]) &\triangleq (\mathsf{protect}_p \in \Xi.\overline{\delta} +\!\!+ \Xi.\overline{\mu} \wedge S(z_b).p \triangleq \mathtt{protected}) \\
&\quad\ \vee (\mathsf{stalled}(\overline{\Xi}, S, \mathsf{read}_b\ v \leftarrow z_b[z_o])) \\
\mathsf{stalled}(\Xi :: \overline{\Xi}, S, \mathsf{write}_b\ v \mapsto z_b[z_o]) &\triangleq (\mathsf{protect}_p \in \Xi.\overline{\delta} +\!\!+ \Xi.\overline{\mu} \wedge S(z_b).p \triangleq \mathtt{protected}) \\
&\quad\ \vee (\mathsf{stalled}(\overline{\Xi}, S, \mathsf{write}_b\ v \mapsto z_b[z_o])) \\
\mathsf{stalled}(\Phi, S, \delta) &\triangleq \bot
\end{aligned}$$

$$\boxed{\mathsf{addEvent}(\overline{\Xi}, \delta) : \overline{\Xi} \times \delta \to \overline{\Xi}}$$

$$\begin{aligned}
\mathsf{addEvent}(\bullet, \delta) &\triangleq \bullet \\
\mathsf{addEvent}(\Xi :: \overline{\Xi}, \delta) &\triangleq \mathsf{addEvent}(\Xi, \delta) :: \overline{\Xi}
\end{aligned}$$

$$\boxed{\mathsf{addEvent}(\Xi, \delta) : \Xi \times \delta \to \Xi}$$

$$\begin{aligned}
\mathsf{addEvent}((S, \widehat{\overline{K} :: e, \overline{\delta}}), \delta') &\triangleq (S, \widehat{\overline{K} :: e, \overline{\delta}} +\!\!+ \delta') \\
\mathsf{addEvent}((S, \overline{K} :: e, \overline{\delta}, \overline{\mu}), \delta') &\triangleq (S, \overline{K} :: e, \overline{\delta}, \overline{\mu}) \\
&\quad\ \text{when } \delta' \neq \mu' \\
\mathsf{addEvent}((S, \overline{K} :: e, \overline{\delta}, \overline{\mu}), \mathsf{read}_b\ v \leftarrow v_r) &\triangleq (S, \widehat{\overline{K} :: e, \overline{\delta}} +\!\!+ \overline{\mu}) \\
&\quad\ \text{when } v_r \in \mathsf{writeLocs}(\overline{\delta}) \\
\mathsf{addEvent}((S, \overline{K} :: e, \overline{\delta}, \overline{\mu}), \mathsf{read}_b\ v \leftarrow v_r) &\triangleq (S, \overline{K} :: e, \overline{\delta}, \overline{\mu}) \\
&\quad\ \text{when } v_r \notin \mathsf{writeLocs}(\overline{\delta}) \\
\mathsf{addEvent}((S, \overline{K} :: e, \overline{\delta}, \overline{\mu}), \mathsf{write}_b\ v \mapsto v_w) &\triangleq (S, \overline{K} :: e, \overline{\delta}, \overline{\mu} +\!\!+ \mathsf{write}_b\ v \mapsto v_w) \\
\mathsf{addEvent}((S, \overline{K} :: e, \overline{\delta}, \overline{\mu}), \mathsf{new}_p\ z@z_b) &\triangleq (S, \overline{K} :: e, \overline{\delta}, \overline{\mu} +\!\!+ \mathsf{new}_p\ z@z_b) \\
\mathsf{addEvent}((S, \overline{K} :: e, \overline{\delta}, \overline{\mu}), \mathsf{protect}_p) &\triangleq (S, \overline{K} :: e, \overline{\delta}, \overline{\mu} +\!\!+ \mathsf{protect}_p)
\end{aligned}$$

Figure 2.8. Speculative operational semantics for λ_{spec}: auxiliary definitions.

$$
\begin{array}{rcl}
\mathsf{unlabel}(\delta^{\ell}) & \triangleq & \delta \\
\mathsf{unlabel}(\tau^{\ell \to \ell'}) & \triangleq & \tau \\
\\
\mathsf{writeLocs}(\overline{c}) & \triangleq & \{v_w \mid \mathsf{write}_{\in}\ v \mapsto v_w \overline{c}\} \\
\\
\mathsf{commit}(S, \bullet) & \triangleq & S \\
\mathsf{commit}(S, \delta + \overline{\delta'}) & \triangleq & \mathsf{commit}(\mathsf{commit}(S, \delta), \overline{\delta'}) \\
\mathsf{commit}(S, \mathsf{write}_b\ v \mapsto z_b[z_o]) & \triangleq & S(z_b).[z_o := v] \\
\mathsf{commit}(S, \mathsf{protect}_p) & \triangleq & S[p := p] \\
\mathsf{commit}(S, \mathsf{new}_p\ z@z_b) & \triangleq & S[z_b := \{size = z, v = \bot, p = p\}] \\
\mathsf{commit}(S, \delta) & \triangleq & S \\
\\
\mathsf{makeFrame}_{\top}(S, \overline{K} :: e, \overline{\delta}) & \triangleq & (S,\ \overline{K} :: e,\ \overline{\delta},\ \bullet) \\
\mathsf{makeFrame}_{\bot}(S, \overline{K} :: e, \overline{\delta}) & \triangleq & \widehat{(S,\ \overline{K} :: e,\ \overline{\delta})}
\end{array}
$$

Figure 2.9. Speculative operational semantics for λ_{spec}: auxiliary definitions, continued.

non-speculative state (S) with a microarchitectural state (a) and a stack of speculation frames (Ξ). Speculation frames come in two forms, a "mispeculation" frame, $\widehat{(S,\ e,\ \overline{\delta})}$, capturing that the current speculation is invalid and will be rolled back to the state S and expression e and "in progress" frames, $(S,\ e,\ \overline{\delta},\ \overline{\mu})$, capturing that the current speculation is potentially valid. The $\overline{\delta}$ and $\overline{\mu}$ components capture the trace of events of the skipped expression and the subsequent memory events, respectively. The skipped events are used if we commit a speculative frame, "replaying" the events that were speculatively passed over, and the memory events are used to determine whether speculation is invalid (discussed below).

The decision of whether and how to speculate is handled by a microarchitectural state "speculation" function ($spec : A \times e \to A \times d$). The function takes the current microarchitectural state and the current state of the executing function, updates the microarchitectural state, and returns a speculation directive, d. This directive says whether we will be speculating with the guessed value v ($\mathsf{spec}\ v$), not speculating

(nonspec), or performing a fence operation (fence).

Speculation

To see how speculation plays out we will go over the three corresponding rules: SPEC-NONSPEC, SPEC-SPEC, and SPEC-TRY-COMMIT and how they capture speculation in the Spectre-PHT bypassing of a bounds check when evaluating `if` $i\{100\} < 10$ `then` $!\,b[i\{100\}]$ `else` 0 (the size of the block b is 10). The term $i\{100\}$ captures a delayed substitution: earlier in the execution the value 100 was substituted for i.[4] We first evaluate $i\{100\}$ and apply the rule SPEC-NONSPEC: the speculator returns that it does not want to speculate on this expression and evaluation proceeds normally (via the SPEC-β rule), so our branch condition is now $100 < 10$. Here the speculator consults its microarchitectural state, sees that every other time we have done this check the result has been true, and thus returns spec 1 (and a new microarchitectural state) and the rule SPEC-SPEC applies. SPEC-SPEC runs the skipped expression, capturing any memory events (writes, reads, etc.) but does not commit them, instead saving them in a new speculation frame via makeFrame. makeFrame checks if the speculated value matches the real value: in this case it does not and therefore our new frame is a misspeculation frame saving the current state and continuation on our stack Ξ. Evaluation then continues with the speculated value 1 and another two SPEC-NONSPEC steps evaluate $!\,b[100]$.

In these SPEC-NONSPEC steps the rule SPEC-β handles two additional aspects beyond the evaluation. First, it checks that the instruction is not stalled via the stalled function. This captures that fence instructions will not execute speculatively and that, with Memory Protection Keys (MPK), writes and reads to `protected` memory will not execute if the protection level is speculatively uncertain (the rules for fences and reads are shown in Figure 2.7). SPEC-β also adds the new event to the speculative stack

[4]We use delayed substitutions (β-SUBST) to capture the fact that, when executing on hardware, argument substitution will be compiled to a register access or memory lookup and as such should not be treated as an immediate value.

($\overline{\Xi} = \mathsf{addEvent}(\Phi.\Xi, \mathsf{unlabel}(\epsilon))$), updating whether the current speculation is invalid or not. addEvent is shown in Figure 2.8. For reads it checks if the read is to the location of a speculatively skipped write: if so the current speculation is invalid and will be rolled back (but continues executing until a fence). The displayed rule for writes shows how other events are added to an in progress frame to be used to check the validity of previous speculation.

Back in our example, were we to continue evaluating we would reach the classic Spectre-PHT out of bounds read, however we will instead assume that the speculator decides to stop speculating and returns the fence directive. The rule SPEC-FENCE thus applies and we turn to the judgment $\mathsf{fence}\ \langle \Phi \mid !\,b[100] \rangle$ to $\langle \Phi_1 \mid e_1 \rangle_{\overline{\delta}}$. This judgment, defined in Figure 2.7, returns the state and expression with which we will continue evaluation as well as the trace of events from evaluating the speculatively skipped expression. If the speculation was invalidated (FENCE-ROLLBACK) then we will return to the continuation where speculation began (this applies to our example and we return to the saved continuation $\texttt{if}\ 100 < 10\ \texttt{then}\ !\,b[i\{100\}]\ \texttt{else}\ 0$ and the respective state at the time of speculation). If the speculation had been valid then FENCE-COMMIT would apply. This commits any memory events that were speculatively skipped and checks whether the new, now committed events invalidate previous speculation (we allow nested speculation thus the speculative “stack”).

2.1.3 Concurrent Observer Semantics

To define concurrent observer capabilities we layer another semantics on top of the speculative and non-speculative semantics, capturing the intuitive notion that a read-only attacker in a concurrent thread may, at *any* time, observe *any* visible memory location. The new judgments (shown in Figure 2.10) consist of a singular rule that, before any step, adds a read event for every memory location visible to a concurrent thread. As MPK guarantees thread local protection this consists of every location that is not in the

$$\boxed{\langle S \mid \overline{K^\ell} :: e^\ell \rangle \xRightarrow{\overline{\epsilon}}_C \langle S \mid \overline{K^\ell} :: e^\ell \rangle}$$

$$\frac{\langle S \mid \overline{K^\ell} :: e^{\ell'} \rangle \xRightarrow{\epsilon} \langle S \mid \overline{{K_2}^{\ell_2}} :: {e_2}^{\ell_2'} \rangle \qquad \overline{\delta} = [\text{branch } v \mid \texttt{public} \sqsubseteq S(z_b) \wedge z_o \in [S(z_b).size) \wedge v = S(z_b).v(z_o)]}{\langle S \mid \overline{K^\ell} :: e^{\ell'} \rangle \xRightarrow{\overline{\delta}^{\text{app}} + \epsilon}_C \langle S \mid \overline{{K_2}^{\ell_2}} :: {e_2}^{\ell_2'} \rangle}$$

$$\boxed{\langle \Phi \mid \overline{K} :: e \rangle \xhookrightarrow{\overline{\delta}}_C \langle \Phi \mid \overline{K} :: e \rangle}$$

$$\frac{\langle \Phi \mid \overline{K} :: e \rangle \xhookrightarrow{\overline{\delta}} \langle \Phi' \mid \overline{K'} :: e' \rangle \qquad \overline{\delta'} = [\text{branch } v \mid \langle \Phi[S.p := \texttt{public}] \mid !\, z_b[z_o] \rangle \xhookrightarrow{\mu} \langle \Phi' \mid v \rangle]}{\langle \Phi \mid \overline{K} :: e \rangle \xhookrightarrow{\overline{\delta'} + \overline{\delta}}_C \langle \Phi' \mid \overline{K'} :: e' \rangle}$$

Figure 2.10. Concurrent observer semantics for λ_{spec}.

`protected` memory region, even if our "main" thread currently has access to protected memory. This models the attacker making *all* possible concurrent observations.

2.1.4 Modeling Spectre Attacks

We demonstrate how λ_{spec} can model the different known Spectre vulnerabilities, showing C code demonstrating the attack and then a corresponding reduction in λ_{spec}.

Spectre v1 (PHT): bounds check bypass

The original Spectre variant involves bypassing a bounds check via the speculative behavior of the Pattern History Table (PHT). The attack was originally demonstrated by P. Kocher et al. [2019] and is tracked by CVE-2017-5753 [Intel Corporation 2017b].

```
// array is of length 10
void main() {
  ...
  if (index < 10) {
```

```
    array[index];
    ...
  }
  ...
}
```

The attacker controls `index` and sets it to 100. Having trained the branch predictor with indices under 10, the processor predicts that the branch condition will be true and speculatively runs `array[index]`, thus reading out of bounds before eventually rolling back, thus allowing an attacker to perform an out-of-bounds memory read.

In λ_{spec} we have a variable `array` mapped to a buffer of size 10. As shown below, the code is equivalent, with the speculation oracle "guessing" that the bounds check passes on Line 2.2. The reduction of the dereference in Line 2.4 is then out-of-bounds, so there is a non-deterministic read to an arbitrary memory location.

$$\texttt{array} \mapsto \{size = 10, \ldots\}$$

$$\texttt{if}\ \texttt{index} < 10\ \texttt{then}\ !\,(\texttt{array[index]}); e_1\ \texttt{else}\ e_2$$

$$\hookrightarrow\!\!\!\twoheadrightarrow^* \quad \textsc{spec-nonspec} \tag{2.1}$$

$$\texttt{if}\ 100 < 10\ \texttt{then}\ !\,(\texttt{array[index]}); e_1\ \texttt{else}\ e_2$$

$$\hookrightarrow\!\!\!\twoheadrightarrow \quad \textsc{spec-spec}(spec(100 < 10) = \textsf{spec}\ 1) \tag{2.2}$$

$$\texttt{if}\ 1\ \texttt{then}\ !\,(\texttt{array[index]}); e_1\ \texttt{else}\ e_2$$

$$\xhookrightarrow{\textsf{branch}\ 1}\!\!\!\twoheadrightarrow \quad \textsc{spec-nonspec} \tag{2.3}$$

$$!\,(\texttt{array[index]}); e_1$$

$$\xhookrightarrow{\textsf{read}_{\textsf{oob}}\ \ldots \mapsfrom \ldots}\!\!\!\twoheadrightarrow \quad \textsc{spec-nonspec} \tag{2.4}$$

$$e_1$$

$$\hookrightarrow\!\!\!\twoheadrightarrow \quad \textsc{spec-try-commit}(\textsc{fence-rollback}) \tag{2.5}$$

$$\texttt{if}\ 100 < 10\ \texttt{then}\ !\,(\texttt{array[index]}); e_1\ \texttt{else}\ e_2$$

$$\hookrightarrow\!\!\!\twoheadrightarrow \quad \textsc{spec-nonspec} \tag{2.6}$$

$$\texttt{if}\ 0\ \texttt{then}\ !\,(\texttt{array[index]}); e_1\ \texttt{else}\ e_2$$

$$\xhookrightarrow{\textsf{branch}\ 0}\!\!\!\twoheadrightarrow \quad \textsc{spec-nonspec} \tag{2.7}$$

$$e_2$$

Spectre v2 (BTB): branch target injection

Spectre variant 2 targets the Branch Target Buffer (BTB), with an attacker taking advantage of processor speculation on indirect jump targets. The attack was originally demonstrated by P. Kocher et al. [2019] and is tracked by CVE-2017-5715 [Intel Corporation 2017a].

```c
// f is a function pointer
void main() {
  ...
  (*f)();
  ...
}

void gadget() {
  ...
}
```

Our example of this attack uses an indirect function call and a `gadget` function that the attacker will exploit. The processor speculates on the target of the call f and speculatively jumps to `gadget`, executing the body until it eventually rolls back and jumps to the correct function.

In λ_{spec} we model this as there being some expression e_f that computes an indirect function target and an exploitable function `gadget`. In Line 2.8, the speculation oracle "guesses" that the target of the indirect function call will be `gadget`.

e_f is an expression that computes which function to call

$\texttt{gadget} \mapsto \lambda x.e$

$e_f(1)$

$$\hookrightarrow\!\!\!\!\twoheadrightarrow \quad \textsc{spec-spec}(spec(e_f) = \mathsf{spec}\ \texttt{gadget}) \tag{2.8}$$

$\texttt{gadget}(1)$

$$\xhookrightarrow{\text{call gadget}}\!\!\!\!\twoheadrightarrow \quad \textsc{spec-nonspec} \tag{2.9}$$

$e[1/x]$

$$\hookrightarrow\!\!\!\!\twoheadrightarrow^* \quad \textsc{spec-nonspec} \tag{2.10}$$

$\ldots$

$$\hookrightarrow\!\!\!\!\twoheadrightarrow \quad \textsc{spec-try-commit}(\textsc{fence-rollback}) \tag{2.11}$$

$e_f(1)$

Spectre v4 (STL): speculative store bypass

Spectre variant 4 targets the memory disambiguation predictor, with an attacker taking advantage of processor speculation on whether stores alias. The attack was originally demonstrated by Horn [2018] and is tracked by CVE-2018-3639 [Intel Corporation 2018b].

```
// x and y are pointers
void main() {
  ...
  x = y;
  *x = 0;
  *y = 10;
```

```
    *x;       // speculatively read x = 0
    ...
}
```

The locations x and y are aliased. When reading from x, the processor does not know which writes are to aliased locations and speculatively guesses that the write to y is disjoint from the location of x. The processor thus speculatively executes the read, reading 0 for the value of x, continuing with execution with this incorrect value until it eventually rolls back and executes with the proper value of 10.

In λ_{spec}, we have two variables **x** and **y** which are aliased to the same location ℓ. We define the microarchitectural state ($\Phi.a$) to include a map tracking a predicted most recent store for each location. In Line 2.12, the microarchitectural state is (correctly) updated to track that the value 0 has been stored to the location ℓ. In Line 2.13, the microarchitectural state misses the aliasing, leaving the predicted store to ℓ as having the value 0 (instead of the correct 10). Then, in Line 2.14 the speculation oracle uses the microarchitectural state to (incorrectly) predict the result of the load as 0, with execution speculatively continuing with the incorrect value.

$x, y = \ell$

The speculative state contains a predicted map of locations to last write.

$$x := 0; y := 10; !\,x; \ldots$$

$$\xhookrightarrow{\mathsf{write_{ib}}\ 0 \mapsto \ell}\!\!\!\rightarrow \quad \textsc{spec-nonspec}(\Phi.a = \{\ell \mapsto 0\}) \tag{2.12}$$

$$y := 10; !\,x; \ldots$$

$$\xhookrightarrow{\mathsf{write_{ib}}\ 0 \mapsto \ell}\!\!\!\rightarrow \quad \textsc{spec-nonspec}(\Phi.a = \{\ell \mapsto 0\}) \tag{2.13}$$

$$!\,x; \ldots$$

$$\hookrightarrow\!\!\!\rightarrow \quad \textsc{spec-spec}(spec(!\,x) = \mathsf{spec}\ 0) \tag{2.14}$$

$$0; \ldots$$

$$\hookrightarrow\!\!\!\rightarrow \quad \textsc{spec-try-commit}(\textsc{fence-rollback}) \tag{2.15}$$

$$!\,x; \ldots$$

$$\xhookrightarrow{\mathsf{read_{ib}}\ 10 \hookleftarrow \ell}\!\!\!\rightarrow \quad \textsc{spec-nonspec} \tag{2.16}$$

$$10; \ldots$$

Spectre v5 (RSB): ret2spec

Spectre variant 5 targets the return stack buffer, with an attacker taking advantage of processor speculation on the indirect jump when returning from a function. The attack was originally demonstrated by Maisuradze and Rossow [2018] and is tracked by CVE-2017-5715 [Intel Corporation 2017a].

```
void main() {
  ...
  return 2 + foo();
}
```

```
int foo() {
  ...
  return 1;
}

void gadget() {
  ...
}
```

When foo is called from main, the return address (pointing at e) is pushed to the stack. When returning from foo the speculation predicts the return address to be gadget, jumps there, then executes code erroneously before rolling back and jumping to e.

In λ_{spec} we define three functions: foo, rest, and gadget. The first, foo, is the same as the C function above: it performs some computation then returns 1. We model the return by passing the return "address" as a continuation, k, with rest capturing the rest of the main function that will be returned (jumped) to. This could also be modeled by pushing the return continuations ("addresses") to a stack in memory, modeling an explicit return stack discipline. In Line 2.19, the speculator predicts that the return address is not the address of rest but is instead the address of gadget.

$$\texttt{foo} \mapsto \lambda k.e_{\texttt{foo}}; k(1) \qquad \texttt{rest} \mapsto \lambda x.2 + x \qquad \texttt{gadget} \mapsto \lambda x.e$$

$\texttt{foo}(\texttt{rest})$

$$\xhookrightarrow{\text{call } \texttt{foo}} \quad \textsc{spec-nonspec} \tag{2.17}$$

$e_{\texttt{foo}}; (k\{\texttt{rest}\})(1)$

$$\hookrightarrow^{*} \quad \textsc{spec-nonspec} \tag{2.18}$$

$(k\{\texttt{rest}\})(1)$

$$\hookrightarrow \quad \textsc{spec-spec}(spec(k\{\texttt{rest}\}) = \mathsf{spec}\ \texttt{gadget}) \tag{2.19}$$

$\texttt{gadget}(1)$

$$\xhookrightarrow{\text{call } \texttt{gadget}} \quad \textsc{spec-nonspec} \tag{2.20}$$

$e[1/x]$

$$\hookrightarrow^{*} \quad \textsc{spec-nonspec} \tag{2.21}$$

$\ldots$

$$\hookrightarrow \quad \textsc{spec-try-commit}(\textsc{fence-rollback}) \tag{2.22}$$

$(k\{\texttt{rest}\})(1)$

$$\hookrightarrow \quad \textsc{spec-nonspec} \tag{2.23}$$

$\texttt{rest}(1)$

$$\xhookrightarrow{\text{call } \texttt{rest}} \quad \textsc{spec-nonspec} \tag{2.24}$$

$2 + x\{1\}$

$$\hookrightarrow^{*} \quad \textsc{spec-nonspec} \tag{2.25}$$

$\ldots$

$$
\begin{array}{rcll}
\text{API contexts} & \Gamma & ::= & \overline{(f : z)} \\
\text{libraries} & L & ::= & \overline{f \mapsto (z_f, \lambda \overline{x}.e)} \\
\text{secret contexts} & \Delta & ::= & \overline{x \mapsto (z_{loc}, z_{len})} \\
\text{heaplets} & H & : & \mathbb{Z} \rightharpoonup m \\
\\
\text{app traces} & A & ::= & \tau^{\texttt{lib}\rightarrow\texttt{app}} +\!\!+ \overline{\delta^{\texttt{app}}} +\!\!+ \tau^{\texttt{app}\rightarrow\texttt{lib}} \\
\text{lib traces} & L & ::= & \tau^{\texttt{app}\rightarrow\texttt{lib}} +\!\!+ \overline{\delta^{\texttt{lib}}} +\!\!+ \tau^{\texttt{lib}\rightarrow\texttt{app}} \\
\text{traces} & T & ::= & A \circ L \circ T \\
 & & | & \tau^{\texttt{lib}\rightarrow\texttt{app}} +\!\!+ \overline{\delta^{\texttt{app}}} +\!\!+ \textsf{end}\ v^{\texttt{app}\rightarrow\texttt{lib}}
\end{array}
$$

Figure 2.11. Syntax of programs and traces for λ_{spec}.

2.2 Robust Constant Time

Next we formalize the security semantics of cryptographic libraries when used within an application.

2.2.1 Programs and Traces

Figure 2.11 shows the syntax for defining libraries and applications. An *API context*, Γ is a map from function names to the number of arguments that function takes and defines the external API for a library. Given an API context Γ a *library* L is a set of functions and their heap locations for each of the external names in Γ. We capture this with the well-formedness judgment $\Gamma \vDash L$, which additionally allows L to contain internal functions (defined in Figure 2.12).

We assume that secrets are stored in memory. A *secret context* Δ is a map from secret variable names x (used to refer to that secret block in application code) to a pair of integer values denoting the location (address) and length of the corresponding block. Given an API context Γ and a secret context Δ, an *application* is a pair of a *heaplet* H (a partial heap containing initial state and application functions), and a "main" expression e with free variables from Γ and Δ. We define a well-formedness judgment for applications

$$\boxed{\Gamma \vDash L}$$

$$\frac{}{\bullet \vDash \bullet} \qquad \frac{\mathsf{length}(\overline{x}) = z \qquad \Gamma \vDash L \qquad z_{loc} \notin \mathsf{locs}(L)}{(f : z) :: \Gamma \vDash f \mapsto (z_f, \lambda \overline{x}.e) :: L} \qquad \frac{(f : z) :: \Gamma \vDash L \qquad z_g \notin \mathsf{locs}(L)}{(f : z) :: \Gamma \vDash g \mapsto (z_g, \lambda \overline{x}.e) :: L}$$

$$\boxed{\Gamma, \Delta \vdash (H, e)}$$

$$\frac{\mathsf{fv}(H) \cup \mathsf{fv}(e) \subseteq \mathrm{dom}(\Gamma) \uplus \mathrm{dom}(\Delta) \qquad \forall \lambda_\ell \overline{x}.e \in \mathrm{cod}(H).\ell = \mathsf{app}}{\Gamma, \Delta \vdash (H, e)}$$

$$\boxed{L \vDash S}$$

$$\frac{\forall \lambda_\ell \overline{x}.e \in \mathrm{cod}(S.H).\ \ell = \mathsf{app}}{\bullet \vDash S} \qquad \frac{S(z_f) = \lambda_{\mathtt{lib}} \overline{x}.e \qquad L \vDash S[H := S.H - \{z_f\}]}{f \mapsto (z_f, \lambda \overline{x}.e) :: L \vDash S}$$

$$\boxed{\Delta \mid H \vDash S = S}$$

$$\frac{S.H = H \qquad S.p = \mathsf{app}}{\bullet \mid H \vDash S = S} \qquad \frac{\begin{array}{c} S(z_{loc}).size = S'(z_{loc}).size = z_{len} \\ \Delta \mid H \vDash S[H := S.H - \{z_{loc}\}] = S'[H := S'.H - \{z_{loc}\}] \end{array}}{x \mapsto (z_{loc}, z_{len}) :: \Delta \mid H \vDash S = S'}$$

$$\boxed{L \mid \Delta \mid H \vDash S = S}$$

$$\boxed{L \mid \Delta \mid H \vDash S}$$

$$\frac{\Delta \mid H \vDash S = S'}{\bullet \mid \Delta \mid H \vDash S = S'} \qquad \frac{\begin{array}{c} S.H(z_f) = S.H'(z_f) = \lambda_{\mathtt{lib}} \overline{x}.e \\ L \mid \Delta \mid H \vDash S[H := S.H - \{z_f\}] = S'[H := S'.H - \{z_f\}] \end{array}}{f \mapsto (z_f, \lambda \overline{x}.e) :: L \mid \Delta \mid H \vDash S = S'}$$

$$\frac{L \mid \Delta \mid H \vDash S = S}{L \mid \Delta \mid H \vdash S}$$

Figure 2.12. Well-formedness of λ_{spec} programs.

$$
\begin{array}{rcl}
e[\bullet] & \triangleq & e \\
e[x \mapsto (z_b, z) :: \Delta] & \triangleq & (e[z_b/x])[\Delta] \\
H[\Delta] & \triangleq & z \mapsto H(z)[\Delta] \\
\{size, p, v\}[\Delta] & \triangleq & \{size, p, v[\Delta]\} \\
(\lambda_\ell \overline{x}.e)[\Delta] & \triangleq & \lambda_\ell \overline{x}.(e[\Delta])
\end{array}
$$

$$
\begin{array}{rcl}
e[\bullet] & \triangleq & e \\
e[f \mapsto (z_f, \lambda_{\texttt{lib}} \overline{x}.e) :: L] & \triangleq & (e[z_f/f])[L] \\
H[L] & \triangleq & z \mapsto H(z)[L] \\
\{size, p, v\}[L] & \triangleq & \{size, p, v[L]\} \\
(\lambda_\ell \overline{x}.e)[L] & \triangleq & \lambda_\ell \overline{x}.(e[L])
\end{array}
$$

Figure 2.13. Substitution in λ_{spec}.

$$
\frac{\langle S \mid \bullet :: e^{\text{app}} \rangle \overset{T}{\Rightarrow}{}^{*} \langle S' \mid \bullet :: v^{\text{app}} \rangle}{\langle S \mid e \rangle \downarrow^{T} \langle S' \mid v \rangle}
\qquad
\frac{\langle S \mid \bullet :: e^{\text{app}} \rangle \overset{\overline{\epsilon}}{\Rightarrow}{}^{*}_{C} \langle S' \mid \bullet :: v^{\text{app}} \rangle}{\langle S \mid e \rangle \downarrow^{\overline{\epsilon}}_{C} \langle S' \mid v \rangle}
$$

$$
\frac{\langle \Phi \mid \bullet :: e \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\!\rightarrow}{}^{*} \langle \Phi' \mid \bullet :: v \rangle \qquad \Phi'.\Xi = \bullet}{\langle \Phi \mid e \rangle \downarrow^{\overline{\delta}}_{S} \langle \Phi' \mid v \rangle}
\qquad
\frac{\langle \Phi \mid \bullet :: e \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\!\rightarrow}{}^{*}_{C} \langle \Phi' \mid \bullet :: v \rangle \qquad \Phi'.\Xi = \bullet}{\langle \Phi \mid e \rangle \downarrow^{\overline{\delta}}_{SC} \langle \Phi' \mid v \rangle}
$$

$$
\begin{array}{rcl}
\mathsf{traces}(\langle S \mid e \rangle) & \triangleq & \{\mathsf{begin} +\!\!+ T +\!\!+ \mathsf{end}\ v \mid \langle S \mid e \rangle \downarrow^{T} \langle S' \mid v \rangle\} \\
\mathsf{concurrentTraces}(\langle S \mid e \rangle) & \triangleq & \{\mathsf{begin} +\!\!+ \overline{\epsilon} +\!\!+ \mathsf{end}\ v \mid \langle S \mid e \rangle \downarrow^{\overline{\epsilon}}_{C} \langle S' \mid v \rangle\} \\
\mathsf{specTraces}(\langle \Phi \mid e \rangle) & \triangleq & \{\mathsf{begin} +\!\!+ \overline{\delta} +\!\!+ \mathsf{end}\ v \mid \langle \Phi \mid e \rangle \downarrow^{\overline{\delta}}_{S} \langle \Phi' \mid v \rangle\} \\
\mathsf{concurrentSpecTraces}(\langle \Phi \mid e \rangle) & \triangleq & \{\mathsf{begin} +\!\!+ \overline{\delta} +\!\!+ \mathsf{end}\ v \mid \langle \Phi \mid e \rangle \downarrow^{\overline{\delta}}_{SC} \langle \Phi' \mid v \rangle\}
\end{array}
$$

Figure 2.14. Definition of traces for λ_{spec}.

$\Gamma, \Delta \vdash (H, e)$ (detailed in Figure 2.12).

A *whole program* is then composed of a library L plugged into an application (H, e). To define this we build a judgment $L \mid \Delta \mid H \vDash S$ (defined in Figure 2.12). This captures that the initial state S 1. contains all of the functions defined in L, 2. contains locations for all of the secrets in Δ, and 3. contains the application heaplet H. We then define substitution operations $e[\Delta][L]$ and $S[\Delta][L]$ replacing the API variable names from Γ with the function locations in L and the secret variables in Δ with their actual block locations (defined in Figure 2.13).

Our operational semantics captures a sequence of events, but for whole programs we factor this sequence into a *program trace* (T) with additional structure (shown in Figure 2.11). This trace captures the decomposition of execution into alternating sequences of application and library domain events, with transition events the boundaries between them. Application traces A are thus a transition event from library to application, followed by a sequence of application domain events, and then a transition back to the library. Library traces are defined similarly and the trace of an entire program is then alternating sequences of these application and library traces. For program traces we define a gluing concatenation operator $\overline{\epsilon_1}\epsilon \circ \epsilon\overline{\epsilon_2}$ which matches the sequence $\overline{\epsilon_1}\epsilon\overline{\epsilon_2}$. We use this to capture that the transition event ending an application trace and starting the subsequent library trace are in fact the same transition event. We then define trace metafunctions capturing the set of all program traces for a given program and semantics (see Figure 2.14).

2.2.2 Robust Constant Time

Our core security property, robust constant time, much like classic constant time, comes in two flavors: speculative and non-speculative (we also separate the associated concurrent versions). Intuitively, robust constant time captures that a library behaves correctly when plugged into an "unknown" context (in this case an application)

$$
\begin{array}{rcl}
\mathsf{ct}(\epsilon^{\ell}) & \triangleq & \mathsf{ct}(\epsilon) \\
\mathsf{ct}(\tau^{\ell \to \ell'}) & \triangleq & \mathsf{ct}(\tau) \\
\mathsf{ct}(\mathsf{begin}) & \triangleq & 0 \\
\mathsf{ct}(\mathsf{end}\ v) & \triangleq & \mathsf{end}\ v \\
\mathsf{ct}(\mathsf{call}\ z) & \triangleq & \mathsf{call}\ z \\
\mathsf{ct}(\mathsf{ret}\ v) & \triangleq & 0 \\
\mathsf{ct}(\mathsf{new}_p\ z@z_b) & \triangleq & \mathsf{new}_p\ z@z_b \\
\mathsf{ct}(\mathsf{read}_b\ v \hookleftarrow z_b[z_o]) & \triangleq & \mathsf{read} \hookleftarrow z_b[z_o] \\
\mathsf{ct}(\mathsf{write}_b\ v \mapsto z_b[z_o]) & \triangleq & \mathsf{write} \mapsto z_b[z_o] \\
\mathsf{ct}(\mathsf{branch}\ v) & \triangleq & \mathsf{branch}\ v \\
\mathsf{ct}(\mathsf{fence}) & \triangleq & 0 \\
\mathsf{ct}(\mathsf{protect}_p) & \triangleq & 0 \\
\mathsf{ct}(0) & \triangleq & 0
\end{array}
$$

Figure 2.15. Constant time events for λ_{spec}.

representing an attacker. For cryptographic libraries the attacker attempts to exploit vulnerabilities in the application to extract secrets, so we parameterize our definition by an *attacker*: a predicate on applications, $\mathsf{pred}_{\Gamma,\Delta} : \wp(H, e)$ that captures a vulnerability as a set of applications with that vulnerability. We instantiate $\mathsf{pred}_{\Gamma,\Delta}$ in Section 2.2.3 to capture read-only and memory-safe attackers.

Robust constant time, then, is a parameterized, robust version of classic constant time definitions. We define $\mathsf{ct} : \overline{\epsilon} \to \overline{c}$ in Figure 2.15, a meta-function that erases the components of the trace that are not visible from timing-based attacks. Robust constant time can thus be defined as follows:

Definition 8 (Robust constant time (RCT))**.** *We say a library* $\Gamma \vDash L$ *is* robustly constant time *for an attacker class* $\mathsf{pred}_{\Gamma,\Delta}$ *if, for all secret contexts* Δ*, applications* $\Gamma, \Delta \vdash (H, e)$ *such that* $\mathsf{pred}_{\Gamma,\Delta}(H, e)$*, and initial states* S_0*,* S_0' *such that* $L \mid \Delta \mid H \vDash S_0 = S_0'$ *we have that* $\mathsf{ct}(\mathsf{traces}(\langle S_0[\Delta][L] \mid e[\Delta][L]\rangle)) = \mathsf{ct}(\mathsf{traces}(\langle S_0'[\Delta][L] \mid e[\Delta][L]\rangle))$.

The judgment $L \mid \Delta \mid H \vDash S_0 = S_0'$ (see Figure 2.12) is a variant of our well-formedness

judgment for initial states capturing that S_0 and S'_0 *only* vary at the secret locations contained in Δ. Robust speculative constant time is defined similarly, however we consider the speculative traces and a speculative attacker oracle $\mathsf{spec} : A \times e \rightarrow A \times d$ (capturing varying speculative attacks).

Definition 9 (Robust speculative constant time (RSCT))**.** *We say a library* $\Gamma \vDash L$ *is* robustly speculative constant time *with respect to a speculation oracle* $\mathsf{spec} : A \times S \times e \rightarrow A \times d$ *if, for all secret contexts* Δ*, memory-safe applications* $\Gamma, \Delta \vdash (H, e)$*, initial states* S_0*,* S'_0 *such that* $L \mid \Delta \mid H \vDash S_0 = S'_0$*, microarchitectural states* $a : A$*,* $\Phi_0 = \{S = S_0[\Delta][L], a = a, \Xi = \bullet\}$*, and* $\Phi'_0 = \{S = S'_0[\Delta][L], a = a, \Xi = \bullet\}$*, we have that* $\mathsf{ct}(\mathsf{specTraces}(\langle \Phi_0 \mid e[\Delta][L] \rangle)) = \mathsf{ct}(\mathsf{specTraces}(\langle \Phi'_0 \mid e[\Delta][L] \rangle))$.

The concurrent versions simply use the set of concurrent traces. We show the definitions below.

Definition 10 (Robust constant time (RCT) for concurrent observers)**.** *We say a library* $\Gamma \vDash L$ *is* robustly constant time for concurrent observers *if, for all secret contexts* Δ*, read-only applications* $\Gamma, \Delta \vdash (H, e)$*, and initial states* S_0*,* S'_0 *such that* $L \mid \Delta \mid H \vDash S_0 = S'_0$ *we have that* $\mathsf{ct}(\mathsf{concurrentTraces}(\langle S_0[\Delta][L] \mid e[\Delta][L] \rangle)) = \mathsf{ct}(\mathsf{concurrentTraces}(\langle S'_0[\Delta][L] \mid e[\Delta][L] \rangle))$.

Definition 11 (Robust speculative constant time (RSCT) for concurrent observers)**.** *We say a library* $\Gamma \vDash L$ *is* robustly speculative constant time for concurrent observers *with respect to a speculation oracle* $\mathsf{spec} : A \times S \times e \rightarrow A \times d$ *if, for all secret contexts* Δ*, memory-safe applications* $\Gamma, \Delta \vdash (H, e)$*, initial states* S_0*,* S'_0 *such that* $L \mid \Delta \mid H \vDash S_0 = S'_0$*, microarchitectural states* $a : A$*,* $\Phi_0 = \{S = S_0[\Delta][L], a = a, \Xi = \bullet\}$*, and* $\Phi'_0 = \{S = S'_0[\Delta][L], a = a, \Xi = \bullet\}$*, we have that* $\mathsf{ct}(\mathsf{concurrentSpecTraces}(\langle \Phi_0 \mid e[\Delta][L] \rangle)) = \mathsf{ct}(\mathsf{concurrentSpecTraces}(\langle \Phi'_0 \mid e[\Delta][L] \rangle))$.

2.2.3 Attackers

As discussed we wish to protect libraries against different kinds of application vulnerabilities: read-only vulnerabilities (corresponding to the attacker model libraries

$\boxed{\Gamma \vdash \textsf{read-only}\ T}$

READ-ONLY-REC

$$\frac{\begin{array}{c} A = \tau_1{}^{\texttt{lib}\rightarrow\texttt{app}} +\!\!\!+ \overline{\delta_A}^{\texttt{app}} +\!\!\!+ \tau_2{}^{\texttt{app}\rightarrow\texttt{lib}} \qquad L = \tau_2{}^{\texttt{app}\rightarrow\texttt{lib}} +\!\!\!+ \overline{\delta_L}^{\texttt{lib}} +\!\!\!+ \tau_3{}^{\texttt{lib}\rightarrow\texttt{app}} \\ \tau_2 = \textsf{call}\ z_f \Rightarrow z_f \in \mathrm{dom}(\Gamma) \\ \textsf{wf-read-only}\ \overline{\delta_A} \qquad \textsf{wf-read-only}\ \overline{\delta_L} \implies \Gamma \vdash \textsf{read-only}\ T \end{array}}{\Gamma \vdash \textsf{read-only}\ A \circ L \circ T}$$

READ-ONLY-END

$$\frac{\textsf{wf-read-only}\ \overline{\delta}}{\Gamma \vdash \textsf{read-only}\ \tau^{\texttt{lib}\rightarrow\texttt{app}} +\!\!\!+ \overline{\delta^{\texttt{app}}} +\!\!\!+ (\textsf{end}\ v)^{\texttt{app}\rightarrow\texttt{lib}}}$$

$\boxed{\Gamma \vdash \textsf{mem-safe}\ T}$

MEMORY-SAFE-REC

$$\frac{\begin{array}{c} A = \tau_1{}^{\texttt{lib}\rightarrow\texttt{app}} +\!\!\!+ \overline{\delta_A}^{\texttt{app}} +\!\!\!+ \tau_2{}^{\texttt{app}\rightarrow\texttt{lib}} \qquad L = \tau_2{}^{\texttt{app}\rightarrow\texttt{lib}} +\!\!\!+ \overline{\delta_L}^{\texttt{lib}} +\!\!\!+ \tau_3{}^{\texttt{lib}\rightarrow\texttt{app}} \\ \tau_2 = \textsf{call}\ z_f \Rightarrow z_f \in \mathrm{dom}(\Gamma) \\ \textsf{wf-mem-safe}\ \overline{\delta_A} \qquad \textsf{wf-read-only}\ \overline{\delta_L} \implies \Gamma \vdash \textsf{read-only}\ T \end{array}}{\Gamma \vdash \textsf{mem-safe}\ A \circ L \circ T}$$

MEMORY-SAFE-END

$$\frac{\textsf{wf-mem-safe}\ \overline{\delta}}{\Gamma \vdash \textsf{mem-safe}\ \tau^{\texttt{lib}\rightarrow\texttt{app}} +\!\!\!+ \overline{\delta^{\texttt{app}}} +\!\!\!+ (\textsf{end}\ v)^{\texttt{app}\rightarrow\texttt{lib}}}$$

Figure 2.16. Definition of robust constant time (RCT) attackers.

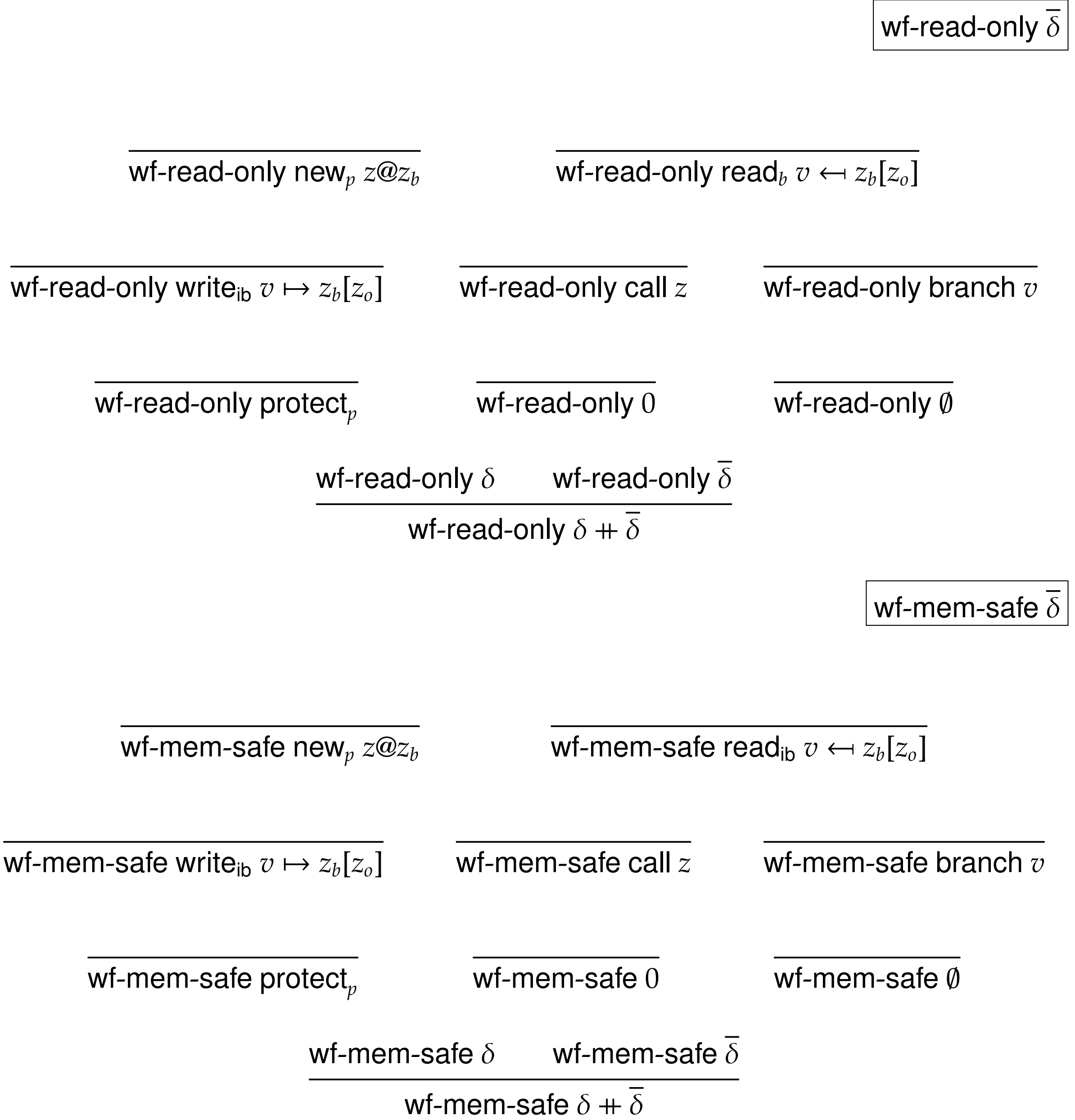


Figure 2.17. Well-formedness of traces for robust constant time (RCT) attackers.

like `LibSodium` assume), speculative vulnerabilities, and concurrent observers. Speculative vulnerabilities and concurrent observers are captured by our speculative and concurrent semantics leaving us with non-speculative read-only attackers (which we contrast with memory-safe attackers such as those written in memory-safe languages). We will use the fact that our traces T capture the back and forth of actions of the application and library and the hand-offs between them to instantiate the predicate of Definition 8. We then define read-only and memory-safe attackers by restricting the set of unsafe behaviors during the application subtraces: Read-only attackers can read memory they were not given access to but cannot write to it (and thus cannot carry out *active* attacks). In contrast, memory-safe attackers may neither read nor write memory they were not given access to. We define both read-only and memory-safe attackers as excluding control-flow exploits.

We wish to capture sets of attackers by their trace properties, however the attacker predicate is defined as a property of applications which are only a partial program and cannot be run. As such we must first link with a library before we can assess the application's (mis)behavior. But we cannot simply take an arbitrary library: an ill-formed library can break application invariants. To untangle this knot we simultaneously define the relevant *restrictions* on the application we are classifying with the *assumptions* that it may make about the library that it is running against.

Definition 12 (Read-only attackers)**.** *We say an application* $\Gamma, \Delta \vdash (H, e)$ *is a* read-only attacker *if, for all libraries* $\Gamma \vDash L$*, initial states* S_0 *such that* $L \mid \Delta \mid H \vDash S_0$*, and traces* $T \in \mathsf{traces}(\langle S_0[\Delta][L] \mid e[\Delta][L]\rangle)$*,* $\Gamma \vdash \mathsf{read\text{-}only}\ T$*.*

Figure 2.16 shows the judgment $\Gamma \vdash \mathsf{read\text{-}only}\ T$ which handles the restrictions on the application and assumptions on the library components of the trace T. The rule READ-ONLY-REC captures the back and forth of assumptions and restrictions: it decomposes the next application and library trace sequences, imposes the read-only restrictions on

the application events (wf-read-only $\overline{\delta_A}$), requires that the call into the library is an API function, and then, under the assumption that the library events do not write out of bounds, inductively requires that the rest of the trace is read-only. The definition of wf-read-only can be found in Figure 2.17: it captures that the application trace can only contain in bounds write events, but may contain arbitrary read events. Memory-safe attackers are defined similarly, with the predicate adjusted to also preclude reads from unexposed blocks.

2.3 A Robust Compiler

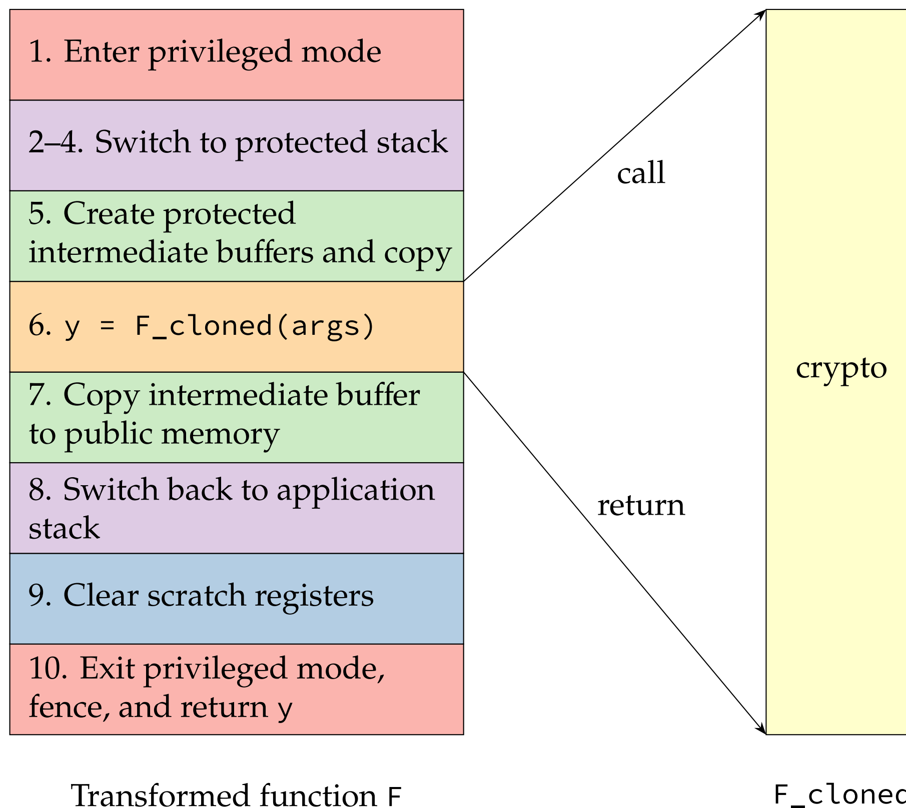
Guided by our formal models we develop RoboCop, a compiler providing robust constant time protections for cryptographic libraries against different attackers. RoboCop is built on top of the LLVM framework [Lattner and Adve 2004] and uses Intel® MPK to guarantee that secret data (cryptographic secrets and the data derived from them) is only accessible while executing trusted cryptographic library code. While RoboCop mainly operates on the library, there are two components that involve the application. First, cryptographic secrets often originate and are managed by application code, and it is therefore necessary to allocate these secrets in protected memory. To do so we provide manual MPK allocation APIs and also adapt techniques from CryptoMPK [Jin et al. 2021] to provide an alternative, automatic application transformation that securely allocates secrets. Further, for the efficiency of our library protections, we reuse a single stack allocated in protected memory. To enable this we develop a simple LLVM pass that allocates this stack on program entry.

2.3.1 Making Libraries Robust

Cryptographic code operates directly on secret data and, to prevent timing-based leaks, is required to be constant time. With this baseline security requirement we operate under the assumption that cryptographic code is *trusted*. The task of RoboCop then is to

```
int stream(u8 *c, u64 clen, u8 *n, u8 *k) {
  mpk_allow_access();
  switch_to_protected_stack();
  u8 *c_internal = mpk_malloc(clen);
  int y = stream_cloned(c_internal, clen, n, k);
  memcpy(c_internal, c, clen);
  switch_to_unprotected_stack();
  clear_scratch_registers();
  mpk_disable_access();
  fence;
  return y;
}
```

(a) Protections applied to LibSodium's Salsa20.



(b) Wrapping of cryptographic function.

Figure 2.18. RoboCop protections.

ensure that the secret data remains inaccessible even if there are vulnerabilities within the client application code. These protections are provided in three steps: 1. Cryptographic developers label the external API functions. 2. RoboCop wraps these API functions to handle the memory isolation. 3. RoboCop replaces all dynamic memory functions (`malloc` and similar) with custom MPK compatible versions, ensuring that all memory allocated within the cryptographic library is kept within the protected domain.

Figure 2.18b shows our wrapping of cryptographic API functions. For every exported function `F` in the library, a clone, `F_cloned`, is generated containing the original implementation of `F`. *Internal* calls to `F` are replaced with calls to `F_cloned`: `F` becomes the external API wrapper for use by the client. `F` is responsible for switching into and out of the protected memory region.

The new `F` takes the following domain switching steps: 1. We enable access to the protected memory region with a `wrpkru` instruction. The specification for MPK [Intel Corporation 2023] ensures this is speculatively secure: `wrpkru` will not execute speculatively and protected memory cannot be accessed until `wrpkru` is committed. 2. We get the address of the protected stack and save the current stack pointer. 3. We copy any stack arguments from the unprotected frame to the new protected stack. 4. We switch the stack pointer to the protected stack frame. 5. If concurrent protections are enabled, we allocate an internal copy buffer for the external buffers (discussed below). 6. We call `F_cloned`. 7. After the cryptographic function returns, we copy any internal buffers back to the original, public buffers. 8. We restore the previous stack pointer. 9. We clear all scratch registers which may potentially contain transient secret computation. 10. We disable access to protected memory, fence if speculative protections are enabled, and return to the application. Together these ensure that all data produced and used by the cryptographic code is within the protected memory region and the region is inaccessible to application code.

Concurrent Protections

Rather than allocate extra memory, cryptographic algorithms sometimes carry out intermediate computations within output (e.g. ciphertext) buffers. In a single-threaded context this is safe: there is no way for client code to access these buffers before they contain their final, cryptographically secure value. In a concurrent context, work like Spectre Declassified [Shivakumar et al. 2023] has shown that an attacker can recover secret information by observing intermediate results. To defend against these attacks we add a concurrent protection option to RoboCop. Here library developers annotate API arguments that are used for intermediate computation, and RoboCop allocates a buffer in protected memory for the arguments, performs the intermediate work within the protected domain, and then copies the declassified result back out to the unprotected memory.

2.3.2 Proving RoboCop Secure

On its own (concurrent) robust (speculative) constant time serves as a useful security property for understanding when library protections are secure against different attackers. However we are interested in automatically providing robust protections via RoboCop. To do so we need to understand our "source language": the assumptions we make about the source of our compilation. For RoboCop we assume that the source is *classically* (speculative) constant time and will prove that RoboCop then guarantees robust (speculative) constant time. This gives library developers flexibility: they can safely use any tool or handwritten technique to guarantee that their library implementation is constant time and then RoboCop lifts this to the *robust* counterpart.

In defining a "source language" that captures classic constant time, note that the classic notion of constant time is that executing a library function produces invariant traces. As such classic constant time is in fact a restricted form of robust constant time,

where, for a given API context Γ, the main function is defined by the grammar $e_\Gamma ::= f(\overline{v})$ for $f \in \Gamma$. That is, "source applications" are simply individual function calls into the library. There is a slight caveat: classical constant time also requires that secrets have not leaked into the application state. With this in mind we define classical constant time as the following variation on robust constant time:

Definition 13 (Classical constant time)**.** *We say a library* $\Gamma \vDash L$ *is* classically constant time *if, for all secret contexts* Δ*, classical "applications"* $\Gamma, \Delta \vdash (H, e_\Gamma)$*, and initial states* S_0*,* S_0' *such that* $L \mid \Delta \mid H \vDash S_0 = S_0'$*, we have that for all traces* $\langle S_0[\Delta][L] \mid e_\Gamma[\Delta][L]\rangle \downarrow^{\overline{\delta}} \langle S_1 \mid v\rangle$ *there exists a trace* $\langle S_0'[\Delta][L] \mid e_\Gamma[\Delta][L]\rangle \downarrow^{\overline{\delta'}} \langle S_1' \mid v\rangle$ *such that* $\mathsf{ct}(\overline{\delta}) = \mathsf{ct}(\overline{\delta'})$ *and* $S_1(\mathrm{dom}(H)) = S_1'(\mathrm{dom}(H))$.

The speculative version is defined similarly:

Definition 14 (Classical speculative constant time)**.** *We say a library* $\Gamma \vDash L$ *is* classically speculative constant time *with respect to a speculation oracle* $\mathsf{spec} : A \times S \times e \to A \times d$ *if, for all secret contexts* Δ*, classical "applications"* $\Gamma, \Delta \vdash (H, e_\Gamma)$*, initial states* S_0*,* S_0' *such that* $L \mid \Delta \mid H \vDash S_0 = S_0'$*, microarchitectural states* $a : A$*,* $\Phi_0 = \{S = S_0[\Delta][L], a = a, \Xi = \bullet\}$*, and* $\Phi_0' = \{S = S_0'[\Delta][L], a = a, \Xi = \bullet\}$*, we have that for all traces* $\langle \Phi_0 \mid e_\Gamma[\Delta][L]\rangle \downarrow_{\mathcal{S}}^{\overline{\delta}} \langle \Phi_1 \mid v\rangle$ *there exists a trace* $\langle \Phi_0' \mid e_\Gamma[\Delta][L]\rangle \downarrow_{\mathcal{S}}^{\overline{\delta'}} \langle \Phi_1' \mid v\rangle$ *such that* $\mathsf{ct}(\overline{\delta}) = \mathsf{ct}(\overline{\delta'})$*,* $\Phi_1.S(\mathrm{dom}(H)) = \Phi_1'.S(\mathrm{dom}(H))$*, and* $\Phi_1.a = \Phi_1'.a$*.*

Our compiler correctness properties are then that the compiler guarantees (speculative) (concurrent) robust constant under the assumption that the library is (speculatively) classically constant time.[5]

A Formal Model of RoboCop

Formally, we represent RoboCop as a parameterized compiler $\mathbb{C}$ which transforms source libraries ($\Gamma \vDash L$) into protected targets L. We have four compilers: 1. $\mathbb{C}_{\mathsf{ro}}$, which

[5]This property can be equivalently viewed as preservation of robust constant time from our source language of classical "applications" to the target language containing the contexts of our attacker model.

$$\boxed{\Delta \mid H \vDash_{\texttt{protected}} S = S}$$

$$\frac{S.H = H \qquad S.p = \texttt{app}}{\bullet \mid H \vDash_{\texttt{protected}} S = S} \qquad \frac{\begin{array}{c} S(z_{loc}).size = S'(z_{loc}).size = z_{len} \\ S(z_{loc}).p = S'(z_{loc}).p = \texttt{protected} \\ \Delta \mid H \vDash_{\texttt{protected}} S[H := S.H - \{z_{loc}\}] = S'[H := S'.H - \{z_{loc}\}] \end{array}}{x \mapsto (z_{loc}, z_{len}) :: \Delta \mid H \vDash_{\texttt{protected}} S = S'}$$

Figure 2.19. Initial well-formedness of states with protected memory.

protects libraries from read-only attackers, 2. $\mathbb{C}_{\text{spec}}$ which protects against speculative attackers, and 3. $\mathbb{C}_{\text{ro-co}}$, and 4. $\mathbb{C}_{\text{spec-co}}$ which also protect against concurrent observers. $\mathbb{C}_{\text{ro}}$ transforms all internal uses of $\texttt{new}_p\ e$ into $\texttt{new}_{\texttt{protected}}\ e$, and wraps each external API function with $\texttt{protect}_{\texttt{protected}}$ and $\texttt{protect}_{\texttt{public}}$. $\mathbb{C}_{\text{spec}}$ is the same as $\mathbb{C}_{\text{ro}}$ but also inserts a fence before returns. The concurrent compilers additionally reallocate all external buffers used by the library within protected memory, and insert memory copying to and from the external and protected buffers. Formal definitions of the compilers are in Appendix A.3.

To capture that the application manages the secret buffers and must allocate them in protected memory we slightly modify the initial state well-formedness judgment to capture that each secret block in Δ is allocated in the `protected` memory region (see Figure 2.19). We additionally assume that applications and libraries do not contain any $\texttt{protect}_p$ expressions prior to being run through our compiler. We prove that each compiler is secure and guarantees its corresponding robust constant time property by constructing semantic interpretations of the traces. The theorem statement that $\mathbb{C}_{\text{ro}}$ guarantees read-only robust constant time is shown below (the theorems and proofs are in Appendix A.4):

Theorem 5 ($\mathbb{C}_{\text{ro}}$ guarantees read-only robust constant time)**.** *If* $\Gamma \vDash L$ *is classically constant time and does not contain any* $\texttt{protect}_p$ *subterms, then* $\mathbb{C}_{\text{ro}}(\Gamma \vDash L)$ *is robustly constant time for*

read-only attackers (that do not contain `protect`$_p$*).*

2.4 Evaluation

To evaluate the cost of guaranteeing robust constant time we ask the following questions:

Q1: What is the overhead of *robust* constant time against read-only attackers? (§ 2.4.1)

Q2: What is the overhead of *robust* constant time against speculative attackers? (§ 2.4.1)

Q3: What is the overhead of *robust* constant time against concurrent observers? (§ 2.4.2)

Benchmarks To study the performance of RoboCop on a wide range of cryptographic code we modify the SUPERCOP [VAMPIRE 2022] cryptographic benchmarking tool. SUPERCOP's benchmarking suite is broken down into operations: we focus on its collections of implementations of authenticated encryption (**aead**), Diffie-Hellman key exchange (**dh**), public key encryption (**encrypt**), key encapsulation mechanism (**kem**), public key signatures (**sign**), and stream cipher (**stream**) algorithms. Within each of these operations SUPERCOP collects multiple implementations of each algorithm: e.g. the **stream** data set contains several implementations of both Salsa20 and ChaCha20. The original design intent of SUPERCOP is then to find the fastest each cryptographic algorithm can run on a given machine. To this end its benchmarking tries every implementation of an algorithm with every compiler in its test set. These implementations are benchmarked multiple times across a range of input data sizes, recording data for the fastest implementation-compiler pair.

RoboCop is implemented as a set of LLVM passes, so we restrict the compiler test set to Clang with the `-O3`, `-O2`, `-Os`, and `-O` flags. We manually remove all non-C/++implementations, as RoboCop does not handle assembly or Rust code. Beyond this manual removal, SUPERCOP automatically prunes implementations that fail to

compile or run, e.g. implementations that rely on non-standard behavior of the GCC compiler and do not compile with Clang. The findings presented below contain all data remaining after the manual removal of non-C/C++ implementations and SUPERCOP's automatic pruning, all of which occurred before data analysis.

With its broad suite of algorithms and its find-the-fastest methodology, SUPERCOP is a more *robust* means of benchmarking cryptographic software security techniques and we encourage future designers to use it for benchmarking. We found its selecting from multiple compilation levels particularly beneficial as, due to the cascading effects of optimizations, comparing at the same optimization level is not truly a head-to-head comparison. Indeed we found that sometimes the fastest optimization level would differ between RoboCop and the baseline, with several instances of `-O3` producing significantly slower code in combination with the protections than `-O`.

Machine and software setup

We run all benchmarks on a 13th Gen Intel® Core™ i9-13900KS, with 125GB RAM, and running Linux kernel version 6.3.0. We run SUPERCOP configured to collect data only on cores with the same frequency and our data is collected from 5.6 GHz cores. RoboCop adds new passes to LLVM 16.0.2 and is split into two passes: the library pass adds the protections and the application pass allocates a stack in protected memory on program entry. SUPERCOP defines API functions for each operation: these are annotated as the external API for RoboCop. We manually label secret key buffers and insert protected allocations for them in the protected versions. For the concurrent protection benchmarks we label the ciphertext arguments on the **dh** and **stream** APIs as being used for internal computation. The speculative protections differ from the read-only protections solely in the insertion of a single fence before returning from the library. We use a modified version of jemalloc 5.2.0 [Jin et al. 2021] patched to provide MPK allocation functions. Our baseline replaces the libc malloc implementation with

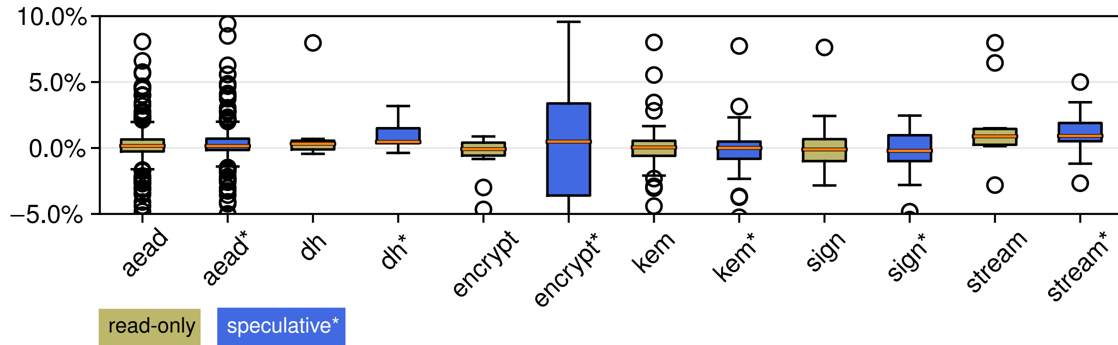


Figure 2.20. Box plot of overheads for read-only and speculative protections* compared to unprotected baseline. Cutoff at 10% overhead, outliers beyond this point are discussed directly.[6]

an unpatched version of jemalloc.

Summary of results

We find that robust constant time protections can generally be guaranteed with minimal overhead (less than 5% in almost all cases for both read-only and speculative protections), with a small number of outliers with a peak of 40% overhead. At small data sizes highly optimized stream ciphers also carry a large overhead (with a median around 33% for read-only and 37% for speculative protections), however these workloads take on the order of a few hundred cycles. We also find that concurrent protections add additional overhead but the resulting costs remain minimal (the majority have overheads under 6%).

2.4.1 Read-Only and Speculative Attackers

We measure the cost of ensuring robust constant time against read-only and speculative attackers across six data sets (shown in Table 2.1 and Figure 2.20). For the largest data size for each benchmark we find that the median overhead across all

[6]For clarity: 21/385 read-only and 23/385 speculative **aead** implementations lie above the whiskers and below the cutoff.

Table 2.1. Overheads for read-only and speculative protections vs. unprotected baseline. N is the number of algorithms in the dataset, Size is the size of the operation's input in bytes, Q_1 and Q_3 are the first and third quartile overheads, and the median overhead is of the overheads of the mean runtimes.

(a) Overheads for read-only protections.

Benchmark	N	Size	Q_1	Median	Q_3
aead	385	2048	-0.26%	**0.16%**	0.65%
dh	9	fixed	-0.11%	**0.32%**	0.56%
encrypt	15	4237	-0.57%	**-0.09%**	0.41%
kem	47	fixed	-0.59%	**0.04%**	0.55%
sign	40	4237	-1.00%	**-0.11%**	0.67%
stream	11	4096	0.24%	**0.88%**	1.45%

(b) Overheads for speculative protections.

Benchmark	N	Size	Q_1	Median	Q_3
aead	385	2048	-0.15%	**0.15%**	0.71%
dh	9	fixed	0.36%	**0.46%**	1.50%
encrypt	15	4237	-3.61%	**0.48%**	3.38%
kem	47	fixed	-0.82%	**0.00%**	0.49%
sign	40	4237	-1.00%	**-0.22%**	0.97%
stream	11	4096	0.51%	**0.92%**	1.90%

Table 2.2. Read-only and speculative protection overheads for **stream** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline.

(a) Read-only protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
1	29.07%	**33.40%**	55.56%	4.29e+02
128	15.08%	**21.66%**	24.78%	7.69e+02
256	8.23%	**12.01%**	17.52%	1.44e+03
512	5.04%	**6.71%**	9.88%	2.11e+03
1024	3.39%	**5.27%**	7.28%	4.11e+03
2048	1.54%	**2.03%**	2.73%	7.72e+03
4096	0.24%	**0.88%**	1.45%	1.54e+04

(b) Speculative protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
1	27.63%	**37.49%**	55.43%	4.29e+02
128	15.63%	**21.25%**	25.35%	7.69e+02
256	9.50%	**13.48%**	15.79%	1.44e+03
512	5.03%	**6.80%**	8.81%	2.11e+03
1024	3.48%	**4.94%**	6.60%	4.11e+03
2048	1.60%	**2.45%**	3.82%	7.72e+03
4096	0.51%	**0.92%**	1.90%	1.54e+04

Table 2.3. Read-only and speculative protection overheads for **aead** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline.

(a) Read-only protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
1	0.59%	**3.92%**	10.07%	7.14e+03
128	0.14%	**1.46%**	5.00%	1.60e+04
256	0.01%	**0.80%**	2.67%	2.69e+04
512	-0.03%	**0.45%**	1.70%	4.93e+04
1024	-0.07%	**0.29%**	0.98%	9.42e+04
2048	-0.26%	**0.16%**	0.65%	1.82e+05

(b) Speculative protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
1	0.63%	**4.01%**	10.18%	7.14e+03
128	0.13%	**1.65%**	4.99%	1.60e+04
256	0.02%	**0.82%**	2.65%	2.69e+04
512	0.00%	**0.56%**	1.68%	4.93e+04
1024	-0.08%	**0.34%**	1.05%	9.42e+04
2048	-0.15%	**0.15%**	0.71%	1.82e+05

Table 2.4. Read-only and speculative protection overheads for **encrypt** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline.

(a) Read-only protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
29	-0.58%	**0.50%**	0.64%	5.67e+05
59	-0.27%	**0.38%**	0.69%	5.65e+05
117	-0.07%	**0.49%**	0.71%	5.59e+05
231	-0.24%	**0.38%**	0.65%	5.50e+05
453	-0.50%	**0.26%**	0.72%	9.03e+05
709	-0.70%	**0.21%**	0.49%	1.13e+06
2711	-0.49%	**0.06%**	0.73%	3.92e+06
4237	-0.57%	**-0.09%**	0.41%	6.09e+06

(b) Speculative protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
29	-3.53%	**0.65%**	3.59%	5.67e+05
59	-3.36%	**0.68%**	3.85%	5.65e+05
117	-3.67%	**0.64%**	3.06%	5.59e+05
231	-3.60%	**0.58%**	3.53%	5.50e+05
453	-3.67%	**0.59%**	2.97%	9.03e+05
709	-3.60%	**0.58%**	3.79%	1.13e+06
2711	-3.57%	**0.59%**	3.53%	3.92e+06
4237	-3.61%	**0.48%**	3.38%	6.09e+06

Table 2.5. Read-only and speculative protection overheads for **sign** ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline.

(a) Read-only protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
29	-1.09%	**-0.12%**	0.96%	2.99e+08
59	-1.20%	**-0.21%**	0.67%	2.99e+08
117	-0.98%	**-0.22%**	0.48%	3.08e+08
231	-1.08%	**-0.22%**	0.50%	3.09e+08
453	-1.16%	**-0.29%**	0.48%	3.09e+08
709	-1.13%	**-0.26%**	0.48%	3.09e+08
2711	-1.22%	**-0.22%**	0.48%	2.99e+08
4237	-1.00%	**-0.11%**	0.67%	3.e+08

(b) Speculative protection overheads.

Size	Q_1	**Median**	Q_3	Baseline cycles
29	-0.57%	**0.27%**	1.83%	2.99e+08
59	-1.05%	**-0.08%**	1.27%	2.99e+08
117	-0.65%	**-0.04%**	1.55%	3.08e+08
231	-0.62%	**0.05%**	1.88%	3.09e+08
453	-0.96%	**0.01%**	1.71%	3.09e+08
709	-0.96%	**-0.04%**	1.39%	3.09e+08
2711	-0.99%	**-0.09%**	1.28%	2.99e+08
4237	-1.00%	**-0.22%**	0.97%	3.e+08

benchmarked algorithms is below 1% for both read-only and speculative protections and that 75% of all implementations for each primitive have overheads under 2% for read-only protections and 4% for speculative protections. For algorithms with varying input lengths, SUPERCOP measures performance across a wide range of lengths. We show the varying overheads across these sizes for stream ciphers in Table 2.2. The overhead increases as data sizes get smaller, with a median overhead of 33% for encrypting a single byte with read-only protections and 37% for speculative protections. Fortunately, at this data size encryption only takes a few hundred cycles so this high relative overhead remains a minimal raw cost.

For 75% of implementations, read-only protections have overheads below 2% and speculative protections below 4%, there are some outliers above 10%: one **kem** implementation has a read-only overhead of 17% and a speculative overhead of 26%; two **sign** implementations have read-only overheads of 15% and 16% and speculative overheads of 16% and 35%; and four **aead** implementations have read-only overheads between 26% and 37% and speculative overheads between 28% and 42%. The two **sign** outliers are based on the learning with errors problem [Alkim et al. 2020] and exhibit high runtime variance in the baseline that includes the measured overhead of RoboCop. For the remaining outliers, the only consistent measure we observe in the performance statistics (as measured using the perf tool) are an increase on the order of 10% in page faults. We hypothesize the observed speedups are due to 1. noise as the baseline number of cycles is often small and 2. the separate protected allocator. We've previously observed better locality properties with separate library allocators.

2.4.2 Concurrent Attackers

To protect against concurrent attackers it is necessary for buffers containing intermediate values derived from secrets to remain within the MPK protected memory. In its read-only attacker protections mode RoboCop ensures all memory originating

Table 2.6. Overhead of read-only and speculative protections vs. overhead with concurrent protections.

(a) Diffie-Hellman key exchange (**dh**) algorithms

	NON-CONCURRENT			CONCURRENT		
Base protections	Q_1	**Median**	Q_3	Q_1	**Median**	Q_3
read-only	-0.11%	**0.32%**	0.56%	0.24%	**0.74%**	1.26%
speculative	0.36%	**0.46%**	1.50%	0.10%	**0.69%**	1.07%

(b) Stream (**stream**) ciphers

		NON-CONCURRENT			CONCURRENT		
Base protections	Size	Q_1	**Median**	Q_3	Q_1	**Median**	Q_3
read-only	4096	0.24%	**0.88%**	1.45%	1.77%	**2.39%**	5.67%
speculative	4096	0.51%	**0.92%**	1.90%	1.61%	**2.14%**	2.60%

from cryptographic code meets this requirement, however some cryptographic implementations use external, public buffers as internal, intermediate (private) buffers. To measure the cost of protecting these intermediate buffers we benchmark the **dh** and **stream** data sets with RoboCop's concurrent protections mode. We annotate the top-level SUPERCOP API functions `crypto_dh` and `crypto_stream` to mark the ciphertext argument as being used for intermediate computation. In the case of **stream** the size of this buffer is dynamically determined so we mark the plaintext length argument as the size for allocating a secure intermediate buffer. Table 2.6 shows the median overheads for the read-only protections compared to the median overheads for the concurrent protections. For our **dh** data set protecting these intermediate buffers increases the median overhead from 0.32% to 0.74% for read-only attackers and from 0.46% to 0.69% for speculative attackers. For our stream cipher data set at the largest data size the increase is greater, with the median increasing from 0.88% to 2.39% for read-only attacker and 0.92% to 2.14% for speculative attackers. We hypothesize that the higher overhead

for stream ciphers is due to the dynamic allocation whereas the statically sized buffer of the Diffie-Hellman API allows optimizations in both allocation and copying back to the external buffer.

Our benchmarking treats every algorithm within each data set as if they use public buffers for intermediate computations. In practice RoboCop lets library designers label specifically which/if buffers are used for intermediate computations. This ensures that code that does not use the external buffers never has to pay the price for the extra allocation and copying and that the same library code base can be used in all contexts: in single-threaded mode RoboCop can omit the allocation and copy but in concurrent mode it handles the creation of a protected intermediate buffer.

2.5 Limitations

Our implementation of RoboCop has a few limitations: 1. While RoboCop handles observers in concurrent threads it does not handle running concurrent *cryptographic library* code. To handle this we could allocate a single protected stack per thread. It would be safe to reuse a single MPK protection key across all threads as concurrent threads would only have access while running cryptographic library code. 2. RoboCop assumes that no other code is using MPK. 3. RoboCop does not handle ensuring that protected memory does not get dumped on crashes, written to disk, or that it is cleared at the end of execution. These could be handled in one place by operating on MPK protected pages. 4. RoboCop assumes the speculative behavior of MPK follows Intel's specification and `wrpkru` will never execute speculatively and protected memory will never be accessed speculatively until protections have been fully committed [Intel Corporation 2023]. Hardware bugs such as meltdown-pk [Canella et al. 2019] violate these assumptions, and we rely on hardware fixes for such bugs (for instance official patches have already been provided for the machine used for our evaluation). 5. RoboCop currently only works

on C/C++ code. While we believe compilers are an effective means of both separating security concerns and de-duplicating implementation work, it should be possible for library authors to implement parameterized RCT protections via preprocessor macros, templates, traits, etc. We encourage cryptographic library developers to use our attacker models to ground their protections and to offer more efficient, parameterized protections moving forward.

2.6 Related Work

Memory isolation

MPK has been used to provide in-process memory isolation [Hedayati et al. 2019; Vahldiek-Oberwagner et al. 2019], including between safe and unsafe Rust code [Gülmez et al. 2023; Kirth et al. 2022; Rivera et al. 2021]. MPK protections can be modified by unprivileged instructions, as such RoboCop assumes that applications do not have access to these instructions and including through control-flow hijacks. Countermeasures like binary rewriting [Vahldiek-Oberwagner et al. 2019], system call filtering [Schrammel, Weiser, Sadek, et al. 2022; Voulimeneas et al. 2022], and CFI schemes [Burow, Carr, et al. 2017; Burow, Zhang, et al. 2019] could be used to lift these assumptions.

Similar to RoboCop, CryptoMPK [Jin et al. 2021] leverages glsmpk to protect the confidentiality of secret cryptographic data from memory disclosure vulnerabilities. We believe that it can guarantee robust constant time against read-only attackers, though their work is not framed in this manner. CryptoMPK differs from RoboCop in several fundamental ways: First, it is instead a whole-program analysis and transformation, identifying “crypto buffers” throughout the program and toggling glsmpk protection around use sites. As such it inserts significantly more context switches than RoboCop, and dynamically allocates and frees secret stack buffers. RoboCop’s trusted library model avoids these costs, allowing a single context switch on library entry and exit, completely

avoiding the need to dynamically allocate stack buffers and leading to significant performance gains. Second, by avoiding a whole program analysis RoboCop's model allows a much simpler implementation. This allows RoboCop to be applied to more complicated code and even hard to analyze assembly code (though we have not implemented this). Lastly, CryptoMPK does not handle robust speculative protections nor robust protections against concurrent attackers. As an optimization, CryptoMPK chooses not to securely allocate small secret stack buffers and instead inserts zeroing code. This zeroing code has the same trade offs between protecting against Spectre and performance as in `LibSodium`, and the buffers are also visible to concurrent attackers. CryptoMPK provides a `mxor` annotation to ensure that eventually declassified buffers are not marked as tainted when they are mixed with secret key data. This has the result of exposing these buffers to concurrent attackers.

Secure zeroization is often deployed as a manual protection in cryptographic libraries. Unfortunately, implementing secure memory zeroing in a high-level language is essentially impossible [Percival 2014; RustCrypto 2023; Z. Yang et al. 2017]. Recent work [Olmos et al. 2024] shows how to develop a compiler pass to implement secure zeroization. RoboCop avoids these issues by restricting all secret data to a protected memory region, thus avoiding the need for zeroization (apart from register zeroing).

(Speculative) constant time

Many domain-specific languages and compilers have been developed to produce high-assurance cryptographic code [Almeida et al. 2017; Barthe, Blazy, et al. 2019; Cauligi, Soeller, et al. 2019; Protzenko et al. 2017; Watt, Renner, et al. 2019]. Spectre attacks [P. Kocher et al. 2019] significantly reduced the guarantees of these tools and prompted defenses against speculative leaks [Cauligi, Disselkoen, Moghimi, et al. 2022]. Jasmin implements Selective Speculative Load Hardening to protect against Spectre-PHT [Shivakumar et al. 2023], performing stack zeroization and register clearing [Olmos

et al. 2024]. Blade inserts a minimum number of protections to prevent leaks via Spectre-PHT gadgets [Vassena et al. 2021]. Swivel hardens WebAssembly sandboxes against speculative sandbox breakout and sandbox poisoning attacks [Narayan, Disselkoen, Moghimi, et al. 2021; Yavarzadeh et al. 2023]. Serberus mitigates all currently known Spectre variants in code that follows the *static* constant-time discipline, which additionally prohibits secret function arguments and return values [Mosier et al. 2024]. These tools serve as complements to RoboCop: in combination they can be used to guarantee end-to-end protections against speculative attackers as discussed in § 2.2.2. Notably, there is an exception to this statement in the case of Serberus: In handling Spectre-RSB [Koruyeh et al. 2018], Serberus makes an implicit assumption that the return stack buffer is empty when entering cryptographic code, arising from an assumption that the cryptographic code represents the entire program. This can be remedied by RSB filling on entry to cryptographic code.

Foundations for cryptographic software security

Researchers have developed trace-based leakage models to reason about timing leaks in cryptographic code [Barthe, Grégoire, et al. 2018; Molnar et al. 2006]. These models have then been extended with prediction oracles [Guarnieri, Köpf, Morales, et al. 2020], speculative semantics, and directives [Cauligi, Disselkoen, Gleissenthall, et al. 2020; Cheang et al. 2019; Guarnieri, Köpf, Reineke, et al. 2021] to capture leaks via (combinations of) different Spectre gadgets [Fabian et al. 2022]. Cauligi, Disselkoen, Moghimi, et al. [2022] survey a number of these semantic models. Of the medium and high-level semantics they survey [Barthe, Cauligi, et al. 2021; Colvin and Winter 2020; Disselkoen et al. 2019; Patrignani and Guarnieri 2021; Ponce-de-Leon and Kinder 2022; Vassena et al. 2021], all model Spectre-PHT [P. Kocher et al. 2019], while only Barthe, Cauligi, et al. [2021] and Ponce-de-Leon and Kinder [2022] model Spectre-STL [Horn 2018]. None model Spectre-BTB [P. Kocher et al. 2019] or Spectre-RSB [Maisuradze and

Rossow 2018]: λ_{spec} answers the call to capture additional Spectre variants in higher-level models. Our speculative semantics also serve as the first model of the speculative behavior of MPK, outside of the description in Intel's manual [Intel Corporation 2023]. Guanciale et al. [2020], present a low-level, microarchitectural, speculative semantics based on speculatively predicting the value of microinstructions: λ_{spec} can be viewed as a lifting of this idea to a high-level language. Our notion of RCT is inspired by previous work on secure compilers [Abate et al. 2019], which are formally defined as compilers that preserve classes of (hyper)-properties in adversarial contexts. Patrignani and Guarnieri [2021] develop secure robust compilation criteria to formally examine the security guarantees of protections inserted by major compilers against Spectre-PHT, our approach to a secure compiler property follows this line of work.

2.7 Acknowledgments

This chapter, in full, is adapted from material as it appears in "Robust Constant-Time Cryptography." Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. Proc. ACM Program. Lang. 9, PLDI, 2025. The dissertation author was the primary investigator and author of this paper.

Appendix A

Proofs

A.1 Proofs of Security for Heavyweight transitions

Throughout the following we will use the shorthand $ctx_{\Psi} \triangleq \Psi.M(ctx^{*})$.

Theorem 6. *NaCl has the disjoint noninterference property.*

Proof. Follows immediately from the fact that all reads and writes are guarded to be within $M_{\texttt{lib}}$, all values in $M_{\texttt{lib}}$ have label `lib`, and all jumps remain within the library code. □

Lemma 1. *The trampoline context is in H_{app}.*

Lemma 2. $ctx \geq ctx_0$.

Lemma 3. *If $\Psi_1(\!|c|\!)_{\texttt{lib}}$ and $\Psi_1 \xrightarrow{p}{}^{*} \Psi_2$, then $\Psi''.M_{\text{app}} = \Psi'.M_{\text{app}}$.*

Proof. There are two cases for p: $p = \texttt{app}$ and $p = \texttt{lib}$. If $p = \texttt{app}$, then $\Psi_2 = \Psi_1$ and therefore trivially $\Psi_2.M_{\text{app}} = \Psi_1.M_{\text{app}}$. If $p = \texttt{lib}$, then for all Ψ such that $\Psi_1 \to^{*} \Psi \to^{+} \Psi_2$, $\Psi.C(\Psi.pc) = (\texttt{lib}, _)$. By the structure of a NaCl program, this ensures that $\Psi_2.M_{\text{app}} = \Psi_1.M_{\text{app}}$. □

Lemma 4. *If $\Psi \xrightarrow{p}{}^{*} \Psi'$ then $\Psi.p = \Psi'.p$.*

Lemma 5. *If $\Psi \xrightarrow{wb} \Psi'$ where $\Psi \to^{*} \Psi'' \to \Psi'$, then $\Psi''.p \neq \Psi.p$ and $\Psi'.p = \Psi.p$.*

Proof. We proceed by simultaneous induction on the well-bracketed transition $\Psi \xrightarrow{\text{wb}} \Psi'$ and the length of $\Psi_0 \rightarrow^* \Psi \xrightarrow{\text{wb}} \Psi'$.

Case No callbacks

We have that $\Psi(\!|\mathtt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2, and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p}{}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\!|\mathtt{gateret}|\!)$. Here $\Psi'' = \Psi_2$. By inspection of the reduction for $\mathtt{gatecall}_n\ e$ we know that $\Psi_1.p \neq \Psi.p$ and therefore by Lemma 4 $\Psi''.p \neq \Psi.p$. By inspection of the reduction for $\mathtt{gateret}$, $\Psi'.p = \Psi.p$.

Case Callbacks

We have that $\Psi(\!|\mathtt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square}{}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\!|\mathtt{gateret}|\!)$. Here $\Psi'' = \Psi_2$. By inspection of the reduction for $\mathtt{gatecall}_n\ e$ we know that $\Psi_1.p \neq \Psi.p$. We now show, by induction on $\Psi_1 \xrightarrow{\square}{}^* \Psi_2$, that $\Psi_2.p = \Psi_1.p \neq \Psi.p$.

Proof. If there are no steps then clearly $\Psi_2.p = \Psi_1.p \neq \Psi.p$. There are two possible cases for $\Psi_1 \xrightarrow{\square}{}^* \Psi_3 \xrightarrow{\square} \Psi_4$. When $\Psi_3 \xrightarrow{p} \Psi_4$, Lemma 4 gives us that $\Psi_4.p = \Psi_3.p = \Psi_1.p \neq \Psi.p$. When $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, our outer inductive hypothesis gives us that $\Psi_4.p = \Psi_3.p = \Psi_1.p \neq \Psi.p$. ■

Lastly, by inspection of the reduction for $\mathtt{gateret}$, $\Psi'.p = \Psi.p$.

□

Lemma 6. *If $\Psi \xrightarrow{\square}{}^* \Psi'$, then $\Psi'.p = \Psi.p$.*

Proof. By induction, Lemma 4, and Lemma 5. □

Lemma 7 (Context Integrity)**.** *Let $\Psi_0 \in Program$, $\Psi_0 \rightarrow^* \Psi$, and $\Psi \xrightarrow{wb} \Psi'$. Then*

1. *if $\Psi.p = \mathtt{app}$, then $\Psi.M([ctx_{\Psi_0}, ctx_{\Psi}]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_{\Psi} = ctx_{\Psi'}$,*

2. *if* $\Psi.p = \texttt{lib}$, *then* $\Psi.M([ctx_{\Psi_0}, ctx_{\Psi}]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ *and* $ctx_{\Psi} = ctx_{\Psi'}$.

Proof. We proceed by mutual, simultaneous induction on the well-bracketed transition $\Psi_1 \xrightarrow{\text{wb}} \Psi_2$ and the length of $\Psi_0 \to^* \Psi \xrightarrow{\text{wb}} \Psi'$.

First we consider the case where $\Psi(\!|c|\!)_{\text{app}}$.

Case No callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2, and p such that $\Psi \to \Psi_1 \xrightarrow{p}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By Lemma 3 we have that $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$. By assumption we have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)_{\text{app}}$ and therefore by Lemma 5 we have that $\Psi_2(\!|\texttt{gateret}|\!)_{\texttt{lib}}$. By Lemma 1, Lemma 2, and inspection of the reduction rules for $\texttt{gatecall}_n\ e$ and $\texttt{gateret}$ we have that $\Psi.M([ctx_{\Psi_0}, ctx_{\Psi}]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_{\Psi} = ctx_{\Psi'}$.

Case Callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2 such that $\Psi \to \Psi_1 \xrightarrow{\Box}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we have that $ctx_{\Psi_1} = ctx_{\Psi} + \text{len}(\mathbb{CSR})$. We now show, by induction on $\Psi_1 \xrightarrow{\Box}{}^* \Psi_2$, that $ctx_{\Psi_2} = ctx_{\Psi_1}$ and $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1}]) = \Psi_2.M([ctx_{\Psi_0}, ctx_{\Psi_2}])$.

Proof. If there are no steps then clearly $ctx_{\Psi_2} = ctx_{\Psi_1}$ and all of $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$. There are two possible cases for $\Psi_1 \xrightarrow{\Box}{}^* \Psi_3 \xrightarrow{\Box} \Psi_4$, and notice that in both $\Psi_3.p = \Psi_1.p = \texttt{lib}$ (by Lemma 6). When $\Psi_3 \xrightarrow{p} \Psi_4$, Lemma 3 gives us that $\Psi_3.M_{\text{app}} = \Psi_4.M_{\text{app}}$ and then Lemma 1 gives us that $ctx_{\Psi_4} = ctx_{\Psi_3} = ctx_{\Psi_1}$. When $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, case 2 of our inductive hypothesis gives us that $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1}]) = \Psi_3.M([ctx_{\Psi_0}, ctx_{\Psi_3}]) = \Psi_4.M([ctx_{\Psi_0}, ctx_{\Psi_4}])$ and $ctx_{\Psi_4} = ctx_{\Psi_3} = ctx_{\Psi_1}$. ∎

Finally, by Lemma 6 we get that $\Psi_2 = \texttt{lib}$ and then inspection of the reduction rule for $\texttt{gateret}$ gives us that $\Psi.M([ctx_{\Psi_0}, ctx_{\Psi}]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_{\Psi} = ctx_{\Psi'}$.

Second we consider the case where $\Psi(\!|c|\!)_{\texttt{lib}}$.

Case No callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2, and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By the structure of a NaCl program we have that $ctx_{\Psi_1} = ctx_{\Psi_2}$ and $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1}]) = \Psi_2.M([ctx_{\Psi_0}, ctx_{\Psi_2}])$. By assumption we have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)_{\texttt{lib}}$ and therefore by Lemma 5 we have that $\Psi_2(\!|\texttt{gateret}|\!)_{\mathsf{app}}$. By Lemma 2 and inspection of the reduction rules for $\texttt{gatecall}_n\ e$ and $\texttt{gateret}$ we have that $\Psi.M([ctx_{\Psi_0}, ctx_{\Psi})) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}))$ and $ctx_{\Psi} = ctx_{\Psi'}$.

Case Callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\Box}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we have that $ctx_{\Psi_1} = ctx_{\Psi} + 1$. We now show, by induction on $\Psi_1 \xrightarrow{\Box}^* \Psi_2$, that $ctx_{\Psi_2} = ctx_{\Psi_1}$ and $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1})) = \Psi_2.M([ctx_{\Psi_0}, ctx_{\Psi_2}))$.

Proof. If there are no steps then clearly $ctx_{\Psi_2} = ctx_{\Psi_1}$ and all of $\Psi_1.M_{\mathsf{app}} = \Psi_2.M_{\mathsf{app}}$. There are two possible cases for $\Psi_1 \xrightarrow{\Box}^* \Psi_3 \xrightarrow{\Box} \Psi_4$, and notice that in both $\Psi_3.p = \Psi_1.p = \mathsf{app}$ (by Lemma 6). When $\Psi_3 \xrightarrow{p} \Psi_4$, the structure of a NaCl program gives us that $ctx_{\Psi_1} = ctx_{\Psi_2}$ and $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1}]) = \Psi_2.M([ctx_{\Psi_0}, ctx_{\Psi_2}])$. When $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, case 1 of our inductive hypothesis gives us that $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1})) = \Psi_3.M([ctx_{\Psi_0}, ctx_{\Psi_3})) = \Psi_4.M([ctx_{\Psi_0}, ctx_{\Psi_4}))$ and $ctx_{\Psi_4} = ctx_{\Psi_3} = ctx_{\Psi_1}$. ■

Finally, by Lemma 6 we get that $\Psi_2 = \mathsf{app}$ and then inspection of the reduction rule for $\texttt{gateret}$ gives us that $\Psi.M([ctx_{\Psi_0}, ctx_{\Psi}]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_{\Psi} = ctx_{\Psi'}$.

□

Theorem 7. *NaCl has callee-save register integrity.*

Proof. Follows from Lemma 7, Lemma 5, and inspection of the reduction rules for $\texttt{gatecall}_n\ e$ and $\texttt{gateret}$. □

Lemma 8. *Let* $\Psi_0 \in \mathit{Program}$, $\pi = \Psi_0 \to^* \Psi$, *and* $\Psi \xrightarrow{wb} \Psi'$, *then*

$$\Psi'.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi)).$$

Proof. First we consider the case where $\Psi(\!|c|\!)_{\text{app}}$.

Case No callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2, and p such that $\Psi \to \Psi_1 \xrightarrow{p}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$. This is adding to the top of the stack, so by the structure of a NaCl program we have that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. By Lemma 3 we have that $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$ and therefore Lemma 1 gives us that $ctx_{\Psi_2} = ctx_{\Psi_1}$ and $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\text{app}}(\pi))$. If we inspect the trampoline code we see that, right before we execute the $\texttt{ret}$, we have set sp to $\Psi_2.M(ctx_{\Psi_2}) = \Psi_1.M(ctx_{\Psi_1}) = \Psi.sp + 1$. Thus, after returning the only part of the application stack that we modify is $\Psi.sp+1$. This, and the fact that $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$ gives us that $\Psi.M(\text{return-address}_{\text{app}}(\pi)) = \Psi'.M(\text{return-address}_{\text{app}}(\pi))$.

Case Callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2 such that $\Psi \to \Psi_1 \xrightarrow{\square}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$. This is adding to the top of the stack, so by the structure of a NaCl program we have that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. We now show, by in-

duction on $\Psi_1 \xrightarrow{\square}{}^* \Psi_2$ that $\Psi_2.M(ctx_{\Psi_2}) = \Psi.sp{+}1$ and $\Psi_2.M(\text{return-address}_{\mathsf{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\mathsf{app}}(\pi))$.

Proof. If there are no steps then $\Psi_2 = \Psi_1$ and both goals hold immediately. There are two possible cases for $\Psi_1 \xrightarrow{\square}{}^* \Psi_3 \xrightarrow{\square} \Psi_4$ and notice that in both $\Psi_3.p = \Psi_1.p = \texttt{lib}$ (by Lemma 6). If $\Psi_3 \xrightarrow{p} \Psi_4$ then Lemma 3 gives us that $\Psi_4.M_{\mathsf{app}} = \Psi_3.M_{\mathsf{app}}$ and our goal holds (as all of $\text{return-address}_{\mathsf{app}}(\pi)$ is in S_{app}). If $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, then Lemma 7 gives us that $ctx_{\Psi_3} = ctx_{\Psi_4}$ and $\Psi_3.M([ctx_{\Psi_0}, ctx_{\Psi_3}]) = \Psi_4.M([ctx_{\Psi_0}, ctx_{\Psi_4}])$ and therefore that $\Psi_4.M(ctx_{\Psi_4}) = \Psi.sp + 1$. $\text{return-address}_{\mathsf{app}}(\Psi_0 \to^* \Psi_3) = \text{return-address}_{\mathsf{app}}(\pi) \uplus \Psi.sp + 1$, so our inductive hypothesis gives us that $\Psi_4.M(\text{return-address}_{\mathsf{app}}(\pi)) = \Psi_3.M(\text{return-address}_{\mathsf{app}}(\pi))$. ■

Second we consider the case where $\Psi(\!|c|\!)_{\texttt{lib}}$.

Case No callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2, and p such that $\Psi \to \Psi_1 \xrightarrow{p}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we see that $\Psi_1.M(\text{return-address}_{\mathsf{app}}(\pi)) = \Psi.M(\text{return-address}_{\mathsf{app}}(\pi))$. By the structure of a NaCl program we have that any call stack elements that are added during the callback will be popped before the `gateret`. Thus, $\Psi_2.M(\text{return-address}_{\mathsf{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\mathsf{app}}(\pi))$. Inspection of the reduction rule for `gateret` then gives us that

$$\begin{aligned} \Psi'.M(\text{return-address}_{\mathsf{app}}(\pi)) &= \Psi_2.M(\text{return-address}_{\mathsf{app}}(\pi)) \\ &= \Psi.M(\text{return-address}_{\mathsf{app}}(\pi)). \end{aligned}$$

Case Callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square}{}^{*} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we see that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. We now show, by induction on $\Psi_1 \xrightarrow{\square}{}^{*} \Psi_2$ that

$$\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\text{app}}(\pi)).$$

Proof. If there are no steps then $\Psi_2 = \Psi_1$ and the goal holds immediately. There are two possible cases for $\Psi_1 \xrightarrow{\square}{}^{*} \Psi_3 \xrightarrow{\square} \Psi_4$ and notice that in both $\Psi_3.p = \Psi_1.p = \texttt{app}$ (by Lemma 6). If $\Psi_3 \xrightarrow{p} \Psi_4$ then the structure of a NaCl program gives us that any call stack elements that are added during the callback will be popped before the $\texttt{gateret}$, and therefore our inductive invariant is maintained. If $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, then notice that $\text{return-address}_{\text{app}}(\Psi_0 \rightarrow^{*} \Psi_3) = \text{return-address}_{\text{app}}(\pi)$, so our inductive hypothesis gives us that $\Psi_4.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_3.M(\text{return-address}_{\text{app}}(\pi))$. ■

Inspection of the reduction rule for $\texttt{gateret}$ then gives us that

$$\begin{aligned}\Psi'.M(\text{return-address}_{\text{app}}(\pi)) &= \Psi_2.M(\text{return-address}_{\text{app}}(\pi)) \\ &= \Psi.M(\text{return-address}_{\text{app}}(\pi)).\end{aligned}$$

□

Theorem 8. *NaCl has return address integrity.*

Proof. We have that $\Psi_0 \in \mathit{Program}$, $\pi = \Psi_0 \rightarrow^{*} \Psi$, $\Psi.p = \texttt{app}$, and $\Psi \xrightarrow{\text{wb}} \Psi'$ and wish to show that $\Psi.M(\text{return-address}_{\text{app}}(\pi)) = \Psi'.M(\text{return-address}_{\text{app}}(\pi))$, $\Psi'.sp = \Psi.sp$, and $\Psi'.pc = \Psi.pc + 1$.

Lemma 8 gives us that $\Psi.M(\text{return-address}_{\text{app}}(\pi)) = \Psi'.M(\text{return-address}_{\text{app}}(\pi))$. We proceed by simultaneous induction on the well-bracketed transition $\Psi \xrightarrow{\text{wb}} \Psi'$ and the length of $\Psi_0 \to^* \Psi \xrightarrow{\text{wb}} \Psi'$ to show that $\Psi'.sp = \Psi.sp$, and $\Psi'.pc = \Psi.pc + 1$.

Case No callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2, and p such that $\Psi \to \Psi_1 \xrightarrow{p}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we see that $\Psi_1.M(\Psi.sp+1) = \Psi.pc+1$ and $\Psi_1.M(ctx_{\Psi_1}) = \Psi.sp+1$. By Lemma 3 we have that $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$ and therefore Lemma 1 gives us that $ctx_{\Psi_2} = ctx_{\Psi_1}$. If we inspect the trampoline code we see that, right before we execute the `ret`, we have set sp to $\Psi_2.M(ctx_{\Psi_2}) = \Psi_1.M(ctx_{\Psi_1}) = \Psi.sp + 1$. Thus, after returning we have that $\Psi'.pc = \Psi.pc + 1$, $\Psi'.sp = \Psi.sp$.

Case Callbacks

We have that $\Psi(\!|\texttt{gatecall}_n\ e|\!)$ and there exist Ψ_1, Ψ_2 such that $\Psi \to \Psi_1 \xrightarrow{\square}{}^* \Psi_2 \to \Psi'$ where $\Psi_2(\!|\texttt{gateret}|\!)$. By inspection of the reduction rule for $\texttt{gatecall}_n\ e$ we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$ and $\Psi_1.M(ctx_{\Psi_1}) = \Psi.sp + 1$. We now show, by induction on $\Psi_1 \xrightarrow{\square}{}^* \Psi_2$ that $\Psi_2.M(\Psi.sp+1) = \Psi.pc+1$ and $\Psi_2.M(ctx_{\Psi_2}) = \Psi.sp+1$.

Proof. If there are no steps then $\Psi_2 = \Psi_1$ and both hold immediately. There are two possible cases for $\Psi_1 \xrightarrow{\square}{}^* \Psi_3 \xrightarrow{\square} \Psi_4$ and notice that in both $\Psi_3.p = \Psi_1.p = \texttt{lib}$ (by Lemma 6). If $\Psi_3 \xrightarrow{p} \Psi_4$ then Lemma 3 gives us that $\Psi_4.M_{\text{app}} = \Psi_3.M_{\text{app}}$ and both hold (as the invariants are on M_{app}). If $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, then Lemma 7 gives us that $ctx_{\Psi_3} = ctx_{\Psi_4}$ and $\Psi_3.M([ctx_{\Psi_0}, ctx_{\Psi_3}]) = \Psi_4.M([ctx_{\Psi_0}, ctx_{\Psi_4}])$ and therefore that $\Psi_4.M(ctx_{\Psi_4}) = \Psi.sp + 1$. $\text{return-address}_{\text{app}}(\Psi_0 \to^* \Psi_3) = \text{return-address}_{\text{app}}(\pi) \uplus \Psi.sp+1$ so Lemma 8 gives us that $\Psi_4.M(\Psi.sp+1) = \Psi_3.M(\Psi.sp+1) = \Psi.pc+1$. ■

Finally, if we inspect the trampoline code we see that, right before we execute the

`ret`, we have set sp to $\Psi_2.M(ctx_{\Psi_2}) = \Psi.sp + 1$. Thus, after returning we have that $\Psi'.pc = \Psi.pc + 1$, $\Psi'.sp = \Psi.sp$.

□

A.2 Proofs of Security for Zero-Cost WebAssembly

Lemma 9 (FTLR for functions)**.** *Let $W \in \text{World}$ and c be the code for a compiled WebAssembly function WF such that WF expects application functions in the interface with locations and types $\pi_2(W)$ and in the library with locations and types $\pi_3(W)$. Further let $instrs = \biguplus_{B \in WF.blocks}[B.start, B.end]$ and $F = \{instrs := instrs, entry := WF.entry.start, type := WF.|args|\}$. Then $(W, F, c) \in \mathcal{F}$.*

Proof. We first unroll the assumptions of $(W, F, c) \in \mathcal{F}$ reusing the variable names defined there. We will maintain that any steps do not step to `oerror` so WOLOG we will continually assume $n' \leq W.n$ such that n' greater than the number of steps we have taken, otherwise the case $\Phi \overset{\rho'}{\rightsquigarrow}{}^{n'} \Phi' \implies \Phi' \neq \texttt{oerror}$ holds.

By the structure of a compiled WebAssembly function and assumption we have that the stack and stack pointer represent $WF.|args|$ arguments. The abstract interpretation ensures that if we write to $H_{\texttt{lib}}$ then that value has label `lib` so the checks pass. We are further assured that we do not read or below the stack frame. The structure of a compiled WebAssembly block then lets us proceed until we reach one of 1. a function call to a library function WF' such that $WF'.entry.start \in \overline{F_l.entry}$, 2. an application function F' such that $F'.entry \in \overline{F_i.entry}$, 3. or the end of the block.

1. The abstract interpretation ensures that we have initialized the arguments

$$WF'.|args| = \pi_3(W)(WF'.entry.start)$$

or failed a dynamic type check and terminated (thus stepping to a terminal state that is not an `oerror`). We thus set $\rho_2 = \rho'$ and see that we have constructed $\rho'_2 = \rho_2 +\!\!+ \{base := sp - WF'.|args|, ret\text{-}addr\text{-}loc := sp, csr\text{-}vals := R(\mathbb{CSR})\}$. We further set $ret\text{-}addr = pc + 1$, $A = WF'.|args|$, $sp = sp$, $R = R$, $M = M$, $C = C$, $\overline{F_i} = \overline{F_i}$, and $\overline{F_l} = \overline{F_l}$. By the abstract interpretation we have that all of the remaining checks in $\mathcal{F}$ pass and that the instantiated Φ is equal to our current state. We therefore instantiate $(\triangleright F_l, C|_{F_l.instrs}) \in \mathcal{F}$. If this uses the remaining steps then we are done. Otherwise we get that we return to $pc + 1$ with all values restored and no new app values written to the library memory, and our walk through the block may continue.

2. Identical to the case for (1).

3. The end of a block is followed by a direct jump to another block B', an indirect block IB, or we are at an exit block. In the case of another block B' we have by the structure of compiled WebAssembly code that we have instantiated $B'.inputs$. We thus jump to the block and follow the same proof structure as detailed here. The same is true of an intermediate block IB except with the extra steps of setting up the inputs jumping to another block B''. Lastly if we have reached the end of an exit block then we have not touched the pushed return address or callee-save registers and the stack pointer is in the expected location. We thus execute `ret` or `gateret` and pass the overlay monitor checks.

□

Lemma 10 (FTLR for libraries)**.** *For any number of steps $n \in \mathbb{N}$ and compiled WebAssembly library L, $(n, L) \in \mathcal{L}$.*

Proof. By unrolling the definition of $\mathcal{L}$ and Lemma 9. □

Theorem 9 (Adequacy of $\mathcal{L}$). *For any number of steps $n \in \mathbb{N}$, library L such that $(n, L) \in \mathcal{L}$, program $\Phi_0 \in$ Program using L, and $n' \leq n$, if $\Phi_0 \rightsquigarrow^{n'} \Phi'$ then $\Phi' \neq \texttt{oerror}$.*

Proof. Straightforward: by assumption for steps in the application, and by assumption about application code properly calling the library code and the unrolling of $\mathcal{L}$ and $\mathcal{F}$. ☐

A.3 Formal Definitions of RoboCop Compilers

$$
\begin{aligned}
\textsf{on-return}_f(v) &= f(v) \\
\textsf{on-return}_f(x) &= f(x) \\
\textsf{on-return}_f(e_1; e_2) &= e_1; \textsf{on-return}_f(e_2) \\
\textsf{on-return}_f(\texttt{fence}) &= f(\texttt{fence}) \\
\textsf{on-return}_f(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) &= \texttt{if}\ e\ \texttt{then}\ \textsf{on-return}_f(e_1)\ \texttt{else}\ \textsf{on-return}_f(e_2) \\
\textsf{on-return}_f(e) &= \texttt{let}\ x = e\ \texttt{in}\ f(x)
\end{aligned}
$$

$$
\begin{aligned}
\textsf{add-protect}(e) &= \texttt{protect}_{\texttt{public}}; e \\
\textsf{add-fence}(e) &= \texttt{fence}; e
\end{aligned}
$$

$$
\begin{aligned}
\textsf{copy-in}(\bullet, e) &= e \\
\textsf{copy-in}(f \mapsto \overline{(x,\, e_{len})} :: rest, e) &= \texttt{let}\ y = \texttt{new}_{\texttt{public}}\ e_{len}\ \texttt{in}\ e_{copy} \\
&\qquad \text{where } e_{copy} = \textsf{copy}(y, x, e_{len}); \textsf{copy-in}(e[y/x])
\end{aligned}
$$

$$\textsf{copy-out}(\bullet, e) = e$$

$$\textsf{copy-out}(f \mapsto \overline{(x,\ e_{len})} :: rest, e) = \texttt{copy}(x, y, e_{len}); \textsf{copy-out}(e)$$

$$\mathbb{C}_{\textsf{ro}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) = \lambda\overline{x}.e_{new}$$

$$\text{where } e_{subst} = e[\texttt{new}_{\texttt{protected}}\ e'/\texttt{new}_p\ e']$$

$$\text{where } e_{body} = \textsf{on-return}_{\textsf{add-protect}}(e_{subst})$$

$$\text{where } e_{new} = \texttt{protect}_{\texttt{protected}}; e_{body}$$

$$\text{if } f \in \Gamma$$

$$\mathbb{C}_{\textsf{ro}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) = \lambda\overline{x}.(e[\texttt{new}_{\texttt{protected}}\ e'/\texttt{new}_p\ e'])$$

$$\text{if } f \notin \Gamma$$

$$\mathbb{C}_{\textsf{spec}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) = \lambda\overline{x}.\textsf{on-return}_{\textsf{add-fence}}(e_{new})$$

$$\text{where } \lambda\overline{x}.e_{new} = \mathbb{C}_{\textsf{ro}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))$$

$$\text{if } f \in \Gamma$$

$$\mathbb{C}_{\textsf{spec}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) = \mathbb{C}_{\textsf{ro}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))$$

$$\text{if } f \notin \Gamma$$

$$\mathbb{C}_{\textsf{co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) = \lambda\overline{x}.\textsf{on-return}_{\textsf{copy-out}(used)}(e_{new})$$

$$\text{where } e_{new} = \textsf{copy-in}(used, e)$$

$$\text{if } f \in \Gamma$$

$$\mathbb{C}_{\mathsf{co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) \quad = \quad f \mapsto (z_f,\ \lambda\overline{x}.e)$$

$$\text{if } f \notin \Gamma$$

$$\mathbb{C}_{\mathsf{ro\text{-}co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) \quad = \quad \mathbb{C}_{\mathsf{ro}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e_{new})$$

$$\text{where } \lambda\overline{x}.e_{new} = \mathbb{C}_{\mathsf{co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))$$

$$\text{if } f \in \Gamma$$

$$\mathbb{C}_{\mathsf{ro\text{-}co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) \quad = \quad \mathbb{C}_{\mathsf{ro}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))$$

$$\text{if } f \notin \Gamma$$

$$\mathbb{C}_{\mathsf{spec\text{-}co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) \quad = \quad \mathbb{C}_{\mathsf{spec}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e_{new})$$

$$\text{where } \lambda\overline{x}.e_{new} = \mathbb{C}_{\mathsf{co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))$$

$$\text{if } f \in \Gamma$$

$$\mathbb{C}_{\mathsf{spec\text{-}co}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e)) \quad = \quad \mathbb{C}_{\mathsf{spec}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))$$

$$\text{if } f \notin \Gamma$$

$$\mathbb{C}_{\mathsf{A}}(\Gamma \vDash \bullet) \quad = \quad \bullet$$

$$\mathbb{C}_{\mathsf{A}}(\Gamma \vDash f \mapsto (z_f,\ \lambda\overline{x}.e) :: L) \quad = \quad f \mapsto (z_f,\ \mathbb{C}_{\mathsf{A}}(\Gamma, f \mapsto (z_f,\ \lambda\overline{x}.e))) :: \mathbb{C}_{\mathsf{A}}(\Gamma \vDash L)$$

For the concurrent compilers we assume we have been provided a labeling of arguments $used : \overline{f \mapsto \overline{(x,\ e_{len})}}$ for all functions in Γ. The variables are the argument names and the expressions are the length of the buffer. The correctness condition for the labeling is the following: For all secret contexts Δ, classical "applications" $\Gamma, \Delta \vdash (H,\ e_\Gamma)$, and initial states $\mathbb{C}_{\mathsf{co}}(\Gamma \vDash L) \mid \Delta \mid H \vDash S$, if $T \in \mathsf{traces}(\langle S[\Delta][\mathbb{C}_{\mathsf{co}}(\Gamma \vDash L)] \mid e_\Gamma[\Delta][\mathbb{C}_{\mathsf{co}}(\Gamma \vDash L)]\rangle)$

and $\overline{\mu}$ is the set of memory events in T, then $\overline{\mu} = \overline{\mu_1} \mathbin{+\!\!+} \overline{\mu_2} \mathbin{+\!\!+} \overline{\mu_3}$ where

1. $\overline{\mu_1} = \overline{\mathsf{new}_{\texttt{public}}\ v_{len}@z'_b \mathbin{+\!\!+} \mathsf{read}_{\mathsf{ib}}\ v \hookleftarrow z_b[z_o] \mathbin{+\!\!+} \mathsf{write}_{\mathsf{ib}}\ v \mapsto z'_b[z_o]}$ and $z_b \in \mathrm{dom}(H)$

2. for all $\mu_2 = \mathsf{read}_b\ v \hookleftarrow z_b[z_o]$ or $\mu_2 = \mathsf{write}_b\ v \mapsto z_b[z_o]$, $z_b \notin \mathrm{dom}(H)$

3. $\overline{\mu_3} = \overline{\mathsf{read}_{\mathsf{ib}}\ v \hookleftarrow z'_b[z_o] \mathbin{+\!\!+} \mathsf{write}_{\mathsf{ib}}\ v \mapsto z_b[z_o]}$

and if $\langle S[\Delta][L] \mid e_\Gamma[\Delta][L]\rangle \downarrow^{T_1} \langle S' \mid v\rangle$ then there exists a T'_1 such that $\langle S[\Delta][\mathbb{C}_{\mathsf{co}}(\Gamma \vDash L)] \mid e_\Gamma[\Delta][\mathbb{C}_{\mathsf{co}}(\Gamma \vDash L)]\rangle \downarrow^{T_1} \langle S'' \mid v\rangle$.

A.4 Proofs of Security for RoboCop Compilers

A.4.1 Read-only Protections

To prove Theorem 10 we must show that, for *any* read-only attackers and any two initial states that vary only in the values of secrets, when we plug our compiled library into the application the execution under both states does not vary (up to the observable leakage). The difficulty arises in two places: firstly, we must reason about a (mostly) arbitrary application. Secondly, the inherent nondeterminism of our semantics (due to allocation and out of bounds memory semantics), means that we are dealing with sets of traces rather than just a single trace.

To handle these challenges we rely on the fact that our attacker model is defined as properties on the structured traces T. Our proof begins with the two initial states S and S' with the same whole program e and some trace T for $\langle S \mid e\rangle$. We must then show that there is a corresponding trace for $\langle S' \mid e\rangle$. To do so we inductively split the *read-only* trace T into its attacker and library subsequences and define a semantic interpretation that captures that there is a corresponding trace T' for $\langle S' \mid e\rangle$ (see Figure A.1).

Lemma 11 (Functions are fixed)**.** *For all* $\langle S_1 \mid \overline{K_1{}^{\ell_{K1}}} :: e_1{}^{\ell_1}\rangle \overset{\bar{\epsilon}}{\Rightarrow}{}^* \langle S_2 \mid \overline{K_2{}^{\ell_{K2}}} :: e_2{}^{\ell_2}\rangle$,

$$\left\{z_f \mapsto \lambda_\ell \overline{x}.e \mid S_1(z_f) = \lambda_\ell \overline{x}.e\right\} = \left\{z_f \mapsto \lambda_\ell \overline{x}.e \mid S_2(z_f) = \lambda_\ell \overline{x}.e\right\}.$$

Proof. By induction on the operational semantics. □

Lemma 12 (Constant time is memory safe)**.** *If* $\Gamma \vDash L$ *is classically constant time and we have a secret context* Δ*, classical "application"* $\Gamma, \Delta \vdash (H, e_\Gamma)$ *and an initial state* $L \mid \Delta \mid H \vdash S$*, then for all* $T \in \mathsf{trace}(\langle S[\Delta][L] \mid e[\Delta][L] \rangle)$ *and* $\delta \in T$*,* $\mathsf{wf\text{-}mem\text{-}safe}\ \delta$*.*

Proof. By induction on the operational semantics. □

Lemma 13 ($\mathbb{C}_{\mathsf{ro}}$ preserves classic constant time)**.** *If* $\Gamma \vDash L$ *is classically constant time then* $\mathbb{C}_{\mathsf{ro}}\Gamma \vDash L$ *is classically constant time.*

Proof. By induction on the compiler. □

Lemma 14 ($\mathbb{C}_{\mathsf{ro}}$ only allocates protected memory)**.** *If* $\Gamma \vDash L$ *is classically constant time and we have a secret context* Δ*, classical "application"* $\Gamma, \Delta \vdash (H, e_\Gamma)$ *and an initial state* $L \mid \Delta \mid H \vdash S$*, then if* $\langle S[\Delta][L] \mid e[\Delta][L] \rangle \downarrow^T \langle S' \mid v \rangle$*, then* $\{S(z_b) \mid S(z_b).p = \mathtt{public}\} = \{S'(z_b) \mid S'(z_b).p = \mathtt{public}\}$

Proof. By induction on the compiler and the assumption of classical constant time. □

Definition 15. *We define our state invariant,* $\mathsf{inv}(S, S')$ *as follows:*

1. $S.p = S'.p = \mathtt{public}$
2. $\forall z : \mathbb{Z}. S(z) \neq S'(z) \Rightarrow S(z).p = S'(z).p = \mathtt{protected}$
3. $\mathrm{dom}(S) = \mathrm{dom}(S')$

$$\mathcal{A}[\![\tau^{\texttt{lib}\to\texttt{app}} +\!\!+ \overline{\delta^{\mathsf{app}}} +\!\!+ \tau^{\mathsf{app}\to\texttt{lib}}]\!] \triangleq$$

$$\left\{ (S,\, S',\, \overline{K^{\ell}},\, e,\, S_1,\, S_1',\, \overline{K_1{}^{\ell_1}},\, e_1) \;\middle|\; \begin{array}{l} \forall \overline{\epsilon_1}, \overline{\epsilon_2}.\ \overline{\epsilon_1} +\!\!+ \overline{\epsilon_2} = \overline{\delta^{\mathsf{app}}} \Rightarrow \\ \quad \exists S_0, S_0', \overline{K_0{}^{\ell_0}}, e_0. \\ \qquad \langle S \mid \overline{K^{\ell}} :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\epsilon_1}}{}^{*} \langle S_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\mathsf{app}} \rangle \\ \qquad \langle S' \mid \overline{K^{\ell}} :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\epsilon_1}}{}^{*} \langle S_0' \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\mathsf{app}} \rangle \\ \qquad \mathsf{inv}(S_0, S_0') \\ \langle S \mid \overline{K^{\ell}} :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\delta^{\mathsf{app}}} +\!\!+ \tau^{\mathsf{app}\to\texttt{lib}}}{}^{*} \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}} \rangle \\ \langle S' \mid \overline{K^{\ell}} :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\delta^{\mathsf{app}}} +\!\!+ \tau^{\mathsf{app}\to\texttt{lib}}}{}^{*} \langle S_1' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}} \rangle \\ \mathsf{inv}(S_1, S_1') \end{array} \right\}$$

$$\mathcal{L}[\![\tau^{\mathsf{app}\to\texttt{lib}} +\!\!+ \overline{\delta^{\texttt{lib}}} +\!\!+ \tau^{\texttt{lib}\to\mathsf{app}}]\!] \triangleq$$

$$\left\{ (S,\, S',\, \overline{K^{\ell}},\, e,\, S_1,\, S_1',\, \overline{K_1{}^{\ell_1}},\, e_1) \;\middle|\; \begin{array}{l} \exists \delta'. \\ \quad \langle S \mid \overline{K^{\ell}} :: e^{\texttt{lib}} \rangle \xRightarrow{\overline{\delta^{\texttt{lib}}} +\!\!+ \tau^{\texttt{lib}\to\mathsf{app}}}{}^{*} \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}} \rangle \\ \quad \langle S' \mid \overline{K^{\ell}} :: e^{\texttt{lib}} \rangle \xRightarrow{\overline{\delta'^{\texttt{lib}}} +\!\!+ \tau^{\texttt{lib}\to\mathsf{app}}}{}^{*} \langle S_1' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}} \rangle \\ \quad \mathsf{ct}(\overline{\delta^{\texttt{lib}}}) = \mathsf{ct}(\overline{\delta'^{\texttt{lib}}}) \\ \quad \mathsf{inv}(S_1, S_1') \end{array} \right\}$$

$$\mathcal{T}[\![A \circ L \circ T]\!] \triangleq$$

$$\left\{ (\langle S \mid \overline{K^{\ell}} :: e \rangle,\, \langle S' \mid \overline{K^{\ell}} :: e \rangle) \;\middle|\; \begin{array}{l} \exists S_1, S_1', S_2, S_2', \overline{K_1{}^{\ell_1}}, e_1, \overline{K_2{}^{\ell_2}}, e_2. \\ \quad (S,\, S',\, \overline{K^{\ell}},\, e,\, S_1,\, S_1',\, \overline{K_1{}^{\ell_1}},\, e_1) \in \mathcal{A}[\![A]\!] \\ \quad (S_1,\, S_1',\, \overline{K_1{}^{\ell_1}},\, e_1,\, S_2,\, S_2',\, \overline{K_2{}^{\ell_2}},\, e_1) \in \mathcal{L}[\![L]\!] \\ \quad (\langle S_2 \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{app}} \rangle,\, \langle S_2' \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{app}} \rangle)) \in \mathcal{T}[\![T]\!] \end{array} \right\}$$

$$\mathcal{T}[\![\tau^{\texttt{lib}\to\texttt{app}} +\!\!+ \overline{\delta^{\texttt{app}}} +\!\!+ (\mathsf{end}\ v)^{\texttt{app}\to\texttt{lib}}]\!] \triangleq$$

$$\left\{ (\langle S \mid \overline{K^{\ell}} :: e\rangle, \langle S' \mid \overline{K^{\ell}} :: e\rangle) \;\middle|\; \begin{array}{l} \exists S_1, S_1', \overline{K_1{}^{\ell_1}}. \\ \quad \text{let } ps = (S,\, S',\, \overline{K^{\ell}},\, e,\, S_1,\, S_1',\, \overline{K_1{}^{\ell_1}},\, v) \\ \quad \text{in } ps \in \mathcal{A}[\![\overline{\delta^{\texttt{app}}} +\!\!+ (\mathsf{end}\ v)^{\texttt{app}\to\texttt{lib}}]\!] \end{array} \right\}$$

Figure A.1. Semantic interpretation of non-speculative traces.

Lemma 15. *If* $\langle S \mid \overline{K^{\ell}} :: e^{\mathsf{app}}\rangle \overset{\overline{\delta^{\mathsf{app}}}}{\Longrightarrow}{}^{*} \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}}\rangle$, $\mathsf{wf\text{-}read\text{-}only}\ \overline{\delta^{\mathsf{app}}}$, *and* $\mathsf{inv}(S, S')$, *then there exists an* S_1' *such that* $\langle S' \mid \overline{K^{\ell}} :: e^{\mathsf{app}}\rangle \overset{\overline{\delta^{\mathsf{app}}}}{\Longrightarrow}{}^{*} \langle S_1' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}}\rangle$ *and* $\mathsf{inv}(S_1, S_1')$.

Proof. We proceed by induction on the reduction relation. The zero step case is immediate by assumption. For the inductive case we have that $\langle S \mid \overline{K^{\ell}} :: e^{\mathsf{app}}\rangle \overset{\overline{\delta_0{}^{\mathsf{app}}}}{\Longrightarrow}{}^{*} \langle S_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\mathsf{app}}\rangle \overset{\delta^{\mathsf{app}}}{\Longrightarrow} \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}}\rangle$ and must show that $\langle S' \mid \overline{K^{\ell}} :: e^{\mathsf{app}}\rangle \overset{\overline{\delta_0{}^{\mathsf{app}}} +\!\!+ \delta^{\mathsf{app}}}{\Longrightarrow}{}^{*} \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}}\rangle$ such that $\mathsf{inv}(S_1, S_1')$. By our inductive hypothesis we have that there exists an S_0' such that $\langle S' \mid \overline{K^{\ell}} :: e^{\mathsf{app}}\rangle \overset{\overline{\delta_0{}^{\mathsf{app}}}}{\Longrightarrow}{}^{*} \langle S_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\mathsf{app}}\rangle$ and $\mathsf{inv}(S_0, S_0')$. We proceed by case analysis on δ.

The only cases that interact with the state (the only parts that differ) are dereferencing, writes, allocation, and protect. In all other cases we let $S_1' = S_0'$ and then it is immediate that $\langle S_0' \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\mathsf{app}}\rangle \overset{\delta^{\mathsf{app}}}{\Longrightarrow} \langle S_0' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}}\rangle$ and the invariant holds by assumption.

In the case that $\delta = \mathsf{read}_{z_b[z_o]}\ b \hookleftarrow v$ we have, by Conditions 1 and 2, that for all locations z_b' such that $\mathsf{accessible}(S_0', z_b')$, $S_0(z_b') = S_0'(z_b')$. By Condition 3 we then have that $\{z_b' \mid \mathsf{accessible}(S_0, z_b')\} = \{z_b' \mid \mathsf{accessible}(S_0', z_b')\}$. Therefore the dereference can take the same step $\mathsf{read}_{z_b[z_o]}\ b \hookleftarrow v$ under S_0' and we let $S_1' = S_0'$.

The case for $\delta = \mathsf{write}_{z_b[z_o]}\ b \mapsto v$ is identical except that we let $S_1' = S_0'(z_b).v[z_0 := v]$. By Conditions 1 and 2 we have that $S_0'(z_b).p = \texttt{public}$ and therefore we get that $\mathsf{inv}(S_1, S_1')$.

For $\delta = \textsf{new}_p\ z_{len}@z_b$ we rely on Condition 3, which means that z_b is also fresh in S'_0.

We have by assumption that the application does not contain protect statements so $\delta = \textsf{protect}_p$ is a contradiction. □

Lemma 16. *For a classically constant time library $\Gamma \vDash L$, if $\mathbb{C}_{\textsf{ro}}(\Gamma \vDash L) \vDash S$, $\mathbb{C}_{\textsf{ro}}(\Gamma \vDash L) \vDash S'$, $T \in \textsf{traces}(\langle S \mid \overline{K^\ell} :: e^{\textsf{app}}\rangle)$, $\Gamma \vdash \textsf{read-only}\ T$, and $\textsf{inv}(S, S')$, then $(\langle S \mid \overline{K^\ell} :: e\rangle, \langle S' \mid \overline{K^\ell} :: e\rangle) \in \mathcal{T}[\![T]\!]$.*

Proof. We proceed by induction on T. Wolog we show the case for $T = A \circ L \circ T'$ (the $T = \tau^{\texttt{lib}\to\texttt{app}} +\!\!+ \overline{\delta^{\textsf{app}}} +\!\!+ (\textsf{end}\ v)^{\textsf{app}\to\texttt{lib}}$ case follows identical reasoning as the A subtrace.) We split the proof into instantiating the different subtrace relations.

Case $\exists S_1, S'_1, \overline{K_1{}^{\ell_1}}, e_1.\ (S, S', \overline{K^\ell}, e, S_1, S'_1, \overline{K_1{}^{\ell_1}}, e_1) \in \mathcal{A}[\![A]\!]$***:*** By definition we have that $A = \tau^{\texttt{lib}\to\texttt{app}} +\!\!+ \overline{\delta^{\textsf{app}}} +\!\!+ \tau^{\textsf{app}\to\texttt{lib}}$. By assumption we have that for all $\overline{\epsilon_1} +\!\!+ \overline{\epsilon_2} = \overline{\delta^{\textsf{app}}}$ there exists some S_0, $\overline{K_0{}^{\ell_0}}$, and e_0 such that $\langle S \mid \overline{K^\ell} :: e^{\textsf{app}}\rangle \xRightarrow{\overline{\epsilon_1}}{}^* \langle S_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\textsf{app}}\rangle$. Lemma 15 then guarantees that the prefix conditions hold.

To show that the overall application trace conditions hold let $\overline{\epsilon_1} = \overline{\delta^{\textsf{app}}}$. By the above logic we have that there exist S_0, $\overline{K_0{}^{\ell_0}}$, and e_0 such that $\langle S \mid \overline{K^\ell} :: e^{\textsf{app}}\rangle \xRightarrow{\overline{\delta^{\textsf{app}}}}{}^* \langle S_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\textsf{app}}\rangle$, $\langle S' \mid \overline{K^\ell} :: e^{\textsf{app}}\rangle \xRightarrow{\overline{\delta^{\textsf{app}}}}{}^* \langle S'_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\textsf{app}}\rangle$, and $\textsf{inv}(S_0, S'_0)$. By assumption we have that there exists some S_1, $\overline{K_1{}^{\ell_1}}$, and e_1 such that $\langle S_0 \mid \overline{K_0{}^{\ell_0}} :: e_0{}^{\textsf{app}}\rangle \xRightarrow{\tau^{\texttt{lib}\to\texttt{app}}} \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}}\rangle$. By $\Gamma \vdash \textsf{read-only}\ T$ we have that $\tau = \textsf{call}\ z_f$ and $z_f \in \text{dom}(\Gamma)$. By inversion $e_0 = z_f(\overline{v})$. By Lemma 11, $\mathbb{C}_{\textsf{ro}}(\Gamma \vDash L) \vDash S$, and $\mathbb{C}_{\textsf{ro}}(\Gamma \vDash L) \vDash S$ we have that $S(z_f) = S(z'_f)$. The remaining conditions of $\mathbb{A}$ then follow immediately. ■

Case $\exists S_2, S'_2, \overline{K_2{}^{\ell_2}}, e_2.\ (S_1, S'_1, \overline{K_1{}^{\ell_1}}, e_1, S_2, S'_2, \overline{K_2{}^{\ell_2}}, e_2) \in \mathcal{L}[\![L]\!]$***:*** By definition we have that $L = \tau^{\textsf{app}\to\texttt{lib}} +\!\!+ \overline{\delta^{\texttt{lib}}} +\!\!+ \tau_2{}^{\texttt{lib}\to\textsf{app}}$. By the above we have that

$$\langle S \mid \overline{K^\ell} :: e^{\textsf{app}}\rangle \xRightarrow{\overline{\delta^{\textsf{app}}} +\!\!+ \tau^{\textsf{app}\to\texttt{lib}}}{}^* \langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}}\rangle$$

and

$$\langle S' \mid \overline{K^\ell} :: e^{\text{app}} \rangle \xRightarrow{\overline{\delta^{\text{app}}} + \tau^{\text{app}\to\texttt{lib}}}{}^* \langle S_1' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}} \rangle.$$

By assumption we have that $\langle S_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}} \rangle \xRightarrow{\overline{\delta^{\texttt{lib}}} + \tau_2{}^{\texttt{lib}\to\text{app}}}{}^* \langle S_2 \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\text{app}} \rangle$ By $\Gamma \vdash$ read-only T we have that $\tau = \textsf{call}\ z_f$ where $z_f \in \text{cod}(\mathbb{C}_{\text{ro}}(\Gamma \vDash L))$. We may therefore apply the assumption that L is classically constant time and Lemma 13 to get that there exist S_2', $\overline{K_2{}^{\ell_2}}$, e_2, and $\overline{\delta'}$ such that $\langle S_1' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\texttt{lib}} \rangle \xRightarrow{\overline{\delta'^{\texttt{lib}}} + \tau_2{}^{\texttt{lib}\to\text{app}}}{}^* \langle S_2' \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\text{app}} \rangle$ where $\textsf{ct}(\overline{\delta'^{\texttt{lib}}}) = \textsf{ct}(\overline{\delta'^{\texttt{lib}}})$. $\textsf{inv}(S_2, S_2')$ follows by the fact that new events must be the same due to the constant timeness and Lemma 14. ■

Case $(\langle S_2 \mid \overline{K_2{}^{\ell_2}} :: e_2 \rangle, \langle S_2' \mid \overline{K_2{}^{\ell_2}} :: e_2 \rangle) \in \mathcal{T}[\![T']\!]$***:*** By the prior states being in $\mathbb{L}$ we have that $\textsf{inv}(S_2, S_2')$. By inversion on $\Gamma \vdash$ read-only T we have that $\Gamma \vdash$ read-only T'. By Lemma 11 $\mathbb{C}_{\text{ro}}(\Gamma \vDash L) \vDash S$ and $\mathbb{C}_{\text{ro}}(\Gamma \vDash L) \vDash S$. We then apply our inductive hypothesis for T' and our proof is complete. ■

□

Lemma 17. *For a classically constant time library $\Gamma \vDash L$, secret context Δ, read-only application $\Gamma, \Delta \vDash (H, e)$, and initial states $\mathbb{C}_{\text{ro}}(\Gamma \vDash L) \mid \Delta \mid H \vDash S_0 = S_0'$, for all $T \in$ $\textsf{traces}(\langle S_0[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)] \mid e[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)] \rangle)$,*

$$(\langle S_0[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)] \mid e[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)] \rangle, \langle S_0'[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)] \mid e[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)] \rangle) \in \mathcal{T}[\![T]\!].$$

Proof. We first show that $\textsf{inv}(S_0[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)], S_0'[\Delta][\mathbb{C}_{\text{ro}}(\Gamma \vDash L)])$. Condition 3 holds by inversion on $L \mid \Delta \mid H \vDash S_0 = S_0'$. Condition 2 holds by inversion on $\Delta \mid H \vDash_{\texttt{protected}} S_0 = S_0'$. Condition 1 holds by inversion on $L \mid \Delta \mid H \vDash S_0 = S_0'$. By assumption and Lemma 12 we have that $\Gamma \vdash$ read-only T. Our goal then follows by Lemma 16. □

Lemma 18 (FTLR)**.** *If $(\langle S \mid \overline{K^\ell} :: e \rangle, \langle S' \mid \overline{K^\ell} :: e \rangle) \in \mathcal{T}[\![T]\!]$, then $T \in$ $\textsf{traces}(\langle S \mid \overline{K^\ell} :: e^{\text{app}} \rangle)$ and there exists a trace $T' \in$ $\textsf{traces}(\langle S' \mid \overline{K^\ell} :: e^{\text{app}} \rangle)$ such that $\textsf{ct}(T) = \textsf{ct}(T')$.*

Proof. By induction on T. □

Theorem 10 ($\mathbb{C}_{\mathsf{ro}}$ guarantees read-only robust constant time)**.** *If* $\Gamma \vDash L$ *is classically constant time and does not contain any* $\mathtt{protect}_p$ *subterms, then* $\mathbb{C}_{\mathsf{ro}}(\Gamma \vDash L)$ *is robustly constant time for read-only attackers (that do not contain* $\mathtt{protect}_p$*).*

Proof. By Lemma 16 and Lemma 18. □

A.4.2 Speculative Protections

Our proof of the Theorem 11 uses a similar technique of semantically interpreting traces as relations between the two states. Here we split the speculative traces into subsequences of non-speculative events (where there was no speculation), mispeculated events (where the speculation was rolled back), and "correctly speculated" events (where the speculative guess was eventually committed). We then show that the non-speculative and "correctly" speculated events are related to an underlying non-speculative trace and thus constant time by the proof of Theorem 10. To show the latter, we rely on the inserted `fence` instruction ensuring that we cannot speculatively return into the application. For mispeculated events we rely on the invariant that the application cannot change the protection level and a lemma that speculative execution cannot read protected memory unless the access level was changed non-speculatively. The proof requires strengthening the statement of classical speculative constant time in two ways, both of which could go overlooked in a classical setting when not considering running cryptographic code in an application context. Firstly, we require that the library be speculatively constant time under an arbitrary initial microarchitectural state. This ensures that the protections take into account any speculative (mis)behavior from the application before the handoff to the library.[1] Secondly, we strengthen the classical statement to add the assumption that the library is speculatively constant time *under an*

[1]We discuss an example where this was overlooked in an existing tool in Section 2.6.

extended function context, i.e. in the presence of unprotected application functions. This requires that the speculative protections account for *cross-domain* speculation attacks, for example, speculatively jumping to unprotected application code.

Definition 16. *We say a step* $\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{\overline{\delta}} \langle \Phi_2 \mid e_2 \rangle$ *is* speculating *if* $\Phi_1.\Xi \neq \bullet$ *and* $\Phi_2.\Xi \neq \bullet$. *We say a subtrace* $\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^{*} \langle \Phi_2 \mid e_2 \rangle$ *is* speculating *if all steps from* $\langle \Phi_1 \mid e_1 \rangle$ *to* $\langle \Phi_2 \mid e_2 \rangle$ *along* $\overline{\delta}$ *are speculating.*

$$
\begin{array}{rcll}
\sigma & ::= & (\delta, \mathcal{N}) & \text{non-speculative trace} \\
& | & (\overline{\delta}, \mathcal{S}) & \text{speculatively rolled-back trace} \\
& | & [\overline{\sigma}, \overline{\delta}] & \text{speculatively committed trace} \\
& & & \text{with } \overline{\delta} \text{ speculated on}
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma & ::= & (\epsilon, \mathcal{N}) \\
& | & (\overline{\delta}, \mathcal{S}) \\
& | & [\overline{\Sigma}, \overline{\epsilon}]
\end{array}
$$

$$\boxed{\mathsf{crunch} : \overline{\sigma} \to \overline{\delta}}$$

$$
\begin{array}{rcl}
\mathsf{crunch}\,(\bullet) & \triangleq & \bullet \\
\mathsf{crunch}\,((\delta, \mathcal{N}) :: \overline{\sigma}) & \triangleq & \delta +\!\!+ \mathsf{crunch}(\overline{\sigma}) \\
\mathsf{crunch}\left((\overline{\delta}, \mathcal{S}) :: \overline{\sigma}\right) & \triangleq & 0 :: \overline{\delta} +\!\!+ 0 :: \mathsf{crunch}(\overline{\sigma}) \\
\mathsf{crunch}\left([\overline{\sigma}, \overline{\delta}] :: \overline{\sigma'}\right) & \triangleq & 0 :: \mathsf{crunch}(\overline{\sigma}) +\!\!+ \overline{\delta} +\!\!+ \mathsf{crunch}(\overline{\sigma'})
\end{array}
$$

$$\boxed{\mathsf{crunch} : \overline{\Sigma} \to \overline{\delta}}$$

$$\mathsf{crunch}\left(\overline{\Sigma}\right) \triangleq \mathsf{crunch}\left(\mathsf{unlabel}(\overline{\Sigma})\right)$$

$$\boxed{\mathsf{nonspec} : \overline{\Sigma} \to \overline{\epsilon}}$$

$$\begin{aligned}
\mathsf{nonspec}\,(\bullet) &\triangleq \bullet \\
\mathsf{nonspec}\left((\epsilon, \mathcal{N}) :: \overline{\Sigma}\right) &\triangleq \epsilon :: \mathsf{nonspec}\left(\overline{\Sigma}\right) \\
\mathsf{nonspec}\left((\overline{\delta}, \mathcal{S}) :: \overline{\Sigma}\right) &\triangleq \mathsf{nonspec}\left(\overline{\Sigma}\right) \\
\mathsf{nonspec}\left([\overline{\Sigma}, \overline{\epsilon}] :: \overline{\Sigma'}\right) &\triangleq \overline{\epsilon} +\!\!+ \mathsf{nonspec}(\overline{\Sigma}) +\!\!+ \mathsf{nonspec}(\overline{\Sigma'})
\end{aligned}$$

$$\boxed{\mathsf{specValid}\ \overline{\Xi}}$$

$$\frac{}{\mathsf{specValid}\ \bullet} \qquad \frac{\mathsf{specValid}\ (\mathsf{addEvents}(\overline{\Xi}, \overline{\delta} +\!\!+ \overline{\mu}))}{\mathsf{specValid}\ (S, e, \overline{\delta}, \overline{\mu}) :: \overline{\Xi}}$$

$$\boxed{(\langle \Phi \mid e \rangle \xhookrightarrow{\overline{\delta}}\!\!\!\!\rightarrow^{+} \langle \Phi \mid e \rangle)|_{\mathcal{S}} = \sigma}$$

$$\frac{\mathsf{length}(\Phi_1.\Xi) = \mathsf{length}(\Phi_2.\Xi)}{(\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{\delta}\!\!\!\!\rightarrow \langle \Phi_2 \mid e_2 \rangle)|_{\mathcal{S}} = (\delta, \mathcal{N})}$$

$$\frac{\begin{array}{c}\mathsf{length}(\Phi_1.\Xi) + 1 = \mathsf{length}(\Phi_2.\Xi) = \mathsf{length}(\Phi_3.\Xi) = \mathsf{length}(\Phi_4.\Xi) + 1 \\ \Phi_3.\Xi = (\Phi_1.S, e_1, \overline{\delta_3}, \overline{\mu}) :: \overline{\Xi} \qquad (\langle \Phi_2 \mid e_2 \rangle \xhookrightarrow{\overline{\delta_2}}\!\!\!\!\rightarrow^{*} \langle \Phi_3 \mid e_3 \rangle)|_{\mathcal{S}^*} = \overline{\sigma}\end{array}}{(\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{0}\!\!\!\!\rightarrow \langle \Phi_2 \mid e_2 \rangle \xhookrightarrow{\overline{\delta_2}}\!\!\!\!\rightarrow^{*} \langle \Phi_3 \mid e_3 \rangle \xhookrightarrow{\overline{\delta_3}}\!\!\!\!\rightarrow \langle \Phi_4 \mid e_4 \rangle)|_{\mathcal{S}} = [\overline{\sigma}, \overline{\delta_3}]}$$

$$\frac{\begin{array}{c}\mathsf{length}(\Phi_1.\Xi) + 1 = \mathsf{length}(\Phi_2.\Xi) = \mathsf{length}(\Phi_3.\Xi) = \mathsf{length}(\Phi_4.\Xi) + 1 \\ \Phi_3.\Xi = (\Phi_1.\widehat{S, e_1}, \overline{\delta}) :: \overline{\Xi} \qquad (\langle \Phi_2 \mid e_2 \rangle \xhookrightarrow{\overline{\delta_2}}\!\!\!\!\rightarrow^{*} \langle \Phi_3 \mid e_3 \rangle)|_{\mathcal{S}^*} = \overline{\sigma}\end{array}}{(\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{0}\!\!\!\!\rightarrow \langle \Phi_2 \mid e_2 \rangle \xhookrightarrow{\overline{\delta_2}}\!\!\!\!\rightarrow^{*} \langle \Phi_3 \mid e_3 \rangle \xhookrightarrow{0}\!\!\!\!\rightarrow \langle \Phi_4 \mid e_1 \rangle)|_{\mathcal{S}} = (\mathsf{crunch}(\overline{\sigma}), \mathcal{S})}$$

$$\boxed{(\langle \Phi \mid e \rangle \xhookrightarrow{\overline{\delta}}\!\!\!\!\rightarrow^* \langle \Phi \mid e \rangle)|_{\mathcal{S}^*} = \overline{\sigma}}$$

$$\frac{}{(\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{\bullet}\!\!\!\!\rightarrow^0 \langle \Phi_1 \mid e_1 \rangle)|_{\mathcal{S}^*} = \bullet}$$

$$\frac{(\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{\overline{\delta_1}}\!\!\!\!\rightarrow^+ \langle \Phi_2 \mid e_2 \rangle)|_{\mathcal{S}} = \sigma_1 \qquad (\langle \Phi_2 \mid e_2 \rangle \xhookrightarrow{\overline{\delta_2}}\!\!\!\!\rightarrow^* \langle \Phi_3 \mid e_3 \rangle)|_{\mathcal{S}^*} = \overline{\sigma_2}}{(\langle \Phi_1 \mid e_1 \rangle \xhookrightarrow{\overline{\delta_1}}\!\!\!\!\rightarrow^+ \langle \Phi_2 \mid e_2 \rangle \xhookrightarrow{\overline{\delta_2}}\!\!\!\!\rightarrow^* \langle \Phi_3 \mid e_3 \rangle)|_{\mathcal{S}^*} = \sigma_1 :: \overline{\sigma_2}}$$

$$\boxed{\textsf{last-nonspec} : \overline{\Sigma} \rightharpoonup \epsilon}$$

$$\left((\epsilon, \mathcal{N}) :: \overline{\Sigma}\right) \triangleq \begin{cases} \epsilon' & \text{when } \textsf{last-nonspec}\left(\overline{\Sigma}\right) = \epsilon' \\ \epsilon & \text{otherwise} \end{cases}$$

$$\textsf{last-nonspec}\left((\overline{\delta}, \mathcal{S}) :: \overline{\Sigma}\right) \triangleq \textsf{last-nonspec}\left(\overline{\Sigma}\right)$$

$$\textsf{last-nonspec}\left([\overline{\Sigma}, \overline{\epsilon}] :: \overline{\Sigma'}\right) \triangleq \begin{cases} \epsilon' & \text{when } \textsf{last-nonspec}\left(\overline{\Sigma'}\right) = \epsilon' \\ \epsilon' & \text{when } \textsf{last-nonspec}\left(\overline{\Sigma}\right) = \epsilon' \\ \epsilon & \text{when } \textsf{last}(\overline{\epsilon}) = \epsilon \end{cases}$$

$$\boxed{\textsf{label} : \epsilon \to \ell}$$

$$\textsf{label}\left(\delta^{\ell}\right) \triangleq \ell$$

$$\textsf{label}\left(\tau^{\ell \to \ell'}\right) \triangleq \ell'$$

$$\boxed{\textsf{label} : \overline{\Sigma} \to \ell}$$

$$\textsf{label}\left(\bullet\right) \triangleq \text{app}$$

$$\mathsf{label}\left(\overline{\Sigma}\right) \triangleq \mathsf{label}\left(\mathsf{last\text{-}nonspec}\left(\overline{\Sigma}\right)\right)$$

$$\boxed{\mathsf{label}_\ell : \Sigma \to \ell}$$

$$\mathsf{label}_\ell(\Sigma) \triangleq \begin{cases} \mathsf{label}(\epsilon) & \text{when } \mathsf{last\text{-}nonspec}(\Sigma :: \bullet) = \epsilon \\ \ell & \text{otherwise} \end{cases}$$

$$\boxed{\langle \Phi \mid \overline{K^\ell} :: e^\ell \rangle \xrightarrow{\Sigma} \langle \Phi \mid \overline{K^\ell} :: e^\ell \rangle}$$

$$\frac{\begin{array}{c}\langle \Phi_1.S \mid \overline{K_1{}^{\ell_{K1}}} :: e_1{}^{\ell_1} \rangle \xRightarrow{\mathsf{nonspec}(\Sigma)}{}^* \langle \Phi_2.S \mid \overline{K_2{}^{\ell_{K2}}} :: e_2{}^{\mathsf{label}_{\ell_1}(\Sigma)} \rangle \\ \langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\mathsf{crunch}(\Sigma)}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle \\ (\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\mathsf{crunch}(\Sigma)}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}} = \mathsf{unlabel}(\Sigma)\end{array}}{\langle \Phi_1 \mid \overline{K_1{}^{\ell_{K1}}} :: e_1{}^{\ell_1} \rangle \xrightarrow{\Sigma} \langle \Phi_2 \mid \overline{K_2{}^{\ell_{K2}}} :: e^{\mathsf{label}_{\ell_1}(\Sigma)} \rangle}$$

Lemma 19. *If* $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle$ *and* $(\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}} = \sigma$*, then* $\Phi_2.S = \mathsf{commit}(\Phi_1.S, \mathsf{nonspec}(\sigma))$.

Proof. By simultaneous induction on $(\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}} = \sigma$ and $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle$ □

Lemma 20. *If* $(\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}} = \sigma$*, then* $\overline{\delta} = \mathsf{crunch}(\overline{\sigma})$.

Proof. By induction on $(\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}} = \sigma$. □

Lemma 21. *If* $(\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}^*} = \overline{\sigma_1}$ *and*

$$(\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}^*} = \overline{\sigma_2},$$

then $\overline{\sigma_1} = \overline{\sigma_2}$.

Proof. By induction on the derivation of σ_1. □

Lemma 22. *If* $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle$ *and* $\Phi_1.\Xi = \Phi_2.\Xi = \bullet$*, then there exists a unique* $\overline{\sigma}$ *such that* $(\langle \Phi_1 \mid e_1 \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid e_2 \rangle)|_{S^*} = \overline{\sigma}$*.*

Proof. Existence follows by induction on the operational semantics with the state of $\mathsf{specValid}\ \overline{\Xi}$ as an invariant. Uniqueness follows by Lemma 21. □

Lemma 23. *If* $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle$ *and* $\Phi_1.\Xi = \Phi_2.\Xi = \bullet$*, then for all* ℓ_{K1} *and* ℓ_1 *there exist unique* $\overline{\Sigma}$*, and* ℓ_{K2} *such that* $\langle \Phi_1 \mid \overline{{K_1}^{\ell_{K1}}} :: {e_1}^{\ell_1} \rangle \overset{\overline{\Sigma}}{\rightarrow}{}^* \langle \Phi_2 \mid \overline{{K_2}^{\ell_{K2}}} :: {e_2}^{\mathsf{label}(\overline{\Sigma})} \rangle$ *and* $\mathsf{crunch}(\overline{\Sigma}) = \overline{\delta}$*.*

Proof. By Lemma 22 there exists a unique $\overline{\sigma}$ such that $(\langle \Phi_1 \mid e_1 \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid e_2 \rangle)|_{S^*} = \overline{\sigma}$. We construct the labels and non-speculative reduction by induction on $\overline{\sigma}$. The remaining conditions follow by assumption and Lemma 20 □

Lemma 24 (Speculation can't change protection)**.** *If* $\langle \Phi_0 \mid e_0 \rangle \overset{0}{\hookrightarrow\!\!\!\rightarrow} \langle \Phi_1 \mid e_1 \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid e_2 \rangle$ *such that* $\Phi_0.\Xi = \bullet$*,* $\Phi_0.S.p = \texttt{public}$*,* $\Phi_1.\Xi \neq \bullet$*, and the subtrace* $\langle \Phi_1 \mid e_1 \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid e_2 \rangle$ *is speculating, then* $\mathsf{read}_b\ v \hookleftarrow z_b[z_o] \in \overline{\delta} \Rightarrow \Phi_0.S(z_b) \neq \texttt{protected}$ *and* $\mathsf{write}_b\ v \mapsto z_b[z_o] \in \overline{\delta} \Rightarrow \Phi_0.S(z_b) \neq \texttt{protected}$*.*

Proof. By induction on the operational semantics with the invariant that that if $\Phi_2.S.p = \texttt{protected}$, then $\mathsf{protect}_p$ exists in $\Phi_2.\Xi$. □

Definition 17. *We define our speculative state invariant,* $\mathsf{sinv}(\Phi, \Phi')$ *as follows:*

1. $\mathsf{inv}(\Phi.S, \Phi'.S)$

2. $\Phi.a = \Phi'.a$

3. $\Phi.\Xi = \Phi'.\Xi$

$$\mathcal{E}_{\mathsf{app}}[\![(\epsilon,\ \mathcal{N})]\!] \triangleq \left\{ \begin{array}{l} (\Phi, \overline{K^\ell}, e, \Phi', \overline{K^\ell}, e, \\ \ \Phi_1, \overline{{K_1}^{\ell_1}}, \Phi'_1, e_1, \overline{{K_1}^{\ell_1}}, e_1) \end{array} \middle| \begin{array}{l} \langle \Phi \mid \overline{K^\ell} :: e^{\mathsf{app}} \rangle \xrightarrow{(\epsilon, \mathcal{N})} \langle \Phi_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathsf{app}} \rangle \\ \langle \Phi' \mid \overline{K^\ell} :: e^{\mathsf{app}} \rangle \xrightarrow{(\epsilon, \mathcal{N})} \langle \Phi'_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathsf{app}} \rangle \\ \mathsf{sinv}(\Phi_1, \Phi'_1) \end{array} \right\}$$

$$\mathcal{E}_{\mathtt{lib}}[\![(\delta^{\mathtt{lib}},\ \mathcal{N})]\!] \triangleq \left\{ \begin{array}{l} (\Phi, \overline{K^\ell}, e, \Phi', \overline{K'^{\ell'}}, e', \\ \ \Phi_1, \overline{{K_1}^{\ell_1}}, e_1, \Phi'_1, \overline{{K'_1}^{\ell'_1}}, e'_1) \end{array} \middle| \begin{array}{l} \exists \delta'. \\ \quad \langle \Phi \mid \overline{K^\ell} :: e^{\mathtt{lib}} \rangle \xrightarrow{(\delta^{\mathtt{lib}}, \mathcal{N})} \langle \Phi_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathtt{lib}} \rangle \\ \quad \langle \Phi' \mid \overline{K'^{\ell'}} :: e'^{\mathtt{lib}} \rangle \xrightarrow{(\delta'^{\mathtt{lib}}, \mathcal{N})} \langle \Phi'_1 \mid \overline{{K'_1}^{\ell'_1}} :: {e'_1}^{\mathtt{lib}} \rangle \\ \quad \mathsf{ct}(\delta) = \mathsf{ct}(\delta') \end{array} \right\}$$

$$\mathcal{E}_{\mathtt{lib}}[\![(\tau^{\mathtt{lib}\to\mathsf{app}},\ \mathcal{N})]\!] \triangleq \left\{ \begin{array}{l} (\Phi, \overline{K^\ell}, e, \Phi', \overline{K^\ell}, e, \\ \ \Phi_1, \overline{{K_1}^{\ell_1}}, e_1, \Phi'_1, \overline{{K_1}^{\ell_1}}, e_1) \end{array} \middle| \begin{array}{l} \langle \Phi \mid \overline{K^\ell} :: e^{\mathtt{lib}} \rangle \xrightarrow{(\tau^{\mathtt{lib}\to\mathsf{app}}, \mathcal{N})} \langle \Phi_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathsf{app}} \rangle \\ \langle \Phi' \mid \overline{K^\ell} :: e^{\mathtt{lib}} \rangle \xrightarrow{(\tau^{\mathtt{lib}\to\mathsf{app}}, \mathcal{N})} \langle \Phi'_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathsf{app}} \rangle \\ \mathsf{sinv}(\Phi_1, \Phi'_1) \end{array} \right\}$$

$$\mathcal{E}_{\mathsf{app}}[\![(\bar{\delta},\ \mathcal{S})]\!] \triangleq \left\{ \begin{array}{l} (\Phi, \overline{K^\ell}, e, \Phi', \overline{K^\ell}, e, \\ \ \Phi_1, \overline{K^\ell}, e, \Phi'_1, \overline{K^\ell}, e) \end{array} \middle| \begin{array}{l} \langle \Phi \mid \overline{K^\ell} :: e^{\mathsf{app}} \rangle \xrightarrow{(\bar{\delta}, \mathcal{S})} \langle \Phi_1 \mid \overline{K^\ell} :: e^{\mathsf{app}} \rangle \\ \langle \Phi' \mid \overline{K^\ell} :: e^{\mathsf{app}} \rangle \xrightarrow{(\bar{\delta}, \mathcal{S})} \langle \Phi'_1 \mid \overline{K^\ell} :: e^{\mathsf{app}} \rangle \\ \mathsf{sinv}(\Phi_1, \Phi'_1) \end{array} \right\}$$

$$\mathcal{E}_{\texttt{lib}}[\![(\bar{\delta}, \mathcal{S})]\!] \triangleq$$

$$\left\{ \begin{array}{l} (\Phi, \overline{K^{\ell}}, e, \Phi', \overline{K'^{\ell'}}, e', \\ \ \Phi_1, \overline{K^{\ell}}, e, \Phi'_1, \overline{K'^{\ell'}}, e') \end{array} \middle| \begin{array}{l} \exists \overline{\delta'}. \\ \quad \langle \Phi \mid \overline{K^{\ell}} :: e^{\texttt{lib}} \rangle \xrightarrow{(\bar{\delta}, \mathcal{S})} \langle \Phi_1 \mid \overline{K^{\ell}} :: e^{\texttt{lib}} \rangle \\ \quad \langle \Phi' \mid \overline{K'^{\ell'}} :: e'^{\texttt{lib}} \rangle \xrightarrow{(\overline{\delta'}, \mathcal{S})} \langle \Phi'_1 \mid \overline{K'^{\ell'}} :: e'^{\texttt{lib}} \rangle \\ \quad \mathsf{ct}(\bar{\delta}) = \mathsf{ct}(\overline{\delta'}) \end{array} \right\}$$

$$\mathcal{E}_{\mathsf{app}}[\![[\overline{\Sigma}, \overline{\epsilon}]]\!] \triangleq$$

$$\left\{ \begin{array}{l} (\Phi, \overline{K^{\ell}}, e, \Phi', \overline{K^{\ell}}, e, \\ \ \Phi_1, \overline{{K_1}^{\ell_1}}, e_1, \Phi'_1, \overline{{K_1}^{\ell_1}}, e_1) \end{array} \middle| \begin{array}{l} \exists \overline{\epsilon'}. \\ \quad \langle \Phi \mid \overline{K^{\ell}} :: e^{\mathsf{app}} \rangle \xrightarrow{[\overline{\Sigma}, \overline{\epsilon}]} \langle \Phi_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathsf{label}([\overline{\Sigma}, \overline{\epsilon}])} \rangle \\ \quad \langle \Phi' \mid \overline{K^{\ell}} :: e^{\mathsf{app}} \rangle \xrightarrow{[\overline{\Sigma}, \overline{\epsilon'}]} \langle \Phi'_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\mathsf{label}([\overline{\Sigma}, \overline{\epsilon}])} \rangle \\ \quad \mathsf{label}_{\mathsf{app}}([\overline{\Sigma}, \overline{\epsilon}]) = \mathsf{app} \Rightarrow \mathsf{sinv}(\Phi_1, \Phi'_1) \\ \quad \mathsf{ct}(\overline{\epsilon}) = \mathsf{ct}(\overline{\epsilon'}) \end{array} \right\}$$

$$\mathcal{E}_{\texttt{lib}}[\![[\overline{\Sigma}, \overline{\epsilon}]]\!] \triangleq$$

$$\left\{ \begin{array}{l} (\Phi, \overline{K^{\ell}}, e, \Phi', \overline{K'^{\ell'}}, e', \\ \ \Phi_1, \overline{{K_1}^{\ell_1}}, e_1, \Phi'_1, \overline{{K'_1}^{\ell'_1}}, e'_1) \end{array} \middle| \begin{array}{l} \exists \overline{\Sigma'}, \overline{\epsilon'}. \\ \quad \langle \Phi \mid \overline{K^{\ell}} :: e^{\texttt{lib}} \rangle \xrightarrow{[\overline{\Sigma}, \overline{\epsilon}]} \langle \Phi_1 \mid \overline{{K_1}^{\ell_1}} :: {e_1}^{\texttt{lib}} \rangle \\ \quad \langle \Phi' \mid \overline{K'^{\ell'}} :: e'^{\texttt{lib}} \rangle \xrightarrow{[\overline{\Sigma'}, \overline{\epsilon'}]} \langle \Phi'_1 \mid \overline{{K'_1}^{\ell'_1}} :: {e'_1}^{\texttt{lib}} \rangle \\ \quad \mathsf{ct}(\mathsf{crunch}([\overline{\Sigma}, \overline{\epsilon}])) = \mathsf{ct}(\mathsf{crunch}([\overline{\Sigma'}, \overline{\epsilon'}])) \end{array} \right\}$$

$$\mathcal{S}_{\ell}[\![\Sigma :: \overline{\Sigma'}]\!] \triangleq$$

$$\left\{ (\langle \Phi \mid \overline{K^{\ell_K}} :: e^{\ell} \rangle, \langle \Phi' \mid \overline{K'^{\ell'_K}} :: e'^{\ell} \rangle) \;\middle|\; \begin{array}{l} \exists \Phi_1, \overline{K_1{}^{\ell_{K1}}}, e_1, \Phi'_1, \overline{K'_1{}^{\ell'_{K1}}}, e'_1, \\ \quad \Phi_2, \overline{K_2{}^{\ell_{K2}}}, e_2, \Phi'_2, \overline{K'_2{}^{\ell'_{K2}}}, e'_2. \\ \quad (\Phi, \overline{K^{\ell_K}}, e, \Phi', \overline{K'^{\ell'_K}}, e', \\ \qquad \Phi_1, \overline{K_1{}^{\ell_{K1}}}, e_1, \Phi'_1, \overline{K'_1{}^{\ell'_{K1}}}, e'_1) \in \mathcal{E}_{\ell}[\![\Sigma]\!] \\ \quad (\Phi_1, \Phi'_1, \overline{K_1{}^{\ell_{K1}}}, e_1, \\ \qquad \Phi_2, \Phi'_2, \overline{K_2{}^{\ell_{K2}}}, e_2) \in \mathcal{S}_{\mathsf{label}(\Sigma)}[\![\overline{\Sigma'}]\!] \end{array} \right\}$$

$$\mathcal{S}_{\mathsf{app}}[\![\bullet]\!] \triangleq \left\{ (\langle \Phi \mid \overline{K^{\ell_K}} :: e^{\mathsf{app}} \rangle, \langle \Phi' \mid \overline{K^{\ell_K}} :: e^{\mathsf{app}} \rangle) \;\middle|\; \mathsf{sinv}(\Phi, \Phi') \right\}$$

$$\mathcal{S}_{\mathtt{lib}}[\![\bullet]\!] \triangleq \left\{ (\langle \Phi \mid \overline{K^{\ell_K}} :: e^{\mathtt{lib}} \rangle, \langle \Phi' \mid \overline{K'^{\ell'_K}} :: e'^{\mathtt{lib}} \rangle) \right\}$$

Lemma 25 ($\mathbb{C}_{\mathsf{spec}}$ preserves classic speculative constant time)**.** *If $\Gamma \vDash L$ is classically speculative constant time then $\mathbb{C}_{\mathsf{spec}}\Gamma \vDash L$ is classically speculative constant time.*

Proof. By induction on the compiler. □

Lemma 26 ($\mathbb{C}_{\mathsf{spec}}$ prevents speculative returns)**.** *If $\Gamma \vDash L$ is classically speculative constant time with respect to a speculation oracle $\mathsf{spec} : A \times S \times e \to A \times d$, then for all secret contexts Δ, classical "applications" $\Gamma, \Delta \vdash (H, e_\Gamma)$, initial states S_0, S'_0 such that $L \mid \Delta \mid H \vDash S_0$, microarchitectural states $a : A$, $\Phi_0 = \{S = S_0[\Delta][L], a = a, \Xi = \bullet\}$, and $e_0 = e_\Gamma[\Delta][\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L)]$, then $\mathsf{begin} \mathbin{+\!\!+} \overline{\delta} \mathbin{+\!\!+} \mathsf{end}\ v \in \mathsf{specTraces}(\langle \Phi_0 \mid e_0 \rangle)$ implies that there exists a $\overline{\Sigma}$ such that $\mathsf{nonspec}(\overline{\Sigma}) = \mathsf{begin} \mathbin{+\!\!+} \overline{\delta} \mathbin{+\!\!+} \mathsf{end}\ v$ and the tail of $\overline{\Sigma}$ is of the form $(\mathsf{fence}^{\mathtt{lib}}, \mathcal{N}) \mathbin{+\!\!+} (\overline{\delta_1}, \mathcal{S})^* \mathbin{+\!\!+} (\mathsf{ret}\ v^{\mathtt{lib}\to\mathsf{app}}, \mathcal{N}) \mathbin{+\!\!+} (\mathsf{end}\ v^{\mathsf{app}\to\mathtt{lib}}, \mathcal{N})$ or $(\mathsf{fence}^{\mathtt{lib}}, \mathcal{N}) \mathbin{+\!\!+} [(\overline{\delta_1}, \mathcal{S})^* \mathbin{+\!\!+} (\mathsf{ret}\ v^{\mathtt{lib}\to\mathsf{app}}, \mathcal{N}), 0] \mathbin{+\!\!+} (\mathsf{end}\ v^{\mathsf{app}\to\mathtt{lib}}, \mathcal{N})$.*

Proof. By induction on the compiler. □

Lemma 27 ($ extension)**.** *If*

$$\langle \Phi \mid \bullet :: e^{\mathsf{app}} \rangle \xrightarrow{\overline{\Sigma_1}}{}^{*} \langle \Phi_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{label}(\overline{\Sigma_1})} \rangle \xrightarrow{\Sigma_2} \langle \Phi_2 \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{label}_{\mathsf{label}(\overline{\Sigma_1})}(\Sigma_2)} \rangle$$

and $(\langle \Phi \mid \bullet :: e^{\mathsf{app}} \rangle, \langle \Phi' \mid \bullet :: e^{\mathsf{app}} \rangle) \in \mathcal{S}_{\mathsf{app}}[\![\overline{\Sigma_1}]\!]$, *then if*

1. *There exist* $\overline{\Sigma_1'}$, Φ_1', $\overline{K_1'{}^{\ell_1'}}$, *and* e_1' *such that* $\langle \Phi' \mid \bullet :: e^{\mathsf{app}} \rangle \xrightarrow{\overline{\Sigma_1'}}{}^{*} \langle \Phi_1' \mid \overline{K_1'{}^{\ell_1'}} :: e_1'{}^{\mathsf{label}(\overline{\Sigma_1})} \rangle$.
2. *If* $\mathsf{label}(\overline{\Sigma_1}) = \mathsf{app}$, *then* $\overline{K_1{}^{\ell_1}} :: e_1 = \overline{K_1'{}^{\ell_1'}} :: e_1'$ *and* $\mathsf{sinv}(\Phi_1, \Phi_1')$.
3. *Let*

$$\langle \Phi.S \mid \bullet :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\delta_0} +\!\!+ (\mathsf{call}\ z_f)^{\mathsf{app}\to\mathtt{lib}} +\!\!+ \overline{\delta_1}}{}^{*} \langle \Phi_1.S \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{label}(\overline{\Sigma_1})} \rangle$$

   such that $(\mathsf{call}\ z_f')^{\mathsf{app}\to\mathtt{lib}} \notin \overline{\delta_1}$ *and* $\overline{\delta_0} +\!\!+ (\mathsf{call}\ z_f)^{\mathsf{app}\to\mathtt{lib}} +\!\!+ \overline{\delta_1} = \mathsf{nonspec}(\overline{\Sigma_1})$. *Then*

$$\langle \Phi'.S \mid \bullet :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\delta_0'} +\!\!+ (\mathsf{call}\ z_f)^{\mathsf{app}\to\mathtt{lib}} +\!\!+ \overline{\delta_1'}}{}^{*} \langle \Phi_1'.S \mid \overline{K_1'{}^{\ell_1'}} :: e_1'{}^{\mathsf{label}(\overline{\Sigma_1})} \rangle$$

   such that $(\mathsf{call}\ z_f')^{\mathsf{app}\to\mathtt{lib}} \notin \overline{\delta_1'}$ *and* $\overline{\delta_0'} +\!\!+ (\mathsf{call}\ z_f)^{\mathsf{app}\to\mathtt{lib}} +\!\!+ \overline{\delta_1'} = \mathsf{nonspec}(\overline{\Sigma_1'})$.
4. $\mathsf{ct}(\mathsf{crunch}(\overline{\Sigma_1})) = \mathsf{ct}(\mathsf{crunch}(\overline{\Sigma_1'}))$

imply that there exist Φ_2', $\overline{K_2'{}^{\ell_2'}}$, *and* e_2' *such that* $(\Phi_1, \overline{K_1{}^{\ell_1}}, e_1, \Phi_1', \overline{K_1'{}^{\ell_1'}}, e_1', \Phi_2, \overline{K_2{}^{\ell_2}}, e_2, \Phi_2', \overline{K_2'{}^{\ell_2'}}, e_2') \in \mathcal{E}_{\mathsf{label}(\overline{\Sigma_1})}[\![\Sigma_2]\!]$, *then* $(\langle \Phi \mid \bullet :: e^{\mathsf{app}} \rangle, \langle \Phi' \mid \bullet :: e^{\mathsf{app}} \rangle) \in \mathcal{S}_{\mathsf{app}}[\![\overline{\Sigma_1} +\!\!+ \Sigma_2]\!]$.

Proof. By induction on $\overline{\Sigma_1}$. □

Lemma 28. *If* $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0}\!\!\!\rightarrow \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle \xhookrightarrow{\overline{\delta}}\!\!\!\rightarrow^{*} \langle \Phi_3 \mid \overline{K_3} :: e_3 \rangle$, $\Phi_1.\Xi = \bullet$, $\Phi_2.\Xi \neq \bullet$, $\langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle \xhookrightarrow{\overline{\delta}}\!\!\!\rightarrow^{*} \langle \Phi_3 \mid \overline{K_3} :: e_3 \rangle$ *is speculating, and* $\mathsf{sinv}(\Phi_1, \Phi_1')$, *then* $\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0}\!\!\!\rightarrow \langle \Phi_2' \mid \overline{K_2} :: e_2 \rangle \xhookrightarrow{\overline{\delta}}\!\!\!\rightarrow^{*} \langle \Phi_3' \mid \overline{K_3} :: e_3 \rangle$ *and* $\mathsf{sinv}(\Phi_3, \Phi_3')$.

Proof. We proceed by induction on $\overline{\delta}$. We have by $\mathsf{sinv}(\Phi_1, \Phi_1')$ that $\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0}\!\!\!\rightarrow \langle \Phi_2' \mid \overline{K_2} :: e_2 \rangle$, completing our base case. In the case where $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0}\!\!\!\rightarrow \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle \xhookrightarrow{\overline{\delta}}\!\!\!\rightarrow^{*} \langle \Phi_3 \mid \overline{K_3} :: e_3 \rangle \xhookrightarrow{\delta_3}\!\!\!\rightarrow \langle \Phi_4 \mid \overline{K_4} :: e_4 \rangle$ we must show that $\mathsf{sinv}(\Phi_3, \Phi_3')$ implies that $\langle \Phi_3' \mid$

$\overline{K_3} :: e_3\rangle \overset{\delta_3}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi_4' \mid \overline{K_4} :: e_4\rangle$. This follows by case analysis on e_3, the fact that $\mathsf{inv}(\Phi_3.S, \Phi_3'.S)$, and Lemma 24. □

Lemma 29. *For a classically speculatively constant time library $\Gamma \vDash L$, secret context Δ, memory-safe application $\Gamma, \Delta \vDash (H, e_a)$, initial states $\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L) \mid \Delta \mid H \vDash S_0 = S_0'$, $S = S_0[\Delta][\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L)]$, $S' = S_0'[\Delta][\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L)]$, $\Phi = \{S = S, a = a, \Xi = \bullet\}$; $\Phi' = \{S = S', a = a, \Xi = \bullet\}$, and $e = e_a[\Delta][\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L)]$, if $\langle \Phi \mid \bullet :: e^{\mathsf{app}}\rangle \xrightarrow{\overline{\Sigma}}{}^* \langle \Phi_1 \mid \overline{K_1}^{\ell_1} :: e_1{}^{\mathsf{label}(\overline{\Sigma})}\rangle$, then $(\langle \Phi \mid \bullet :: e^{\mathsf{app}}\rangle, \langle \Phi' \mid \bullet :: e^{\mathsf{app}}\rangle) \in \mathcal{S}_{\mathsf{app}}[\![\overline{\Sigma}]\!]$.*

Proof. We proceed by induction on $\overline{\Sigma}$. In the base case we have $\overline{\Sigma} = \bullet$, where $\mathsf{sinv}(\Phi, \Phi')$ follows from our assumptions.

In our inductive case we have $\overline{\Sigma} = \overline{\Sigma_1} +\!\!+ \Sigma_2$ and therefore $\langle \Phi \mid \bullet :: e^{\mathsf{app}}\rangle \xrightarrow{\overline{\Sigma_1}}{}^* \langle \Phi_1 \mid \overline{K_1}^{\ell_1} :: e_1{}^{\mathsf{label}(\overline{\Sigma_1})}\rangle \xrightarrow{\Sigma_2} \langle \Phi_2 \mid \overline{K_2}^{\ell_2} :: e_2{}^{\mathsf{label}_{\mathsf{label}(\overline{\Sigma_1})}(\Sigma_2)}\rangle$. By our inductive hypothesis we have that $(\langle \Phi \mid \bullet :: e^{\mathsf{app}}\rangle, \langle \Phi' \mid \bullet :: e^{\mathsf{app}}\rangle) \in \mathcal{S}_{\mathsf{app}}[\![\overline{\Sigma_1}]\!]$. By Lemma 27 we may assume that there exist $\overline{\Sigma_1'}$, Φ_1', $\overline{K_1'}^{\ell_1'}$, and e_1' such that $\langle \Phi' \mid \bullet :: e^{\mathsf{app}}\rangle \xrightarrow{\overline{\Sigma_1'}}{}^* \langle \Phi_1' \mid \overline{K_1'}^{\ell_1'} :: e_1'{}^{\mathsf{label}(\overline{\Sigma_1})}\rangle$ and $\mathsf{label}(\overline{\Sigma_1}) = \mathsf{app} \Rightarrow \overline{K_1}^{\ell_1} :: e_1 = \overline{K_1'}^{\ell_1'} :: e_1' \wedge \mathsf{sinv}(\Phi_1, \Phi_1')$. We must then show that there exist Φ_2', $\overline{K_2'}^{\ell_2'}$, and e_2' such that $(\Phi_1, \overline{K_1}^{\ell_1}, e_1, \Phi_1', \overline{K_1'}^{\ell_1'}, e_1', \Phi_2, \overline{K_2}^{\ell_2}, e_2, \Phi_2', \overline{K_2'}^{\ell_2'}, e_2') \in \mathcal{E}_{\mathsf{label}(\overline{\Sigma_1})}[\![\Sigma_2]\!]$.

We proceed by case analysis on $\mathsf{label}(\overline{\Sigma_1})$ and Σ_2. We first consider all of the cases where $\mathsf{label}(\overline{\Sigma_1}) = \mathsf{app}$. By assumption $\overline{K_1}^{\ell_1} :: e_1 = \overline{K_1'}^{\ell_1'} :: e_1'$ and $\mathsf{sinv}(\Phi_1, \Phi_1')$. We pick $\overline{K_2'}^{\ell_2'} :: e_2' = \overline{K_2}^{\ell_2} :: e_2$. By inversion on $\langle \Phi_1 \mid \overline{K_1}^{\ell_1} :: e_1{}^{\mathsf{label}(\overline{\Sigma_1})}\rangle \xrightarrow{\Sigma_2} \langle \Phi_2 \mid \overline{K_2}^{\ell_2} :: e_2{}^{\mathsf{label}_{\mathsf{label}(\overline{\Sigma_1})}(\Sigma_2)}\rangle$ we have

1. $\langle \Phi_1.S \mid \overline{K_1}^{\ell_{K1}} :: e_1{}^{\ell_1}\rangle \xRightarrow{\mathsf{nonspec}(\Sigma)}{}^* \langle \Phi_2.S \mid \overline{K_2}^{\ell_{K2}} :: e_2{}^{\mathsf{label}_{\ell_1}(\Sigma)}\rangle$
2. $\langle \Phi_1 \mid \overline{K_1} :: e_1\rangle \overset{\mathsf{crunch}(\Sigma)}{\hookrightarrow\!\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2\rangle$
3. $(\langle \Phi_1 \mid \overline{K_1} :: e_1\rangle \overset{\mathsf{crunch}(\Sigma)}{\hookrightarrow\!\!\!\!\rightarrow}{}^* \langle \Phi_2 \mid \overline{K_2} :: e_2\rangle)|_S = \mathsf{unlabel}(\Sigma)$

Case $\Sigma_2 = (\epsilon, \mathcal{N})$ ***:*** We must show that there exists a Φ_2' such that

1. $\langle \Phi_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}} \rangle \xrightarrow{(\epsilon, \mathcal{N})} \langle \Phi_2 \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{app}} \rangle$

2. $\langle \Phi_1' \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{app}} \rangle \xrightarrow{(\epsilon, \mathcal{N})} \langle \Phi_2' \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{app}} \rangle$

3. $\mathsf{sinv}(\Phi_2, \Phi_2')$

The first follows immediately by assumption.

For the second we first show that there exists an S_2' such that $\langle \Phi_1'.S \mid \overline{K_1{}^{\ell_{K1}}} :: e_1{}^{\ell_1} \rangle \xRightarrow{\epsilon}{}^* \langle S_2' \mid \overline{K_2{}^{\ell_{K2}}} :: e_2{}^{\mathsf{label}_{\ell_1}((\epsilon, \mathcal{N}))} \rangle$. This follows by Lemma 16 and the definitions of $\mathbb{T}$ and $\mathbb{A}$. We then let $\Phi_2' = \{S = S_2', a = \mathit{spec}(\Phi_1.a, e_1).2, \Xi = \Phi_2.\Xi\}$. $\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\mathsf{unlabel}(\epsilon)}{}^* \langle \Phi_2' \mid \overline{K_2} :: e_2 \rangle$ by the same logic as Lemma 15 (by inversion $\mathit{spec}(\Phi_1.a, e_1) = \mathsf{nonspec}$ so the rule SPEC-β applies). $(\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{\mathsf{unlabel}(\epsilon)}{}^* \langle \Phi_2' \mid \overline{K_2} :: e_2 \rangle)|_{\mathcal{S}} = \mathsf{unlabel}(\epsilon)$ by unrolling of definitions and assumption.

The speculative invariant follows by Lemma 16, the definitions of $\mathbb{T}$ and $\mathbb{A}$ and the fact that $\Phi_1.a = \Phi_1'.a$ and $\Phi_1.\Xi = \Phi_1.\Xi$. ■

Case $\Sigma_2 = (\overline{\delta}, \mathcal{S})$ ***:*** By inversion we have that $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0::\overline{\delta}}{}^* \langle \Phi_3 \mid \overline{K_3} :: e_3 \rangle \xhookrightarrow{0} \langle \Phi_2 \mid \overline{K_1} :: e_1 \rangle$. By Lemma 28 we have that $\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0::\overline{\delta}}{}^* \langle \Phi_3' \mid \overline{K_3} :: e_3 \rangle$ and $\mathsf{sinv}(\Phi_3, \Phi_3')$. Therefore $\langle \Phi_3' \mid \overline{K_3} :: e_3 \rangle \xhookrightarrow{0} \langle \Phi_2' \mid \overline{K_1} :: e_1 \rangle$. The other conditions follow immediately. ■

Case $\Sigma_2 = [\overline{\Sigma_3}, \overline{\epsilon}]$ ***:*** We must show that there exists a $\overline{\epsilon'}$ such that $\langle \Phi_1' \mid \overline{K_1{}^{\ell_1}} :: K[e_1]^{\mathsf{app}} \rangle \xrightarrow{[\overline{\Sigma}, \overline{\epsilon'}]} \langle \Phi_2' \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{label}([\overline{\Sigma}, \overline{\epsilon}])} \rangle$, $\mathsf{label}_{\mathsf{app}}([\overline{\Sigma}, \overline{\epsilon}]) = \mathsf{app} \Rightarrow \mathsf{sinv}(\Phi_2, \Phi_2')$, and $\mathsf{ct}(\overline{\epsilon}) = \mathsf{ct}(\overline{\epsilon'})$.

We case split on whether $(\mathsf{call}\ z_f)^{\mathsf{app}\to\mathtt{lib}} \in \overline{\epsilon}$. If it isn't then we let $\overline{\epsilon'} = \overline{\epsilon}$. By inversion we have that $\langle \Phi_1 \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0} \langle \Phi_3 \mid \overline{K_3} :: e_3 \rangle \xhookrightarrow{\overline{\delta}}{}^* \langle \Phi_4 \mid \overline{K_4} :: e_4 \rangle \xhookrightarrow{\overline{\epsilon}} \langle \Phi_2 \mid \overline{K_2} :: e_2 \rangle$. By Lemma 16 and the definitions of $\mathbb{T}$ and $\mathbb{A}$ on the underlying non-speculative subtrace for e_1 we have that $\langle \Phi_1' \mid \bullet :: e_1 \rangle \xhookrightarrow{\mathsf{unlabel}(\overline{\epsilon})}{}^* \langle \Phi_1'' \mid \bullet :: v \rangle$ and therefore $\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0} \langle \Phi_3' \mid \overline{K_3} :: e_3 \rangle$ with $\mathsf{sinv}(\Phi_3, \Phi_3')$. By Lemma 28 we have that $\langle \Phi_1' \mid \overline{K_1} :: e_1 \rangle \xhookrightarrow{0::\overline{\delta}}{}^* \langle \Phi_4' \mid \overline{K_4} :: e_4 \rangle$ and $\mathsf{sinv}(\Phi_3, \Phi_3')$. Our goals then follow immediately.

If $(\mathsf{call}\ z_f)^{\mathsf{app}\to\mathtt{lib}} \in \overline{\epsilon}$, then by Lemma 16 we have that there is some underlying non-speculative trace $\overline{\epsilon'}$ such that $\mathsf{ct}(\overline{\epsilon}) = \mathsf{ct}(\overline{\epsilon'})$ which contains the entire call into the

library. By the definition of classical constant time, Lemma 14, and $\mathsf{sinv}(\Phi_1, \Phi'_1)$, this call leaves no traces in unprotected memory (and therefore no reads or writes to unprotected memory may be invalidated). Therefore we may once again apply the same reasoning to get that $\langle \Phi'_1 \mid \overline{K_1{}^{\ell_1}} :: K[e_1]^{\mathsf{app}} \rangle \xrightarrow{[\overline{\Sigma}, \overline{\epsilon'}]} \langle \Phi'_2 \mid \overline{K_2{}^{\ell_2}} :: e_2{}^{\mathsf{label}([\overline{\Sigma}, \overline{\epsilon}])} \rangle$. ■

We next consider all of the cases where $\mathsf{label}(\overline{\Sigma_1}) = \mathtt{lib}$. By assumption we may let

$$\langle \Phi.S \mid \bullet :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\delta_0} + (\mathsf{call}\ z_f)^{\mathsf{app} \to \mathtt{lib}} + \overline{\delta_1}}{}^{*} \langle \Phi_1.S \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{label}(\overline{\Sigma_1})} \rangle$$

such that $(\mathsf{call}\ z'_f)^{\mathsf{app} \to \mathtt{lib}} \notin \overline{\delta_1}$ and $\overline{\delta_0} + (\mathsf{call}\ z_f)^{\mathsf{app} \to \mathtt{lib}} + \overline{\delta_1} = \mathsf{nonspec}(\overline{\Sigma_1})$ and then have that

$$\langle \Phi'.S \mid \bullet :: e^{\mathsf{app}} \rangle \xRightarrow{\overline{\delta'_0} + (\mathsf{call}\ z_f)^{\mathsf{app} \to \mathtt{lib}} + \overline{\delta'_1}}{}^{*} \langle \Phi'_1.S \mid \overline{K'_1{}^{\ell'_1}} :: e'_1{}^{\mathsf{label}(\overline{\Sigma_1})} \rangle$$

such that $(\mathsf{call}\ z'_f)^{\mathsf{app} \to \mathtt{lib}} \notin \overline{\delta'_1}$ and $\overline{\delta'_0} + (\mathsf{call}\ z_f)^{\mathsf{app} \to \mathtt{lib}} + \overline{\delta'_1} = \mathsf{nonspec}(\overline{\Sigma'_1})$. By the assumption that the attacker is memory safe, $z_f \in \mathrm{dom}(\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L))$.

Case $\Sigma_2 = (\delta^{\mathtt{lib}}, \mathcal{N})$ ***:*** Follows by the above assumption, the assumption that the library is classically speculative constant time, and Lemma 25. ■

Case $\Sigma_2 = (\tau^{\mathtt{lib} \to \mathsf{app}}, \mathcal{N})$ ***:*** Follows by the above assumption, the assumption that the library is classically speculative constant time, Lemma 25, and the same reasoning using Lemma 14 as in Lemma 16. ■

Case $\Sigma_2 = (\overline{\delta}, \mathcal{S})$ ***:*** Follows by the above assumption, the assumption that the library is classically speculative constant time, Lemma 25, and Lemma 26. ■

Case $\Sigma_2 = [\overline{\Sigma_3}, \overline{\epsilon}]$ ***:*** Follows by the above assumption, the assumption that the library is classically speculative constant time, Lemma 25, and Lemma 26. ■

□

Lemma 30 (FTLR). *If* $(\langle \Phi \mid \bullet :: e^{\text{app}} \rangle, \langle \Phi' \mid \bullet :: e^{\text{app}} \rangle) \in \mathcal{S}_{\text{app}}[\![\overline{\Sigma}]\!]$*, then there exist* $\overline{\Sigma'}$, Φ_1, $\overline{K_1{}^{\ell_1}}$, e_1, Φ'_1, $\overline{K'_1{}^{\ell'_1}}$*, and* e'_1 *such that* $\langle \Phi \mid \bullet :: e^{\text{app}} \rangle \xrightarrow{\overline{\Sigma}}{}^{*} \langle \Phi_1 \mid \overline{K_1{}^{\ell_1}} :: e_1{}^{\mathsf{label}(\overline{\Sigma})} \rangle$ *and* $\langle \Phi' \mid \bullet :: e^{\text{app}} \rangle \xrightarrow{\overline{\Sigma'}}{}^{*} \langle \Phi'_1 \mid \overline{K'_1{}^{\ell'_1}} :: e'_1{}^{\mathsf{label}(\overline{\Sigma})} \rangle$ *and* $\mathsf{ct}(\mathsf{crunch}(\overline{\Sigma})) = \mathsf{ct}(\mathsf{crunch}(\overline{\Sigma'}))$.

Proof. By induction on $\overline{\Sigma}$. □

Theorem 11 ($\mathbb{C}_{\mathsf{spec}}$ guarantees robust speculative constant time)**.** *If* $\Gamma \vDash L$ *is classically speculative constant time for a speculation oracle spec and does not contain any* `protect`$_p$ *subterms, then* $\mathbb{C}_{\mathsf{spec}}(\Gamma \vDash L)$ *is robustly speculatively constant time (for attackers that do not contain* `protect`$_p$*).*

Proof. By Lemma 23, Lemma 29, and Lemma 30. □

A.4.3 Concurrent Protections

Theorem 12 ($\mathbb{C}_{\mathsf{ro\text{-}co}}$ guarantees robust constant time for concurrent observers)**.** *If* $\Gamma \vDash L$ *is classically constant time and does not contain any* `protect`$_p$ *subterms, then* $\mathbb{C}_{\mathsf{ro\text{-}co}}(\Gamma \vDash L)$ *is robustly constant time for concurrent observers (that do not contain* `protect`$_p$*).*

Theorem 13 ($\mathbb{C}_{\mathsf{spec\text{-}co}}$ guarantees robust speculative constant time for concurrent observers)**.** *If* $\Gamma \vDash L$ *is classically speculative constant time for a speculation oracle spec and does not contain any* `protect`$_p$ *subterms, then* $\mathbb{C}_{\mathsf{spec\text{-}co}}(\Gamma \vDash L)$ *is robustly speculatively constant time (for attackers that do not contain* `protect`$_p$*).*

Both proofs exactly follow the structure of their non-concurrent counterparts with the addition of maintaining the state invariant during library subtraces as well. This ensures that *at all times* the only memory that varies is protected from concurrent observers.

Bibliography

Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jeremy Thibault. June 2019. "Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation". In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 2019 IEEE 32nd Computer Security Foundations Symposium (CSF). IEEE, Hoboken, NJ, USA, (June 2019), 256–25615. ISBN: 978-1-72811-407-1. DOI: 10.1109/CSF.2019.00025.

Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. May 1, 1996. "Efficient and language-independent mobile programs". In: *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (PLDI '96). Association for Computing Machinery, New York, NY, USA, (May 1, 1996), 127–136. ISBN: 978-0-89791-795-7. DOI: 10.1145/231379.231402.

Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Oct. 22, 2006. "Deconstructing process isolation". In: *Proceedings of the 2006 workshop on Memory system performance and correctness* (MSPC '06). Association for Computing Machinery, New York, NY, USA, (Oct. 22, 2006), 1–10. ISBN: 978-1-59593-578-6. DOI: 10.1145/1178597.1178599.

Fritz Alder, Jo Van Bulck, David Oswald, and Frank Piessens. Dec. 8, 2020. "Faulty Point Unit: ABI Poisoning Attacks on Intel SGX". In: *Proceedings of the 36th Annual Computer Security Applications Conference* (ACSAC '20). Association for Computing Machinery, New York, NY, USA, (Dec. 8, 2020), 415–427. ISBN: 978-1-4503-8858-0. DOI: 10.1145/3427228.3427270.

Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Juliane Krämer, Patrick Longa, and Jefferson E. Ricardini. 2020. "The Lattice-Based Digital Signature Scheme qTESLA". In: *Applied Cryptography and Network Security*. Ed. by Mauro Conti, Jianying Zhou, Emiliano

Casalicchio, and Angelo Spognardi. Springer International Publishing, Cham, 441–460. isbn: 978-3-030-57808-4. doi: 10.1007/978-3-030-57808-4_22.

José Bacelar Almeida et al.. Oct. 30, 2017. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (CCS '17). Association for Computing Machinery, New York, NY, USA, (Oct. 30, 2017), 1807–1823. isbn: 978-1-4503-4946-8. doi: 10.1145/3133956.3134078.

Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. June 4, 2011. "Language-independent sandboxing of just-in-time compilation and self-modifying code". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). Association for Computing Machinery, New York, NY, USA, (June 4, 2011), 355–366. isbn: 978-1-4503-0663-8. doi: 10.1145/1993498.1993540.

Alexandre Bartel and John Doe. Sept. 28, 2018. *Twenty years of Escaping the Java Sandbox*. (Sept. 28, 2018). Retrieved June 12, 2025 from https://www.exploit-db.com/exploits/45517.

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Dec. 20, 2019. "Formal verification of a constant-time preserving C compiler". *Supplementary material for Article: Formal Verification of a Constant-Time Preserving C Compiler*, 4, (Dec. 20, 2019), 7:1–7:30, POPL, (Dec. 20, 2019). doi: 10.1145/3371075.

Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. May 2021. "High-Assurance Cryptography in the Spectre Era". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2021), 1884–1901. doi: 10.1109/SP40001.2021.00046.

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. July 2018. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 2018 IEEE 31st Computer Security Foundations Symposium (CSF). ISSN: 2374-8303. (July 2018), 328–343. doi: 10.1109/CSF.2018.00031.

J.F. Bastien. Dec. 18, 2018. *Automatic variable initialization*. GitHub. (Dec. 18, 2018). Retrieved June 12, 2025 from https://github.com/llvm/llvm-project/commit/14daa20be1ad89639ec209d969232d19cf698845.

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. "Dune: Safe User-level Access to Privileged {CPU} Features". In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 335–348. ISBN: 978-1-931971-96-6. Retrieved June 12, 2025 from https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay.

D. Bernstein. 2005. "Cache-timing attacks on AES". In: Retrieved June 12, 2025 from https://www.semanticscholar.org/paper/Cache-timing-attacks-on-AES-Bernstein/352e74019d86163d73618f03429ae452ab429629.

Daniel J. Bernstein. 2008. "The Salsa20 Family of Stream Ciphers". In: *New Stream Cipher Designs: The eSTREAM Finalists*. Lecture Notes in Computer Science. Ed. by Matthew Robshaw and Olivier Billet. Springer, Berlin, Heidelberg, 84–97. ISBN: 978-3-540-68351-3. DOI: 10.1007/978-3-540-68351-3_8.

Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. 2019. "Compiling Sandboxes: Formally Verified Software Fault Isolation". In: *Programming Languages and Systems*. Ed. by Luís Caires. Springer International Publishing, Cham, 499–524. ISBN: 978-3-030-17184-1. DOI: 10.1007/978-3-030-17184-1_18.

Frédéric Besson, Thomas Jensen, and Julien Lepiller. 2018. "Modular Software Fault Isolation as Abstract Interpretation". In: *Static Analysis*. Ed. by Andreas Podelski. Springer International Publishing, Cham, 166–186. ISBN: 978-3-319-99725-4. DOI: 10.1007/978-3-319-99725-4_12.

Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *5th {USENIX} Symposium on Networked Systems Design & Implementation*. {NSDI} 2008. {USENIX} Association, 309–322. https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/bittau/bittau.pdf.

Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2020. “Provably-Safe Multilingual Software Sandboxing using WebAssembly”. (2020).

Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Apr. 4, 2017. “Control-Flow Integrity: Precision, Security, and Performance”. *ACM Comput. Surv.*, 50, 1, (Apr. 4, 2017), 16:1–16:33. doi: 10.1145/3054924.

Nathan Burow, Xinping Zhang, and Mathias Payer. May 2019. “SoK: Shining Light on Shadow Stacks”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. IEEE, (May 2019), 985–999. doi: 10.1109/SP.2019.00076.

[SW] Bytecode Alliance, *WebAssembly Micro Runtime* 2020. url: https://github.com/bytecodealliance/wasm-micro-runtime.

Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: 28th USENIX Security Symposium (USENIX Security 19), 249–266. isbn: 978-1-939133-06-9. Retrieved Apr. 25, 2024 from https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity”. In: 24th {USENIX} Security Symposium ({USENIX} Security 15), 161–176. isbn: 978-1-939133-11-3. Retrieved June 19, 2020 from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Oct. 11, 2009. “Fast byte-granularity software fault isolation”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (SOSP ’09). Association for Computing Machinery, New York, NY, USA, (Oct. 11, 2009), 45–58. isbn: 978-1-60558-752-3. doi: 10.1145/1629575.1629581.

Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. June 11, 2020. “Constant-time foundations for the new spectre

era". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). Association for Computing Machinery, New York, NY, USA, (June 11, 2020), 913–926. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385970.

Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. May 2022. "SoK: Practical Foundations for Software Spectre Defenses". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2022), 666–680. DOI: 10.1109/SP46214.2022.9833707.

Sunjay Cauligi, Gary Soeller, et al.. June 8, 2019. "FaCT: a DSL for timing-sensitive computation". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019). Association for Computing Machinery, New York, NY, USA, (June 8, 2019), 174–189. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314605.

Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. June 2019. "A Formal Approach to Secure Speculation". In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 2019 IEEE 32nd Computer Security Foundations Symposium (CSF). ISSN: 2374-8303. (June 2019), 288–28815. DOI: 10.1109/CSF.2019.00027.

Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. May 2016. "Shreds: Fine-Grained Execution Units with Private Memory". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2016), 56–71. DOI: 10.1109/SP.2016.12.

Chromium Team. Apr. 9, 2019. *Inner sandbox escape on 64-bit Windows via KiUserExceptionDispatcher*. (Apr. 9, 2019). Retrieved June 12, 2025 from https://issuetracker.google.com/issues/42401312.

Chromium Team. 2020. *Memory safety*. The Chromium Projects. Retrieved May 7, 2025 from https://www.chromium.org/Home/chromium-security/memory-safety/.

Chromium Team. July 24, 2012. *Security: NaClSwitch() leaks NaClThreadContext pointer to x86-32 untrusted code*. (July 24, 2012). Retrieved June 12, 2025 from https://issuetracker.google.com/issues/42402545.

Chromium Team. Apr. 6, 2011. *Signal handling change allows inner sandbox escape on x86-32 Linux in Chrome*. (Apr. 6, 2011). Retrieved June 12, 2025 from https://issuetracker.google.com/issues/42401287.

Chromium Team. Aug. 3, 2010. *Uninitialized sendmsg syscall arguments in sel_ldr*. (Aug. 3, 2010). Retrieved June 12, 2025 from https://issuetracker.google.com/issues/42400629?pli=1.

Robert J. Colvin and Kirsten Winter. 2020. "An Abstract Semantics of Speculative Execution for Reasoning About Security Vulnerabilities". In: *Formal Methods. FM 2019 International Workshops* (Lecture Notes in Computer Science). Ed. by Emil Sekerinski et al. Springer International Publishing, Cham, 323–341. ISBN: 978-3-030-54997-8. DOI: 10.1007/978-3-030-54997-8_21.

[SW] Frank Denis, *libsodium* 2024. URL: https://doc.libsodium.org/.

Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. May 2019. "The Code That Never Ran: Modeling Attacks on Speculative Evaluation". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2019), 1238–1255. DOI: 10.1109/SP.2019.00047.

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. "XFI: Software Guards for System Address Spaces". In: OSDI '06, 75–88. https://www.usenix.org/legacy/events/osdi06/tech/erlingsson.html.

Xaver Fabian, Marco Guarnieri, and Marco Patrignani. Nov. 7, 2022. "Automatic Detection of Speculative Execution Combinations". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (CCS '22). Association for Computing Machinery, New York, NY, USA, (Nov. 7, 2022), 965–978. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560555.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Apr. 18, 2006. "Language support for fast and reliable message-based communication in singularity OS". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (EuroSys '06). Association for Computing

Machinery, New York, NY, USA, (Apr. 18, 2006), 177–190. isbn: 978-1-59593-322-5. doi: 10.1145/1217935.1217953.

Bryan Ford. 2005. "VXA: A Virtual Architecture for Durable Compressed Archives". In: *4th USENIX Conference on File and Storage Technologies—Abstract*. FAST 2005, 295–308. https://www.usenix.org/legacy/events/fast05/tech/ford.html.

Bryan Ford and Russ Cox. 2008. "Vx32: Lightweight User-level Sandboxing on the x86". *USENIX '08: 2008 USENIX Annual Technical Conference*.

Nathan Froyd. Feb. 25, 2020. *Securing Firefox with WebAssembly*. Mozilla Hacks. (Feb. 25, 2020). Retrieved June 12, 2025 from https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly.

Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Dec. 11, 2020. "Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge". In: *Proceedings of the 21st International Middleware Conference* (Middleware '20). Association for Computing Machinery, New York, NY, USA, (Dec. 11, 2020), 265–279. isbn: 978-1-4503-8153-6. doi: 10.1145/3423211.3425680.

J. A. Goguen and J. Meseguer. Apr. 1982. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. 1982 IEEE Symposium on Security and Privacy. ISSN: 1540-7993. (Apr. 1982), 11–11. doi: 10.1109/SP.1982.10014.

Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Nov. 2, 2020. "Speculative Probing: Hacking Blind in the Spectre Era". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (CCS '20). Association for Computing Machinery, New York, NY, USA, (Nov. 2, 2020), 1871–1885. isbn: 978-1-4503-7089-9. doi: 10.1145/3372297.3417289.

Google Project Zero. Jan. 12, 2021. *Project Zero: Introducing the In-the-Wild Series*. Project Zero. (Jan. 12, 2021). Retrieved May 8, 2025 from https://googleprojectzero.blogspot.com/2021/01/introducing-in-wild-series.html.

Nuwan Goonasekera, William Caelli, and Colin Fidge. 2015. "LibVM: an architecture for shared library sandboxing". *Software: Practice and Experience*, 45, 12, 1597–1617. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2294. doi: 10.1002/spe.2294.

Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. July 4, 2015. "Memory-safe Execution of C on a Java VM". In: *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security* (PLAS'15). Association for Computing Machinery, New York, NY, USA, (July 4, 2015), 16–27. isbn: 978-1-4503-3661-1. doi: 10.1145/2786558.2786565.

Roberto Guanciale, Musard Balliu, and Mads Dam. Oct. 30, 2020. "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security. ACM, Virtual Event USA, (Oct. 30, 2020), 1853–1869. isbn: 978-1-4503-7089-9. doi: 10.1145/3372297.3417246.

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. May 2020. "Spectector: Principled Detection of Speculative Information Flows". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2020), 1–19. doi: 10.1109/SP40000.2020.00011.

Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. May 2021. "Hardware-Software Contracts for Secure Speculation". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2021), 1868–1883. doi: 10.1109/SP40001.2021.00036.

Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. Oct. 2023. "Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust". In: *2023 IEEE Secure Development Conference (SecDev)*. 2023 IEEE Secure Development Conference (SecDev). (Oct. 2023), 54–66. doi: 10.1109/SecDev56634.2023.00020.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. June 14, 2017. "Bringing the web up to speed

with WebAssembly". *SIGPLAN Not.*, 52, 6, (June 14, 2017), 185–200. doi: 10.1145/3140587.3062363.

Lars T. Hansen. 2019. *Cranelift: Performance parity with Baldr on x86-64*. Retrieved June 12, 2025 from https://bugzilla.mozilla.org/show_bug.cgi?id=1539399.

Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. "Hodor: {Intra-Process} Isolation for {High-Throughput} Data Plane Libraries". In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), 489–504. isbn: 978-1-939133-03-8. Retrieved June 12, 2025 from https://www.usenix.org/conference/atc19/presentation/hedayati-hodor.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. June 2009. "Fault isolation for device drivers". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. ISSN: 2158-3927. (June 2009), 33–42. doi: 10.1109/DSN.2009.5270357.

Jann Horn. 2018. *speculative execution, variant 4: speculative store bypass*. Retrieved Mar. 17, 2025 from https://project-zero.issues.chromium.org/issues/42450580.

Galen C. Hunt and James R. Larus. Apr. 1, 2007. "Singularity: rethinking the software stack". *SIGOPS Oper. Syst. Rev.*, 41, 2, (Apr. 1, 2007), 37–49. doi: 10.1145/1243418.1243424.

Intel Corporation. 2023. *{Intel® 64} and {IA-32} Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/content-details/774476/intel-64-and-ia-32-architectures-software-developer-s-manual-volume-1-basic-architecture.html.

Intel Corporation. 2018a. *Complex Shadow-Stack Updates, Intel® Control-flow Enforcement Technology Specification*. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

Intel Corporation. 2017a. *CVE-2017-5715*. https://nvd.nist.gov/vuln/detail/CVE-2017-5715.

Intel Corporation. 2017b. *CVE-2017-5753*. https://nvd.nist.gov/vuln/detail/CVE-2017-5753.

Intel Corporation. 2018b. *CVE-2018-3639*. https://nvd.nist.gov/vuln/detail/CVE-2018-3639.

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. "Not So Fast: Analyzing the Performance of {WebAssembly} vs. Native Code". In: 2019 USENIX Annual Technical

Conference (USENIX ATC 19), 107–120. ISBN: 978-1-939133-03-8. Retrieved June 12, 2025 from https://www.usenix.org/conference/atc19/presentation/jangda.

Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Hang Zhang, Dawu Gu, Siqi Ma, Zhiyun Qian, and Juanru Li. Sept. 9, 2021. "Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK". In: 2022 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. IEEE Computer Society, (Sept. 9, 2021), 473–488. ISBN: 978-1-66541-316-9. DOI: 10.1109/SP46214.2022.00028.

Evan Johnson. 2021. *Update VeriWasm version*. GitHub. Retrieved June 12, 2025 from https://github.com/bytecodealliance/lucet/pull/684.

Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. "Доверяй, но проверяй: SFI safety for native-compiled Wasm". In: *Proceedings 2021 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Internet Society, Virtual. ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.24078.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Dec. 27, 2017. "RustBelt: securing the foundations of the rust programming language". *Proceedings of the ACM on Programming Languages*, 2, (Dec. 27, 2017), 1–34, POPL, (Dec. 27, 2017). DOI: 10.1145/3158154.

Paul A. Karger. Apr. 1, 1989. "Using registers to optimize cross-domain call performance". In: *Proceedings of the third international conference on Architectural support for programming languages and operating systems* (ASPLOS III). Association for Computing Machinery, New York, NY, USA, (Apr. 1, 1989), 194–204. ISBN: 978-0-89791-300-3. DOI: 10.1145/70082.68201.

Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Mar. 28, 2022. "PKRU-safe: automatically locking down the heap between safe and unsafe languages". In: *Proceedings of the Seventeenth European Conference on Computer Systems* (EuroSys '22). Association for Computing Machinery, New York, NY, USA, (Mar. 28, 2022), 132–148. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519582.

Paul Kocher et al.. May 2019. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2019), 1–19. DOI: 10.1109/SP.2019.00002.

Paul C. Kocher. 1996. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology — CRYPTO ’96*. Ed. by Neal Koblitz. Springer, Berlin, Heidelberg, 104–113. ISBN: 978-3-540-68697-2. DOI: 10.1007/3-540-68697-5_9.

Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. “Spectre Returns! Speculation Attacks using the Return Stack Buffer”. In: 12th USENIX Workshop on Offensive Technologies (WOOT 18). Retrieved Nov. 16, 2023 from https://www.usenix.org/conference/woot18/presentation/koruyeh.

Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. July 2014. “Portable Software Fault Isolation”. In: *2014 IEEE 27th Computer Security Foundations Symposium*. 2014 IEEE 27th Computer Security Foundations Symposium. ISSN: 2377-5459. (July 2014), 18–32. DOI: 10.1109/CSF.2014.10.

Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. “Code-Pointer Integrity”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 147–163. ISBN: 978-1-931971-16-4. Retrieved July 18, 2020 from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

C. Lattner and V. Adve. Mar. 2004. “LLVM: a compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. (Mar. 2004), 75–86. DOI: 10.1109/CGO.2004.1281665.

Xavier Leroy. Dec. 2009. “A Formally Verified Compiler Back-end”. *Journal of Automated Reasoning*, 43, 4, (Dec. 2009), 363–446. DOI: 10.1007/s10817-009-9155-4.

James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. “{Light-Weight} Contexts: An {OS} Abstraction for Safety and Performance”. In: 12th USENIX Symposium on Operating Systems Design and

Implementation (OSDI 16), 49–64. ISBN: 978-1-931971-33-1. Retrieved June 12, 2025 from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton.

H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2018. *System V Application Binary InterfaceAMD64 Architecture Processor Supplement(With LP64 and ILP32 Programming Models)*. https://software.intel.com/content/dam/develop/external /us/en/documents/intro-to-intel-avx-183287.pdf.

Steven Lucco, Oliver Sharp, and Robert Wahbe. Dec. 11, 1995. "Omniware: A Universal Substrate for Web Programming". In: *Proceedings of the Fourth International Conference on World Wide Web* (WWW4). Association for Computing Machinery, New York, NY, USA, (Dec. 11, 1995), 359–368. ISBN: 978-1-56592-169-6. DOI: 10.1145/3592626.3592655.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. May 2010. "Object Capabilities and Isolation of Untrusted Web Applications". In: *2010 IEEE Symposium on Security and Privacy*. 2010 IEEE Symposium on Security and Privacy. ISSN: 2375-1207. (May 2010), 125–140. DOI: 10.1109/SP.2010.16.

Giorgi Maisuradze and Christian Rossow. Oct. 15, 2018. "ret2spec: Speculative Execution Using Return Stack Buffers". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS '18). Association for Computing Machinery, New York, NY, USA, (Oct. 15, 2018), 2109–2122. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243761.

Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Sept. 2021. "Bypassing memory safety mechanisms through speculative control flow hijacks". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021 IEEE European Symposium on Security and Privacy (EuroS&P). (Sept. 2021), 633–649. DOI: 10.1109/EuroSP51992.2021.00048.

A.A. Matos and G. Boudol. June 2005. "On Declassification and the Non-Disclosure Policy". In: *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. 18th IEEE Computer Security Foundations Workshop (CSFW'05). ISSN: 2377-5459. (June 2005), 226–240. DOI: 10.1109/CSFW.2005.21.

Stephen McCamant and Greg Morrisett. 2006. "Evaluating SFI for a CISC Architecture". In: 15th {USENIX} Security Symposium ({USENIX} Security). Retrieved July 18, 2020 from https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture.

Tyler McMullen. 2020. "Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing". PriSC. (2020).

Kathleen Metrick, Jared Semrau, and Shambavi Sadayappan. Apr. 13, 2020. *Think Fast: Time Between Disclosure, Patch Release and Vulnerability Exploitation — Intelligence for Vulnerability Management, Part Two*. Google Cloud Blog. (Apr. 13, 2020). Retrieved May 8, 2025 from https://cloud.google.com/blog/topics/threat-intelligence/time-between-disclosure-patch-release-and-vulnerability-exploitation.

Adrian Mettler, David Wagner, and Tyler Close. Mar. 1, 2010. "Joe-E: A Security-Oriented Subset of Java". *NDSS Symposium 2010*, (Mar. 1, 2010).

Mark S. Miller, Michael Samuel, Ben Laurie, Ihab Awad, and Mike Stay. June 7, 2008. *Caja: Safe active content in sanitized JavaScript*. (June 7, 2008). http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

Matt Miller. Aug. 12, 2019. "Trends and Challenges in the Vulnerability Mitigation Landscape". 13th USENIX Workshop on Offensive Technologies (WOOT 2019). (Aug. 12, 2019).

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". In: *Information Security and Cryptology - ICISC 2005*. Ed. by Dong Ho Won and Seungjoo Kim. Springer, Berlin, Heidelberg, 156–168. ISBN: 978-3-540-33355-5. DOI: 10.1007/11734727_14.

Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richards Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999. *TALx86: A Realistic Typed Assembly Language*.

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Jan. 2002. "Stack-Based Typed Assembly Language". *Journal of Functional Programming*, 12, 1, (Jan. 2002), 43–88. Publisher: Cambridge University Press. DOI: 10.1017/S0956796801004178.

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. June 11, 2012. "RockSalt: better, faster, stronger SFI for the x86". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '12). Association for Computing Machinery, New York, NY, USA, (June 11, 2012), 395–404. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254111.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. May 1, 1999. "From System F to Typed Assembly Language". *ACM Transactions on Programming Languages and Systems*, 21, 3, (May 1, 1999), 527–568. DOI: 10.1145/319301.319345.

Nicholas Mosier, Hamed Nemati, John C. Mitchell, and Caroline Trippel. May 2024. "Serberus: Protecting Cryptographic Code from Spectres at Compile-Time". In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2024), 4200–4219. DOI: 10.1109/SP54263.2024.00048.

Mozilla. 2021. *Firefox Public Data Report*. Retrieved June 12, 2025 from https://data.firefox.com/dashboard/hardware.

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. "Retrofitting Fine Grain Isolation in the Firefox Renderer". In: 29th USENIX Security Symposium (USENIX Security 20), 699–716. ISBN: 978-1-939133-17-5. Retrieved June 12, 2025 from https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

Shravan Narayan, Craig Disselkoen, Daniel Moghimi, et al.. 2021. "Swivel: Hardening {WebAssembly} against Spectre". In: 30th USENIX Security Symposium (USENIX Security 21), 1433–1450. ISBN: 978-1-939133-24-3. Retrieved June 12, 2025 from https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. Dec. 4, 2019. *Gobi: WebAssembly as a Practical Path to Library Sandboxing*. (Dec. 4, 2019). arXiv: 1912.02285[cs]. doi: 10.48550/arXiv.1912.02285.

Ben Niu and Gang Tan. Nov. 3, 2014. "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (CCS '14). Association for Computing Machinery, New York, NY, USA, (Nov. 3, 2014), 1317–1328. isbn: 978-1-4503-2957-6. doi: 10.1145/2660267.2660281.

Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. 2024. "High-assurance zeroization". *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024, 1, 375–397. Number: 1. doi: 10.46586/tches.v2024.i1.375-397.

Oracle. 2019. *Java Platform, Standard Edition: Java Virtual Machine Guide*. https://docs.oracle.com/en/java/javase/13/vm/java-virtual-machine-guide.pdf.

Marco Patrignani, Amal Ahmed, and Dave Clarke. Feb. 27, 2019. "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work". *ACM Computing Surveys*, 51, 6, (Feb. 27, 2019), 1–36. doi: 10.1145/3280984.

Marco Patrignani and Marco Guarnieri. Nov. 12, 2021. "Exorcising Spectres with Secure Compilers". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (CCS '21). Association for Computing Machinery, New York, NY, USA, (Nov. 12, 2021), 445–461. isbn: 978-1-4503-8454-4. doi: 10.1145/3460120.3484534.

Mathias Payer and Thomas R. Gross. Mar. 9, 2011. "Fine-grained user-space security through virtualization". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (VEE '11). Association for Computing Machinery, New York, NY, USA, (Mar. 9, 2011), 157–168. isbn: 978-1-4503-0687-4. doi: 10.1145/1952682.1952703.

Colin Percival. Sept. 6, 2014. *Zeroing buffers is insufficient*. (Sept. 6, 2014). Retrieved June 12, 2025 from https://www.daemonology.net/blog/2014-09-06-zeroing-buffers-is-insufficient.html.

Hernán Ponce-de-Leon and Johannes Kinder. May 2022. "Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2022), 235–248. doi: 10.1109/SP46214.2022.9833774.

Jonathan Protzenko et al.. Aug. 29, 2017. "Verified low-level programming embedded in F*". *Proc. ACM Program. Lang.*, 1, (Aug. 29, 2017), 17:1–17:29, ICFP, (Aug. 29, 2017). doi: 10.1145/3110261.

Weizhong Qiang, Yong Cao, Weiqi Dai, Deqing Zou, Hai Jin, and Benxi Liu. Dec. 2017. "Libsec: A Hardware Virtualization-Based Isolation for Shared Library". In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). (Dec. 2017), 34–41. doi: 10.1109/HPCC-SmartCity-DSS.2017.5.

Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Dec. 6, 2021. "Keeping Safe Rust Safe with Galeed". In: *Proceedings of the 37th Annual Computer Security Applications Conference* (ACSAC '21). Association for Computing Machinery, New York, NY, USA, (Dec. 6, 2021), 824–836. isbn: 978-1-4503-8579-4. doi: 10.1145/3485832.3485903.

[SW] RustCrypto, *Zeroize* version 1.7.0, Nov. 16, 2023. url: https://docs.rs/zeroize/1.7.0/zeroize/.

Henrik Rydgård. Jan. 28, 2020. *Windows (Fastcall) calling convention: Callee-saved XMM (FP) registers are not actually saved*. GitHub. (Jan. 28, 2020). Retrieved June 12, 2025 from https://github.com/bytecodealliance/wasmtime/issues/1177.

David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. "Jenny: Securing Syscalls for {PKU-based} Memory Isolation Systems". In: 31st USENIX Security Symposium (USENIX Security 22), 936–952. isbn: 978-1-939133-31-1. Retrieved June 12, 2025 from https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel.

David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. “Donky: Domain Keys – Efficient {In-Process} Isolation for {RISC-V} and x86”. In: 29th USENIX Security Symposium (USENIX Security 20), 1677–1694. isbn: 978-1-939133-17-5. Retrieved June 12, 2025 from https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel.

David Sehr, Robert Muth, Karl Schimpf, Cliff Biffle, Victor Khimenko, Bennet Yee, Brad Chen, and Egor Pasko. 2010. “Adapting Software Fault Isolation to Contemporary CPU Architectures”. In: *19th {USENIX} Security Symposium ({USENIX} Security 10)*. {USENIX} Association.

Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Oct. 28, 1996. “Dealing with disaster: surviving misbehaved kernel extensions”. In: *Proceedings of the second USENIX symposium on Operating systems design and implementation* (OSDI ’96). Association for Computing Machinery, New York, NY, USA, (Oct. 28, 1996), 213–227. isbn: 978-1-880446-82-9. doi: 10.1145/238721.238779.

Basavesh Ammanaghatta Shivakumar et al.. May 2023. “Spectre Declassified: Reading from the Right Place at the Wrong Time”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2023), 1753–1770. doi: 10.1109/SP46215.2023.10179355.

Joseph Siefers, Gang Tan, and Greg Morrisett. Oct. 4, 2010. “Robusta: taming the native beast of the JVM”. In: *Proceedings of the 17th ACM conference on Computer and communications security* (CCS ’10). Association for Computing Machinery, New York, NY, USA, (Oct. 4, 2010), 201–211. isbn: 978-1-4503-0245-6. doi: 10.1145/1866307.1866331.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Jan. 2, 2019. “StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities”. *Proceedings of the ACM on Programming Languages*, 3, (Jan. 2, 2019), 1–28, POPL, (Jan. 2, 2019). doi: 10.1145/3290332.

Gang Tan. 2017. “Principles and Implementation Techniques of Software-Based Fault Isolation”. *Foundations and Trends® in Privacy and Security*, 1, 3, 137–198. doi: 10.1561/3300000013.

[SW] The Coq Development Team, *Rocq* version 9.0.0, Jan. 10, 2025. URL: https://rocq-prover.org/.

The LLVM Foundation. 2021a. *Control Flow Integrity*. Clang 12 Documentation. Retrieved June 12, 2025 from https://clang.llvm.org/docs/ControlFlowIntegrity.html.

The LLVM Foundation. 2021b. *SafeStack*. Clang 12 Documentation. Retrieved June 12, 2025 from https://clang.llvm.org/docs/SafeStack.html.

Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". *Proceedings of the 23rd {USENIX} Security Symposium*, 941–955. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf.

Alex Tsariounov. 2021. *Shielding Linux Resources*. Retrieved June 12, 2025 from https://documentation.suse.com/sle-rt/15-SP1/html/SLE-RT-all/cha-shielding-intro.html.

Reini Urban. Jan. 23, 2019. *libsodium_memzero with memory barrier*. GitHub. (Jan. 23, 2019). https://github.com/jedisct1/libsodium/issues/802.

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)". In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. {USENIX} Association, 1221–1238. ISBN: 978-1-939133-06-9.

[SW] VAMPIRE, *System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives* version 20221122, Nov. 22, 2022. URL: https://bench.cr.yp.to/supercop.html.

Varda. Oct. 1, 2018. *WebAssembly on Cloudflare Workers*. The Cloudflare Blog. (Oct. 1, 2018). Retrieved June 12, 2025 from https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Jan. 4, 2021. "Automatically eliminating speculative leaks from cryptographic code with blade". *Proceedings of the ACM on Programming Languages*, 5, (Jan. 4, 2021), 49:1–49:30, POPL, (Jan. 4, 2021). DOI: 10.1145/3434330.

Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. Mar. 28, 2022. "You shall not (by)pass! practical, secure, and fast PKU-based sandboxing". In: *Proceedings of the Seventeenth European Conference on Computer Systems* (EuroSys '22). Association for Computing Machinery, New York, NY, USA, (Mar. 28, 2022), 266–282. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519560.

W3C. Dec. 5, 2019. *WebAssembly Core Specification*. (Dec. 5, 2019). Retrieved June 12, 2025 from https://www.w3.org/TR/wasm-core-1/.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. "Efficient Software-Based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (SOSP '93). Association for Computing Machinery, Asheville, North Carolina, USA, 203–216. ISBN: 978-0-89791-632-5. DOI: 10.1145/168619.168635.

Robert N.M. Watson et al.. May 2015. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. ISSN: 2375-1207. (May 2015), 20–37. DOI: 10.1109/SP.2015.9.

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Jan. 2, 2019. "CT-wasm: type-driven secure cryptography for the web ecosystem". *Proofs, Source, and Evaluation Scripts for CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem*, 3, (Jan. 2, 2019), 77:1–77:29, POPL, (Jan. 2, 2019). DOI: 10.1145/3290390.

Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Oct. 10, 2019. "Weakening WebAssembly". *Proc. ACM Program. Lang.*, 3, (Oct. 10, 2019), 133:1–133:28, OOPSLA, (Oct. 10, 2019). DOI: 10.1145/3360559.

WebAssembly Community Group. 2021. *Exception Handling*. Retrieved June 12, 2025 from https://github.com/WebAssembly/exception-handling.

Johannes Wikner and Kaveh Razavi. 2022. "{RETBLEED}: Arbitrary Speculative Code Execution with Return Instructions". In: 31st USENIX Security Symposium (USENIX Security 22), 3825–3842. ISBN: 978-1-939133-31-1. Retrieved June 12, 2025 from https://www.usenix.org/conference/usenixsecurity22/presentation/wikner.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. June 4, 2011. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). Association for Computing Machinery, New York, NY, USA, (June 4, 2011), 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532.

Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. "Dead Store Elimination (Still) Considered Harmful". In: 26th USENIX Security Symposium (USENIX Security 17), 1025–1040. ISBN: 978-1-931971-40-9. Retrieved Sept. 16, 2022 from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/yang.

Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. May 2023. "Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. (May 2023), 1220–1237. DOI: 10.1109/SP46215.2023.10179415.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. May 17, 2009. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (SP '09). IEEE Computer Society, USA, (May 17, 2009), 79–93. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.25.

[SW] Alon Zakai, *WasmBoxC* 2020. URL: https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html.

Bin Zeng, Gang Tan, and Greg Morrisett. Oct. 17, 2011. "Combining control-flow integrity and static analysis for efficient and validated data sandboxing". In: *Proceedings of the 18th ACM conference on Computer and communications security* (CCS '11). Association for Computing Machinery, New York, NY, USA, (Oct. 17, 2011), 29–40. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046713.

Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Oct. 9, 2011. "ARMor: fully verified software fault isolation". In: *Proceedings of the ninth ACM international conference on Embedded software* (EMSOFT '11). Association for Computing Machinery, New York, NY, USA, (Oct. 9, 2011), 289–298. ISBN: 978-1-4503-0714-7. DOI: 10.1145/2038642.2038687.

Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Nov. 3, 2014. "ARMlock: Hardware-based Fault Isolation for ARM". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (CCS '14). Association for Computing Machinery, New York, NY, USA, (Nov. 3, 2014), 558–569. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660344.